The algxpar package*

Jander Moreira
moreira.jander@gmail.com

July 29, 2020

Abstract

The algxpar packages is an extension of the algorithmicx package to handle multiline text with the proper indentation.

Contents

*This document corresponds to algxpar v0.91, dated 2020/05/30.

Change History

1 Introduction

I teach algorithms and programming and adopted the algorithmicx package (algpseudocode) to typeset my code, as it provides a clean, easy to read pseudolanguage algorithms with a minimum effort to write.

As part of the teaching process, I use very verbose commands in my algorithms before the students start to use more sintetic text. For example, I use "*Iniciate a counter c with the value* 0", what will become "$c \leftarrow 0$" later. This leads to sentences that often span the text for multiple lines, specially in two-column documents with nested structures.

Unfortunatelly, algorithmicx has no support for multiline statements natively, but it can adapted to use \parboxes to achive this goal.

This package, therefore, extends macros to handle multiple lines in a seamlessly way. Some new commands and features are also added.

2 Instalation

The package algxpar is provided by the files `algxpar.ins` and `algxpar.dtx`.

If the `.sty` file is not available, it can be generated by running the following at a command line prompt.

```
latex algxpar.ins
```

Then the generated `algxpar.sty` must be copied to a directory searched by LaTeX. Package dependencies can be checked in section 6.

3 Usage

The package must be loaded using

```
\usepackage[⟨options⟩]{algxpar}
```

The only option to the package is `brazilian`, which sets the pseudocode "reserved words" to Brazilian Portuguese, so \While is rendered **enquanto** instead

of **while**, for example. No other language is supported so far, but a translation can be easily achieved (see section 7).

4 Writing pseudocode

The algorithms must be written using the `algorithmic` environment and use basically the same set of macros defined by algpseudcode.

```
\begin{algorithmic}
    ⟨contents⟩
\end{algorithmic}
```

Example

Consider the following code.

```
\begin{algorithmic}
\Function{Max}{$a, b$}
    \If{$a > b$}
        \Statep{\Return $a$}
    \Else
        \Statep{\Return $b$}
    \EndIf
\EndFunction
\end{algorithmic}
```

The corresponding typeset is shown below.

function MAX(a, b)
  if $a > b$ **then**
    retorne a
  else
    retorne b
  end if
end function

4.1 Header

A header for the algorithm is proposed so the algorithm can provide a description, its inputs and outputs, as well as the preconditions and post-conditions Therefore, new macros are defined.

`\Description` A description can be provided for the sake of code documentation. The macro `\Description` is used to provide such a text. The input requirements for the algorithm uses the clause `\Input` and the produced by the code should be expressed with `\Output`. Also, the possibility to use `\Require` and `\Ensure` remains.

`\Input`
`\Output`
`\Require`
`\Ensure`

Examples

```
\Description Evaluates and prints the factorial of $n$
\Input A non-negative integer number $n$
\Output The value of the factorial $n$
```

Description: Evaluates and prints the factorial of n
Input: A non-negative integer number n
Output: The value of the factorial n

```
\Require $n \in \{1, 2, \ldots, 10\}$
\Ensure $k = \max(1, 2, \ldots, 10)$
```

Pre: $n \in \{1, 2, \ldots, 10\}$
Post: $k = \max(1, 2, \ldots, 10)$

4.2 Constants and identifiers

`\True` `\False` `\Nil` `\Id`

Some additional macros were added: `\True`, `\False`, and `\Nil`, producing TRUE, FALSE, and NIL, respectively.

The macro `\Id{`⟨*id*⟩`}` was included to support long variable names, such as *maxval* or *count*, for example. This macro handles better ligatures and accented characters than the regular math mode. `$offered$` results in $offered$ and `\Id{offered}` produces *offered*. With accented characters, `$magnético$` and `\Id{magnético}` result in *magntico* and *magnético*, respectively.

`\TextString`

For literal constants, usually represented quoted in programs and algorithms, the macro `\TextString{`⟨*text*⟩`}` is provided, so `\TextString{Error}` produces "`Error`".

`\VisibleSpace`

An additional macro called `\VisibleSpace` is also provided to produce ␣. Sometimes the number of spaces is relevant in text strings, so one can write `\TextString{a\VisibleSpace\VisibleSpace\VisibleSpace b}` to get "a␣␣␣b".

The macros `\Id` and `\TextString` work in text and math modes.

4.3 Assignment, reading and writing

`\gets`

The default symbol for assigning values to variables is ←, provided by `\gets`. This is a clearer option, once the equal sign is left just for comparisons.

`\Read` `\Write`

Although not common in algorithms published in scientific journals, explicit reading and writing is necessary for basic algorithms. Therefore `\Read` and `\Write` fulfills this need.

```
\Statep{\Read\ $a, b$}
\Statep{$s \gets a + b$}
\Statep{\Write\ $s$}
```

▷

read a, b
$s \leftarrow a + b$
write s

\Set \Setl

Besides \gets, the macros \Set and \Setl can be used for assignments.

\Set{⟨*id*⟩}{⟨*value*⟩} is a shortcut to \Id{id} \gets value. The "long" version for the assigment is \Setl{⟨*id*⟩}{⟨*value*⟩}, to get the verbose "Set *id* to *value*".

4.4 Comments

Comments use the symbol ▷ preceding the commented text and stay close to the left margin. Comment macros are intended to be used with \State or \Statex, when no multiline handling is done. Comments with multiline control are considered starting at section 4.5.

