The Pygments Pseudocode Lexer

Version 3.0

Franz Glasner

2026-05-23

Last updated on 2026-05-23 (rev cd22c2e8390d)

Contents

CHAPTER 1

Introduction

Metadata

Version
3.0

Date
2026-05-23

Copyright

- © 2026 Franz Glasner
- © 2015 Simon Wachter

License
MIT License

Revision
cd22c2e8390d

This package contains Pygments[1] lexers for some basic pseudocode.

Initially a fork of *pygments-lexer-pseudocode* it has been considerably changed and expanded.

It now contains the following lexers:

Language Name	Description	Extension(s)	Aliases / Short Name(s)	Lexer Class
AlgPseudocode	Pseudocode lexer inspired by CTAN's Algpseudocodex[2]	*.algpseudocode, *.algpseudo	algpseudocode, algpseudo	AlgPseudocodeLexer
AlgPseudocodeFR	AlgPseudocode with french keyword expansion	*.algpseudo-fr, *.algpseudocode-fr	algpseudocode-fr, algpseudo-fr	AlgPseudocodeLexer_FR
AlgPseudocodeDE	AlgPseudocode with german keyword expansion	*.algpseudo-de, *.algpseudocode-de	algpseudocode-de, algpseudo-de	AlgPseudocodeLexer_DE
FrPseudocode	The original lexer from *pygments-lexer-pseudocode* – renamed and slightly changed	*.fr-algo, *.fr-pseudocode	fr-pseudocode, fr-pseudo, fr-algorithm, fr-algo	FrPseudocodeLexer

[1] https://pygments.org/

It additionally contains the following filters:

Filter Name	Description	Filter Class
tokenreplace	A configurable token replacer: replace a given token type by another given token type in a stream	TokenReplaceFilter
errortogenericerror	Replace all `Error` tokens in a stream by `Generic.Error` tokens	ErrorToGenericErrorTokenFilter

1.1 Installation

The lexers and filters are available as a Python[3] wheel package:

```
pip install pygments-lexer-pseudocode2
```

Alternatively, to build from the source code repository:

```
python -m build
```

1.2 Usage

After installation all the Pseudocode lexers and filters automatically register itself with the plugin system of Pygments. Their respective aliases and filename extensions are made known to Pygments.

Usage is easy:

Pygmentizing a file with a known file extension:

```
pygmentize document.algpseudocode
```

Or you can manually select the Pseudocode lexer you want to use by using a command line flag:

```
pygmentize -l algpseudocode somefile
```

Within Sphinx[4] use a lexer like this:

```
.. code-block:: algpseudocode
```

1.3 Licenses

The package is licensed under the terms of the MIT Licence.

Some code snippets are licensed under the terms of the BSD 2-Clause “Simplified” License as they are taken from the Python lexer that is included in Pygments[5].

[2] https://ctan.org/pkg/algpseudocodex
[3] https://www.python.org/
[4] https://www.sphinx-doc.org
[5] https://pygments.org/

CHAPTER 2

Lexers

2.1 Overview

The package contains the following lexers:

Language Name	Description	Extension(s)	Aliases / Short Name(s)	Lexer Class
AlgPseudocode	Pseudocode lexer inspired by CTAN's Algpseudocodex[6]	*.algpseudocode, *.algpseudo	algpseudocode, algpseudo	AlgPseudocodeLexer
AlgPseudocodeFR	AlgPseudocode with french keyword expansion	*.algpseudo-fr, *.algpseudocode-fr	algpseudocode-fr, algpseudo-fr	AlgPseudocodeLexer_FR
AlgPseudocodeDE	AlgPseudocode with german keyword expansion	*.algpseudo-de, *.algpseudocode-de	algpseudocode-de, algpseudo-de	AlgPseudocodeLexer_DE
FrPseudocode	The original lexer from *pygments-lexer-pseudocode* – renamed and slightly changed	*.fr-algo, *.fr-pseudocode	fr-pseudocode, fr-pseudo, fr-algorithm, fr-algo	FrPseudocodeLexer

2.2 AlgPseudocode and Language Variants

These lexers are heavily inspired by CTAN's Algpseudocodex[7]. They recognize all sorts of single- and multi-line comments in addition to expressions and commands that are inspired by Algpseudocodex[8].

They are used in Sphinx[9] using their aliases. The code block:

```
.. code-block:: algpseudocode

   \PROGRAM {The Pseudoprogram} \IS

   \END PROGRAM {The Pseudoprogram}
```

[6] https://ctan.org/pkg/algpseudocodex
[7] https://ctan.org/pkg/algpseudocodex
[8] https://ctan.org/pkg/algpseudocodex
[9] https://www.sphinx-doc.org

will be rendered as:

```
PROGRAM The Pseudoprogram IS

END OF PROGRAM The Pseudoprogram
```

And the same code block with the german variant (using `.. code-block:: algpseudocode-de` as language alias):

```
PROGRAMM The Pseudoprogram IST

ENDE DES PROGRAMMS The Pseudoprogram
```

2.2.1 States

The AlgPseudocode lexer and its language variants AlgPseudocodeDE and AlgPseudocodeFR basically work in three states (aka modes or contexts): *default*, *expression* and *text*.

