SUPPER, a “modern” stropping regime for Algol 68

GNU68-2025-004 (draft)

by Jose E. Marchesi

Foreword

The following specification has been released under the auspices of the GNU Algol 68 Working Group, and has been scrutinized to ensure that

a. it is strictly upwards-compatible with Algol 68,
b. it is consistent with the philosophy and orthogonal framework of the language, and
c. it fills a clearly discernible gap in the expressive power of that language.

The source of this document can be found at `https://git.sr.ht/~jemarch/gnu68`.

The informal description of this proposal introduces the proposed new language features, providing a rationale and usage examples.

The formal definition of this proposal uses the existing formalism and conventions of the Standard Hardware Representation for Algol 68, and it is expressed as modifications to the Standard Hardware Representation.

Finally, the implementation notes of this proposal describes a way in which the features added by this specification can be implemented. No implementer should feel committed to do things as described there; the same language facilities may well be implementable in other ways, more suitable to specific implementations.

1 Informal Description

Representation languages and stropping

The Algol 68 algorithmic language establishes that certain source constructs, namely mode indications and operator indications, consist in a sequence of *bold letters* and *bold digits*, known as a *bold word*. In contrast, other constructs like identifiers, field selectors and labels are composed of regular or non-bold letters and digits, known as a *tag*.

What is precisely a bold letter or digit, and how it differs from a non-bold letter or digit, is not specified by the Report. This is no negligence, but a conscious effort at abstracting the definition of the so-called *strict language* from its representation. This allows having several different representations of the same language.

Some representations of Algol 68 are intended to be published in books, be it paper or electronic devices, and be consumed by persons. These are called *publication languages*. In publication languages bold letters and digits are typically represented by actual bold alphanumeric typographic marks, or sometimes underlined alphanumeric marks.

Other representations of Algol 68 are intended to be both produced and consumed by computers. These are called *hardware languages*, and would very likely use some compact binary representation in which the distinction between bold and regular letters and digits becomes irrelevant.

Finally, we have representations of Algol 68 that are intended to be primarily written by programmers and to be primarily processed by programs such as compilers, static analyzers or interpreters. These representations are called *programming languages*, and use some textual representation that is easy to read, edited and parsed, consisting in a stream of characters encoded in some character set.

Unfortunately, distinguishing a bold alphabet in programming languages is not easy, because computer systems today do not yet provide readily usable and ergonomic bold or underline alphanumeric marks in text files, despite the existence of Unicode and very fancy and sophisticated editing environments. The lack of appropriate input methods surely plays a role on this pitiful state of affairs. Thus, the programming representation languages of Algol 68 should resort to a technique known as *stropping* in order to differentiate bold letters and digits from non-bold letters and digits. A particular set of rules specifying the representation of these characters is known as a *stropping regime*.

There are three classical stropping regimes for Algol 68, which were standardized and specified long ago in the Standard Hardware Representation normative document. These are *POINT stropping*, *RES stropping* and *UPPER stropping*.

The following sections review these existing stropping regimes in a cursory way, to then introduce a new stropping regime that is the subject of this specification. For more details on the standard stropping regimes the reader is referred to the Standard Hardware Representation.

POINT stropping

POINT stropping is in a way the most fundamental of the three standard regimes. It was designed to work in installations with limited character sets that provided just one alphabet, usually printed in upper-case, a set of digits, and a very restricted set of other symbols.

```
.PROC RECSEL OUTPUT RECORDS = .VOID:
.BEGIN .BITS FLAGS
          := (INCLUDE DESCRIPTORS | REC F DESCRIPTOR | REC F NONE);
       .RECRSET RES = REC DB QUERY (DB, RECUTL TYPE,
                                    RECUTL QUICK, FLAGS);
       .RECWRITER WRITER := REC WRITER FILE NEW (STDOUT);

       SKIP COMMENTS .OF WRITER := .TRUE;
       .IF RECUTL PRINT SEXPS
       .THEN MODE .OF WRITER := REC WRITER SEXP .FI;
       REC WRITE (WRITER, RES)
.END
```

Figure 1.1: Example of code in POINT stropping

In POINT stropping a bold word is represented by its constituent letters and digits preceded by a point character. For example, the symbol `bold begin symbol` in the strict language, which is represented as **begin** in the reference language, would be represented as `.BEGIN` in POINT stropping.

More examples are summarized in the following table.