\Comment \Commentl \CommentIn

The macro \Comment{⟨*text*⟩} puts ⟨*text*⟩ at the end of the line.

A variant, \Commentl{⟨*text*⟩}, places the commented text without moving it to the left margin. It is a "local" comment.

A third option is \CommentIn{⟨*text*⟩}, that places the comment locally, but finishes it with ◁. Yes, that is really ugly.

```
\State\Commentl{Simple counter}
\State $c \gets 1$\Comment{initialize conter}
\State $n \gets \Call{FirstInstance}{}$
\While{$n < 0$}
    \State $c \gets c + 1$\Comment{counts one more}
    \State $n \gets \mbox{\CommentIn{all new} } \Call{NewInstance}{}$
\EndWhile
```

▷ *Simple counter*
$c \leftarrow 1$ ▷ *initialize conter*
$n \leftarrow$ FirstInstance( )
while $n < 0$ **do** ▷ *x*
    $c \leftarrow c + 1$ ▷ *counts one more*
    $n \leftarrow$ ▷ *all new* ◁ NewInstance( )
end while

4.5 Statements

\Statep \State \Statex

The statements should use \Statep{⟨*text*⟩}, which defines a hang indent for continued lines. The algorithmicx's \State and \Statex can be used as well.

In opposition to \State and \Statex, which uses justified text, \Statep aligns only to the left, what is aesthetically better than justification in my opinion.

Since `\Statep` uses a `\parbox` to span the text over multiple lines, no room is left for a comment. When needed a comment can be added through the optional argument: `\Statep[`⟨*comment*⟩`]{`⟨*text*⟩`}`.

Example

```
\Statep{Calculate the value of $x$ using $k$ and $m$,
 considering the stochastic distribution}
\Statep[$k \neq 0$, $m > k$]{Calculate the value of $x$
using $k$ and $m$, considering the stochastic distribution}
```

Calculate the value of x using k and m, considering the stochastic distribution

Calculate the value of x using k and m, considering the stochastic distribution ▷ $k \neq 0$, $m > k$

4.6 Conditionals

The traditional **if-then-else** structure is suported, handling nested commands as well. An **else if** construction avoids nesting **if**s and getting too much indentation. The macros are: `\If`, `\Else`, and `\ElsIf`.

`\If` `\Else` `\If[`⟨*comment*⟩`]{`⟨*condition*⟩`}` is used for conditional execution and is ended with a `\EndIf`. The optional ⟨*comment*⟩ is typeset to the left and the ⟨*condition*⟩ is put in a `\parbox`. Regular `\Comment` and `\Commentl` can be used after `\Else`.

`\ElsIf` The **else if** clause is specified by `\ElsIf[`⟨*comment*⟩`]{`⟨*condition*⟩`}`.

`\Switch` `\EndSwitch` `\Case` `\EndCase` `\Otherwise` `\EndOtherwise` Flow control using a selection structure are provided by the macro `\Switch[`⟨*comment*⟩`]{`⟨*selector*⟩`}`, ended with `\EndSwitch`. Each matching clause uses `\Case[`⟨*comment*⟩`]{`⟨*value*⟩`}` and `\EndCase`. The default uses `\Otherwise` and `\EndOtherwise`.

To specify ranges, the macro `\Range[`⟨*step*⟩`]{`⟨*start*⟩`}{`⟨*end*⟩`}` can be used. For example, `\Range{1}{10}` outputs 1..10 and `\Range[2]{0}{10}` prints 0..10:2.

Examples

```
\If{$a < 0$}
    \Statep{$a \gets 0$}
\EndIf
```

---

if $a < 0$ **then**
    $a \leftarrow 0$
end if

```
\If[closing doors]{the building is empty and the
  security system is active}
    \Statep{$\Id{status} \gets \TextString{ok}$}
\Else
    \Statep{$\Id{status} \gets \TextString{not ok}$}
\EndIf
```

---

if the building is empty and the security system is active **then** ▷ *closing doors*
  status ← "ok"
else
  status ← "not ok"
end if

```
\If[desired status]{$n \geq 0.8$}
    \Statep{$\Id{status} \gets \TextString{excelent}$}
\ElsIf{$n \geq 0.7$}
    \Statep{$\Id{status} \gets \TextString{great}$}
\ElsIf{$n \geq 0.5$}
    \Statep{$\Id{status} \gets \TextString{good}$}
\ElsIf{$n \geq 0.2$}
    \Statep{$\Id{status} \gets \TextString{not so good}$}
\Else\Comment{minimum not achieved}
    \Statep{$\Id{status} \gets \TextString{call for help}$}
\EndIf
```

---

if $n \geq 0.8$ **then** ▷ *desired status*
  status ← "excelent"
else if $n \geq 0.7$ **then**
  status ← "great"
else if $n \geq 0.5$ **then**
  status ← "good"
else if $n \geq 0.2$ **then**
  status ← "not so good"
else ▷ *minimum not achieved*
  status ← "call for help"
end if

```
\Switch[$1 \leq \Id{month} \leq 12$]{\Id{month}}
    \Case{2}
        \If{\Call{IsLeapYear}{\Id{year}}}
            \Statep{$n_{days} \gets 29$}
        \Else
            \Statep{$n_{days} \gets 28$}
        \EndIf
```