In *expressions* it automatically recognizes:

- Strings (single-quote, double-quote, triple-single-quote, triple-double-quote, Python[10] style)
- Numbers (also Python[11] style)
- (Mathematical) operators and symbols
- `\TEXT{...}` or `\T{...}`

  Used to switch to a text-mode that prohibits automatic expression highlighting.

  A closing curly brace can be quoted with `\}` to not end the text mode prematurely.
- `\EXPR`, `\EXPRESSION` or `\E` as nested construct
- `\NAME`, `\CALL` and `\GETS`
- `\REM` and `\REMARK` for remarks (aka comments)
- Names (*Name.Entity*)
- *Explicit Token Types* (page 10)

The *default*-mode is an extension of *expression*: in addition to *expressions* it recognizes all sorts of single- and multi-line comments and commands that are inspired by Algpseudocodex[12].

In *text* context it recognizes:

- `\EXPRESSION`, `\EXPR` or `\E`

  Use to switch to expression-mode.

  A closing curly brace can be quoted with `\}` to not end the expression mode prematurely.
- `\TEXT` (aka `\T`) as nested construct
- `\REM` and `\REMARK` for remarks (aka comments)
- *Explicit Token Types* (page 10)

[10] https://www.python.org/
[11] https://www.python.org/
[12] https://ctan.org/pkg/algpseudocodex

2.2.2 Lexer Options

`prohibit_raiseonerror_filter`

Type: `bool`

Default: `None`

If `True` the *raiseonerror* filter is not allowed to be applied by Sphinx[13] when `Lexer.add_filter()` is called.

This setting does not apply to filters that are set by the standard lexer option *filters*.

`no_end`

Type: `bool`

Default: `False`

If `True` all the `\ENDxxx` commands will be skipped and yield no output.

`strict_tokentype`

Type:: `bool`

Default: `True`

Control whether *Explicit Token Types* (page 10) yield `pygments.token.Token.Generic.Error` tokens (when `True`, this is the default) or a token type that is synthesized on the fly by `pygments.token.string_to_tokentype()` (when `False`).

`gets`

Type: `str` or `None`

Default: `None` (yields ←)

The operator symbol to be printed by the command `\GETS`.

An often used alternative is `:=`.

`remark`

Type: `str` or `None`

Default: `None` (yields ▷)

The symbol to be printed as when starting comments with `\REMARK` or `\REM`.

To use a lexer with non-default options in Sphinx[14] see section *Customized Lexers in Sphinx* (page 12).

2.2.3 Comments

- with the `\REMARK` or `\REM` keywords (until the end of the line; the output includes a leading symbol, by default ▷)
- multi-line comments with `/* ... */`; they can be **nested**
- multi-line comments with `(* ... *)`; they can be **nested**
- single-line comments with `//` or `#` (until the end of the line)

```
/*
 * A single multiline comment
 */

/*
```

(continues on next page)

[13] https://www.sphinx-doc.org
[14] https://www.sphinx-doc.org

(continued from previous page)

```
 * A multiline comment
 *
 * /* This is a nested multi-line comment */
 *
 */

(*
 * A multiline comment
 *
 * (* This is a nested multi-line comment *)
 *
 *)

// A single-line comment

# A single-line comment

▷ A remark is a single-line comment with a leading symbol
```

2.2.4 Literals

Strings and numbers as in Python[15]. String prefixes `u` and `b` are supported—prefixes `r`, `f` and `t` are not supported.

To have non-string-delimiting single- and double-quotes in the output you have to escape them using `\'` or `\”`. This must be used to typeset something as `f'(x) = 0`.

```
0  1234567890 0xdead 0b100001 0o720  2.7 2.7e-54

”A string with an escaped double-quote \” ”

'Another string with an escaped single-quote \' '

”””A multiline
string
”””

'''Another multiline string

'''

b”A \x20 byte string”

u'An explicit Unicode \u1234 string'

”  a non string

'  a non string also
```

[15] https://www.python.org/

2.2.5 (Mathematical) Symbols and Operators

Some ASCII symbol combinations are recognized and replaced by a Unicode symbol:

```
<=>     ⇔
<->     ↔
<-      ←
->      →
=>      ⇒
<=      ≤
>=      ≥
<>      ≠
!=      ≠
:=      :=
=:      =:
?=      ≟
```

Unicode codepoints with property `Sm` are recognized as mathematical symbols and highlighted accordingly.

2.2.6 Punctuation

Runs of dots `.`, `..`, `...`, `....`, ... are handled properly in expressions and yield a punctuation token. They are not replaced by corresponding Unicode symbols.