Strict language	**Reference language**	**POINT stropping**
`true symbol`	**true**	`.TRUE`
`false symbol`	**false**	`.FALSE`
`integral symbol`	**int**	`.INT`
`completion symbol`	**exit**	`.EXIT`
`bold-letter-c-...`	**crc32**	`.CRC32`

In POINT stropping a tag is represented by writing its constituent non-bold letters and digits in order. But they are organized in several *taggles*.

Each taggle is a sequence of one or more letters and digits, optionally followed by an underscore character. For example, the tag `PRINT` is composed of a single taggle, but the tag `PRINT_TABLE` is composed of a first taggle `PRINT_` followed by a second taggle `TABLE`.

To improve readability it is possible to insert zero or more white space characters between the taggles in a tag. Therefore, the tag `PRINT_TABLE` could have been written `PRINT TABLE`, or even `PRINT_ TABLE`. This is the reason why Algol 68 identifiers, labels and field selectors can and do usually feature white spaces in them.

It is important to note that both the trailing underscore characters in taggles and the white spaces in a tag do not contribute anything to the denoted tag: these are just stropping artifacts aimed to improve readability. Therefore `FOOBAR` `FOO BAR`, `FOO_BAR` and `FOO_BAR_` are all representations of the same tag, that represents the `letter-f-letter-o-letter-o-letter-b-letter-a-letter-r` language construct.

See Figure 1.1 for an example of an Algol 68 procedure encoded in POINT stropping.

RES stropping

The early installations where Algol 68 ran not only featured a very restricted character set, but also suffered from limited storage and complex to use and time consuming input methods such as card punchers and readers. It was important for the representation of programs to be as compact as possible.

The RES stropping regime was very likely introduced due to that reason. As its name implies, it reduces the number of bold words that require being stropped by introducing *reserved words*, which are the the bold words specified in the section 9.4.1 of the Report as a representation of certain symbols, such as **at**, **begin**, **if**, **int** and many others.

```
PROC RECSEL OUTPUT RECORDS = VOID:
BEGIN BITS FLAGS
         := (INCLUDE DESCRIPTORS | REC F DESCRIPTOR | REC F NONE);
      .RECRSET RES = REC DB QUERY (DB, RECUTL TYPE,
                                   RECUTL QUICK, FLAGS);
      .RECWRITER WRITER := REC WRITER FILE NEW (STDOUT);

      SKIP COMMENTS OF WRITER := TRUE;
      IF RECUTL PRINT SEXPS
      THEN MODE .OF WRITER := REC WRITER SEXP FI;
      REC WRITE (WRITER, RES)
END
```

Figure 1.2: Example of code in RES stropping

RES stropping encodes bold words and tags like POINT stropping, but if a bold word is a reserved word then it can then be written without a preceding point, achieving this way a more compact, and easier to read, representation for programs.

Introducing reserved words has the obvious disadvantage that some tags cannot be written the obvious way due to the possibility of conflicts. For example, to represent a tag `if` it is not possible to just write `IF`, because it conflicts with a reserved word, but this can be overcome easily (if not very elegantly) by writing `IF_` instead.

See Figure 1.2 for an example of an Algol 68 procedure encoded in RES stropping.

Note how user-defined mode indications an operator indications still require explicit stropping.

UPPER stropping

At some point computers added support for more than one alphabet by introducing character sets with both upper and lower case letters, along with convenient ways to both input and display these, namely a shift key and proper terminals.

```
PROC recsel output records = VOID:
BEGIN BITS flags
         := (include descriptors | rec f descriptor | rec f none);
      RECRSET res = rec db query (db, recutl type,
                                  recutl quick, flags);
      RECWRITER writer := rec writer file new (stdout);

      skip comments of writer := TRUE;
      IF recutl print sexps
      THEN mode OF writer := rec writer sexp FI;
      rec write (writer, res)
END
```

Figure 1.3: Example of code in UPPER stropping

In UPPER stropping the letters in bold word are represented by upper-case letters, whereas the letters in tags are represented by lower-case letters.

The notions of upper- and lower-case are obviously not applicable to digits, but since the language syntax assures that it is not possible to have a bold word that starts with a digit, digits are considered to be bold by convention if they follow a bold letter or another bold digit.

See Figure 1.3 for an example of an Algol 68 procedure encoded in UPPER stropping.

Note how in this regime it is almost never necessary to introduce bold tags with points. All in all, it looks much more natural to contemporary readers. UPPER stropping is in fact the stropping regime of choice today. It is difficult to think of any reason why anyone would resort to use POINT or RES stropping nowadays.

