The `algorithmicx` package*

Szász János
szaszjanos@users.sourceforge.net

January 16, 2019

Abstract

The `algorithmicx` package provides many possibilities to customize the layout of algorithms. You can use one of the predefined layouts (**pseudocode**, **pascal** and **c** and others), with or without modifications, or you can define a completely new layout for your specific needs.

Contents

1 Introduction

All this has begun in my last year at the university. The only thing that I knew of LaTeX was that it exists, and that it is "good". I started using it, but I needed to typeset some algorithms. So I begun searching for a good algorithmic style, and I have found the `algorithmic` package. It was a great joy for me, and I started to use it... Well... Everything went nice, until I needed some block that wasn't defined in there. What to do? I was no LaTeX guru, in fact I only knew the few basic macros. But there was no other way, so I opened the style file, and I copied one existing block, renamed a few things, and voilà! This (and some other small changes) where enough for me...

One year later — for one good soul — I had to make some really big changes on the style. And there on a sunny day came the idea. What if I would write some macros to let others create blocks automatically? And so I did! Since then the style was completely rewritten... several times...

I had fun writing it, may you have fun using it! I am still no LaTeX guru, so if you are, and you find something really ugly in the style, please mail me! All ideas for improvements are welcome!

Thanks go to Benedek Zsuzsa, Ionescu Clara, Szőcs Zoltán, Cseke Botond, Kanoc and many-many others. Without them I would have never started or continued **algorithmicx**.

*This is the documentation for the version 1.2 of the package. This package is released under LPPL.

2 General informations

2.1 The package

The package **algorithmicx** itself doesn't define any algorithmic commands, but gives a set of macros to define such a command set. You may use only **algorithmicx**, and define the commands yourself, or you may use one of the predefined command sets.

These predefined command sets (layouts) are:

algpseudocode has the same look[1] as the one defined in the `algorithmic` package. The main difference is that while the `algorithmic` package doesn't allow you to modify predefined structures, or to create new ones, the `algorithmicx` package gives you full control over the definitions (ok, there are some limitations — you can not send mail with a, say, `\For` command).

algcompatible is fully compatible with the `algorithmic` package, it should be used only in old documents.

algpascal aims to create a formatted pascal program, it performs automatic indentation (!), so you can transform a pascal program into an **algpascal** algorithm description with some basic substitution rules.

algc – yeah, just like the **algpascal**... but for c... This layout is incomplete.

To create floating algorithms you will need `algorithm.sty`. This file may or may not be included in the `algorithmicx` package. You can find it on CTAN, in the `algorithmic` package.

2.2 The algorithmic block

Each algorithm begins with the `\begin{algorithmic}[lines]` command, the optional `lines` controls the line numbering: 0 means no line numbering, 1 means number every line, and n means number lines n, $2n$, $3n$... until the `\end{algorithmic}` command, witch ends the algorithm.

2.3 Simple lines

A simple line of text is beginned with `\State`. This macro marks the begin of every line. You don't need to use `\State` before a command defined in the package, since these commands use automatically a new line.

To obtain a line that is not numbered, and not counted when counting the lines for line numbering (in case you choose to number lines), use the `Statex` macro. This macro jumps into a new line, the line gets no number, and any label will point to the previous numbered line.

We will call *statament*s the lines starting with `\State`. The `\Statex` lines are not statements.

[1] almost :-)

2.4 Placing comments in sources

Comments may be placed everywhere in the source using the `\Comment` macro (there are no limitations like those in the `algorithmic` package), feel the freedom! If you would like to change the form in witch comments are displayed, just change the `\algorithmiccomment` macro:

```
\algrenewcommand{\algorithmiccomment}[1]{\hskip3em$\rightarrow$ #1}
```

will result:

1: $x \leftarrow x + 1$ → Here is the new comment

2.5 Labels and references

Use the `\label` macro, as usual to label a line. When you use `\ref` to reference the line, the `\ref` will be subtitued with the corresponding line number. When using the **algorithmicx** package togedher with the **algorithm** package, then you can label both the algorithm and the line, and use the `\algref` macro to reference a given line from a given algorithm:

```
\algref{<algorithm>}{<line>}
```

```
The \textbf{while} in algorithm
\ref{euclid} ends in line
\ref{euclidendwhile}, so
\algref{euclid}{euclidendwhile}
is the line we seek.
```

The **while** in algorithm **??** ends in line **??**, so **??.??** is the line we seek.

2.6 Breaking up long algorithms

Sometimes you have a long algorithm that needs to be broken into parts, each on a separate float. For this you can use the following:

`\algstore{<savename>}` saves the line number, indentation, open blocks of the current algorithm and closes all blocks. If used, then this must be the last command before closing the algorithmic block. Each saved algorithm must be continued later in the document.

