The algxpar package

0.99.2a

Jander Moreira – moreira.jander@gmail.com

2025/07/31

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

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

.

Change History

0.9 (2019-11-12)

Initial version.

0.91 (2020-08-01)

`\Id` has been recoded to work in both text and math modes, preventing hyphenation 12

`\Set` can be used for assignments. 13

0.99 (2023-06-28)

The code was reviewd and almost entirely rewritten.

The "look and feel" of the pseudocode can be set in the `\begin{algorithmic}` 7

`\Set` is deprecated and will no longer be supported 13

`\Setl` has been removed from the package 13

Added support to style the main components of the pseudocode, such as keywords, comments and text and widths. 18

New mechanism to translation with separate files for each language 23

Added support to style named constants and function/procedures identifiers. . . . 26

Macros `\TextString` and `\VisibleSpace` accidentally removed. 26

0.99.1 (2024/05/05)

Added experimental support to show vertical indentantion lines. 18

`line style` option added to style indent lines. 19

0.99.2 (2024/10/23)

Part of the code was rewritten using etoolbox.

Wrong spacing between parboxes were fixed. 22

0.99.2a (2025/07/31)

Fixed length computation in non-commented lines to consider the token length.

`\Id` updated to natively handle the underscore character without escaping. 12

1 Introduction

I teach algorithms and programming, and for writing my algorithms, I've adopted the algorithmicx package (algpseudocode). It allows me to create pseudocode that's both clear and easy to read, requiring minimal effort to ensure it looks visually appealing.

Teaching algorithms involves a different approach to pseudocode compared to its typical use in scientific papers, where solutions are often presented in a more formal, concise manner. Students typically work with abstract algorithms before learning a specific programming language, so the focus is more on the logic of the solution than on the variables themselves. Additionally, teaching strategies that emphasize refinement and iteration call for a less programmatic and more descriptive style of code. For instance, instead of writing something like "$s \leftarrow s + c$," we might say, "*accumulate current expenses in the total sum of costs*," as the latter provides a clearer explanation of the logic without requiring students to understand the specifics of variable manipulation.

However, this more verbose approach often results in longer sentences that can span multiple lines. Given that pseudocode places a premium on visual clarity—particularly in control structures and indentation—it became necessary to create a package that accommodates multi-line statements while maintaining readability.

The algorithmicx and algpseudocode packages do not natively support multi-line statements. Therefore, this package extends several macros to manage multi-line code properly, adding new commands and features to improve its functionality.

```
\begin{algorithmic}[1]
    \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
    \Statex
    \Statep{Initialize a table with all byte values}[each table position holds a
    ↪ single byte]
    \Statep{Initialize \Id{sequence} with the first byte from the input stream}
    \While{there are bytes in the input}[wait until all bytes have been
    ↪ processed]
        \Statep{Retrieve a single byte from the input and store it in \Id{byte}}
        \If{the concatenation of \Id{sequence} and \Id{byte} exists in the
        ↪ table}
            \Statep{Set \Id{sequence} to $\Id{sequence} +
            ↪ \Id{byte}$}[concatenate without generating any output]
        \Else
            \Statep{Output the code for \Id{sequence}}[i.e., the binary
            ↪ representation of its index in the table]
            \Statep{Add the concatenation of \Id{sequence} and \Id{byte} to the
            ↪ table}[the table learns a new, longer sequence]
            \Statep{Set \Id{sequence} to \Id{byte}}[begin a new sequence with
            ↪ the remaining byte]
        \EndIf
    \EndWhile
    \Statep{Output the code for \Id{sequence}}[the remaining bit sequence]
\end{algorithmic}
```

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 byte values ▷ *each table position holds a single byte*
2: Initialize *sequence* with the first byte from the input stream
3: **while** there are bytes in the input **do** ▷ *wait until all bytes have been processed*
4: Retrieve a single byte from the input and store it in *byte*
5: **if** the concatenation of *sequence* and *byte* exists in the table **then**
6: Set *sequence* to *sequence* + *byte* ▷ *concatenate without generating any output*
7: **else**
8: Output the code for *sequence* ▷ *i.e., the binary representation of its index in the table*
9: Add the concatenation of *sequence* and *byte* to the table ▷ *the table learns a new, longer sequence*
10: Set *sequence* to *byte* ▷ *begin a new sequence with the remaining byte*
11: **end if**
12: **end while**
13: Output the code for *sequence* ▷ *the remaining bit sequence*

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)
etoolbox	(https://ctan.org/pkg/etoolbox)
pgfmath	(https://ctan.org/pkg/pgf)
pgfopts	(https://ctan.org/pkg/pgf)
ragged2e	(https://ctan.org/pkg/ragged2e)
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.