2.2.7 Commands

- Start with a backslash character `\`
- Case-insensitive
- Yield mostly the `pygments.token.Token.Keyword` token type
- Translated if a translation is found
- Depending on the command—may have required or optional parameters

  Parameter handling is as follows:

  - Parameters are enclosed in curly braces `{` and `}`
  - Escaping within the braces is possible using the backslash `\` as escape character
  - Parameters are separated from the keyword/command by a (possibly empty) run of space or TAB characters. This is true for required and optional parameters.
- Unrecognized commands typically result in a `pygments.token.Token.Error` token.

More on escaping rules you can find in *this chapter* (page 12).

Commands With Required Parameters

```
\PROGRAM {A Program}  or  \PROG {A Program}                                 ␣
↪PROGRAM A Program
\ALGORITHM{An Algorithm}  or  \ALGO{An Algorithm}                           ␣
↪ALGORITHM An Algorithm
\PROCEDURE{A Procedure}  or  \PROC{A Procedure}                             ␣
↪PROCEDURE A Procedure
\FUNCTION{A Function}  or  \FUNC{A Function}  or  \FN{A Function}           ␣
↪FUNCTION A Function
\CLASS{A Class}                                                             ␣
```

(continues on next page)

(continued from previous page)

```
↪CLASS A Class

\STATEMENT{the expression}  \STATE{the expression}  \BLOCK{the expression}     ◆␣
↪the expression

expr1: \EXPRESSION{expression a in b}   expr2: \EXPR{expression b in a}       ␣
↪expr1: expression a in b   expr2: expression b in a

\TEXTSTATEMENT{the text}  \TEXTSTATE{the text}  \TSTATEMENT{the text}  \TSTATE{the␣
↪text}  \TEXTBLOCK{the text}  \TBLOCK{the text}             ▪ the text

\INPUT{Input 1}                          Input: Input 1
\INPUTS{Input 2}                         Inputs: Input 2

\OUTPUT{Output 1}                        Output: Output 1
\OUTPUTS{Output 2}                       Outputs: Output 2

\ENSURE{Whatever should be ensured!}     Ensure: Whatever should be ensured!

\REQUIRE{Whatever should be required.}   Require: Whatever should be required.

\RETURNS{Return 2}                       Returns: Return 2

\CALL{a function}(p1, p2)                a function(p1, p2)

\NAME{an entity name}                    an entity name
```

For a special command with two required parameters see *Explicit Token Types* (page 10).

Commands With Optional Parameters

Some END-commands have optional parameters:

```
\ENDPROGRAM  \ENDPROG              END OF PROGRAM
\ENDALGORITHM  \ENDALGO            END OF ALGORITHM
\ENDPROCEDURE  \ENDPROC            END OF PROCEDURE
\ENDFUNCTION  \ENDFUNC  \ENDFN     END OF FUNCTION
\ENDCLASS                          END OF CLASS
```

They are used like this:

```
\CLASS{Foo Bar Class} ... \END CLASS {Foo Bar Class}   yields   CLASS Foo Bar Class␣
↪... END OF CLASS Foo Bar Class

\CLASS{Foo Bar Class} ... \END CLASS                   yields   CLASS Foo Bar Class␣
↪... END OF CLASS
```

See also:

For other syntax variants concerning *END* see also section *END-Commands* (page 9).

Commands Without Parameters

"Normal" Commands

```
\IF                                 IF
\THEN                               THEN
\ELSE                               ELSE
\ELSEIF or \ELSIF  or  \ELIF        ELSE IF or ELSE IF or ELSE IF
\DO                                 DO
\WHILE                              WHILE
\FORALL                             FOR ALL
\FOR                                FOR
\FROM                               FROM
\TO                                 TO
\STEP                               STEP
\IN                                 IN
\LOOP                               LOOP
\REPEAT                             REPEAT
\UNTIL                              UNTIL

\RETURN                             RETURN

\BEGIN                              BEGIN
\END                                END

\IS                                 IS
\WITH                               WITH

\GETS                               ←

\REMARK   or   \REM                 ▷ A comment with a leading symbol
```

\REMARK or \REM is special: all characters to the end of the line are taken as comment; curly braces are not needed—in fact: they are interpreted to be part of the comment.

END-Commands

The separator character can be empty, a run of ASCII spaces, a run of TAB characters, a single underscore _ or a single hyphen -.

All of the following examples are equally valid and result in the same output:

\ENDIF, \END IF, \END-IF, \END_IF or \END IF

```
\ENDIF             END IF     ▷ empty

\END IF            END IF     ▷ a single space

\END  IF           END IF     ▷ two spaces

\END-IF            END IF     ▷ a single hyphen

\END_IF            END IF     ▷ a single underscore

\END   IF          END IF     ▷ a single TAB character
```

The list of END-commands (here always just with - as separator):

```
\END-PROGRAM  \END-PROG             END OF PROGRAM
\END-ALGORITHM  \END-ALGO           END OF ALGORITHM
```

(continues on next page)

(continued from previous page)

```
\END-PROCEDURE  \END-PROC            END OF PROCEDURE
\END-FUNCTION  \END-FUNC  \END-FN    END OF FUNCTION
\END-CLASS                           END OF CLASS
\END-IF                              END IF
\END-WHILE                           END WHILE
\END-FOR                             END FOR
\END-FORALL                          END FOR ALL
\END-LOOP                            END LOOP
```

Note: The output of these END-commands can be suppressed by setting the lexer option `no_end` to `True`.