`\algstore*{<savename>}` Like the above, but the algorithm must not be continued.

`\algrestore{<savename>}` restores the state of the algorithm saved under `<savename>` in this algorithmic block. If used, then this must be the first command in an algorithmic block. A save is deleted while restoring.

`\algrestore*{<savename>}` Like the above, but the save will not be deleted, so it can be restored again.

See example in the **Examples** section.

2.7 Multiple layouts in the same document

You can load multiple algorithmicx layouts in the same document. You can switch between the layouts using the `\alglanguage{<layoutname>}` command. After this command all new algorithmic environments will use the given layout until the layout is changed again.

3 The predefined layouts

3.1 The algpseudocode layout

If you are familiar with the `algorithmic` package, then you'll find it easy to switch. You can use the old algorithms with the **algcompatible** layout, but please use the **algpseudocode** layout for new algorithms.

To use **algpseudocode**, simply load `algpseudocode.sty`:

```
\usepackage{algpseudocode}
```

You don't need to manually load the **algorithmicx** package, as this is done by **algpseudocode**.

The first algorithm one should write is the first algorithm ever (ok, an improved version), *Euclid's algorithm*:

Algorithm 1 Euclid's algorithm

```
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
```

Created with the following source:

```
\begin{algorithm}
\caption{Euclid's algorithm}\label{euclid}
\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\label{euclidendwhile}
```

```
   \State \textbf{return} $b$\Comment{The gcd is b}
\EndProcedure
\end{algorithmic}
\end{algorithm}
```

The `\State` stands at the beginning of each simple statement; the respective statement is put in a new line, with the needed indentation. The `\Procedure` ... `\EndProcedure` and `\While` ... `\EndWhile` blocks (like any block defined in the **algpseudocode** layout) automatically indent their content. The indentation of the source doesn't matter, so

```
\begin{algorithmic}[1]
\Repeat
\Comment{forever}
\State this\Until{you die.}
\end{algorithmic}
```

1: **repeat** ▷ forever
2: this
3: **until** you die.

But, generally, it is a good idea to keep the source indented, since you will find errors much easier. And your tex file looks better!

All examples and syntax descriptions will be shown as the previous example — the left side shows the LaTeX input, and the right side the algorithm, as it appears in your document. I'm cheating! Don't look in the `algorithmicx.tex` file! Believe what the examples state! I may use some undocumented and dirty stuff to create all these examples. You might be more confused after opening `algorithmicx.tex` as you was before.

In the case of syntax descriptions the text between < and > is symbolic, so if you type what you see on the left side, you will not get the algorithm on the right side. But if you replace the text between < > with a proper piece of algorithm, then you will probably get what you want. The parts between [and] are optional.

3.1.1 The for block

The **for** block may have one of the forms:

```
\For{<text>}
   <body>
\EndFor
```

1: **for** <text> **do**
2: <body>
3: **end for**

```
\ForAll{<text>}
   <body>
\EndFor
```

1: **for all** <text> **do**
2: <body>
3: **end for**

Example:

```
\begin{algorithmic}[1]
\State $sum\gets 0$
\For{$i\gets 1, n$}
   \State $sum\gets sum+i$
\EndFor
\end{algorithmic}
```

1: $sum \leftarrow 0$
2: **for** $i \leftarrow 1, n$ **do**
3: $\quad sum \leftarrow sum + i$
4: **end for**

3.1.2 The while block

The **while** block has the form:

```
\While{<text>}
   <body>
\EndWhile
```

1: **while** <text> **do**
2: $\quad$ <body>
3: **end while**

Example:

```
\begin{algorithmic}[1]
\State $sum\gets 0$
\State $i\gets 1$
\While{$i\le n$}
   \State $sum\gets sum+i$
   \State $i\gets i+1$
\EndWhile
\end{algorithmic}
```

1: $sum \leftarrow 0$
2: $i \leftarrow 1$
3: **while** $i \leq n$ **do**
4: $\quad sum \leftarrow sum + i$
5: $\quad i \leftarrow i + 1$
6: **end while**

3.1.3 The repeat block

The **repeat** block has the form:

```
\Repeat
   <body>
\Until{<text>}
```

1: **repeat**
2: $\quad$ <body>
3: **until** <text>

Example:

```
\begin{algorithmic}[1]
\State $sum\gets 0$
\State $i\gets 1$
\Repeat
   \State $sum\gets sum+i$
   \State $i\gets i+1$
\Until{$i>n$}
\end{algorithmic}
```

1: $sum \leftarrow 0$
2: $i \leftarrow 1$
3: **repeat**
4: $\quad sum \leftarrow sum + i$
5: $\quad i \leftarrow i + 1$
6: **until** $i > n$