`\usepackage[`⟨*package options list*⟩`]{algxpar}`

Currently, the list of package options includes the following.

⟨*language name*⟩

Algorithm keywords are typically created in English by default, with the English keyword set always being loaded. When available, one can use keyword sets in other languages by specifying the desired language.
Currently supported languages:

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

```
% Loads Brazilian keyword set and sets it as default
\usepackage[brazilian]{algxpar}
```

`language` = ⟨*language name*⟩

This option selects the keyword set corresponding to ⟨*language name*⟩ as the document's default. It is available as a general option (see `language`).
This option is useful when other languages are loaded.

```
% Loads Brazilian keyword set but keeps English as default
\usepackage[brazilian, language = english]{algxpar}
```

`noend` = `true` | `false` Default: `true`; initially: `true`

The `noend` option suppresses the line marking the end of a block, while preserving the indentation.
See more information in `end` and `noend` options.

```
% Suppresses all end-lines that close a block
\usepackage[noend]{algxpar}
```

3 Writting pseudocode

Algorithms, following the structure of the algorithmicx package, are written within the `algorithmic` environment. The option to use a number to determine line numbering is preserved from the original version.

An algorithm consists of instructions, control structures like conditionals and loops, as well as documentation and comments.

```
\begin{algorithmic}
    \Description Calculation of the factorial of a natural number
    \Input $n \in \mathbb{N}$
    \Output $n!$
    \Statex
    \Statep{\Read $n$}
    \Statep{$\Id{factorial} \gets 1$}[$0! = 1! = 1$]
    \For{$k \gets 2$ \To $n$}[from 2 up]
        \Statep{$\Id{factorial} \gets \Id{factorial} \times k$}[$(k-1)! \times
        ↪  k$]
    \EndFor
    \Statep{\Write \Id{factorial}}
\end{algorithmic}
```

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 Euclid's algorithm as shown in the algorithmicx package documentation[2].

```
\begin{algorithmic}[1]
    \Procedure{Euclid}{$a,b$}
        \Comment{The g.c.d. of a and b}
        \State $r\gets a\bmod b$
        \While{$r\not=0$}
            \Comment{We have the answer if r is 0}
            \State $a\gets b$
            \State $b\gets r$
            \State $r\gets a\bmod b$
        \EndWhile
        \State \textbf{return} $b$\Comment{The gcd is b}
    \EndProcedure
\end{algorithmic}
```

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**

[2] A label has been suppressed here.

```
    return b                                                    ▷ The gcd is b
end procedure
```

Comments are added *in loco* with the `\Comment` 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.

```
\begin{algorithmic}[1]
    \Procedure[The g.c.d. of a and b]{Euclid}{$a,b$}  % <-- Comment
        \State $r\gets a\bmod b$
        \While[We have the answer if r is 0]{$r\not=0$}  % <-- Comment
            \State $a\gets b$
            \State $b\gets r$
            \State $r\gets a\bmod b$
        \EndWhile
        \Statep[The gcd is b]{\Keyword{return} $b$}   % <-- Comment
    \EndProcedure
\end{algorithmic}
```

```
1: procedure EUCLID(a, b)                                       ▷ The g.c.d. of a and b
2:     r ← a mod b
3:     while r ≠ 0 do                                           ▷ We have the answer if r is 0
4:         a ← b
5:         b ← r
6:         r ← a mod 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. hus, as of v0.99, the comment can be placed after the text (as the second parameter of the macro), certainly making writing algorithms more user-friendly. To maintain backward compatibility, the use of comments before text is still supported, although it is discouraged.

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

```
\begin{algorithmic}[1]
    \Procedure{Euclid}{$a,b$}[The g.c.d. of a and b]  % <-- Comment
        \State $r\gets a\bmod b$
        \While{$r\not=0$}[We have the answer if r is 0]  % <-- Comment
            \State $a\gets b$
            \State $b\gets r$
            \State $r\gets a\bmod b$
        \EndWhile[end of the loop]  % <-- Comment
        \Statep{\Keyword{return} $b$}[The gcd is b]  % <-- Comment
    \EndProcedure
\end{algorithmic}
```

