ma
the
ma
tisch

cen
trum

AFDELING INFORMATICA IW 8/73 JUNE

T.J. DEKKER and D. GRUNE

# PROPOSALS FOR THE REPRESENTATION OF ALGOL 68 PROGRAMS

amsterdam 1973

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 8/73 JUNE

T.J. DEKKER and D. GRUNE

# PROPOSALS FOR THE REPRESENTATION OF ALGOL 68 PROGRAMS

2e boerhaavestraat 49 amsterdam

*Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.*

*The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.*

ACM - Computing Reviews - category: 4.22

# Proposals for the representation of ALGOL 68 programs.

June 8th, 1973.

Theodorus J. Dekker,
University of Amsterdam.

Dick Grune,
Mathematical Centre, Amsterdam.

Abstract

Criteria are given for "acceptable" representations of programs. The resulting requirements are applied to ALGOL 68. It appears that they can be satisfied by very superficial changes in ALGOL 68, mainly in "string item".

## 1. Motives.

The essential part of a computer language is its semantics. It is less fundamental how this semantics is controlled (through a syntax) as it is less fundamental how this control is represented (through a hardware representation). Syntax and representation have much in common:

a. In both fields decisions are essentially arbitrary: there is no scientific proof that the form "i:= 3" is better suited for an assignation than "set i to 3 tes" or "3=: i". In practice it is impossible to prove that someone is wrong.
b. In both fields ambiguities lurk everywhere (and are patched up after discovery).
c. Different people tend to reach very different solutions.

In both fields standardization has a healthy influence:

a. Implementers no longer have to spend their time on taking arbitrary decisions.
b. Since there is only one scheme on which more people can concentrate, we have a better chance of eventually removing all ambiguities.
c. It greatly eases the education of programmers and the exchange of programs.

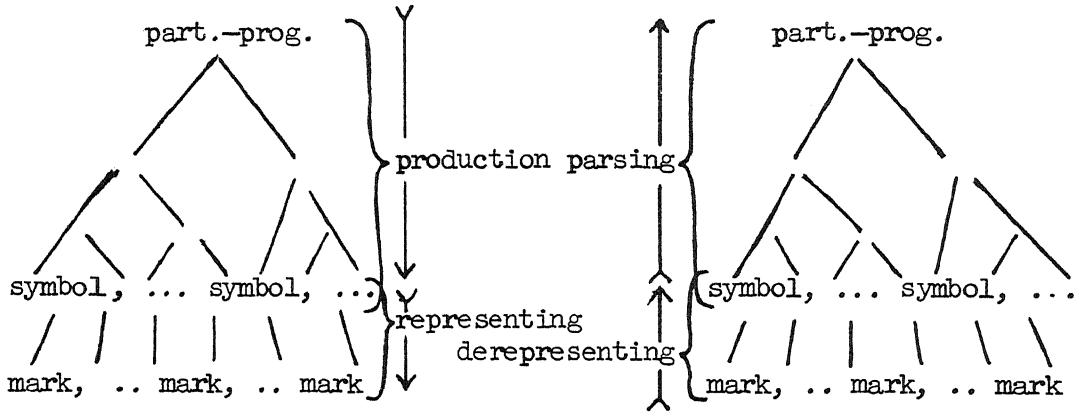
The syntax of ALGOL 68 is well standardized. The standardization of representations, however, leaves much to be desired: consequently in that field chaos rages.

We therefore submit this (annotated) proposal for standardization of representations. The text is of a mixed nature: proposals for changes are indicated in the margin by a solid line, proposals for suggestions are indicated by a dotted line.

## 2. Considerations.

First we must get our terminology straight. Through the process of "production" we obtain from 'particular-program' a sequence of 'NOTION-symbol's to be called "symbols"s. Through the process of "representing" we obtain from this sequence of symbols a sequence of (generally)

readable, printable, punchable hardware phenomena, to be called "mark"s. Through the process of "derepresenting" (lexical analysis) we can reconstruct from the sequence of marks the sequence of symbols. Through the process of "parsing" we can reconstruct from the sequence of symbols the 'particular-program'.