3.1.4 The if block

The **if** block has the form:

```
\If{<text>}
   <body>
[
\ElsIf{<text>}
   <body>
...
]
[
\Else
   <body>
]
\EndIf
```

1: **if** <text> **then**
2: <body>
[
3: **else if** <text> **then**
4: <body>
...
]
[
5: **else**
6: <body>
]
7: **end if**

Example:

```
\begin{algorithmic}[1]
\If{$quality\ge 9$}
   \State $a\gets perfect$
\ElsIf{$quality\ge 7$}
   \State $a\gets good$
\ElsIf{$quality\ge 5$}
   \State $a\gets medium$
\ElsIf{$quality\ge 3$}
   \State $a\gets bad$
\Else
   \State $a\gets unusable$
\EndIf
\end{algorithmic}
```

1: **if** $quality \geq 9$ **then**
2: $a \leftarrow perfect$
3: **else if** $quality \geq 7$ **then**
4: $a \leftarrow good$
5: **else if** $quality \geq 5$ **then**
6: $a \leftarrow medium$
7: **else if** $quality \geq 3$ **then**
8: $a \leftarrow bad$
9: **else**
10: $a \leftarrow unusable$
11: **end if**

3.1.5 The procedure block

The **procedure** block has the form:

```
\Procedure{<name>}{<params>}
   <body>
\EndProcedure
```

1: **procedure** <NAME>(<params>)
2: <body>
3: **end procedure**

Example: See Euclid's algorithm on page **??**.

3.1.6 The function block

The **function** block has the same syntax as the **procedure** block:

```
\Function{<name>}{<params>}
   <body>
\EndFunction
```

1: **function** <NAME>(<params>)
2: <body>
3: **end function**

3.1.7 The loop block

The **loop** block has the form:

```
\Loop
   <body>
\EndLoop
```

1: **loop**
2: <body>
3: **end loop**

3.1.8 Other commands in this layout

The starting conditions for the algorithm can be described with the **require** instruction, and its result with the **ensure** instruction.

A procedure call can be formatted with `\Call`.

```
\Require something
\Ensure something
\Statex
\State \Call{Create}{10}
```

Require: something
Ensure: something

1: CREATE(10)

Example:

```
\begin{algorithmic}[1]
\Require $x\ge5$
\Ensure $x\le-5$
\Statex
\While{$x>-5$}
   \State $x\gets x-1$
\EndWhile
\end{algorithmic}
```

Require: $x \geq 5$
Ensure: $x \leq -5$

1: **while** $x > -5$ **do**
2: $x \leftarrow x - 1$
3: **end while**

3.1.9 Package options

The `algpseudocode` package supports the following options:

compatible/noncompatible *Obsolete, use the algcompatible layout instead.* If you would like to use old algorithms, written with the `algorithmic` package without (too much) modification, then use the **compatible** option. This option defines the uppercase version of the commands. Note that you still need to remove the `[...]` comments (these comments appeared due to some limitations in the `algorithmic` package, these limitations and comments are gone now). The default **noncompatible** does not define the all uppercase commands.

noend/end

With **noend** specified all **end ...** lines are omitted. You get a somewhat smaller algorithm, and the ugly feeling, that something is missing... The **end** value is the default, it means, that all **end ...** lines are in their right place.

3.1.10 Changing command names

One common thing for a pseudocode is to change the command names. Many people use many different kind of pseudocode command names. In **algpseudocode** all keywords are declared as `\algorithmic<keyword>`. You can change them to output the text you need:

```
\algrenewcommand\algorithmicwhile{\textbf{am\'\i g}}
\algrenewcommand\algorithmicdo{\textbf{v\'egezd el}}
\algrenewcommand\algorithmicend{\textbf{v\'ege}}
\begin{algorithmic}[1]
\State $x \gets 1$
\While{$x < 10$}
   \State $x \gets x + 1$
\EndWhile
\end{algorithmic}
```

1: $x \leftarrow 1$
2: **amíg** $x < 10$ **végezd el**
3: $\quad x \leftarrow x + 1$
4: **vége amíg**

In some cases you may need to change even more (in the above example **amíg** and **vége** should be interchanged in the `\EndWhile` text). Maybe the number of the parameters taken by some commands must be changed too. this can be done with the command text customizing macros (see section **??**). Here I'll give only some examples of the most common usage:

```
\algrenewcommand\algorithmicwhile{\textbf{am\'\i g}}
\algrenewcommand\algorithmicdo{\textbf{v\'egezd el}}
\algrenewcommand\algorithmicend{\textbf{v\'ege}}
\algrenewtext{EndWhile}{\algorithmicwhile\ \algorithmicend}
\begin{algorithmic}[1]
\State $x \gets 1$
\While{$x < 10$}
   \State $x \gets x - 1$
\EndWhile
\end{algorithmic}
```

