Short-circuit logical pseudo-operators for Algol 68

GNU68-2025-003 (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 Revised Report, and it is expressed as modifications to the Report.

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

Rationale

Most programming languages widely used today provide short-circuited versions of the logical OR and AND functions. In many cases these are the only versions of the logical operators which are provided. They allow to express many conditions in a brief and clean way, relying on the control flow logic implied by the operators. This is particularly rewarding when the operations get nested, as is often the case.

The operands of an Algol 68 formula are always elaborated collaterally. The standard logical operators **and** and **or** are not an exception. When faced with a situation where short-circuited logic is required, one is forced to make the control flow explicit.

Let us suppose we have a string value and we wish to determine whether the last character stored within it is a newline character. However, the string value may be empty, so we have to check its size before indexing it in order to avoid a bounds run-time error.

As discussed above, using the standard logical operators are not an option, because of collateral elaboration of the operands of a formula:

```
if upb str > 0 and str[upb str] = "'/" # Wrong #
then c ... c fi
```

Instead, we have to encode the lazy logic by mean of explicit control flow clauses. An obvious clumsy and verbose way to do this is something like:

```
if upb str > 0
then if str[upb str] = "'/"
     then C ... C fi
fi
```

A much better idiom is to use the abbreviated style of conditional clause this way:

```
if (upb str > 0 | str[upb str] = "'/" | false)
then C ... C fi
```

If the implementation we use guarantees that the **skip** boolean value is always **false**, like GCC does, then we can use an even more compact (if non-portable) version of the idiom:

```
if (upb str > 0 | str[upb str] = "'/")
```

```
then C ... C fi
```

When it is a short-circuited OR what is needed, it is always possible to negate the first operand and use the idiom above. For example, if we want to check whether a given name is nil or if its referred value is zero, we can do:

```
if (ptr :/=: nil | ptr = 0)
then C ... C fi
```

The andth and orel pseudo-operators

This extension, which comes from ALGOL68RS and also the Algol 68 Genie interpreter, takes the form of two constructs that take the form of pseudo-operators:

`tertiary1` **andth** `tertiary2`
: Meaning "and then", elaborates `tertiary1` and if it yields true, then elaborates `tertiary2` and yields its value. Otherwise it yields false.

`tertiary1` **orel** `tertiary2`
: Meaning "or else", elaborates `tertiary1` and if it yields false, then elaborates `tertiary2` and yields its value. Otherwise it yields true.

Note that the way the pseudo-operators are implemented, they have a lower priority than any proper operator. This means that care must be given when using them with the **and** and **or** operators. For example, the following code almost certainly doesn't elaborate as intended, and it may result in nil being dereferenced:

```
if ptr :/=: nil andth ptr => 0 and ptr <= 10
then C ... C fi
```

Parenthesis are necessary to assure the intended safe semantics:

```
if ptr :/=: nil andth (ptr => 0 and ptr <= 10)
then C ... C fi
```

2 Formal Description

The short-circuited logical functions are units.

```
5.1

A) UNIT{32d} :: ... ; and function{57a} ; or function {57b}.
```

These logical functions are pseudo-operators providing logical AND and OR functions with short-circuited elaboration.

```
5.7 Short-circuit logical functions

{ The short-circuit logical functions are pseudo-operators providing
   logical AND and OR functions with short-circuited elaboration.  }

5.7.1 Syntax

a) boolean NEST and function{5A} :
     meek boolean NEST TERTIARY1, andth{94c} token ; meek boolean NEST TERTIARY2.

b) boolean NEST or function{5A} :
     meek boolean NEST TERTIARY1, orel{94c} token ; meek boolean NEST TERTIARY2.

c) *boolean NEST short circuit function :
     boolean NEST and function{a} ; boolean NEST or function{b}.

5.7.2 Semantics
```

```
a) The yield of an and-function F, in an environ E, is determined
   as follows:
   . let t be the yield of TERTIARY1 in E.
   . If t = false,
     then the yield of F is false;
     otherwise the yield of F is the yield of TERTIARY2 in E.

b) The yield of an or-function F, in an environ E, is determined
   as follows:
   . let t be the yield of TERTIARY1 in E.
   . If t = true,
     then the yield of F is true;
     otherwise the yield of F is the yield of TERTIARY2 in E.
```

Two new symbols have been invented, with a proposed representation in the reference language.

```
9.4.1.c
  andth symbol{57b}                    ANDTH
  orel symbol{57b}                     OREL
```

3 Implementation Notes

None whatsoever.