```
1: procedure EUCLID(a, b)                                       ▷ The g.c.d. of a and b
2:     r ← a mod b
3:     while r ≠ 0 do                                           ▷ We have the answer if r is 0
4:         a ← b
5:         b ← r
6:         r ← a mod b
7:     end while                                                ▷ end of the loop
```

```
8:     return b                                                    ▷ The gcd is b
9: end procedure
```

Using `\Comment` 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.

```
\begin{algorithmic}
    <keyword font = \scshape\bfseries, comment width = nice>
    \If{$a > b$}[check conditions]
        \While{$a > 0$}
            <keyword color = blue!70>
            \Statep{\Call{Process}{$a$}}[process current data]
        \EndWhile
    \EndIf
\end{algorithmic}
```

```
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 `\State` and `\Statex` 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 `\Statep` macro is available, which should be used instead of `\State`.

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

> The `\Statep` 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 apply only to this macro.

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.

```
\begin{algorithmic}
    \State \Read $v$
    \If{$v < 0$}[is it negative?]
        \Statep{$v \gets -v$}[make it positive]
    \EndIf
\end{algorithmic}
```

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

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

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

`\EndIf[`⟨*comment*⟩`]<`⟨*options*⟩`>`

`\EndIf` closes its respective `\If`.
Any ⟨*options*⟩ specified apply only to this macro.

`\Else[`⟨*comment*⟩`]<`⟨*options*⟩`>`

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

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

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

3.4.2 The switch block

```
\begin{algorithmic}
    \Statep{Get \Id{Optiondef}}
    \Switch{\Id{Optiondef}}
    \Case{1}[inserts new record]
    \Statep{\Call{Insert}{\Id{record}}}
    \EndCase
    \Case{2}[deletes a record]
    \Statep{\Call{Delete}{\Id{key}}}
    \EndCase
    \Otherwise
    \Statep{Print ``invalid option''}
    \EndOtherwise
    \EndSwitch
\end{algorithmic}
```

```
Get Optiondef
switch Optiondef
    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
```

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

The `\Switch` is closed by a matching `\EndSwitch`.
Any ⟨*options*⟩ specified in this macro will impact this command and all items within the inner block, propagating up to and including the closing macro.

`\EndSwitch[`⟨*comment*⟩`]<`⟨*options*⟩`>`

This macro closes a `\Switch` block.
Any ⟨*options*⟩ specified apply only to this macro.

`\Case[`⟨*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 ⟨*options*⟩ specified in this macro will impact this command and all items within the inner block, propagating up to and including the closing macro.

`\EndCase[`⟨*comment*⟩`]<`⟨*options*⟩`>`

This macro closes a corresponding `\Case` statement.
Any ⟨*options*⟩ specified apply only to this macro.

`\Otherwise[`⟨*comment*⟩`]<`⟨*options*⟩`>`

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

`\EndOtherwise[`⟨*comment*⟩`]<`⟨*options*⟩`>`

This macro closes a corresponding `\Otherwise` statement.
Any ⟨*options*⟩ specified apply only to this macro.

3.4.3 The for block

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

```
\begin{algorithmic}
    \For{$i \gets 0$ \To $n$}
        \Statep{Do something with $i$}
    \EndFor
    \ForAll{$\Id{item} \in C$}
        \Statep{Do something with \Id{item}}
    \EndFor
    \ForEach{\Id{item} in queue $Q$}
        \Statep{Do something with \Id{item}}
    \EndFor
\end{algorithmic}
```

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

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

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

\EndFor[⟨*comment*⟩]<⟨*Optiondef*⟩>

This macro closes a corresponding \For, \ForEach or \ForAll.
Any ⟨*options*⟩ specified apply only to this macro.

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

Same as \For.

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

Same as \For.

3.4.4 The while block

\While is the loop with testing condition at the top.

```
\begin{algorithmic}
    \While{$n > 0$}
        \Statep{Do something}
        \Statep{$n \gets n - 1$}
    \EndWhile
\end{algorithmic}
```

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

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

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

`\EndWhile[`⟨*comment*⟩`]<`⟨*options*⟩`>`

This macro closes a matching `\While` block.
Any ⟨*options*⟩ specified apply only to this macro.