1: $x \leftarrow 1$
2: **amíg** $x < 10$ **végezd el**
3: $\quad x \leftarrow x - 1$
4: **amíg vége**

```
\algnewcommand\algorithmicto{\textbf{to}}
\algrenewtext{For}[3]%
  {\algorithmicfor\ #1 \gets #2 \algorithmicto\ #3 \algorithmicdo}
\begin{algorithmic}[1]
\State $p \gets 1$
\For{i}{1}{n}
   \State $p \gets p * i$
\EndFor
\end{algorithmic}
```

1: $p \leftarrow 1$
2: **for** $i \leftarrow 1$ **to** n **do**
3: $\quad p \leftarrow p * i$
4: **end for**

You could create a translation package, that included after the **algpseudocode** package translates the keywords to the language you need.

3.2 The algpascal layout

The most important feature of the **algpascal** layout is that *it performs automatically the block indentation*. In section **??** you will see how to define such automatically indented loops. Here is an example to demonstrate this feature:

```
\begin{algorithmic}[1]
\Begin
\State $sum:=0$;
\For{i=1}{n}\Comment{sum(i)}
   \State $sum:=sum+i$;
\State writeln($sum$);
\End.
\end{algorithmic}
```

1: **begin**
2: $sum := 0$;
3: **for** $i = 1$ **to** n **do** {sum(i)}
4: $\quad sum := sum + i$;
5: writeln(sum);
6: **end**.

Note, that the `\For` is not closed explicitly, its end is detected automatically. Again, the indentation in the source doesn't affect the output. In this layout every parameter passed to a command is put in mathematical mode.

3.2.1 The begin ... end block

```
\Begin
   <body>
\End
```

1: **begin**
2: <body>
3: **end**

The `\Begin` ... `\End` block and the `\Repeat` ... `\Until` block are the only blocks in the **algpascal** style (instead of `\Begin` you may write `\Asm`). This means, that every other loop is ended automatically after the following command (another loop, or a block).

3.2.2 The for loop

```
\For{<assign>}{<expr>}
   <command>
```

1: **for** <*assign*> **to** <*expr*> **do**
2: $\quad$ <command>

The **For** loop (as all other loops) ends after the following command (a block counts also as a single command).

```
\begin{algorithmic}[1]
\Begin
   \State $sum:=0$;
   \State $prod:=1$;
   \For{i:=1}{10}
      \Begin
         \State $sum:=sum+i$;
         \State $prod:=prod*i$;
      \End
\End.
\end{algorithmic}
```

1: **begin**
2: $sum := 0$;
3: $prod := 1$;
4: **for** $i := 1$ **to** 10 **do**
5: **begin**
6: $sum := sum + i$;
7: $prod := prod * i$;
8: **end**
9: **end**.

3.2.3 The while loop

```
\While{<expression>}
   <command>
```

1: **while** <*expression*> **do**
2: <command>

3.2.4 The repeat... until block

```
\Repeat
   <body>
\Until{<expression>}
```

1: **repeat**
2: <body>
3: **until** <*expression*>

3.2.5 The if command

```
\If{<expression>}
   <command>
[
\Else
   <command>
]
```

1: **if** <*expression*> **then**
2: <command>
[
3: **else**
4: <command>
]

Every `\Else` matches the nearest `\If`.

3.2.6 The procedure command

```
\Procedure <some text>
```

1: **procedure** <some text>

`\Procedure` just writes the "procedure" word on a new line... You will probably put a `\Begin`... `\End` block after it.

3.2.7 The function command

```
\Function<some text>
```

1: **function** <some text>

Just like **Procedure**.

3.3 The algc layout

Sorry, the **algc** layout is unfinished. The commands defined are:

- `\{... \}` block
- `\For` with 3 params
- `\If` with 1 param
- `\Else` with no params
- `\While` with 1 param
- `\Do` with no params
- `\Function` with 3 params
- `\Return` with no params

4 Custom algorithmic blocks

4.1 Blocks and loops

Most of the environments defined in the standard layouts (and most probably the ones you will define) are divided in two categories:

Blocks are the environments witch contain an arbitrary number of commands or nested blocks. Each block has a name, begins with a starting command and ends with an ending command. The commands in a block are indented by `\algorithmicindent` (or another amount).

If your algorithm ends without closing all blocks, the `algorithmicx` package gives you a nice error. So be good, and close them all!

Blocks are all the environments defined in the `algpseudocode` package, the `\Begin ... \End` block in the `algpascal` package, and some other ones.

Loops (Let us call them loops...) The loops are environments that include only one command, loop or block; a loop is closed automatically after this command. So loops have no ending commands. If your algorithm (or a block) ends before the single command of a loop, then this is considered an empty command, and the loop is closed. Feel free to leave open loops at the end of blocks!