We now require the (de)representing mechanism to be independent of the production/parsing part. This requirement has several advantages. It removes sources of confusion ("for the packing of bold tags we have to know if we are in a string, for finding out if we are in a string we must be able to recognize comments, for the recognition of comments we must be able to pack bold tags, etc."); it gives a clear division of responsibilities; it forces us to define a clean interface on the level of symbols; and, last but not least, it allows us to formulate our next requirement: The (de)representing must be as simple as possible. This will make our third requirement easy to fulfil: straightforward convertibility of programs.

It should be pointed out here that in a classic compiler design such a strict separation of derepresenting and parsing is necessary. The derepresenting mechanism cannot rely on syntactic information from the parser since it is generally one or more symbols ahead of the parser, nor can it provide its own syntactic information since there is no guarantee that parser and derepresenter will always agree on it. Consider, for example, the input ; y:= ar"tan(x); . Most parsers will be clever enough to discard the " whereas the derepresenting routine will not.

## 3. The symbol level.

3.1. Experience has shown that as little significance as possible should be attached to layout; as, however, experience has also shown that some codes do need layout on the level of marks, the symbol level is the lowest level on which layout can be declared meaningless. We therefore require: Layout between symbols has no meaning.
The requirement that (de)representing be simple entails that symbols should only be composite if strictly necessary. Since from the syntax of ALGOL 68 it follows that a bold tag can follow a bold tag but a tag cannot

follow a tag, a bold tag must be a symbol since its decomposition into bold-LETTER-symbols would lead to ambiguities; but a tag need not be a symbol and can safely be decomposed into LETTER-symbols. As an automatic consequence, layout is allowed inside tags but not necessarily inside bold tags.

3.2. Decomposing every item as far as possible has another advantage: it uncovers facts about the language that are now concealed by the syntax. It is not at al clear why a language that has a 'becomes-symbol', a 'colon-symbol' and an 'is-symbol' should prevent the sequence 'becomes-symbol', 'colon-symbol'. However, if the language has:

becomes composite: colon symbol, equals symbol.
is composite: colon symbol, equals symbol, colon symbol.

then 'becomes-composite', 'colon-symbol' looks suspicious. Anyhow, a syntax of this form must already be in use with many people that keep an eye on possible ambiguities.

3.3. At present there are two kinds of bold-TAGs, those for which there is a representation, like tag; and those for which there is not, like begin. A simple representing mechanism cannot be expected to tell the (rather unnatural) difference. The responsibility for preventing for example the definition of an operator begin lies clearly with the "production part" of the language.

3.4. We therefore propose:

a. to change the rule for NOTION-token to

NOTION token: pragment sequence option, NOTION composite.

and then amend 9.4. along the following lines:

for composites whose representation consists of only one mark, e.g., letters:

LETTER composite: LETTER symbol.

for composite for which there are more (non-bold) representations, e.g., "becomes":

becomes composite: {becomes symbol { ≔ {if anybody has it}};}
point symbol, equals symbol;
colon composite, equals symbol.

for composites for which there is a bold representation, e.g., "of":

of composite: of symbol {→} ;
bold letter o letter f symbol.

Such an approach would also pacify those that are now claiming that they cannot see the difference between a 'label-symbol' and a 'colon-symbol'.

b. to replace TAG-symbol in 5.5.1. and in 4.6.1. by TAG and to define TAG analogous to the definition in the Old Report.

c. to rephrase:

TAB:: bold NONRESERVED symbol; SIZETY STANDARD.
TAD:: bold NONRESERVED symbol; DYAD ... etc.
TAM:: bold NONRESERVED symbol; MONAD ... etc.