▷

```
    \EndCase
    \Case{4, 6, 9, 11}
        \Statep{$n_{days} \gets 30$}
    \EndCase
    \Otherwise\Comment{1, 3, 5, 7, 8, 10, 12}
        \Statep{$n_{days} \gets 31$}
    \EndOtherwise
\EndSwitch
```

---

with *month* **of** ▷ $1 \leq month \leq 12$
  case 2 **do**
    if IsLeapYear(*year*) **then**
      $n_{days} \leftarrow 29$
    else
      $n_{days} \leftarrow 28$
    end if
  end case
  case 4, 6, 9, 11 **do**
    $n_{days} \leftarrow 30$
  end case
  otherwise do ▷ *1, 3, 5, 7, 8, 10, 12*
    $n_{days} \leftarrow 31$
  end otherwise
end swith

4.7 Loops

Loops uses **while**, **repeat until**, and **for** flow control.

`\While` `\EndWhile` Loops with condition on top uses `\While[`⟨*comment*⟩`]{`⟨*condition*⟩`}` and are ended with `\EndWhile`.

`\Repeat` `\Until` When loops have their termination condition tested at the bottom, the macros `\Repeat` and `\Until[`⟨*comment*⟩`]{`⟨*condition*⟩`}` are used.

`\For` `\ForAll` `\ForEach` The **for** loop starts with `\For[`⟨*comment*⟩`]{`⟨*condition*⟩`}` and ends with `\EndFor`. To make things more versatile, `\For` can be replaced by `\ForAll` or `\ForEach`.

`\To` `\DownTo` `\Step` Some macros for supporting loops are also provided: `\To`, `\DownTo`, and `\Step`, which defaults to **to**, **downto**, and **step**, repectively.

Examples

```
\While{there is data in the input stream and no
    termination signal was received}
    \Statep{Get element $e$ from the input stream}
```

▷

```
    \Statep{\Call{Process}{$e$}}
\EndWhile
```

---

while there is data in the input stream and no termination signal was received **do** ▷ x
  Get element e from the input stream
  PROCESS(e)
end while

```
\Statep[$n_1, n_2 > 0$]{Let $n_1$ and $n_2$
be the two integers in order to find the greatest
number that divides both}
\Repeat
  \Statep[$n_1 \bmod n_2$]{Set \Id{rest} as the
      rest of the integer
      division of $n_1$ by $n_2$}
  \Statep{Redefine $n_1$ with the value of $n_2$}
  \Statep{Redefine $n_2$ with the value of \Id{rest}}
\Until[terminates]{$\Id{rest} = 0$}
\Statep[greatest common divisor]{Set $m$ to the value of $n_1$}
```

Let n_1 and n_2 be the two integers in order to find the greatest number that divides both ▷ $n_1, n_2 > 0$
repeat
  Set *rest* as the rest of the integer division of n_1 by n_2 ▷ $n_1 \bmod n_2$
  Redefine n_1 with the value of n_2
  Redefine n_2 with the value of *rest*
until $rest = 0$ ▷ *terminates*
Set m to the value of n_1 ▷ *greatest common divisor*

```
\For{$i \gets n-1$ \DownTo\ $0$}
  \Statep{$s \gets s + i$}
\EndFor
```

---

for $i \leftarrow n - 1$ **downto** 0 **do**
  $s \leftarrow s + i$
end for

```
\ForEach[main transactions]{transaction $t$ in the flow
    of transactions for month $m$}
    \Statep{\Call{ProcessTransaction}{$t$}}
\EndFor
```

▷

for each transaction t in the flow of transactions for month m **do** ▷ *main transactions*
    ProcessTransaction(t)
end for

```
\ForAll{$e$ in set $M$}
    \Statep{\Call{ProcessElement}{$e$}}
\EndFor
```

for all e in set M **do**
    ProcessElement(e)
end for

4.8 Procedures and functions

\Procedure
\EndProcedure
\Function
\EndFunction
\Return

Procedure and functions are supported with `\Procedure{`⟨*name*⟩`}{`⟨*arguments*⟩`}` and `\EndProcedure` and `\Function{`⟨*name*⟩`}{`⟨*arguments*⟩`}` and `\EndFunction`. The return value for functions use `\Return`.

Examples

```
\Procedure{PrintError}{$code$}
    \Switch{$code$}
        \Case{1}
            \Statep{\Write\ \TextString{Not found}}
        \EndCase
        \Case{2}
            \Statep{\Write\ \TextString{Access denied}}
        \EndCase
        \Case{3}
            \Statep{\Write\ \TextString{Blocked}}
        \EndCase
        \Otherwise
            \Statep{\Write\ \TextString{Unknown}}
        \EndOtherwise
    \EndSwitch
\EndProcedure
```

procedure PrintError(*code*)
    swith *code* **of**
        case 1 **do**
            write "`Not found`"
        end case
▷

```
        case 2 do
            write "Access denied"
        end case
        case 3 do
            write "Blocked"
        end case
        otherwise do
            write "Unknown"
        end otherwise
    end swith
end procedure
```

```
\Function{CelsiusToFahrenheit}{$t$}
    \Statep{\Return $\dfrac{9}{5}t + 32$}
\EndFunction
```

---

function CelsiusToFahrenheit(t)
  retorne $\frac{9}{5}t + 32$
end function

```
\Function[many parameters]{MyFunction}
   {$a$, $b$, $c$, $d$, $e$, $f$, $g$, $h$, $i$, $j$, $k$, $l$}
    \Statep{\Return $\dfrac{a+b+c+d}{f+g+hi^{j}}kl$}
\EndFunction
```

---

function MyFunction(a, b, c, d, e, f, g, h, i, j, k, l) ▷ *many parameters*
  retorne $\frac{a+b+c+d}{f+g+hi^{j}}kl$
end function

5 Extras

`\NewLine` Sometimes just letting the `\parbox` handle the line breaks is not enough. The macro `\NewLine` can be used to manually break lines.

