CHOOSING THE RIGHT NAMES

• For feature and class names, use full words, not abbreviations, call number, not num.
• Do not hesitate to use several words connected by underscores, as in ANNUAL_RATE.
• For features, there is seldom a need for more than two or possibly three underscore-connected words.
• Do not include in a feature name the name of the underlying data abstraction (which should serve as the class name).
  o The feature giving the part number in class PART should be called just number, not part_number.
• Sometimes, every instance of a certain class contains a field representing an instance of another class. Although you should try to find a more specific name, you may, if this fails, just declare the feature as rate: RATE.
• Local entities and arguments of a routine only have a local scope, so they do not need to be as evocative.

move (i: INTEGER) is
-- Move cursor i positions, or after if i is too large.
  local
    c: CURSOR; counter: INTEGER; p: like FIRST_ELEMENT
  …
remove is
-- Remove current item; move cursor to right neighbor.
  local
    succ, pred, removed: like first_element
  …

• If succ and pred had been features they would have been called successor and predecessor.

Letter case
• Class names appear in all upper case: POINT, LINKED_LIST…
• Names of attributes, routines etc. appear in all lower case: balance, deposit, succ, i.

• Constant attributes have their first letter in upper case and the rest in lower lower case: Pi: INTEGER is 3.1415926524; Welcome_message: STRING is "Welcome!"

• A few reserved words are written with an initial upper case since they are similar to constants, they include Current, Result, Precursor, True and False.

Grammatical categories

• For class names, you should always use a noun, possibly qualified as in LONG_TERM_SAVINGS_ACCOUNT.

• Routine names should faithfully reflect the Command-Query separation principle:
  o Procedures (commands) should be verbs in the infinitive or imperative: make, move, deposit, set_color.
  o Attributes and functions (queries) should never be imperative or infinitive verbs; never call a query get_value, but just value.

• Non-boolean query names should be nouns, such as number.

• A frequent convention for boolean queries is the is_ form, as in is_empty.

HEADER COMMENTS AND INDEXING CLAUSES

Instead of the long comment in
tangent_from (p: POINT): LINE is
  -- Return the tangent line to the circle going through the point p,
  -- if the point is outside of the current circle.

  require
    outside_circle: not has (p)

  ...

just write
  -- Tangent from p.

because of the following reasons:

• The comment for a query, as here, should not start with “Return the…” or “Compute the…””. Simply name what the query returns, typically using a qualified noun.

• We can get rid of the auxiliary words, especially the, where they are not required for understandability.

• Another mistake is to have used the words line to refer to the result and point to refer to the argument: this information is immediately obvious from the declared types, LINE and POINT.

• Header comments for commands (procedures) should end with a period.
For boolean-valued queries, the comment should always be in the form of a question, terminated by a question mark:

\[\text{has } (v \colon G) \colon \text{BOOLEAN is} \]

-- Does \('v'\) appear in list?

• Software entities — attributes, arguments — appearing in comments in the source text should always appear between an opening quote (“backquote”) and a closing quote.

Because an exported attribute should be externally indistinguishable from argumentless functions — remember the Uniform Access principle — it should also have a comment:

\[\text{count} \colon \text{INTEGER} \]

-- Number of students in course

TEXT LAYOUT AND PRESENTATION

The textual layout of the notation follows a comb-like structure; the idea is that a syntactically meaningful part of a class, such as an instruction or an expression, should either:

• Fit on a line together with a preceding and succeeding operators.
• Be indented just by itself on one or more lines.

\[\text{if } c \text{ then } a \text{ else } b \text{ end} \]
or

\[\text{if} \]

\[c\]

\[\text{then} \]

\[a\]

\[\text{else} \]

\[b\]

\[\text{end} \]
or

\[\text{if } c \text{ then} \]

\[a\]

\[\text{else } b \text{ end} \]

Spaces

You will use a space:

• Before an opening parenthesis, but not after: \(f(x)\).
• After a closing parenthesis unless the next character is a period or semicolon; but not before. Hence: \(proc1(x); x := f1(x) + f2(y)\).
• After a comma but not before: \(g(x, y, z)\).

Spaces should appear before and after arithmetic operators, as in \(a + b\).
A layout example

indexing
description: "Example for formating"

class
EXAMPLE

inherit
MY_PARENT
redefine f1, f2 end

MY_OTHER_PARENT
rename
g1 as old_g1, g2 as old_g2
redefine
g1
select
g2
end

create
make

feature -- Initialization
make is

-- Do something.
require
some_condition: correct (x)

local
my_entity: MY_TYPE

do
if a then
b; c
else
other_routine
new_value := old_value / (max2 – max1)
end
end

feature -- Access
my_attribute: SOME_TYPE

-- Explanation of its role (aligned with comment for make)

... 
invariant
upper_bound: x <= y
end