"A metaproduction rule must be added for the metanotion 'NONRESERVED' whose alternatives are of the form 'TAG' such that the notion 'bold-TAG-symbol' is not produced anywhere else in this Report."

d. as a matter of clean terminology to rename 'bold-begin(end,comment, pragmat)-symbol' to 'word-begin(end,comment,pragmat)-symbol', and to add rules like:

word begin symbol: bold letter b letter e letter g letter i letter n symbol.

(or some abbreviation through a convention).

3.5. Other notions giving rise to symbols are 'other-string-item', 'MATCH-other-PRAGMENT-item' and 'letter-OTHERALPHA'. These must now be considered in more detail.

3.6. Concerning 'other-string-item'.

Prime consideration:
Given an environment and a character code known in that environment, an ALGOL 68 program must be able to produce any character output that can normally be produced by and accepted by that environment. That is, it must be able to write input for itself and for other programs and must be as good as normal punching equipment.

3.6.1. The only way of specifying character output (explicitly or implicitly) is through 'row-of-character' and an important instrument in specifying 'row-of-character' is the 'string-item'. Any "character" should therefore correspond to an easy-to-write 'string-item'.

It is very attractive to require a one-to-one correspondence between the punched marks representing the 'string-item's and the "character"s in a 'row-of-character'. This requirement, however, breaks down on three occasions:

a. "character"s without corresponding input punchings,
b. the 'space-symbol',
c. bold-TAGs.

Ad a.
The mode char defines a set of values. To each such value there corresponds an integral value n such that $0 \leq n \leq$ max abs char. We propose that conversely to each integral value $0 \leq n \leq$ max abs char there corresponds a character value (to which there will not necessarily correspond a transputtable mark). (This "character space" might be shared by different

codes in one compiler). The set of values of char is often larger than the set of available punchings. It is, however, awkward to have a set of values without denotations for every value. So there must be a way to denote any value of char by its corresponding integral value. The only way of doing this is by writing down the 'integral-denotation' yielding that integral value. However, this 'integral-denotation' must be marked as such: there must be a special symbol to do this. We shall call this symbol the 'exception-symbol'. The end of the 'integral-denotation' must be demarcated (since 'digit-symbol' may follow 'string-item' but it may not follow 'integral-denotation'): the obvious choice is PACKing it. But then we can allow a list of 'integral-denotation's as well. There is no reason to exclude comment from this item, on the contrary, comment may be very helpful for elucidating the meanings of the 'integral-denotation's.
The above suggests the following rule (to be added in "String denotations"):

general item: exception symbol, integral denotation list PACK.

It may be objected that this is a machine-dependent feature, but so is the use of repr and abs and the correspondence between "character"s and "integer"s.

Ad b.
Since the 'space-symbol' does not denote itself but the 'space-layout-mark' instead, there must be a way to specify the 'space-symbol' itself. In view of the above the solution is obvious: a 'space-symbol' preceeded by an 'exception-symbol' denotes the 'space-symbol' itself. However, the (necessary) task of assigning a mark to the 'space-symbol' is embarrassingly difficult in (almost) all codes: a multi-mark is unacceptable for practical use, and nobody has a bold point.
The requirement that layout between symbols be meaningless prevents us from using the 'space-layout-mark' as 'space-symbol' (which would mean denying the usefulness of a 'space-symbol' at all). The problem would be solved by a more appropriate representation for the 'space-symbol' (or in our new terminology, a better production rule for 'space-composite'). We propose:

space composite {produced by 'string-item'} :
    point symbol.
    {a 'space-composite' denotes the "character" that
     corresponds to the 'space-layout-mark'.}

string point composite {produced by 'string-item'} :
    exception symbol, point symbol.
    {a 'string-point-composite' denotes the "character" that
     corresponds to the 'point-mark'.}

point composite {produced by 'fractional-part' etc.} :
    point symbol.