`DefineCode` `\UseCode` `\ShowCode` It is possible to define pieces of code for later use. Using the environment `DefineCode` with a ⟨*name*⟩, a part of the pseudocode can be specified and used with `\UseCode{`⟨*name*⟩`}`. The ⟨*name*⟩ provided should be unique; when repeated the code is overwritten. The macro `\ShowCode[`⟨*options*⟩`]{`⟨*name*⟩`}` displays the saved code *verbatim*. Any option for `\VerbatimInput` from fancyvrb can be specified in ⟨*options*⟩. All chuncks of code are written to temporary files.

Examples

```
\If{$h > 0$ and\NewLine
        ($n_1 \neq 0$ or $n_2 < n_1$) and \NewLine
        $p \neq \Nil$}
    \Statep{\Call{DoSomething}{}}
\Else
    \Statep{\Call{DoSomethingElse}{}}
\EndIf
```

---

if $h > 0$ and
  ($n_1 \neq 0$ or $n_2 < n_1$) and
  $p \neq$ NIL **then**
    DOSOMETHING( )
else
    DOSOMETHINGELSE( )
end if

```
\begin{DefineCode}{half_in_out}
    \Input A number $n$
    \Output Half of $n$ (i.e., $n/2$)
\end{DefineCode}
\begin{DefineCode}{half_code}
    \Statep[in]{Get $n$}
    \Statep[out]{Print $n/2$}
\end{DefineCode}
```

Inside `algorithmic` one can use the following definitions.

```
\UseCode{half_in_out}
\Statep{\Commentl{Code}}
\UseCode{half_code}
```

Input: A number n
Output: Half of n (i.e., $n/2$)
  ▷ *Code*
  Get n ▷ *in*
  Print $n/2$ ▷ *out*

The source is shown by `\ShowCode{half_code}`.

```
\Statep[in]{Get $n$}
\Statep[out]{Print $n/2$}
```

6 Implementation

This package is algxpar v0.91 – LaTeX 2_ε.

```
\NeedsTeXFormat{LaTeX2e}[2005/12/01]
\ProvidesPackage{algxpar}
[2020/05/30 v0.91 Algorithms with multiline/paragraph support]
\newif\ifaxp@brazilian\axp@brazilianfalse
\DeclareOption{brazilian}{\axp@braziliantrue}
\DeclareOption*{\PackageWarning{algxpar}{Unknown `\CurrentOption'}}
\ProcessOptions\relax
```

ragged2e: for `\RaggedRight`
listings: to get accented characters in verbatim mode (pt_BR)
amsmath, amssymb: for `\triangleright` and `\triangleleft`
xcolor: gray color for `\VisibleSpace`
tcolorbox: verbatim save to file
fancyvrb: verbatim read from file with tabs

```
\RequirePackage{algorithmicx}
\RequirePackage{algpseudocode}
\RequirePackage{ragged2e}
\RequirePackage{listings}
\RequirePackage{amsmath, amssymb}
\RequirePackage{xcolor}
\RequirePackage{tcolorbox} % to save verbatim
\RequirePackage{fancyvrb} % to load verbatim preserving tabs
```

`\True` `\False` `\Nil` `\Id` `\TextString` `\VisibleSpace`

```
\algnewcommand\algorithmictrue{True}
\algnewcommand\algorithmicfalse{False}
\algnewcommand\algorithmicnil{Nil}
\algnewcommand\True{\ensuremath{\textsc{\rmfamily \algorithmictrue}}}
\algnewcommand\False{\ensuremath{\textsc{\rmfamily \algorithmicfalse}}}
\algnewcommand\Nil{\ensuremath{\textsc{\rmfamily \algorithmicnil}}}
\newcommand{\Id}[1]{\ensuremath{\textit{\rmfamily#1}}}
\newcommand{\TextString}[1]{\textrm{\normalfont``{\ttfamily\mbox{#1}}''}}
\algnewcommand{\VisibleSpace}{\textrm{\color{black!70}\textvisiblespace}}
```