Bold taggles

In all three classical stropping regimes it is not possible to write white space characters between the constituent letters and digits of a bold word. It is very common, however, for user-defined mode indications and operator indications to contain several natural words, such as in `TREENODE` or `RECWRITER`. This can be a little difficult to read.

The GNU extension GNU68-2025-002, “Bold taggles in Algol 68”, adds support in all the standard stropping regimes to use underscores in bold words. This is done by redefining bold words to be based on taggles, much like tags.

```
PROC recsel output records = VOID:
BEGIN BITS flags
         := (include descriptors | rec f descriptor | rec f none);
      REC_RSET res = rec db query (db, recutl type,
                                   recutl quick, flags);
      REC_WRITER writer := rec writer file new (stdout);

      skip comments of writer := TRUE;
      IF recutl print sexps
      THEN mode OF writer := rec writer sexp FI;
      rec write (writer, res)
END
```

Figure 1.4: Example of code in UPPER stropping with bold taggles

With this extension, the above mode indications could have been written like `TREE_NODE` an `REC_WRITER`, improving readability.

See Figure 1.4 for an example of an Algol 68 procedure encoded in UPPER stropping with bold taggles.

SUPPER stropping

This proposal describes a new stropping regime that combines the advantages of the RES and the UPPER regimes. The resulting representation of programs aims to be both more appealing to contemporary programmers and also more convenient to be used in today's computing systems.

On one hand, the RES stropping regime made it possible to avoid explicit stropping of a big subset of all the bold words, namely the ones pertaining to the fixed set of "reserved words". However, both user-defined mode indications and operator indications still had to be stropped explicitly, prefixing them with a dot character.

On the other hand, the UPPER stropping regime, cleverly exploiting dual-alphabet installations, implemented the explicit stropping by having bold words encoded using upper-case letters and tags encoded using lower-case letters.

Combining and adapting both approaches we can obtain a stropping regime in which explicit stropping is reduced to the minimum necessary, *i.e.* user-defined mode and operator indications, and in which explicit stropping is done in a way that looks more familiar to today's programmers and less heavy on upper-case letters than in UPPER stropping.

```
proc recsel_output_records = void:
begin bits flags
         := (include_descriptors | rec_f_descriptor | rec_f_none);
      RecRset res = rec_db_query (db, recutl_type,
                                  recutl_uick, flags);
      RecWriter writer := rec_writer_file_new (stdout);

      skip_comments of writer := true;
      if recutl_print_sexps
      then mode_ of writer := rec_writer_sexp fi;
      rec_write (writer, res)
end
```

Figure 1.5: Example of code in SUPPER stropping

In the SUPPER stropping regime bold words are written by writing a sequence of one or more *taggles*. Each taggle is written by writing a letter followed by zero or more other letters and digits and is optionally followed by a trailing underscore character. The first letter in a bold word shall be an upper-case letter. The rest of the letters in the bold word may be either upper- or lower-case.

For example, `RecRset`, `Rec_Rset` and `RECRset` are all different ways to represent the same mode indication. This allows to recreate popular naming conventions such as `CamelCase`.

As in the other stropping regimes, the casing of the letters and the underscore characters are not really part of the mode or operator indication.

Operator indications are also bold words and are written in exactly the same way than mode indications, but it is usually better to always use upper-case letters in operator indications. On one side, it looks better, especially in the case of dyadic operators where the asymmetry of, for example `Equal` would look odd, consider `m1 Equal m2` as opposed to `m1 EQUAL m2`. On the other side, tools like editors can make use of this convention in order to highlight operator indications differently than mode indications.

In the SUPPER stropping regime tags are written by writing a sequence of one or more *taggles*. Each taggle is written by writing a letter followed by zero or more other letters and digits and is optionally followed by a trailing underscore character. All letters in a tag shall be lower-case letters.

For example, the identifier `list` is represented by a single taggle, and it is composed by the letters `l`, `i`, `s` and `t`, in order. In the jargon of the strict language we would spell the tag as `letter-l-letter-i-letter-s-letter-t`.

The label `found_zero` is represented by two taggles, `found_` and `zero`, and it is composed by the letters `f`, `o`, `u`, `n`, `d`, `z`, `e`, `r` and `o`, in order. In the jargon of the strict language we would spell the tag as `letter-f-letter-o-letter-u-letter-n -letter-d-letter-z-letter-e-letter-r-letter-o`.