The same technique can be used to denote the 'exception-mark':

```
exception composite {produced by 'string-item'} :
        exception symbol, comma symbol.
        {an 'exception-composite' denotes the "character" that
         corresponds to the 'exception-mark'.}
```

Note 1.
From the above syntax it follows that 'exception-symbol' will never be followed by a 'LETTER-symbol'. This fact is used in 4.1..

Note 2.
It must be emphasized that the above has nothing to do with a bolding-convention: the 'exception-symbol' and its producing rules belong to the "production part" and are of no concern to the representing mechanism. All codes will need an 'exception-mark', corresponding to the 'exception-symbol'. That this 'exception-mark' can also be conveniently used in a tag-bolding convention, is due to, let us say, a fortunate coincidence.

Note 3.
Once having created the 'exception-symbol', we might consider what more services it could render. It does not occur yet outside 'string-denotation's, so its occurrence outside 'string-denotation's (and of course 'character-denotation's) is unambiguous. Furthermore we want to maintain the rule that an 'exception-symbol' is never followed by a 'LETTER-symbol'. Consequently, the candidates are ₁₀ ) ] ∿ and ° . The ) and ] are pointless in this respect, more representations for 'skip' and 'nil' are not really useful, so the ₁₀ remains. Since the ₁₀ is a sore point in many codes anyhow, it will be useful to define:

```
times ten to the power composite:
        letter e symbol;
        times ten to the power symbol;
        exception symbol.
```

(which would come in place of the 'times-ten-to-the-power-choice').

Ad c.
There is no reason to categorically forbid bold-TAGs in strings. On the contrary, if one did, an implementation that uses capitals for bolding would be hard put to print

```
REF REAL xx;
```

from a string. One could argue that if bold letters are present in the code they should be 'other-string-item's in their own right: this, however, would make a strict separation between derepresenting and parsing impossible. Even the requirement that in a string such letters should be separated by layout, (thus: "R E F.R E A L.xx;") would not help since the single R would still be a bold-TAG, not to speak of the nuisance value of such a feature and its unteachability. So we propose to

allow 'bold-TAG's as 'string-item's with the semantics:

A bold-TAG-symbol denotes
- if the set of "character"s contains a bold alphabet and bold digits:
  the sequence of bold characters that constitute that TAG
- {otherwise undefined, e.g., a sequence of, in some suitable way corresponding, characters}.

3.6.2. We now reach the following definitions of 'string-item' and 'character-glyph':

string item:
character glyph; quote composite; bold TAG symbol; space composite; exception composite; general item; string point composite {instead of point symbol}; other string item.
character glyph:
LETTER symbol; DIGIT symbol; open symbol; close symbol; comma symbol; plus symbol; minus symbol.

'Other-string-item' produces all symbols that correspond to marks in the given code that are not 'quote-symbol', 'point-symbol' or 'exception-symbol' and are not produced by 'character-glyph'.
The occurrence of 'times-ten-to-the-power-symbol' and 'plus-i-times-symbol' in 'character-glyph' would make them "required" (see 4.3.) which is very undesirable regarding small character sets. They should come in through 'other-string-item'.

3.7. Concerning 'MATCH-other-PRAGMENT-item'.
The present definition causes no problems. Nevertheless, it might be useful to give users and implementers some leeway by allowing here an 'incorrect-symbol' that would only be produced by 'MATCH-other-PRAGMENT-item', and that during derepresenting would originate from any mark or sequence of marks that cannot be properly derepresented (like violation of the bolding convention or parity error).

3.8. Concerning 'letter-OTHERALPHA'.
Since one of our aims is the easy convertibility of programs from one hardware representation to another it is not advisable that OTHERALPHA should produce other alphabets. Preferably it should not produce anything at all.

4. The (de)representing mechanism.
The representing mechanism is completely dependent on the given code. This means that it is not possible to define these mechanisms in the Report (but the above has laid a basis for their structure). Consequently our only hope to curb the chaos lies in supplying guidelines for the construction of such mechanisms.