`\Description` `\Input` `\Output` `\Ensure` `\Require`

```
\algnewcommand\algorithmicdescription{\textbf{Description}}
\algnewcommand\algorithmicinput{\textbf{Input}}
\algnewcommand\algorithmicoutput{\textbf{Output}}
\algrenewcommand\algorithmicensure{\textbf{Ensure}}
\algrenewcommand\algorithmicrequire{\textbf{Require}}
\algnewcommand\Description{\item[\algorithmicdescription:]}
\algnewcommand\Input{\item[\algorithmicinput:]}
\algnewcommand\Output{\item[\algorithmicoutput:]}
\algrenewcommand\Ensure{\item[\algorithmicensure:]}
\algrenewcommand\Require{\item[\algorithmicrequire:]}
```

```
\Read
\Write
 \Set
 \Setl
\algnewcommand{\algorithmicread}{\textbf{read}}
\algnewcommand{\algorithmicwrite}{\textbf{write}}
\algnewcommand\algorithmicset{Set}
\algnewcommand\algorithmicsetto{to}
\algnewcommand{\Set}[2]{\Id{#1} $\gets$ #2}
\algnewcommand{\Setl}[2]{\algorithmicset\ #1 \algorithmicsetto\ #2}
\algnewcommand{\Read}{\algorithmicread}
\algnewcommand{\Write}{\algorithmicwrite}

\Comment
\Commentl
\CommentIn
\newcommand{\axp@commentleftsymbol}{$\triangleright$}
\newcommand{\axp@commentrightsymbol}{$\triangleleft$}
\algnewcommand{\CommentIn}[1]{\axp@commentleftsymbol~%
  \textsl{#1}~\axp@commentrightsymbol}
\algnewcommand{\Commentl}[1]{\axp@commentleftsymbol~\textsl{#1}}
\algrenewcommand{\algorithmiccomment}[1]{%
\def\tmp{#1}%
\ifx\tmp\empty\else%
\hfill\Commentl{#1}%
\fi
}

\Statep
\newlength{\axp@stateindent}
\setlength{\axp@stateindent}{\dimexpr\algorithmicindent/2\relax}
\algnewcommand{\Statep}[2][]{\State\algparbox[#1]{#2}{\axp@stateindent}}

\While
\EndWhile
\newlength{\axp@whilewidth}
\algblockdefx{While}{EndWhile}%
[2][]{%
\settowidth{\axp@whilewidth}{\algorithmicwhile\ }%
\algparbox[x#1]{\algorithmicwhile\ #2~\algorithmicdo}{\axp@whilewidth}%
}%
{\algorithmicend\ \algorithmicwhile}

\Repeat
\Until
\newlength{\axp@untilwidth}
\algblockdefx{Repeat}{Until}%
{\algorithmicrepeat}%
[2][]{%
\settowidth{\axp@untilwidth}{\algorithmicuntil\ }%
\axp@algparbox{#1}{\algorithmicuntil\ #2}{\axp@untilwidth}{0}%
}

\If
\Else
\ElsIf
\EndIf
\newlength{\axp@ifwidth}
\newlength{\axp@elseifwidth}
```

```
\algblockdefx[If]{If}{EndIf}%
[2][]{%
\settowidth{\axp@ifwidth}{\algorithmicif\ }%
\algparbox[#1]{\algorithmicif\ #2~\algorithmicthen}{\axp@ifwidth}%
}
{\algorithmicend\ \algorithmicif}
\algcblockx[If]{If}{ElsIf}{EndIf}
[2][]{%
\settowidth{\axp@elseifwidth}{\algorithmicelse\ \algorithmicif\ }%
\algparbox[#1]{\algorithmicelse~\algorithmicif\ #2~\algorithmicthen}{\axp@elseifwidth}%
}
{\algorithmicend\ \algorithmicif}
\algcblockx{If}{Else}{EndIf}
{\textbf{\algorithmicelse}}
{\textbf{\algorithmicend~\algorithmicif}}
```

```
\Switch
\EndSwitch
\Case
\EndCase
\Otherwise
\EndOtherwise
\Range
```

```
\algnewcommand{\algorithmicswitch}{\textbf{switch}}
\algnewcommand{\algorithmicof}{\textbf{of}}
\algnewcommand{\algorithmiccase}{\textbf{case}}
\algnewcommand{\algorithmicotherwise}{\textbf{otherwise}}
\newlength{\axp@switchwidth}
\algblockdefx{Switch}{EndSwitch}%
[2][]{%
\settowidth{\axp@switchwidth}{\algorithmicswitch\ }%
\algparbox[#1]{\algorithmicswitch\ #2~\algorithmicof}{\axp@switchwidth}%
}
{\algorithmicend~\algorithmicswitch}
\newlength{\axp@casewidth}
\algblockdefx{Case}{EndCase}%
[2][]{%
\settowidth{\axp@casewidth}{\algorithmiccase\ }%
\algparbox[#1]{\algorithmiccase\ #2~\algorithmicdo}{\axp@casewidth}%
}
{\algorithmicend~\algorithmiccase}
\algblockdefx{Otherwise}{EndOtherwise}%
{\algorithmicotherwise~\algorithmicdo}%
{\textbf{\algorithmicend\ \algorithmicotherwise}}
\newcommand{\Range}[3][]{%
\ensuremath{%
#2%
\def\temp{#1}%
\mathcal{\ldotp\ldotp}#3
\ifx\temp\empty\relax\else{\ensuremath{\mathcal{:}#1}}\fi%
}%
}
```

```
\For
\ForEch
\ForAll
\EndFor
\To
\DownTo
\Step
```

```
\algnewcommand{\To}{\textbf{to}}
\algnewcommand{\DownTo}{\textbf{downto}}
```

```
\algnewcommand{\Step}{\textbf{step}}
\newlength{\axp@forwidth}
\algblockdefx{For}{EndFor}%
[2][]{%
\settowidth{\axp@forwidth}{\algorithmicfor\ }%
\algparbox[#1]{\algorithmicfor\ #2~\algorithmicdo}{\axp@forwidth}%
}
{\algorithmicend\ \algorithmicfor}
\algnewcommand{\algorithmicforeach}{\textbf{for~each}}
\newlength{\axp@foreachwidth}
\algblockdefx{ForEach}{EndFor}%
[2][]{%
\settowidth{\axp@foreachwidth}{\algorithmicforeach\ }%
\algparbox[#1]{\algorithmicforeach\ #2~\algorithmicdo}{\axp@foreachwidth}%
}
{\algorithmicend\~\algorithmicfor}
\newlength{\axp@forallwidth}
\algblockdefx{ForAll}{EndFor}%
[2][]{%
\settowidth{\axp@forallwidth}{\algorithmicforall\ }%
\algparbox[#1]{\algorithmicforall\ #2~\algorithmicdo}{\axp@forallwidth}%
}%
{\algorithmicend\ \algorithmicfor}
```