2.2.8 Names and Entities

In an *expression* context all other words are interpreted as entity names (token type `pygments.token.Token.Name.Entity`).

Allowed characters in the words follow the corresponding Python[16] rules. As such, many Unicode characters are allowed.

To highlight entity names with whitespace or other "special" characters in it use the `NAME` command.

```
entity_name_1          entity_name_1

entity_name_2          entity_name_2

\NAME{entity-name 3}   entity-name 3

München                München

Genève                 Genève
```

Note: Should you want to change the token type and the associated highlighting you may want to have a look at *TokenReplaceFilter* (page 22).

2.2.9 Explicit Token Types

They allow to handle keywords and operators that are not recognized by default. And they allow the user to explicitely highlight some input text with a low-level command.

They are implemented with the `\ttX{ARG1}{ARG2}` command.

This command has two required parameters:

1. The content of the first argument *ARG1* can be one of
   - A *value* in the `pygments.token.STANDARD_TYPES` dict.

     Its corresponding token type (the associated *key* in this dictionary) will be used as token type for the token.
   - A string representation of an existing token type without the `Token.` prefix (e.g. `String`, `Generic`, `Generic.EmphStrong`, `Text`, `Text.Multiline`).

[16] https://www.python.org/

If a corresponding token type is not found the lexer's behaviour depends on the lexer option `strict_tokentype` (see *Lexer Options* (page 5)):

If `True` (the default) the command yields a `pygments.token.Token.Generic.Error` token type for the given command's content.

If `False` then the Pygments[17] function `pygments.token.string_to_tokentype()` will be called. This function returns either an existing token type or synthesizes a new one on the fly. The associated highlighting with freshly created token types in the output may not be well defined.

For this argument escaping is neither needed nor supported.

2. The content of the second argument will given the token type of the first parameter.

   Standard *Escaping Rules* (page 12) apply to this argument!

Note: The command for explicit token types is **case-sensitive**.

Examples:

```
• \ttX{}{token}                  token      ▷ just a base "Token"

• \ttX{kc}{C}                    C          ▷ C as Keyword.Constant
• \ttX{Keyword.Constant}{C}      C          ▷ C as Keyword.Constant
• \ttX{ow}{∈}                    ∈          ▷ ∈ as Operator.Word
• \ttX{Operator.Word}{∈}         ∈          ▷ ∈ as Operator.Word
• \ttX{kc}{A Constant Keyword}   A Constant Keyword  ▷ An explicit Keyword.Constant
• \ttX{nv}{A Variable Name}      A Variable Name     ▷ An explicit Name.Variable
• \ttX{ni}{An Entity*Name}       An Entity*Name      ▷ An explicit Name.Entity
• \ttX{k}{∈ ∌}      ∈ ∌        ▷ ∈ and ∌ as (ordinary) Keywords
• \ttX{o}{∈ ∌}      ∈ ∌        ▷ ∈ and ∌ as (ordinary) Operators
  /*
   * The line below has ∈_∌ as (peculiar) function name.
   * Their params are automatic (i.e. a normal expression).
   */
• \ttX{nf}{∈_∌}(p1, p2)                ∈_∌(p1, p2)
• \ttX{Name.Function}{∈_∌}(p1, p2)     ∈_∌(p1, p2)
  /*
   * The line below has ∈_∌ as (peculiar) decorator name (as used in Python).
   * Their params are automatic (i.e. a normal expression).
   */
• \ttX{nd}{∈_∌}(p1, p2)                ∈_∌(p1, p2)
• \ttX{Name.Decorator}{∈_∌}(p1, p2)    ∈_∌(p1, p2)
  /*
   * Normal emphasis ("strong")
   */
• \ttX{gs}{this is strong}                      this is strong
• \ttX{Generic.Strong}{this is strong}          this is strong
  /*
   * A strong emphasis.
   */
• \ttX{ges}{A Strong Emphasis!}                 A Strong Emphasis!
• \ttX{Generic.EmphStrong}{A Strong Emphasis!}  A Strong Emphasis!
  /*
   * Escaping is allowed and needed for the closing brace!
   * The example token type is a "String".
   */
• \ttX{s}{Escaping brace \} and backslash \\!}   Escaping brace } and backslash \!
```

(continues on next page)

[17] https://pygments.org/

(continued from previous page)

```
  /*
   * This is a non-existing token type:
   * by default you get some generic error markup with a Generic.Error
   * token and no expansion.
   * See also `Lexer Options` and `strict_tokentype`.
   */
• \ttX{NON-EXISTING}{ϵ_∄}(p1, p2)     \ttX{NON-EXISTING}{ϵ_∄}(p1, p2)
```

An example with a lexer and `strict_tokentype=False` (highlighting obviously is like standard text with the templates used):

```
• \ttX{Generic.Not.Yet.Existing}{ϵ_∄}(p1, p2)     ϵ_∄(p1, p2)
```

Note: Explicit token types work in all *expression* and *text* contexts.