The representing results in a sequence of marks which are elements of a set of marks (called the "code"). This code is divided into two subsets, one containing the marks that correspond to (perceptible) prints (called "non-layout-mark"s) and one containing marks that control the positioning of non-layout-marks or are imperceptible (called "layout-mark"s). Generally a non-layout-mark occupies one position; some codes, however, contain marks that do not occupy a position (either directly or through a trick), like non-shift underline, non-shift umlaut (called "diacritical-mark"s). In order to avoid problems over the subtle difference between for example a single-underlined letter and a double-underlined letter, a mark together with its diacritical-marks must be considered as one mark.

4.1. The main task in designing a representation mechanism is to establish a convention by which bold-TAG-symbols are made recognizable. Since from changes proposed above it follows that the 'exception-symbol' can never be followed by a 'LETTER-symbol', the 'exception-mark' can render good services.

4.2. The second task is to decide on the admissibility of layout-marks. For this purpose the layout-marks in the given code are split into two groups: 'significant-layout-mark's, including the 'space-layout-mark' (generally known as "blank") and possibly others; and 'dummy-layout-mark's, including all those marks that are generated or discarded by the system beyond programmers control (like 'stopcode', 'end-of-card', 'end-of-record', 'blank-tape', 'ring-bell', etc.) and possibly others.
Significant-layout-marks are allowed between symbols, dummy-layout-marks are allowed everywhere.

4.3. The third task is to set up a correspondence between non-bold symbols and non-layout marks. It is highly advisable that this be a one-symbol-one-mark correspondence: any attempt to use multi-marks for one symbol opens up abysses of ambiguities (if not today then in three months). Moreover, one mark cannot be used for more than one symbol: a simple derepresenting would be impossible.
It is, however, perfectly admissible to have more than one mark correspond to one same symbol. In fact this situation may arise in codes for which subsets are defined. For example, in the ASCII64 set the exclamation-mark will probably be used for the stick-symbol but in the ASCII96 set the interrupted-bar-mark is a better candidate. In order to preserve the subset character of ASCII64, both the exclamation-mark and the interrupted-bar-mark should correspond to the stick-symbol in the ASCII96 set.

For the purpose of establishing the desired correspondence, the non-bold symbol-set of the reference language in the Report is divided into two subsets: those symbols for which a corresponding mark is required, in the sense that without these marks it is not conveniently possible to write ALGOL 68 programs (those symbols will be called "indispensable" symbols); and those symbols that are defined but not "indispensable" in the sense of the above (called "dispensable" symbols). For reasons of terminology the symbol-set is also divided into "operator"s (i.e., DYAD-symbols), and "syntactic"s (the rest).

4.3.1. Those that are "indispensable" must be supplied by any representing mechanism. It concerns the following symbols:

syntactics:
- LETTER-symbol
- DIGIT-symbol
- exception-symbol
- open-symbol
- close-symbol
- point-symbol
- comma-symbol
- quote-symbol
- formatter-symbol

operators:
- plus-symbol
- minus-symbol
- times-symbol or asterisk-symbol
- divided-by-symbol
- equals-symbol.

Unfortunately there are 48 of these, one too many for a 48-character set (which has 47 non-layout marks). If we want to cater for 48-character sets we shall have to supply a bold-TAG alternative for at least one of those symbols. Letters, digits, arithmetic operators, parentheses, point, comma and exception do not lend themselves for such an alternative. This leaves us the 'quote-symbol' and the 'formatter-symbol'. A bold-TAG as alternative for the 'quote-symbol' would yield an awkward 'quote-composite' (was 'quote-image'). We therefore propose that the bold-TAG fo be reinstalled for formatter-composite.