3.4.5 The repeat-until block

The loop with testing condition at the bottom is the `\Repeat`/`\Until` block.

```
\begin{algorithmic}
    \Repeat
        \Statep{Do something}
        \Statep{$n \gets n - 1$}
    \Until{$n \leq 0$}
\end{algorithmic}
```

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

`\Repeat[`⟨*comment*⟩`]<`⟨*options*⟩`>`

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

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

In ⟨*text*⟩ is the boolean expression that, when `\True`, will end the loop.
Any ⟨*options*⟩ specified apply only to this macro.

3.4.6 The loop block

A generic loop is build with `\Loop`.

```
\begin{algorithmic}
    \Loop
        \Statep{Do something}
        \Statep{$n \gets n + 1$}
        \If{$n$ is multiple of 5}
            \Statep{\Continue}[restarts loop]
```

```
        \EndIf
        \Statep{Do something else}
        \If{$n \leq 0$}
            \Statep{\Break}[ends loop]
        \EndIf
        \Statep{Keep working}
    \EndLoop
\end{algorithmic}
```

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

`\Loop[`⟨*comment*⟩`]<`⟨*options*⟩`>`

The generic loop starts with `\Loop` and ends with `\EndLoop`. Usually the infinite loop is interrupted by and internal `\Break` or restarted with `\Continue`. Any ⟨*options*⟩ specified in this macro will impact this command and all items within the inner block, propagating up to and including the closing macro.

`\EndLoop[`⟨*comment*⟩`]<`⟨*options*⟩`>`

`\EndLoop` closes a matching `\Loop` block.
Any ⟨*options*⟩ specified apply only to this macro.

3.5 Constants and Identifiers

A few macros for well known constants were defined: `\True` (TRUE), `\False` (FALSE), and `\Nil` (NIL).

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

`\Id{`⟨*identifier*⟩`}`

Updated in 0.99.2a

Identifiers are emphasised: `\Id{my_value}` is *my_value*. Its designed to work in both text and math modes: `$\Id{offer}_k$` is $offer_k$.

3.6 Assignments and I/O

To support teaching-like, basic pseudocode writing, the macros `\Read` and `\Write` are provided.

```
\begin{algorithmic}
    \Statep{\Read $v_1, v_2$}
    \Statep{$\Id{mean} \gets \dfrac{v_1 + v_2}{2}$}[calculate]
    \Statep{\Write \Id{mean}}
```

```
\end{algorithmic}
```

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

The `\Set` macro, although obsolete, can be used for assignments.

`\Set{`⟨*lvalue*⟩`}{`⟨*expression*⟩`}`

Deprecated in 0.99

This macro expands to `\Id{#1} \gets #2`.
It will no longer be supported but will be retained as-is for backward compatibility. Since proper handling of text and math modes is needed and its usage offers no clear benefit, there is no justification to keep it.

`\Setl`:
Removed in 0.99

The macro `\Setl` has been removed.

3.7 Procedures and Functions

Modularization uses `\Procedure` or `\Function`.

```
\begin{algorithmic}
    \Procedure{SaveNode}{\Id{node}}[saves a B\textsuperscript{+}-tree node to
    ↪  disk]
        \If{\Id{node}.\Id{is_modified}}
            \If{$\Id{node}.\Id{address} = -1$}
                \Statep{Set file writting position after file's last
                ↪  byte}[creates a new node on disk]
            \Else
                \Statep{Set file writting position to
                ↪  \Id{node}.\Id{address}}[updates the node]
            \EndIf
            \Statep{Write \Id{node} to disk}
            \Statep{$\Id{node}.\Id{is_modified} \gets \False$}
        \EndIf
    \EndProcedure
\end{algorithmic}
```

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 ← False
  end if
end procedure

```
\begin{algorithmic}
    \Function{Factorial}{$n$}[$n \geq 0$]
        \If{$n \in \{0, 1\}$}
            \Statep{\Return $1$}[base case]
        \Else
            \Statep{\Return $n \times \Call{Factorial}{n-1}$}[recursive case]
        \EndIf
    \EndFunction
```

```
\end{algorithmic}
```

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

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

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

`\EndProcedure[`⟨*comment*⟩`]<`⟨*optons*⟩`>`

This macro closes the `\Procedure` block.
Any ⟨*options*⟩ specified apply only to this macro.