Note: Nesting of explicit token types is *not supported*.

2.2.10 Escaping Rules

The escape character is a backslash `\`.

A backslash can be escaped with `\\` and yields a single backslash token.

Within parameters a closing curly brace `}` ends the current parameters environment. It must be escaped using `\}` if a closing curly brace is part of the argument content.

A single backslash yields a `pygments.token.Token.Generic.Error` token when in *default* and *expression* states (and also in *Explicit Token Types* (page 10)). Contrary—in *text* contexts a single backslash character that does not introduce a command yields a normal text token.

In all contexts a backslash that would normally introduce a known command must be escaped if the content should not recognized as a command.

Single- and double-quotes must be escaped also (`\"` or `\'`) in *default* and *expression* contexts when they should not introduce a string token.

2.2.11 Customized Lexers in Sphinx

Defining lexers with non-default options in Sphinx[18] can be done in its configuration file `conf.py`.

The first option is to apply the Sphinx config value `highlight_options` properly. An existing lexer can be customized by options.

A more flexible alternative is to define a new lexer in the Sphinx application. The very same lexer class can be used with different options:

```
from functools import partial
from pygments_lexer_pseudocode2.lexers.algpseudocode import AlgPseudocodeLexer

def setup(app):

    #
    # Add a custom lexer: AlgPseudocodeLexer with custom init
    # option "no_end".
```

(continues on next page)

[18] https://www.sphinx-doc.org

(continued from previous page)

```
    #
    # In modern Sphinx versions given lexer must be callable and may
    # not be a lexer instance. So use an indirection with "partial"
    # here.
    #
    app.add_lexer("noend-algpseudocode",
                  partial(AlgPseudocodeLexer, no_end=True))
```

Note: Lexers in Sphinx are instantiated with the *raiseonerror* filter applied by default. This is also true for custom lexers that are added by `Sphinx.add_lexer()`. Using the *filters* option the user can associate custom filters with a lexer. These filters have precedence over the default *raiseonerror* filter.

Lexer *instances* that are added to `sphinx.highlighting.lexers` somehow are taken as is by Sphinx and are not augmented with any default filters.

See also chapter *Filters* (page 21).

For older Sphinx versions your mileage may vary.

2.2.12 Some Examples

Synthetic Example

The first example is a synthetic example with many features.

Its source code can be found at example-1.pseudocode.

```
PROGRAM The Pseudocode Lexer IS

  /*
   * The program is here, but it also could be an \ALGORITHM
   *
   * /* YES! Nested multi-line comments work! */
   */

  PROCEDURE A Procedure name IS
    ▪ A text block 1
    ▪ A text block 2 with a nested expression: flag is FALSE
    ◆
        ▷  A remark on its own line within a "BLOCK"

  END OF PROCEDURE A Procedure name

  FUNCTION A Function with {escaped} text IS
    (1234567890 ≠ what) or a and b xor (c in d)   ▷ this is another remark
    ◆ foo is not bar
    ◆ 2 + 6 = 1 (mod 7)

    IF a is nil THEN
      Set something
    ELSE IF
      Set some other thing
    END IF
  END OF FUNCTION A Function with {escaped} text

  CLASS A Class IS
    // This is a one-line comment
    a and b xor (c in d)      ▷ this is another remark
```

(continues on next page)

(continued from previous page)

```
    # This is another one-line comment

    (* Here is a "block" of expressions.
       A block has a leading symbol. *)
    ◆ foo ≟ bar
      Bär ≥ Maus
        and Bär ≠ Maus
    ◆ a 1.2 {x in X} c
    (* Analogous there is a variant that is in text-mode by default.
       It has an other leading symbol. *)
    ▪ We will compute next a xor b or (set X is Empty) ▷ without c!
      or multiply it the \ttx-ges/other/ way.

    ▪ foo bar     ▷ A remark within a text statement until LF!
      nextfoo nextbar
      nextnextfoo nextnextbar
  END OF CLASS A Class

  @A_XXX
  FOO BAR
END OF PROGRAM
```

The highlighted output with a customized *AlgPseudocodeLexer* and its *no_end* option set to `True`:

```
PROGRAM The Pseudocode Lexer IS

  /*
   * The program is here, but it also could be an \ALGORITHM
   *
   * /* YES! Nested multi-line comments work! */
   */

  PROCEDURE A Procedure name IS
    ▪ A text block 1
    ▪ A text block 2 with a nested expression: flag is FALSE
    ◆
        ▷  A remark on its own line within a "BLOCK"



  FUNCTION A Function with {escaped} text IS
    (1234567890 ≠ what) or a and b xor (c in d)   ▷ this is another remark
    ◆ foo is not bar
    ◆ 2 + 6 = 1 (mod 7)

    IF a is nil THEN
      Set something
    ELSE IF
      Set some other thing



  CLASS A Class IS
    // This is a one-line comment
    a and b xor (c in d)      ▷ this is another remark

    # This is another one-line comment

    (* Here is a "block" of expressions.
       A block has a leading symbol. *)
```