4.3.2. At present the following symbols are "dispensable" (represented here by their reference representations):

syntactics:
[ ] : ; @ | ¹⁰ \ ° ¢ # $ →

operators:
* or × ↑ % < > ≠ ↑ ⊥ ↓ & ∧ ∨ ¬ ~

⌈ ⌊ ▯

4.3.3. A recommendable strategy for establishing the required correspondence is:

A. Those non-layout marks in the code that "sufficiently resemble" the reference representation of a given symbol correspond to that symbol; the interpretation of "sufficiently resembling" must be such that
   a. there is a corresponding mark for each "indispensable" symbol,
   b. no mark corresponds to more than one symbol.
B. The remaining non-layout marks are made to correspond to terminal productions of 'other-string-item' and as far as possible to 'OTHERMONAD-symbol's and to terminal productions of 'MATCH-other-PRAGMENT-item'.

If subsets are defined for the code, then the above recipe should be applied to the smallest subset(s) first, and then to other (sub)set(s) in such an order as to preserve the proper subset character. This process may cause more than one mark to correspond to one same (syntactic) symbol (not so for operator symbols).

4.3.4. The above implies that every syntactic symbol in the reference language potentially blocks a possibly valuable operator-mark. To restrict the damage it is useful to reconsider the necessity of some syntactic symbols, especially of those that have an equal or very similar function. There are two such cases: the $_{10}$ and the \ ; and the ¢ and the #. We propose to drop the \ and the ¢, the \ since it is not at all standard for 'times-ten-to-the-power-symbol' and is probably much more useful as an operator (the 'exception-symbol' providing a good alternative anyhow); and the ¢ since # is more widely available (e.g., ASCII and EBCDIC) and ¢ is a point of confusion between codes (again ASCII and EBCDIC).

4.3.5. In the reference language the ∿ is the (approximate) representation of both the 'tilde-symbol' and the 'skip-symbol'. This conflicts with the one-symbol-one-mark principle and moreover cannot easily be mended by proper redefinition: many compilers will get themselves into trouble over the legal construction ∿:=: a . We propose the ∿ for the 'skip-symbol' only: then 'skip' and 'nil' have symmetric representations, and for not there generally are enough alternatives.

4.3.6. Experience has shown that the field where unanimity is hardest to reach is that of operator representations. This is also reflected by the avalanche of operator definitions in the Report. It is tantalizing to see the editors use a mechanism of efficient operator-redefinition and not being able ourselves to reach the same effect other than by the rude and inefficient mechanism of redeclaring:

```
op // = (int n, m) int:  n % m,
```

thereby entailing an additional procedure call upon each application. We therefore propose to add a simple rename mechanism for operators:

```
op (int, int) int // = op %,

        {where CONTRACTITY is EMPTY, operator token,
         PRAM NEST operator with TAD}
```

not causing an additional procedure call.

This feature would:

a. beautify chapter 10 of the Report (less ivory tower),
b. free compiler designers from a lot of awkward decisions, and compiler writers from patching up the compiler afterwards when outside pressure decides that /+ is a necessary operator for dyadic −,
c. please everybody who has special ideas on the appearance of operators,
d. greatly facilitate combining of programs.

4.4.Example:

An actual representing mechanism, e.g. for ASCII64, might then look as follows.

Step 1: after each 'bold-TAG-symbol' that is followed by a 'LETTER-symbol' or a 'DIGIT-symbol' or nothing, a 'significant-layout-mark' ("blank") is inserted.

Step 2: in front of every symbol a sequence of zero or more 'significant-layout-mark's is inserted.