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

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

`\EndFunction[`⟨*comment*⟩`]<`⟨*optons*⟩`>`

This macro closes the `\Function` block.
Any ⟨*options*⟩ specified apply only to this macro.

For calling a procedure or function, `\Call` should be used.

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

`\Call` is used to state a function or procedure call. The module's ⟨*name*⟩ and ⟨*arguments*⟩ are mandatory.
Any ⟨*options*⟩ specified apply only to this macro.

3.8 Comments

The `\Comment` macro, as defined by the algorithmicx package, retains its original behavior and has been redefined to support styling options.

`\Comment{`⟨*text*⟩`}<`⟨*options*⟩`>`

The updated `\Comment` can be used with `\State`, `\Statex`, and `\Statep`. When used with `\Statep`, it should be enclosed within the text braces, though multi-line statements may function differently than expected.
Any ⟨*options*⟩ specified apply only to this macro.

```
\begin{minipage}{7.5cm}
    \begin{algorithmic}
        <comment color = blue>%  for viewing purposes only
        \State Assign the value zero to the variable $x$\Comment{first
        ↪  assignment}
        \Statep{Assign the value zero to the variable $x$\Comment{first
        ↪  assignment}}
        \Statep{Assign the value zero to the variable $x$}[first assignment]%
        ↪  best choice
    \end{algorithmic}
\end{minipage}
```

Assign the value zero to the variable x ▷ *first assignment*
Assign the value zero to the variable x ▷ *first assignment*
Assign the value zero to the variable x ▷ *first assignment*

`\Commentl{`⟨*text*⟩`}<`⟨*options*⟩`>`

While `\Comment` pushes text to the end of the line, the `\Commentl` macro is "local" – it simply places a comment without altering the line structure.
Local comments follows regular text and no line changes are checked.
Any ⟨*options*⟩ specified apply only to this macro.

```
\begin{algorithmic}
    \If{$a > 0$~~\Commentl{special case}\\
    or\\
        $a < b$~~\Commentl{general case}\\}
        \Statep{Process data~~\Commentl{may take a while}}
    \EndIf
\end{algorithmic}
```

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

`\CommentIn{`⟨*text*⟩`}<`⟨*options*⟩`>`

`\CommentIn` is an alternative to *line comments*, which typically extend to the end of the line. This macro defines a comment with both a beginning and an end. A comment starts with ▷ and ends with ◁.
Any ⟨*options*⟩ specified apply only to this macro.

```
\begin{algorithmic}
    \If{$a > 0$ \CommentIn{special case} or $a < b$ \CommentIn{general case}}
        \Statep{Process data~~\Commentl{may take a while}}
    \EndIf
\end{algorithmic}
```

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.

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

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

`\Description` ⟨*description text*⟩

The `\Description` is intended to hold the general description of the pseudocode.

`\Require` ⟨*pre-conditions*⟩

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

`\Ensure` ⟨*post-conditions*⟩

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

```
\begin{algorithmic}
    \Description Calculation of the factorial of a natural number through
    ↪  successive multiplications
    \Input $n$ (integer)
    \Output $n!$ (integer)
\end{algorithmic}
```

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

`\Input` ⟨*inputs*⟩

This works as an alternative to `\Require`, presenting **Input**.

`\Output` ⟨*outputs*⟩

This works as an alternative to `\Ensure`, presenting **Output**.

4 Customization and Fine Tunning

Starting from version 0.99 of algxpar, a set of options has been introduced to customize the presentation of algorithms. For instance, colors and fonts specific to keywords can now be specified, offering a more convenient way to tailor each algorithm.

The `\AlgSet` macro fulfills this purpose.