Loops are most of the environments in the `algpascal` and `algc` packages.

For some rare constructions you can create mixtures of the two environments (see section **??**). Each block and loop may be continued with another one (like the `If` with `Else`).

4.2 Defining blocks

There are several commands to define blocks. The difference is in what is defined beyond the block. The macro `\algblock` defines a new block with starting and ending entity.

```
\algblock[<block>]{<start>}{<end>}
```

The defined commands have no parameters, and the text displayed by them is `\textbf{<start>}` and `\textbf{<end>}`. You can change these texts later (**??**).

With `\algblockdefx` you can give the text to be output by the starting and ending command and the number of parameters for these commands. In the text reference with $\#n$ to the parameter number n. Observe that the text is given in the form you define or redefine macros, and really, this is what happens.

```
\algblockdefx[<block>]{<start>}{<end>}
   [<startparamcount>][<default value>]{<start text>}
   [<endparamcount>][<default value>]{<end text>}
```

This defines a new block called `<block>`, `<start>` opens the block, `<end>` closes the block, `<start>` displays `<start text>`, and has `<startparamcount>` parameters, `<end>` displays `<end text>`, and has `<endparamcount>` parameters. For both `<start>` and `<end>`, if `<default value>` is given, then the first parameter is optional, and its default value is `<default value>`.

If you want to display different text (and to have a different number of parameters) for `<end>` at the end of different blocks, then use the `\algblockx` macro. Note that it is not possible to display different starting texts, since it is not possible to start different blocks with the same command. The `<start text>` defined with `\algblockx` has the same behavior as if defined with `\algblockdefx`. All ending commands not defined with `\algblockx` will display the same text, and the ones defined with this macro will display the different texts you specified.

```
\algblockx[<block>]{<start>}{<end>}
   [<startparamcount>][<default value>]{<start text>}
   [<endparamcount>][<default value>]{<end text>}
```

If in the above definitions the `<block>` is missing, then the name of the starting command is used as block name. If a block with the given name already exists, these macros don't define a new block, instead this it will be used the defined block. If `<start>` or `<end>` is empty, then the definition does not define a new starting/ending command for the block, and then the respective text must be missing from the definition. You may have more starting and ending commands for one block. If the block name is missing, then a starting command must be given.

```
\algblock[Name]{Start}{End}
\algblockdefx[NAME]{START}{END}%
   [2][Unknown]{Start #1(#2)}%
   {Ending}
\algblockdefx[NAME]{}{OTHEREND}%
   [1]{Until (#1)}
\begin{algorithmic}[1]
\Start
   \Start
      \START[One]{x}
      \END
      \START{0}
      \OTHEREND{\texttt{True}}
   \End
   \Start
   \End
\End
\end{algorithmic}
```

1: **Start**
2: **Start**
3: Start One(x)
4: Ending
5: Start Unknown(0)
6: Until (`True`)
7: **End**
8: **Start**
9: **End**
10: **End**

4.3 Defining loops

The loop defining macros are similar to the block defining macros. A loop has no ending command and ends after the first state, block or loop that follows the loop. Since loops have no ending command, the macro `\algloopx` would not have mutch sense. The loop defining macros are:

```
\algloop[<loop>]{<start>}

\algloopdefx[<loop>]{<start>}
   [<startparamcount>][<default value>]{<start text>}
```

Both create a loop named `<loop>` with the starting command `<start>`. The second also sets the number of parameters, and the text displayed by the starting command.

```
\algloop{For}
\algloopdefx{If}[1]{\textbf{If} #1 \textbf{then}}
\algblock{Begin}{End}
\begin{algorithmic}[1]
\For                                    1: For
   \Begin                               2:     Begin
   \If{$a < b$}                         3:         If $a < b$ then
      \For                              4:             For
         \Begin                         5:                 Begin
         \End                           6:                 End
   \Begin                               7:         Begin
   \End                                 8:         End
   \End                                 9:     End
\end{algorithmic}
```

4.4 Continuing blocks and loops

For each block/loop you may give commands that close the block or loop and open another block or loop. A good example for this is the **if** ... **then** ... **else** construct. The new block or loop can be closed or continued, as any other blocks and loops.

To create a continuing block use one of the following:

```
\algcblock[<new block>]{<old block>}{<continue>}{<end>}

\algcblockdefx[<new block>]{<old block>}{<continue>}{<end>}
   [<continueparamcount>][<default value>]{<continue text>}
   [<endparamcount>][<default value>]{<end text>}

\algcblockx[<new block>]{<old block>}{<continue>}{<end>}
   [<continueparamcount>][<default value>]{<continue text>}
   [<endparamcount>][<default value>]{<end text>}
```

All three macros define a new block named `<new block>`. If `<new block>` is not given, then `<continue>` is used as the new block name. It is not allowed to have both `<new block>` missing, and `<continue>` empty. The `<continue>` command ends the `<old block>` block/loop and opens the `<new block>` block. Since `<continue>` may end different blocks and loops, it can have different text at the end of the different blocks/loops. If the `<continue>` command doesn't find an `<old block>` to close, then an error is reported.