Step 3: every symbol is "represented", as follows:

- if it is a 'bold-TAG-symbol' it is represented by an 'apostrophe-mark' (') followed by a sequence of LETTER-marks and DIGIT-marks that correspond to the LETTERs and DIGITs in the TAG, in that same order (the bolding convention).
- if it is one of the following symbols, it is represented by the corresponding mark.

| | | | |
|---|---|---|---|
| stick-symbol | exclamation-mark | (!) | cs |
| quote-symbol | quote-mark | (") | c |
| brief-comment-symbol | tic-tac-toe-mark | (#) | s |
| formatter-symbol | dollar-sign-mark | ($) | cs |
| percent-symbol | percent-mark | (%) | cs |
| ampersand-symbol | ampersand-mark | (&) | cs |
| exception-symbol | apostrophe-mark | (') | |
| open-symbol | left-parenthesis-mark | (() | c |
| close-symbol | right-parenthesis-mark | ()) | c |
| asterisk-symbol | asterisk-mark | (*) | cs |
| plus-symbol | plus-mark | (+) | c |
| comma-symbol | comma-mark | (,) | c |
| minus-symbol | minus-mark | (-) | c |
| point-symbol | point-mark | (.) | c |
| divided-by-symbol | slash-mark | (/) | cs |
| DIGIT-symbol | DIGIT-mark | (0 ... 9) | c |
| colon-symbol | colon-mark | (:) | cs |
| go-on-symbol | semicolon-mark | (;) | cs |
| less-than-symbol | smaller-mark | (<) | cs |
| equals-symbol | equal-mark | (=) | cs |
| greater-than-symbol | greater-mark | (>) | cs |
| question-symbol | question-mark | (?) | mcs |
| at-symbol | at-mark | (@) | cs |
| LETTER-symbol | LETTER-mark | (A ... Z) | c |
| sub-symbol | left-square-bracket-m. | ([) | cs |
| backslash-symbol | backslash-mark | (\) | mcs |
| bus-symbol | right-square-bracket-m. | (]) | cs |
| not-symbol | circumflex-mark | (^) | cs |
| underscore-symbol | underscore-mark | (_) | cs |

(symbols that are 'OTHERMONAD-symbol's are marked 'm', those that are produced by 'MATCH-other-PRAGMENT-item' are marked 'c' and those that are produced by 'other-string-item' are marked 's').

- if it is not one of the above, no representation is provided.

Step 4: in front of every mark a sequence of zero or more 'dummy-layout-mark's is inserted.

Note 1.
If downward compatibility with ASCII48 would have been our goal, we should have defined the $ as an alternative representation of the 'exception-symbol' (and it might at the same time govern a different bolding-convention).

Note 2.
The code can be compatibly extended to ASCII96 by defining, for example:

| | | | |
|---|---|---|---|
| grave-symbol | grave-mark | (`) | mcs |
| LETTER-symbol | small-LETTER-mark | (a ... z)c | |
| left-brace-symbol | left-brace-mark | ({) | mcs |
| stick-symbol | interrupted-bar-mark | (¦) | cs |
| right-brace-symbol | right-brace-mark | (}) | mcs |
| skip-symbol | tilde-mark | (~) | cs |

4.5. The derepresenting mechanism is essentially the above in reverse order. It might be described in ALGOL 68 as follows.

```
mode symbol = union(char, string);

co non-bold symbols will be delivered as chars, bold symbols as strings co

proc symbol = symbol:
(symbol s; while layout mark(s:= symbolette) do skip; s);

proc symbolette = symbol:
if ahead = apostrophe mark
then    ahead:= solid mark;
        if letter(ahead)
        then    string s:= ahead;
                while letgit(ahead:= solid mark) do s+:= ahead;
                s
        else    apostrophe mark
        fi      co bolding convention co
else    char c= ahead; ahead:= solid mark; c
fi;

proc solid mark = char:
(char c; while dummy layout mark(c:= mark) do skip; c);

proc mark = char: (char c; read(c); c);

char ahead := solid mark;
```

provided that appropriate (ASCII64) definitions of "layout mark", "apostrophe mark", "letter", "letgit" and "dummy layout mark" are supplied.

## 5. Summary.

A clean interface is defined between the production part and the representing part of the language. In order to effect this interface minor changes to the production part are proposed. In order to standardize the (environment-dependent) representing part, guidelines concerning its definition and construction are proposed.