The `typst-algo` package.

Typeset algorithms in Typst.

Hugo SALOU — https://github.com/hugo-s29/typst-algo

(last documentation update : 2023-07-04)

Goals. This project aims to be a Typst equivalent of the LaTeX package `algpseudocode`. There is already a Typst package aimed at algorithm typsetting, but `typst-algorithms`'s style is a lot closer to code than algorithms. The main objective of this package is to be able to render an algorithm like Algorithm 1. A step-by-step breakdown of the code (in Listing 1) is available in Section 1.

$n \leftarrow \text{length}(T)$
While T isn't sorted **do**
  $i \leftarrow \mathcal{U}(\{1, ..., n\})$
  $j \leftarrow \mathcal{U}(\{1, ..., n\})$
  Swap elements at index i and index j in T
End While

Algorithm 1: A *very efficient* sorting algorithm

Contents

1. Principles

To typeset an algorithm with `typst-algo`, you use functions for each "instruction" you want to show. In order to better understand, I'll explain step-by-step the code (Listing 1) used to typeset Algorithm 1. In Section 2, there are more complex examples (procedures, for loops, "blocks within blocks," *etc*).

```
#algorithm[
  $n <- "length"(T)$\
  #algo_while[$T$ isn't sorted] #algo_block[
    $i <- cal(U)({1, ..., n})$\
    $j <- cal(U)({1, ..., n})$\
    Swap elements at index $i$ and index $j$ in $T$
  ]
  #algo_end_while
]
```

Listing 1: The code used to typeset Algorithm 1

Firstly, the whole algorithm is wrapped in a function named `algorithm`. This function takes only one argument, the algorithm's content. To write simple lines like $n \leftarrow \text{length}(T)$, you don't need special instructions; you can just add it inside the algorithm's content. However, remember to add a `\` at the end of your line to add a line break.

To write the *while* loop, you use the `algo_while` function. This function takes one argument, the "test" used by the while loop. The while loop's content needs to be added afterwards. If the content cannot be displayed after the while instruction, you need to use the `algo_block` function. (You can look at more examples in Section 2.) In our case, the while loop's body contains three lines so we need to add a *block*. The `algo_block` function works in a similar manner to the `algorithm` function: you can directly write text, or add instructions (see more complex examples in Section 2). You don't need to add a line break after the while instruction, since `algo_block` does it automatically.

After the block is filled with instructions, we can call the `algo_end_while` function, it'll add "End While."

All other instructions work similarly, there's a list of usable functions in Section 3.

2. First examples

In this section, there will be some examples of algorithms typeset with `typst-algo` and the code used.

2.1. An algorithm to approximate π.

Input a value n.
$m \leftarrow 0$
For $i \in \{1, ..., n\}$ **do**
  $x \leftarrow \mathcal{U}([0, 1])$
  $y \leftarrow \mathcal{U}([0, 1])$
  If $x^2 + y^2 \leq 1$ **then** $m \leftarrow m + 1$
End For
Return $4 \cdot m \,/\, n$

```
#algorithm[
  Input a value $n$.\
  $m <- 0$\
  #algo_for[$i in {1,...,n}$]
  #algo_block[
    $x <- cal(U)([0,1])$\
    $y <- cal(U)([0,1])$\
    #algo_if[$x^2 + y^2 <= 1$]
      $m <- m + 1$
  ]
  #algo_end_for
  #algo_return $4 dot m \/ n$
]
```

2.2. The Quine–McCluskey algorithm for solving SAT.

Procedure ASSUME(F, p, v)
This procedure will return $F[p \mapsto v]$ where F is written in CNF, p is one of its variables, and $v \in \mathbb{B}$ is a boolean.[1] The notation $F[p \mapsto v]$ means we are substituting the variable p with the value v.