Create continuing loops with the followings:

```
\algcloop[<new loop>]{<old block>}{<continue>}

\algcloopdefx[<new loop>]{<old block>}{<continue>}
   [<continueparamcount>][<default value>]{<continue text>}
```

```
\algcloopx[<new loop>]{<old block>}{<continue>}
   [<continueparamcount>][<default value>]{<continue text>}
```

These macros create a continuing loop, the `<continue>` closes the `<old block>` block/loop, and opens a `<new loop>` loop.

```
\algblock{If}{EndIf}
\algcblock[If]{If}{ElsIf}{EndIf}
\algcblock{If}{Else}{EndIf}
\algcblockdefx[Strange]{If}{Eeee}{Oooo}
   [1]{\textbf{Eeee} "#1"}
   {\textbf{Wuuuups\dots}}
\begin{algorithmic}[1]
\If
   \If
   \ElsIf
   \ElsIf
      \If
      \ElsIf
      \Else
      \EndIf
   \EndIf
   \If
   \EndIf
\Eeee{Creep}
\Oooo
\end{algorithmic}
```

1: **If**
2: **If**
3: **ElsIf**
4: **ElsIf**
5: **If**
6: **ElsIf**
7: **Else**
8: **EndIf**
9: **EndIf**
10: **If**
11: **EndIf**
12: **Eeee** "Creep"
13: **Wuuuups...**

```
\algloop{If}
\algcloop{If}{Else}
\algblock{Begin}{End}
\begin{algorithmic}[1]
\If
   \Begin
   \End
\Else
   \If
      \Begin
      \End
\end{algorithmic}
```

1: **If**
2: **Begin**
3: **End**
4: **Else**
5: **If**
6: **Begin**
7: **End**

4.5 Even more customisation

With the following macros you can give the indentation used by the new block (or loop), and the number of statements after that the "block" is automatically

closed. This value is ∞ for blocks, 1 for loops, and 0 for statements. There is a special value, 65535, meaning that the defined "block" does not end automatically, but if it is enclosed in a block, then the ending command of the block closes this "block" as well.

```
\algsetblock[<block>]{<start>}{<end>}
   {<lifetime>}{<indent>}

\algsetblockdefx[<block>]{<start>}{<end>}
   {<lifetime>}{<indent>}
   [<startparamcount>][<default value>]{<start text>}
   [<endparamcount>][<default value>]{<end text>}

\algsetblockx[<block>]{<start>}{<end>}
   {<lifetime>}{<indent>}
   [<startparamcount>][<default value>]{<start text>}
   [<endparamcount>][<default value>]{<end text>}

\algcsetblock[<new block>]{<old block>}{<continue>}{<end>}
   {<lifetime>}{<indent>}

\algcsetblockdefx[<new block>]{<old block>}{<continue>}{<stop>}
   {<lifetime>}{<indent>}
   [<continueparamcount>][<default value>]{<continue text>}
   [<endparamcount>][<default value>]{<end text>}

\algcsetblockx[<new block>]{<old block>}{<continue>}{<stop>}
   {<lifetime>}{<indent>}
   [<continueparamcount>][<default value>]{<continue text>}
   [<endparamcount>][<default value>]{<end text>}
```

The `<lifetime>` is the number of statements after that the block is closed. An empty `<lifetime>` field means ∞. The `<indent>` gives the indentation of the block. Leave this field empty for the default indentation. The rest of the parameters has the same function as for the previous macros.

```
\algsetblock[Name]{Start}{Stop}{3}{1cm}
\algsetcblock[CName]{Name}{CStart}{CStop}{2}{2cm}
\begin{algorithmic}[1]
\Start                          1: Start
   \State 1                     2:       1
   \State 2                     3:       2
   \State 3                     4:       3
\State 4                        5: 4
\Start                          6: Start
   \State 1                     7:       1
\Stop                           8: Stop
\State 2                        9: 2
\Start                          10: Start
   \State 1                     11:      1
\CStart                         12: CStart
   \State 1                     13:              1
   \State 2                     14:              2
\State 3                        15: 3
\Start                          16: Start
   \State 1                     17:      1
\CStart                         18: CStart
   \State 1                     19:              1
\CStop                          20: CStop
\end{algorithmic}
```

The created environments behave as follows:

- It starts with `\Start`. The nested environments are indented by 1 cm.
- If it is followed by at least 3 environments (statements), then it closes automatically after the third one.
- If you put a `\Stop` before the automatic closure, then this `\Stop` closes the environment. `CStart` closes a block called `Name` and opens a new one called `CName` and having an indentaion of 2 cm.
- `CName` can be closed with `CStop` or it is closed automatically after 2 environments.