The identifier `crc_32` is likewise represented by two taggles, `crc_` and `32`. Note how the second taggle contains only digits. In the jargon of the strict language we would spell the tag as `letter-c-letter-r-letter-c-digit-three-digit-two`.

The underscore characters are not really part of the tag, but part of the stropping. For example, both `goto found_zero` and `goto foundzero` jump to the same label.

See Figure 1.5 for an example of an Algol 68 procedure encoded in SUPPER stropping.

No white spaces in identifiers

SUPPER is the only Algol 68 stropping regime that doesn't allow having typographical display features (spaces, tabs and newline characters) between the taggles conforming a tag. In other words, it is not allowed to have white spaces as part of identifiers.

This is a shame, but there are two main reasons why it was decided to proceed like this.

First and most importantly, the SUPPER stropping regime is based on reserved words, which are mapped from the representation specified for the symbols of the language that are represented by bold words in the reference language.

The reference language of Algol 68 was designed before the Standard Hardware Representation introduced the RES stropping regime. Likely this is the reason why it specifies so many and so short symbols: the authors assumed that tags would always live in a different name space than bold words.

With reserved words like `to` and `in`, it becomes very difficult to separate identifier taggles with white spaces without bumping into conflicts. Consider for example the not at all unlikely procedure name `checked real to int`. This identifier would be not legal, because the three last taggles conflict with the the standard mode `real`, the syntactic bold word `to` and the standard mode `int`, respectively.

Such conflicts could be avoided by making the offending taggles to be adjacent to underscores, but then who would want to write identifiers like `checked real_ to_ int_` or `checked real_to_ int`? It is much simpler to just mandate for such tags to be written as `checked_real_to_int`. This avoids any possibility of conflict since all the taggles are adjacent to an underscore and therefore cannot collide with reserved words. Also, the risk of programs breaking in the future due to new reserved words getting added to the language gets dramatically reduced.

The second reason is that the notion of allowing blanks to be freely interjected in user-defined identifiers is alien to most if not all programming languages widely used today. It is, unfortunately, an eccentricity, even if a beautiful one, that makes it difficult to leverage existing tools such as editors, code indentation engines, and basically any program that makes assumptions on the general form of programs. For example, convincing a programming editor or IDE that a single identifier may span for more than one logical line in the source file may prove quite challenging and frustrating.

Conformance

This proposal conforms to the requirements specified in the appendix B.3 of the Standard Hardware Representation:

```
B.3  Other Stropping Regimes.

  For compatibility with existing installation practice,
implementations may implement stropping regimes in addition to those
provided by the standard.  However, such additional regimes should be
invoked by pragmat-items distinct from those in *3.5.  All
modifications to the defined regimes -- including extensions -- should
be avoided because they would inhibit error detection and decrease
portability.
```

2 Formal Description

This extension adds a new stropping regime to the Standard Hardware Representation, so a new pragmat-item shall be invented.

```
3.5

... of which there are four.  A new regime is invoked by a pragmat
containing one of the pragmat-items POINT, UPPER, RES or SUPPER, and
takes effect following the closing pragmat symbol.
```

The new stropping regime is then described in its own section, as follows.

```
3.5.4 SUPPER Stropping

Bold words.

- A bold word is written as a sequence of {one or more} taggles
  whose worthy letters and digits correspond, in order, to the bold
  faced letters and digits in the word.

- A bold word must start with an upper-case letter.  Upper- and
  lower-case letters may be otherwise intermixed in a bold word.

- Upper-case letters may be written only in bold words and
  character-glyphs.

Tags.

- A tag is written as a sequence of {one or more} taggles.  The
  taggles cannot be separated by typographical display features.

- A taggle is written by writing, in order, the corresponding
  worthy letters and digits optionally followed by an underscore.

- A taggle must be adjacent to an underscore if its letters and
  digits correspond, in order, to those of a reserved word.

- If a taggle does not end with an underscore, it must be followed
  by a disjunctor.

{Examples:

     Program: PR SUPPER PR begin real x; x := x - 1 end
     Bold: begin, AMode, AmOdE, A_Mode, a_mode, AMODE, Oper, OPER
     Plain: begin_, end_of_file, end_of_file_, xl,
            x_l,
     Error: end of file, end____of__file,
            end_of_file__ }
```

3 Implementation Notes

Stropping is to be implemented purely at the lexical level and it must not have any impact on the letters and digits that constitute bold words and tags. Therefore, in principle, of all the components of a compiler or interpreter only the lexical analyzer (or tokenizer) shall concern itself with the stropping regime.