\Procedure
\EndProcedure
\Function
\EndFunction
\Call

```
\newlength{\axp@procedurewidth}
\newlength{\axp@namewidth}
\algblockdefx{Procedure}{EndProcedure}%
[3][]{%
\settowidth{\axp@procedurewidth}{\algorithmicprocedure~}%
\settowidth{\axp@namewidth}{\textsc{#2}()%
\addtolength{\axp@procedurewidth}{0.6\axp@namewidth}%
\algparbox[#1]{\algorithmicprocedure\ \textsc{#2}(#3)}{\axp@procedurewidth}
}%
{\algorithmicend\ \algorithmicprocedure}
\newlength{\axp@functionwidth}
\algblockdefx{Function}{EndFunction}%
[3][]{%
\settowidth{\axp@functionwidth}{\algorithmicfunction~}%
\settowidth{\axp@namewidth}{\textsc{#2}()%
\addtolength{\axp@functionwidth}{0.6\axp@namewidth}%
\algparbox[#1]{\algorithmicfunction\ \textsc{#2}(#3)}{\axp@functionwidth}
}%
{\algorithmicend\ \algorithmicfunction}
\algrenewcommand\Call[2]{%
\def\argstmp{#2}%
\textsc{#1}\ifx\argstmp\empty\mbox{(\hskip0.5ex)}\else(#2)\fi%
}
```

\NewLine

```
\newcommand{\NewLine}{\\}

DefineCode
  \UseCode
  \ShowCode
\newenvironment{DefineCode}[1]
{\begingroup\tcbverbatimwrite{\jobname_code_#1.tmp}}
{\endtcbverbatimwrite\endgroup}
\newcommand{\UseCode}[1]{\input{\jobname_code_#1.tmp}}
\newcommand{\ShowCode}[2][]{{\small\VerbatimInput[tabsize=4, #1]%
{\jobname_code_#2.tmp}}}


\alglinenumber
\algrenewcommand{\alglinenumber}[1]%
  {\hspace{-1.5em}\color{black!35}{\scriptsize#1}\raisebox{0.2ex}{\tiny$\blacktriangleright$}}

\axp@algparbox
\newlength{\axp@commentwidth}
\setlength{\axp@commentwidth}{0pt}
\newcommand{\algparbox}[3][]{\axp@algparbox{#1}{#2}{#3}{1}}

\newlength{\axp@largestcommentwidth}
\setlength{\axp@largestcommentwidth}{0.3\linewidth}
\newcommand{\axp@algparbox}[4]{%
    \def\temp{#1}%
    \ifx\temp\empty%
        \setlength{\axp@commentwidth}{-2em}%
    \else%
        \settowidth{\axp@commentwidth}{\axp@commentleftsymbol\ #1}%
        \ifdim\axp@commentwidth>\axp@largestcommentwidth\relax%
            \setlength{\axp@commentwidth}{\axp@largestcommentwidth}%
        \fi%
    \fi%
    \renewcommand{\NewLine}{\\\hspace{#3}}%
    \parbox[t]{\dimexpr\linewidth-\axp@commentwidth-%
        (\algorithmicindent)*(\theALG@nested - #4)-2em}%
        {\RaggedRight\setlength{\hangindent}{#3}#2\strut}%
    \ifx\temp\empty\else%
        \hfill\axp@commentleftsymbol\hspace{0.5em}%
        \parbox[t]{\axp@commentwidth}{\slshape\RaggedRight#1}%
    \fi%
    \renewcommand{\NewLine}{\\}%
}

\lstset{
literate=
{á}{{\'a}}1 {é}{{\'e}}1 {í}{{\'i}}1 {ó}{{\'o}}1 {ú}{{\'u}}1
{Á}{{\'A}}1 {É}{{\'E}}1 {Í}{{\'I}}1 {Ő}{{\'O}}1 {Ú}{{\'U}}1
{à}{{\`a}}1 {è}{{\`e}}1 {ì}{{\`i}}1 {ò}{{\`o}}1 {ù}{{\`u}}1
{À}{{\`A}}1 {È}{{\'E}}1 {Ì}{{\`I}}1 {Õ}{{\`O}}1 {Ù}{{\`U}}1
{ä}{{\"a}}1 {ë}{{\"e}}1 {ï}{{\"i}}1 {ö}{{\"o}}1 {ü}{{\"u}}1
```

```
{ã}{{\~a}}1 {õ}{{\~o}}1
{Ã}{{\~A}}1 {Õ}{{\~O}}1
{Ä}{{\"A}}1 {Ë}{{\"E}}1 {Ï}{{\"I}}1 {Ö}{{\"O}}1 {Ü}{{\"U}}1
{â}{{\^a}}1 {ê}{{\^e}}1 {î}{{\^i}}1 {ô}{{\^o}}1 {û}{{\^u}}1
{Â}{{\^A}}1 {Ê}{{\^E}}1 {Î}{{\^I}}1 {Ô}{{\^O}}1 {Û}{{\^U}}1
{ç}{{\c c}}1 {Ç}{{\c C}}1
{ø}{{\o}}1 {å}{{\r a}}1 {Å}{{\r A}}1
{œ}{{\oe}}1 {Œ}{{\OE}}1 {æ}{{\ae}}1 {Æ}{{\AE}}1
{ß}{{\ss}}1
{ű}{{\H{u}}}1 {Ű}{{\H{U}}}1 {ő}{{\H{o}}}1 {Ő}{{\H{O}}}1
{£}{{\pounds}}1
{«}{{\guillemotleft}}1
{»}{{\guillemotright}}1
{ñ}{{\~n}}1 {Ñ}{{\~N}}1 {¿}{{?‘}}1
}
```