4.6 Parameters, custom text

With `\algrenewtext` you can change the number of parameters, and the text displayed by the commands. With `algnotext` you can makes the vole output line disappear, but it works only for ending commands, for beginning commands you will get an incorrect output.

```
\algrenewcommand[<block>]{<command>}
   [<paramcount>][<default value>]{<text>}
```

`\algnotext[<block>]{<ending command>}`

If `<block>` is missing, then the default text is changed, and if `<block>` is given, then the text displayed at the end of `<block>` is changed.

To make a command output the default text at the end of a block (say, you have changed the text for this block), use `\algdefaulttext`.

`\algdefaulttext[<block>]{<command>}`

If the `<block>` is missing, than the default text itself will be set to the default value (this is `\textbf{<command>}`).

4.7 The ONE defining macro

All block and loop defining macros call the same macro. You may use this macro to gain a better acces to what will be defined. This macro is `\algdef`.

`\algdef{<flags>}...`

Depending on the flags the macro can have many forms.

Flag	Meaning
s	starting command, without text
S	starting command with text
c	continuing command, without text
C	continuing command, with default text
xC	continuing command, with block specific text
e	ending command, without text
E	continuing command, with default text
xE	continuing command, with block specific text
N	ending command, with default "no text"
xN	ending command, with no text for this block
b	block(default)
l	loop
L	loop closes after the given number of statements
i	indentation specified

The `<new block>` may be given for any combination of flags, and it is not allowed to have `<new block>` missing and `<start>` missing/empty. For c, C, xC an old block is expected. For s, S, c, C, xC the `<start>` must be given. For e, E, xE, N, xN the `<end>` must be given. For L the `<lifetime>` must be given. For i the `<indent>` must be given. For S, C, xC the starting text and related infos must be given. For E, xE the ending text must be given. For each combination of flags give only the needed parameters, in the following order:

```
\algdef{<flags>}[<new block>]{<old block>}{<start>}{<end>}
   {<lifetime>}{<indent>}
   [<startparamcount>][<default value>]{<start text>}
   [<endparamcount>][<default value>]{<end text>}
```

The block and loop defining macros call `\algdef` with the following flags:

Macro	**Meaning**
\algblock	\algdef{se}
\algcblock	\algdef{ce}
\algloop	\algdef{sl}
\algcloop	\algdef{cl}
\algsetblock	\algdef{seLi}
\algsetcblock	\algdef{ceLi}
\algblockx	\algdef{SxE}
\algblockdefx	\algdef{SE}
\algcblockx	\algdef{CxE}
\algcblockdefx	\algdef{CE}
\algsetblockx	\algdef{SxELi}
\algsetblockdefx	\algdef{SELi}
\algsetcblockx	\algdef{CxELi}
\algsetcblockdefx	\algdef{CELi}
\algloopdefx	\algdef{Sl}
\algcloopx	\algdef{Cxl}
\algcloopdefx	\algdef{Cl}

5 Examples

5.1 A full example using algpseudocode

```
\documentclass{article}
\usepackage{algorithm}
\usepackage{algpseudocode}
\begin{document}
\begin{algorithm}
\caption{The Bellman-Kalaba algorithm}
\begin{algorithmic}[1]
\Procedure {BellmanKalaba}{$G$, $u$, $l$, $p$}
   \ForAll {$v \in V(G)$}
      \State $l(v) \leftarrow \infty$
   \EndFor
   \State $l(u) \leftarrow 0$
   \Repeat
      \For {$i \leftarrow 1, n$}
         \State $min \leftarrow l(v_i)$
         \For {$j \leftarrow 1, n$}
            \If {$min > e(v_i, v_j) + l(v_j)$}
               \State $min \leftarrow e(v_i, v_j) + l(v_j)$
               \State $p(i) \leftarrow v_j$
            \EndIf
         \EndFor
         \State $l'(i) \leftarrow min$
      \EndFor
      \State $changed \leftarrow l \not= l'$
      \State $l \leftarrow l'$
   \Until{$\neg changed$}
\EndProcedure
\Statex
\Procedure {FindPathBK}{$v$, $u$, $p$}
   \If {$v = u$}
      \State \textbf{Write} $v$
   \Else
      \State $w \leftarrow v$
      \While {$w \not= u$}
         \State \textbf{Write} $w$
         \State $w \leftarrow p(w)$
      \EndWhile
   \EndIf
\EndProcedure
\end{algorithmic}
\end{algorithm}
\end{document}
```