(continues on next page)

(continued from previous page)

```
    ◆ foo ≙ bar
      Bär ≥ Maus
        and Bär ≠ Maus
    ◆ a 1.2 {x in X} c
    (* Analogous there is a variant that is in text-mode by default.
       It has an other leading symbol. *)
    ▪ We will compute next a xor b or (set X is Empty) ▷ without c!
      or multiply it the \ttx-ges/other/ way.

    ▪ foo bar     ▷ A remark within a text statement until LF!
      nextfoo nextbar
      nextnextfoo nextnextbar


  @A_XXX
  FOO BAR
```

Dinic's Algorithm

The second example is Wikipedia's description of *Dinic's Algorithm* (see https://en.wikipedia.org/wiki/Dinic%27s_algorithm).

Its source code can be found at algorithm-dinic.pseudocode.

```
ALGORITHM Dinic's Algorithm WITH
  Input: A network G = ((V , E), c, s, t).
  Output: An s-t flow f of maximum value.
IS
  1. Set f(e) = 0 for each e ∈ E.
  2. Construct G_L from G_f of G.
     If dist(t) = ∞, stop and output f.
  3. Find a blocking flow f' in G_L.
  4. Augment flow f by f' and go back to step 2.
END OF ALGORITHM
```

Ford–Fulkerson Algorithm

The third example is Wikipedia's pseudocode of the *Ford–Fulkerson Algorithm* (see https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm).

Its source code can be found at algorithm-ford-fulkerson.pseudocode.

```
ALGORITHM Ford-Fulkerson WITH
  Inputs: Given a network G = (V, E) with flow capacity c, a source node s, and a␣
↪sink node t
  Output: Compute a flow f from s to t of maximum value
IS
  1. f(u, v) ← 0 for all edges (u, v)

  2. While there is a path p from s to t in G_f,
     such that c_f(u, v) > 0 for all edges (u, v) ∈ p:

     1. Find c_f(p) = min{c_f(u, v): (u, v) ∈ p}

     2. For each edge (u, v) ∈ p

        1. f(u, v) ← f(u, v) + c_f(p)    ▷ Send flow along the path
```

(continues on next page)

(continued from previous page)

```
        2. f(v, u) ← f(v, u) - c_f(p)    ▷ The flow might be ”returned” later

END OF ALGORITHM Ford-Fulkerson
```

Edmonds–Karp Algorithm

The fourth example is Wikipedia's pseudocode of the *Edmonds–Karp Algorithm* (see https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm) with a custom lexer which skips all `ENDxxx` keywords.

Its source code can be found at algorithm-edmonds-karp.pseudocode.

```
ALGORITHM Edmonds-Karp WITH
    Input:
      graph:  graph[v] should be the list of edges coming out of vertex v in the
              original graph and their corresponding constructed reverse edges
              which are used for push-back flow.
              Each edge should have a capacity cap, flow, source s and sink t
              as parameters, as well as a pointer to the reverse edge rev.
      s:      Source vertex
      t:      Sink vertex
    Output:
      flow:  Value of maximum flow)
IS
    flow := 0
    REPEAT
        /* Run a breadth-first search (bfs) to find the shortest s-t path.
           We use 'pred' to store the edge taken to get to each vertex,
           so we can recover the path afterwards. */
        q := queue()
        q.push(s)
        pred := array(graph.length)
        WHILE not empty(q) and pred[t] = null DO
            cur := q.pop()
            FOR Edge e in graph[cur] DO
                IF pred[e.t] = null and e.t ≠ s and e.cap > e.flow THEN
                    pred[e.t] := e
                    q.push(e.t)



        IF not (pred[t] = null) THEN
            /* We found an augmenting path.
               See how much flow we can send */
            df := ∞
            FOR (e := pred[t]; e ≠ null; e := pred[e.s]) DO
                df := min(df, e.cap - e.flow)

            /* And update edges by that amount */
            FOR (e := pred[t]; e ≠ null; e := pred[e.s]) DO
                e.flow  := e.flow + df
                e.rev.flow := e.rev.flow - df

            flow := flow + df

    UNTIL pred[t] = null      ▷ i.e., until no augmenting path was found
    RETURN flow
```