7 Customization

By default, the longest width for a comment at the right margin is `0.3\linewidth`. This can be changed using something like the code below.

```
\makeatletter
\setlength{\axp@largestcommentwidth}{⟨new length⟩}
\makeatother
```

The assignment sign can be changed from ← to anything else, as well as the symbols used in comments.

```
\renewcommand{\gets}{\mathop{::=}}
\renewcommand{\axp@commentleftsymbol}{\texttt{//}}
\renewcommand{\axp@commentrightsymbol}{\texttt{*/}}
```

To handle languages, the macro `\algxparset` should be used.

```
\pgfkeys{
algxpar/.cd,
brazilian/.code = {\axp@languagebrazilian},
english/.code = {\axp@languageenglish},
default/.code = {\axp@languageenglish},
}
\newcommand{\algxparset}[1]{
\pgfkeys{
algxpar/.cd,
#1
}
}
\newcommand{\axp@languagebrazilian}{
\algrenewcommand\algorithmicdescription{\textbf{Descrição}}
```

```
\algrenewcommand\algorithmicinput{\textbf{Entrada}}
\algrenewcommand\algorithmicoutput{\textbf{Saída}}
\algrenewcommand\algorithmicrequire{\textbf{Pré}}
\algrenewcommand\algorithmicensure{\textbf{Pós}}
\algrenewcommand{\algorithmicend}{\textbf{fim}}
\algrenewcommand{\algorithmicif}{\textbf{se}}
\algrenewcommand{\algorithmicthen}{\textbf{então}}
\algrenewcommand{\algorithmicelse}{\textbf{senão}}
\algrenewcommand{\algorithmicswitch}{\textbf{escolha}}
\algrenewcommand{\algorithmicof}{\textbf{de}}
\algrenewcommand{\algorithmiccase}{\textbf{caso}}
\algrenewcommand{\algorithmicotherwise}{\textbf{caso~contrário}}
\algrenewcommand{\algorithmicfor}{\textbf{para}}
\algrenewcommand{\algorithmicdo}{\textbf{faça}}
\algrenewcommand{\algorithmicwhile}{\textbf{enquanto}}
\algrenewcommand{\algorithmicrepeat}{\textbf{repita}}
\algrenewcommand{\algorithmicuntil}{\textbf{até que}}
\algrenewcommand{\algorithmicloop}{\textbf{repita}}
\algrenewcommand{\algorithmicforeach}{\textbf{para~cada}}
\algrenewcommand{\algorithmicforall}{\textbf{para~todo}}
\algrenewcommand{\algorithmicfunction}{\textbf{função}}
\algrenewcommand{\algorithmicprocedure}{\textbf{procedimento}}
\algrenewcommand{\algorithmicreturn}{\textbf{retorne}}
\algrenewcommand{\algorithmictrue}{Verdadeiro}
\algrenewcommand{\algorithmicfalse}{Falso}
\algrenewcommand{\algorithmicnil}{Nulo}
\algrenewcommand{\algorithmicread}{\textbf{leia}}
\algrenewcommand{\algorithmicwrite}{\textbf{escreva}}
\algrenewcommand{\algorithmicset}{Defina}
\algrenewcommand{\algorithmicsetto}{como}
\algrenewcommand{\To}{\textbf{até}}
\algrenewcommand{\DownTo}{\textbf{decrescente~até}}
\algrenewcommand{\Step}{\textbf{passo}}
}
\newcommand{\axp@languageenglish}{
\algrenewcommand\algorithmicdescription{\textbf{Description}}
\algrenewcommand\algorithmicinput{\textbf{Input}}
\algrenewcommand\algorithmicoutput{\textbf{Output}}
\algrenewcommand\algorithmicrequire{\textbf{Pre}}
\algrenewcommand\algorithmicensure{\textbf{Post}}
\algrenewcommand{\algorithmicend}{\textbf{end}}
\algrenewcommand{\algorithmicif}{\textbf{if}}
\algrenewcommand{\algorithmicthen}{\textbf{then}}
\algrenewcommand{\algorithmicelse}{\textbf{else}}
\algrenewcommand{\algorithmicswitch}{\textbf{swith}}
\algrenewcommand{\algorithmicof}{\textbf{of}}
\algrenewcommand{\algorithmiccase}{\textbf{case}}
\algrenewcommand{\algorithmicotherwise}{\textbf{otherwise}}
\algrenewcommand{\algorithmicfor}{\textbf{for}}
\algrenewcommand{\algorithmicdo}{\textbf{do}}
```

```
\algrenewcommand{\algorithmicwhile}{\textbf{while}}
\algrenewcommand{\algorithmicrepeat}{\textbf{repeat}}
\algrenewcommand{\algorithmicuntil}{\textbf{until}}
\algrenewcommand{\algorithmicloop}{\textbf{loop}}
\algrenewcommand{\algorithmicforeach}{\textbf{for~each}}
\algrenewcommand{\algorithmicforall}{\textbf{for~all}}
\algrenewcommand{\algorithmicfunction}{\textbf{function}}
\algrenewcommand{\algorithmicprocedure}{\textbf{procedure}}
\algrenewcommand{\algorithmicreturn}{\textbf{retorne}}
\algrenewcommand{\algorithmictrue}{True}
\algrenewcommand{\algorithmicfalse}{False}
\algrenewcommand{\algorithmicnil}{Nil}
\algrenewcommand{\algorithmicread}{\textbf{read}}
\algrenewcommand{\algorithmicwrite}{\textbf{write}}
\algrenewcommand{\algorithmicset}{Set}
\algrenewcommand{\algorithmicsetto}{to}
\algrenewcommand{\To}{\textbf{to}}
\algrenewcommand{\DownTo}{\textbf{downto}}
\algrenewcommand{\Step}{\textbf{step}}
}
\axp@languageenglish % default language
\ifaxp@brazilian\algxparset{brazilian}\fi
```