Algorithm 2 The Bellman-Kalaba algorithm

```
1: procedure BellmanKalaba(G, u, l, p)
2:     for all v ∈ V(G) do
3:         l(v) ← ∞
4:     end for
5:     l(u) ← 0
6:     repeat
7:         for i ← 1, n do
8:             min ← l(v_i)
9:             for j ← 1, n do
10:                if min > e(v_i, v_j) + l(v_j) then
11:                    min ← e(v_i, v_j) + l(v_j)
12:                    p(i) ← v_j
13:                end if
14:            end for
15:            l'(i) ← min
16:        end for
17:        changed ← l ≠ l'
18:        l ← l'
19:    until ¬changed
20: end procedure

21: procedure FindPathBK(v, u, p)
22:    if v = u then
23:        Write v
24:    else
25:        w ← v
26:        while w ≠ u do
27:            Write w
28:            w ← p(w)
29:        end while
30:    end if
31: end procedure
```

5.2 Breaking up an algorithm

```
\documentclass{article}
\usepackage{algorithm}
\usepackage{algpseudocode}
\begin{document}
\begin{algorithm}
\caption{Part 1}
\begin{algorithmic}[1]
\Procedure {BellmanKalaba}{$G$, $u$, $l$, $p$}
   \ForAll {$v \in V(G)$}
      \State $l(v) \leftarrow \infty$
   \EndFor
   \State $l(u) \leftarrow 0$
   \Repeat
      \For {$i \leftarrow 1, n$}
         \State $min \leftarrow l(v_i)$
         \For {$j \leftarrow 1, n$}
            \If {$min > e(v_i, v_j) + l(v_j)$}
               \State $min \leftarrow e(v_i, v_j) + l(v_j)$
               \State \Comment For some reason we need to break here!
\algstore{bkbreak}
\end{algorithmic}
\end{algorithm}

And we need to put some additional text between\dots

\begin{algorithm}[h]
\caption{Part 2}
\begin{algorithmic}[1]
\algrestore{bkbreak}
               \State $p(i) \leftarrow v_j$
            \EndIf
         \EndFor
         \State $l'(i) \leftarrow min$
      \EndFor
      \State $changed \leftarrow l \not= l'$
      \State $l \leftarrow l'$
   \Until{$\neg changed$}
\EndProcedure
\end{algorithmic}
\end{algorithm}
\end{document}
```

Algorithm 3 Part 1

```
1: procedure BellmanKalaba(G, u, l, p)
2:     for all v ∈ V(G) do
3:         l(v) ← ∞
4:     end for
5:     l(u) ← 0
6:     repeat
7:         for i ← 1, n do
8:             min ← l(v_i)
9:             for j ← 1, n do
10:                if min > e(v_i, v_j) + l(v_j) then
11:                    min ← e(v_i, v_j) + l(v_j)
12:                                    ▷ For some reason we need to break here!
```

And we need to put some additional text between...

Algorithm 4 Part 2

```
13:                    p(i) ← v_j
14:                end if
15:            end for
16:            l'(i) ← min
17:        end for
18:        changed ← l ≠ l'
19:        l ← l'
20:    until ¬changed
21: end procedure
```

5.3 Using multiple layouts

```
\documentclass{article}
\usepackage{algorithm}
\usepackage{algpseudocode}
\usepackage{algpascal}
\begin{document}

\alglanguage{pseudocode}
\begin{algorithm}
\caption{A small pseudocode}
\begin{algorithmic}[1]
\State $s \gets 0$
\State $p \gets 0$
\For{$i \gets 1,\, 10$}
   \State $s \gets s + i$
   \State $p \gets p + s$
\EndFor
\end{algorithmic}
\end{algorithm}

\alglanguage{pascal}
\begin{algorithm}
\caption{The pascal version}
\begin{algorithmic}[1]
\State $s := 0$
\State $p := 0$
\For{i = 1}{10}
   \Begin
   \State $s := s + i$
   \State $p := p + s$
   \End
\end{algorithmic}
\end{algorithm}

\end{document}
```

Algorithm 5 A small pseudocode

1: $s \leftarrow 0$
2: $p \leftarrow 0$
3: **for** $i \leftarrow 1$, 10 **do**
4: $\quad s \leftarrow s + i$
5: $\quad p \leftarrow p + s$
6: **end for**

Algorithm 6 The pascal version

1: $s := 0$
2: $p := 0$
3: **for** $i = 1$ **to** 10 **do**
4: $\quad$ **begin**
5: $\quad s := s + i$
6: $\quad p := p + s$
7: $\quad$ **end**

6 Bugs

If you have a question or find a bug you can contact me on:

szaszjanos@users.sourceforge.net

If possible, please create a small LaTeX example related to your problem.