`\AlgSet{`⟨*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 from the point where the macro is called. To limit the scope of a definition made with `\AlgSet` to a specific part of the document, simply enclose it in a TEX group.

```
\AlgSet{algorithmic indent = 1.5cm}
\begin{algorithmic}
    \Statep{\Read $k$}
    \If{$k < 0$}
        \Statep{$k \gets -k$}
    \EndIf
    \Statep{\Write $k$}
\end{algorithmic}
```

read k
if $k < 0$ **then**
$k \leftarrow -k$
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.

```
\begin{algorithmic}
    <keyword font = \sffamily\bfseries\itshape>
    \Statep{\Read $k$}
    \If{$k < 0$}
        \Statep{$k \gets -k$}
    \EndIf
    \Statep{\Write $k$}
\end{algorithmic}
```

read k
if $k < 0$ ***then***
$k \leftarrow -k$
end if
write k

Named styles can also be defined using the pgfkeys syntax.

```
\AlgSet{
    fancy/.style = {
        text color = green!40!black,
        keyword color = blue!75!black,
        comment color = brown!80!black,
        comment symbol = \texttt{//},
```

```
    }
}
\begin{algorithmic}
    <fancy>
    \Statep{\Commentl{Process $k$}}
    \Statep{\Read $k$}
    \If{$k < 0$}
        \Statep{$k \gets -k$}[back to positive]
    \EndIf
    \Statep{\Write $k$}
\end{algorithmic}
```

```
// Process k
read k
if k < 0 then
    k ← −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.

```
\AlgSet{
    highlight/.style = {
        text color = red!60!black,
        keyword color = red!60!black,
    }
}
\begin{algorithmic}
    \Statep{\Commentl{Process $k$}}
    \Statep{\Read $k$}
    \If{$k < 0$}
        <highlight>
        \Statep{$k \gets -k$}[back to positive]
    \EndIf
    \Statep{\Write $k$}
\end{algorithmic}
```

```
▷ Process k
read k
if k < 0 then
    k ← −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 `\AlgSet` or the ⟨*options*⟩ parameter of the various macros.

`indent lines = true | false` Default: `true`; initially: `false`

New in 0.99.1

The `indent lines` option displays vertical indentation lines in the pseudocode. This feature works properly only if the start and end of the block are on the same page. Additionally, at least two compilation steps are needed for the lines to be correctly positioned.
To change the style for the lines, use `line style`
This feature is still experimental and incomplete.

```
\begin{algorithmic}
    <indent lines>
    \For{$i \gets 0$ \To $N - 1$}
        \For{$j \gets$ \To $N - 1$}
            \If{$m_{ij} < 0$}
                \Statep{$m_{ij} \gets 0$}
            \EndIf
        \EndFor
    \EndFor
\end{algorithmic}

\begin{algorithmic}
    <indent lines, noend>
    \For{$i \gets 0$ \To $N - 1$}
        \For{$j \gets$ \To $N - 1$}
            \If{$m_{ij} < 0$}
                \Statep{$m_{ij} \gets 0$}
            \EndIf
        \EndFor
    \EndFor
\end{algorithmic}
```

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

`line style = ⟨style⟩`

New in 0.99.1

Almost anything TikZ can apply to a line can be used to set the indentation lines.

To show indentation lines in algorithms, `indent lines` must be set to true.

```
\begin{algorithmic}
    <indent lines, line style = {orange, dotted, -latex, ultra thick}>
    \For{$i \gets 0$ \To $N - 1$}
        \For{$j \gets$ \To $N - 1$}
            \If{$m_{ij} < 0$}
                \Statep{$m_{ij} \gets 0$}
            \EndIf
        \EndFor
    \EndFor
\end{algorithmic}
```

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
  end for
end for

`language` = ⟨*language*⟩

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

`noend` S

tructured algorithms use blocks to define their structures, marking both the beginning and the end. In pseudocode, it is common to use a line to indicate the end of a block. The `end` option suppresses this line.
The result resembles a program written in Python.

`end` T

his option reverses the behaviour of `noend`, and the closing line of a block presented.

```
\begin{algorithmic}
    <noend>
    \For{$i \gets 0$ \To $N - 1$}
        \For{$j \gets$ \To $N - 1$}
            \If{$m_{ij} < 0$}
                <end>
                \Statep{$m_{ij} \gets 0$}
            \EndIf
        \EndFor
    \EndFor
\end{algorithmic}
```

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 you to modify an existing keyword or define a new one. For more information on keywords and translations, see section 4.2.

```
\begin{algorithmic}
    <
    keywords = {
        terminate = Terminate, %  new keyword
        then = \{, %  redefined
        endif = \}, %  redefined
        while = whilst,  %  redefined
    }
    >
    \While{\True}
        \If{$t < 0$}
            \Statep{Run the \Keyword{terminate} module}
        \EndIf
    \EndWhile
\end{algorithmic}
```

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

`algorithmic indent` = ⟨*width*⟩ Initially: `1.5em`

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

`comment symbol` = ⟨*symbol*⟩ Initially: `$\triangleright$`

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 `\CommentSymbol`. Do not change this symbol by redefining `\CommentSymbol`, as font, shape and color settings will no longer be respected. Always use `comment symbol`.