Let $\ell_{\boldsymbol{T}}$ be the literal p if $v = \boldsymbol{T}$, otherwise $\neg p$.
Let $\ell_{\boldsymbol{F}}$ be the literal p if $v = \boldsymbol{F}$, otherwise $\neg p$.
For $C \in F$ **do**
If $\ell_{\boldsymbol{T}} \in C$ **then** we remove C from F.
Else if $\ell_{\boldsymbol{F}} \in C$ **then** we remove $\ell_{\boldsymbol{F}}$ from C.
End If
End For
End Procedure

Procedure QUINE(F)
If $\emptyset = F$ **then Return** $\boldsymbol{T}$
Else if $\emptyset \in F$ **then Return** $\boldsymbol{F}$
Else if $\exists\{\ell\} \in F$ **then**
If $\ell = p$, with $p \in \mathrm{vars}(F)$ **then Return** QUINE(ASSUME($F, p, \boldsymbol{T}$))
Else if $\ell = \neg p$, with $p \in \mathrm{vars}(F)$ **then Return** QUINE(ASSUME($F, p, \boldsymbol{F}$))
End If
Else
Let $p \in \mathrm{vars}(F)$.
Return QUINE(ASSUME($F, p, \boldsymbol{T}$)) $\vee$ QUINE(ASSUME($F, p, \boldsymbol{F}$))
End If
End Procedure

Algorithm 2: The Quine–McCluskey algorithm for solving SAT

[1] For boolean values, we'll write $\boldsymbol{F}$ for false, and $\boldsymbol{T}$ for true, and thus, $\mathbb{B} = \{\boldsymbol{F}, \boldsymbol{T}\}$.

```
#algorithm[
  #algo_procedure(args: [$F,p,v$])[Assume]
  #algo_block[
    This procedure ... the variable $p$ with the value~$v$.\
    #v(0.5cm)
    Let $ell_bold(T)$ be the literal $p$ if $v = bold(T)$, otherwise $not p$.\
    Let $ell_bold(F)$ be the literal $p$ if $v = bold(F)$, otherwise $not p$.\
    #algo_for[$C in F$] #algo_block[
      #algo_if[$ell_bold(T) in C$] we remove $C$ from $F$.\
      #algo_else_if[$ell_bold(F) in C$] we remove $ell_bold(F)$ from $C$.\
      #algo_end_if
    ]
    #algo_end_for
  ]
  #algo_end_procedure

  #v(0.5cm)

  #algo_procedure(args: [$F$])[Quine]
  #algo_block[
    #algo_if[$nothing = F$] #algo_return $bold(T)$\
    #algo_else_if[$nothing in F$] #algo_return $bold(F)$\
    #algo_else_if[$exists {ell} in F$] #algo_block[
      #algo_if[$ell = p$, with $p in "vars"(F)$]
        #algo_return #algo_call([Quine], args:
          algo_call([Assume], args: [$F,p,bold(T)$]))\
      #algo_else_if[$ell = not p$, with $p in "vars"(F)$]
        #algo_return #algo_call([Quine], args:
          algo_call([Assume], args: [$F,p,bold(F)$]))\
      #algo_end_if
    ]
    #algo_else #algo_block[
      Let $p in "vars"(F)$.\
      #algo_return #algo_call([Quine], args:
        algo_call([Assume], args: [$F,p,bold(T)$]))
      $or$ #algo_call([Quine], args:
        algo_call([Assume], args: [$F,p,bold(F)$]))\
    ]
    #algo_end_if
  ]
  #algo_end_procedure
]
```

Listing 2: Code used to typeset Algorithm 2

3. Reference

- **Conditionals.** `#algo_if[condition]` will produce " **If** condition **then** ". This should be followed by `#algo_end_if` (after the *if* instruction's content).
- **Block.** `#algo_block[block\ content]` will produce

  > block
  > content.

  This can be used inside between any pairs of instructions (*e.g.* "if", "for", "while", ...) if the content needs to be on multiple lines.

- **Procedures.** `#algo_procedure(args: [args])[name]` will produce "**Procedure** NAME(args)". This should be followed by `#algo_end_procedure` (after the procedure's content).
- **Functions.** Similar to procedures, but using `#algo_function` and `#algo_end_function` instead.
- **Calling procedures or functions.** `#algo_call(args: [args])[name]` will appear in your document as "NAME(args)". This can be used to call a procedure or a function.
- **For loops.** `#algo_for[loop_iteration]` will result in " **For** loop_iteration **do** ". This should be followed by `#algo_end_for` (after the for loop's content).
- **While loops.** Similar to for loops, but using `#algo_while` and `#algo_end_while` instead.

If some instruction is missing, please see Section 4 to know how to contribute to `typst-algo`.

4. Contributing

This project is open-source (MIT-licensed). Feel free to contribute if you think a feature is missing, the code could be improved, or anything else.