And now the *Edmonds–Karp Algorithm* with **french** keywords:

```
ALGORITHME Edmonds-Karp AVEC
    Input:
      graph:  graph[v] should be the list of edges coming out of vertex v in the
              original graph and their corresponding constructed reverse edges
              which are used for push-back flow.
              Each edge should have a capacity cap, flow, source s and sink t
              as parameters, as well as a pointer to the reverse edge rev.
      s:      Source vertex
      t:      Sink vertex
    Output:
      flow:  Value of maximum flow)
EST
    flow := 0
    RÉPÉTER
        /* Run a breadth-first search (bfs) to find the shortest s-t path.
           We use 'pred' to store the edge taken to get to each vertex,
           so we can recover the path afterwards. */
        q := queue()
        q.push(s)
        pred := array(graph.length)
        TANTQUE not empty(q) and pred[t] = null FAIRE
            cur := q.pop()
            POUR Edge e in graph[cur] FAIRE
                SI pred[e.t] = null and e.t ≠ s and e.cap > e.flow ALORS
                    pred[e.t] := e
                    q.push(e.t)
                END IF
            FIN POUR
        FIN TANTQUE
        SI not (pred[t] = null) ALORS
            /* We found an augmenting path.
               See how much flow we can send */
            df := ∞
            POUR (e := pred[t]; e ≠ null; e := pred[e.s]) FAIRE
                df := min(df, e.cap - e.flow)
            FIN POUR
            /* And update edges by that amount */
            POUR (e := pred[t]; e ≠ null; e := pred[e.s]) FAIRE
                e.flow  := e.flow + df
                e.rev.flow := e.rev.flow - df
            FIN POUR
            flow := flow + df
        END IF
    JUSQUACEQUE pred[t] = null      ▷ i.e., until no augmenting path was found
    RENVOYER flow
FIN D'ALGORITHME Edmonds-Karp
```

And again the *Edmonds–Karp Algorithm* with **german** keywords:

```
ALGORITHMUS Edmonds-Karp MIT
    Input:
      graph:  graph[v] should be the list of edges coming out of vertex v in the
              original graph and their corresponding constructed reverse edges
              which are used for push-back flow.
              Each edge should have a capacity cap, flow, source s and sink t
              as parameters, as well as a pointer to the reverse edge rev.
      s:      Source vertex
      t:      Sink vertex
    Output:
      flow:  Value of maximum flow)
IST
```

(continues on next page)

(continued from previous page)

```
    flow := 0
    WIEDERHOLE
        /* Run a breadth-first search (bfs) to find the shortest s-t path.
           We use 'pred' to store the edge taken to get to each vertex,
           so we can recover the path afterwards. */
        q := queue()
        q.push(s)
        pred := array(graph.length)
        SOLANGE not empty(q) and pred[t] = null DO
            cur := q.pop()
            FÜR Edge e in graph[cur] DO
                WENN pred[e.t] = null and e.t ≠ s and e.cap > e.flow DANN
                    pred[e.t] := e
                    q.push(e.t)
                ENDE WENN
            ENDE FÜR
        ENDE SOLANGE
        WENN not (pred[t] = null) DANN
            /* We found an augmenting path.
               See how much flow we can send */
            df := ∞
            FÜR (e := pred[t]; e ≠ null; e := pred[e.s]) DO
                df := min(df, e.cap - e.flow)
            ENDE FÜR
            /* And update edges by that amount */
            FÜR (e := pred[t]; e ≠ null; e := pred[e.s]) DO
                e.flow  := e.flow + df
                e.rev.flow := e.rev.flow - df
            ENDE FÜR
            flow := flow + df
        ENDE WENN
    BIS pred[t] = null      ▷ i.e., until no augmenting path was found
    RETURN flow
ENDE DES ALGORITHMUS Edmonds-Karp
```

2.3 FrPseudocode

This is the renamed pseudocode lexer from the original *pygments-lexer-pseudocode* package.

It has been changed somewhat:

- renamed from `Pseudocode` to `FrPseudocode`
- changed aliases to `fr-pseudocode`, `fr-pseudo`, `fr-algorithm` and `fr-algo`
- changed file extension to `.fr-algo` and `.fr-pseudocode`
- changed some exististing arrows and added some more
- numbers parsing is more flexible by following the rules of the Pygments[19] lexer for Python[20]
- also allow `!=` as inequality operator (in addition to `<>`)

It mostly just recognizes some (french) keywords and highlights them.

Comments are supported (`//` and `/* ... */` (single-line only)). "Directives" in "special" comments are to be enclosed in curly braces `{ ... }`.

It also implements some symbol replacements/conversions like `<=` to ≤, `>=` to ≥ or `<>` to ≠.

[19] https://pygments.org/
[20] https://www.python.org/

Example:

The following example

```
/* foo bar */

fonction fonc-1({passage par valeur}param1)
début
  si param1 <= 0 alors
    b = 0
  sinon
    b = 1
    a = param1
    répéter
      a = a - 1
      b = b * 2
    tantque a <> 0
  fin si
  retourner b
fin fonction
```

will be highlighted as

```
/* foo bar */

fonction fonc-1({passage par valeur}param1)
début
  si param1 ≤ 0 alors
    b = 0
  sinon
    b = 1
    a = param1
    répéter
      a = a - 1
      b = b * 2
    tantque a ≠ 0
  fin si
  retourner b
fin fonction
```

2.3.1 Lexer Options

There are no lexer options besides the Pygments[21] standard lexer options.