`comment symbol right` = ⟨*symbol*⟩ Initially: `$\triangleleft$`

This symbol closes a `\CommentIn`. It is set to ◁ and can be accessed using the `\CommentSymbolRight` macro. Avoid changing the symbol by redefining `\CommentSymbolRight`, as this would cause font, shape, and color settings to be ignored. Always use the `comment symbol right` option.

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 `\color` with font options will tend to break the document.

`text font` = ⟨*font, shape and size*⟩ Initially: 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*⟩ Initially: `\slshape`

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

`keyword font` = ⟨*font, shape and size*⟩ Initially: `\bfseries`

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*⟩ Initially: `\scshape`

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 `\Constant` is used.

`module font` = ⟨*font, shape and size*⟩ Initially: `\scshape`

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

4.1.2 Colors

Colors are defined using the xcolors package.

`text color` = ⟨*color*⟩ Initially: .

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

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

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

`keyword color` = ⟨*color*⟩ Initially: .

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

`constant color` = ⟨*color*⟩ Initially: .

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

`module color` = ⟨*color*⟩ Initially: .

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

4.1.3 Paragraphs

Multi-line support are internally handled by `\parbox`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*⟩ Initially: `\RaggedRight`

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

`comment style` = ⟨*style*⟩ Initially: `\RaggedRight`

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

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

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

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

This is the `\hangindent` set inside `\Statep` statements.

`comment width` = `auto`|`nice`|⟨*width*⟩ Initially: `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 `0.4\linewidth`.
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.

```
\begin{algorithmic}
    <language = brazilian>
    \Procedure{Euclid}{$a,b$}
        \State $r\gets a\bmod b$
        \While{$r\not=0$}
            \State $a\gets b$
            \State $b\gets r$
            \State $r\gets a\bmod b$
        \EndWhile
        \Statep{\Keyword{return} $b$}
    \EndProcedure
```

```
\end{algorithmic}
```

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 `\AlgLanguageSet` macro.

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

This macro assigns new values to existing keywords as well as creates new ones. Once defined, keywords cannot be removed.
If a default keyword is not redefined, the English version will be used.
To create a new set, copy the file algxpar-english.kw.tex and modify it as needed. Note that there is a set of keywords for the lines that close each block. These keys are included to offer more flexibility in customizing how these lines are displayed. It is strongly recommended to use the Keyword macro for references to other keywords, so that font, color, and language changes can be made without issues.
In translations, these *compound keywords* are not necessarily required (see the file brazilian.kw.tex, which follows the settings in algxpar-english.kw.tex). However, if they are defined, there will be different versions for each language.

```
% English keywords
% Moreira, J. (moreira.jander@gmail.com)
\AlgLanguageSet{english}{%
    description = Description,
    input = Input,
    output = Output,
    require = Require,
    ensure = Ensure,
    end = end,
    if = if,
    then = then,
    else = else,
    switch = switch,
    of = of,
    case = case,
    otherwise = otherwise,
    do = do,
    while = while,
    repeat = repeat,
    until = until,
    loop = loop,
    foreach = {for~each},
    forall = {for~all},
    for = for,
    to = to,
    downto = {down~to},
    step = step,
    continue = continue,
    break = break,
```

```
    function = function,
    procedure = procedure,
    return = return,
    byref = byref,
    byvalue = byvalue,
    true = True,
    false = False,
    nil = Nil,
    read = read,
    write = write,
    set = set,
}

% Compound keywords
\AlgLanguageSet{english}{
    endwhile = \Keyword{end}~\Keyword{while},
    endfor = \Keyword{end}~\Keyword{for},
    endloop = \Keyword{end}~\Keyword{loop},
    endif = \Keyword{end}~\Keyword{if},
    endswitch = \Keyword{end}~\Keyword{switch},
    endcase = \Keyword{end}~\Keyword{case},
    endotherwise = \Keyword{end}~\Keyword{otherwise},
    endprocedure = \Keyword{end}~\Keyword{procedure},
    endfunction = \Keyword{end}~\Keyword{function},
}
```

The mechanism behind `\AlgLanguageSet` utilizes the `\SetKeyword` macro, which is called to modify the value of a single keyword[3]. To retrieve the value of a specific keyword, the `\Keyword` macro should be used. It produces the formatted value based on the options currently applied to keywords.