8 To do...

There are lots of improvements to make in the code. I recognize it!

Appendix

A An example

```
\Description Inserts a new item in the B-tree structure,
    handling only the root node
\Input The \Id{item} to be inserted
\Output Returns \True\ in case of success, \False\ in
    case of failure (i.e., duplicated keys)
\Function{Insert}{\Id{item}}
    \If{\Id{tree.root address} is \Nil}
        \Statep{\Commentl{Create first node}}
        \Statep[\Nil\ = new node]{$\Id{new root node}
            \gets \Call{GetNode}{\Nil}$}
        \Statep[only item]{Insert \Id{item} in \Id{new
            root node} and set both  its left and right
            childs to \Nil; also set \Id{new root
            node.count} to 1}
```

```
            \Statep[first node is always a leaf]{Set \Id{new
                root node.type} to \Leaf}
            \Statep[flag that node must be updated in file]
                {Set \Id{new root node.modified} to \True}
            \Statep{\Call{WriteNode}{\Id{new root node}}}
            \Statep{$\Id{tree.root address} \gets
                \Id{new root node.address}$}
            \Statep[update root address in file]
                {\Call{WriteRootAddress}{}}
            \Statep{\Return \True}
        \Else
            \Statep{\Commentl{Insert in existing tree}}
            \Statep[]{$\Id{success}$,
            $\Id{promoted item}$, $\Id{new node address} \gets
                \Call{SearchInsert}{\Id{tree.root address},
                \Id{item}}$}
            \If[root has splitted]{\Id{success} and
                ${\Id{new node address}\neq\Nil}$}
                \Statep[new root]{$\Id{new root node} \gets
                    \Call{GetNode}{\Nil}$}
                \Statep{Insert \Id{promoted item} in \Id{new
                    root node} and set \Id{new root node.count}
                    to 1}
                \Statep[tree height grows]{Set \Id{item}'s
                    left child to \Id{tree.root
                    address} and right child to \Id{new
                    node address}}
                \Statep[not a leaf]{Set \Id{new root
                        node.type} to \Internal}
                \Statep{Set \Id{new root node.modified}
                    to \True}
                \Statep{\Call{WriteNode}{\Id{new root
                    node}}}
                \Statep{$\Id{tree.root address} \gets
                    \Id{new root node.address}$}
                \Statep[update root address in
                    file]{\Call{WriteRootAddress}{}}
            \EndIf
            \Statep[insertion status]{\Return \Id{success}}
        \EndIf
    \EndFunction
```

Description: Inserts a new item in the B-tree structure, handling only the root node

Input: The *item* to be inserted

Output: Returns TRUE in case of success, FALSE in case of failure (i.e., ▷

```
duplicated keys)
function INSERT(item)
    if tree.root address is NIL then
        ▷ Create first node
        new root node ← GETNODE(NIL)                    ▷ NIL = new node
        Insert item in new root node and set both       ▷ only item
          its left and right childs to NIL; also set
          new root node.count to 1
        Set new root node.type to LEAF                  ▷ first node is always a
                                                          leaf
        Set new root node.modified to TRUE              ▷ flag that node must be
                                                          updated in file
        WRITENODE(new root node)
        tree.root address ← new root node.address
        WRITEROOTADDRESS( )                             ▷ update root address in
                                                          file
        retorne TRUE
    else
        ▷ Insert in existing tree
        success, promoted item, new node address    ←
          SEARCHINSERT(tree.root address, item)
        if success and                                  ▷ root has splitted
           new node address ≠ NIL then
            new root node ← GETNODE(NIL)                ▷ new root
            Insert promoted item in new root node and set
              new root node.count to 1
            Set item's left child to                    ▷ tree height grows
              tree.root address and right child to
              new node address
            Set new root node.type to INTERNAL          ▷ not a leaf
            Set new root node.modified to TRUE
            WRITENODE(new root node)
            tree.root address ← new root node.address
            WRITEROOTADDRESS( )                         ▷ update root address in
                                                          file
        end if
        retorne success                                 ▷ insertion status
    end if
end function
```

Index