[21] https://pygments.org/

CHAPTER 3

Filters

The package contains the following filters:

Filter Name	Description	Filter Class
tokenreplace	A configurable token replacer: replace a given token type by another given token type in a stream	TokenReplaceFilter
errortogenericerror	Replace all `Error` tokens in a stream by `Generic.Error` tokens	ErrorToGenericErrorTokenFilter

The *AlgPseudocode lexer* (page 3) by default yields an error token for the code block below because it encounters an unknown command. When used within Sphinx[22] it warns about this and—as a consequence—suppresses highlighting for this code block completely (see also *this note* (page 13)):

```
\nonexisting{TEST}
```

This may be changed by using a custom AlgPseudocode lexer that has the `prohibit_raiseonerror_filter` lexer option enabled. Then the the output in Sphinx[23] is as follows:

```
\nonexisting{TEST}
```

Alternatively—with the “errortogenericerror” filter applied the very same block is highlighted as:

```
\nonexisting{TEST}
```

The above custom lexer is to be defined in Sphinx[24] using:

```
import functools
from pygments_lexer_pseudocode2.filters import ErrorToGenericErrorTokenFilter
from pygments_lexer_pseudocode2.lexers.algpseudocode import AlgPseudocodeLexer

def setup(app):
    app.add_lexer(
        ”genericerror-algpseudocode”,
        functools.partial(
            AlgPseudocodeLexer,
            filters=[ErrorToGenericErrorTokenFilter]))
```

[22] https://www.sphinx-doc.org
[23] https://www.sphinx-doc.org
[24] https://www.sphinx-doc.org

3.1 TokenReplaceFilter

Name
tokenreplace

Filter Options

`replacements`
Type: `dict[str | pygments.token._TokenType, str | pygments.token._TokenType]`

A map from tokens to their replacements.

`token_from`
Type: `str` or `pygments.token._TokenType`

The name of a token type (like `Error`) or a token object (like `pygments.token.Token.Error`).

If given the *token_to* options is required and *replacements* will be augmented with their respective values.

`token_to`
Type: `str` or `pygments.token._TokenType`

The name of a token type (like `Generic.Error`) or a token object (like `pygments.token.Token.Generic.Error`).

This option is required if *token_from* is given.

Replace all token types given as *replacements* keys or in *token_from* with the token types given in *replacements* values or in *token_to*.

The values in the token stream are retained.

All `str` types in any of the filter options are converted to real tokens using Pygments[25] function `pygments.token.string_to_tokentype()`.

3.2 ErrorToGenericErrorTokenFilter

Name
errortogenericerror

Filter Options
none

Replace all `pygments.token.Token.Error` tokens in a stream by `pygments.token.Token.Generic.Error` tokens.

The filter is implemented as an application of the *TokenReplaceFilter* (page 22).

[25] https://pygments.org/

CHAPTER 4

Changes

4.1 Version 3 Series

3.0 (2026-05-23)

First release after forking *pygments-lexer-pseudocode*.

[feature] Added the lexer AlgPseudocode (name “algpseudocode”) and its language variants “algpseudocode-fr” and “algpseudocode-de”

[feature] Added two useful filters:

- the generic “tokenreplace”
- “errortogenericerror”

[doc] Added documentation

[test] Implemented unit tests

4.2 Version 2 Series

No releases. Just forked from *pygments-lexer-pseudocode 2.0.1*.

4.3 Breaking Changes

None yet.

CHAPTER 5

Licenses

The project is licensed under the terms of the MIT License:

```
MIT License

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of␣
↪this software and
associated documentation files (the ”Software”), to deal in the Software without␣
↪restriction, including
without limitation the rights to use, copy, modify, merge, publish, distribute,␣
↪sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is furnished to␣
↪do so, subject to the
following conditions:

The above copyright notice and this permission notice shall be included in all␣
↪copies or substantial
portions of the Software.

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,␣
↪INCLUDING BUT NOT
LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND␣
↪NONINFRINGEMENT. IN NO
EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR␣
↪OTHER LIABILITY, WHETHER
IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION␣
↪WITH THE SOFTWARE OR THE
USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Some code snippets are licensed under the terms of the BSD 2-Clause “Simplified” License as they are taken from the Python lexer that is included in Pygments[26]:

```
BSD 2-Clause ”Simplified” License

Copyright <YEAR> <COPYRIGHT HOLDER>

```

(continues on next page)

[26] https://pygments.org/

(continued from previous page)

```
Redistribution and use in source and binary forms, with or without modification,␣
↪are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list␣
↪of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this␣
↪list of conditions and the following disclaimer in the documentation and/or other␣
↪materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY␣
↪EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED␣
↪WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.␣
↪IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,␣
↪INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT␣
↪NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR␣
↪PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,␣
↪WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)␣
↪ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE␣
↪POSSIBILITY OF SUCH DAMAGE.
```