`\SetKeyword[`⟨*language*⟩`]{`⟨*keyword*⟩`}{`⟨*value*⟩`}`

The macro `\SetKeyword` 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.

`\Keyword[`⟨*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.

```
\SetKeyword[german]{if}{wenn} % new
Depending on the language, a keyword can take different forms: \Keyword{if}
↪  (English), \Keyword[german]{if} (German) or \Keyword[brazilian]{if}
↪  (Brazilian Portuguese).
```

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

[3] Macros such as `\algorithmicwhile` from the algorithmicx package are no longer in use.

```
\AlgLanguageSet{spanish}{
    while = mientras,
    do = hacer,
    endwhile = fin,
    if = si,
    then = entonces,
    else = en otro caso,
    endif = fin,
}
\begin{algorithmic}
    <language = spanish, indent lines>
    \If{$a > b$}[check conditions]
        \While{$a > 0$}
            \Statep{\Call{Process}{$a$}}[process current data]
        \EndWhile
    \Else
        \Statep{\Call{Signal}{\Id{pid}}}
    \EndIf
\end{algorithmic}
```

si $a > b$ **entonces** ▷ *check conditions*
  mientras $a > 0$ **hacer**
    PROCESS(a) ▷ *process current data*
  fin
en otro caso
  SIGNAL(pid)
fin

4.3 Other features

These macros were added to support the styles defined for named constants and function and procedure identifiers.

`\Constant{`⟨*name*⟩`}`

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

`\Module{`⟨*name*⟩`}`

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

5 "Forgotten" features

Some features were forgotten from v0.91 to v.99, although it was not intentional. The macros `\TextString` and `\VisibleSpace` simply vanished into thin air.

6 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.

7 Examples

7.1 LZW revisited

```
\AlgSet{
    comment color = purple,
    comment width = nice,
    comment style = \raggedleft,
}
```

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 byte values ▷ *each table position holds a single byte*
2: Initialize *sequence* with the first byte from the input stream
3: **while** there are bytes in the input **do** ▷ *wait until all bytes have been processed*
4: Retrieve a single byte from the input and store it in *byte*
5: **if** the concatenation of *sequence* and *byte* exists in the table **then**
6: Set *sequence* to *sequence* + *byte* ▷ *concatenate without generating any output*
7: **else**
8: Output the code for *sequence* ▷ *i.e., the binary representation of its index in the table*
9: Add the concatenation of *sequence* and *byte* to the table ▷ *the table learns a new, longer sequence*
10: Set *sequence* to *byte* ▷ *begin a new sequence with the remaining byte*
11: **end if**
12: **end while**
13: Output the code for *sequence* ▷ *the remaining bit sequence*

7.2 LZW revisited again

```
\AlgSet{
    keyword font = \ttfamily,
    keyword color = green!40!black,
    text font = \itshape,
    comment font = \footnotesize,
    algorithmic indent = 1.5em,
    indent lines,
    noend,
}
```

`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 byte values                ▷ each table position holds a single byte
2:  Initialize sequence with the first byte from the input stream
3:  while there are bytes in the input do                  ▷ wait until all bytes have been processed
4:      Retrieve a single byte from the input and store it in byte
5:      if the concatenation of sequence and byte exists in the table then
6:          Set sequence to sequence + byte                ▷ concatenate without generating any output
7:      else
8:          Output the code for sequence                   ▷ i.e., the binary representation of its index in the
                                                             table
9:          Add the concatenation of sequence and byte to the table   ▷ the table learns a new, longer sequence
10:         Set sequence to byte                           ▷ begin a new sequence with the remaining byte
11: Output the code for sequence                           ▷ the remaining bit sequence
```

Index