User's Guide to Blacklisting

Heikki Järvinen, Sami Saarinen, Per Undén

ECMWF

October 9, 1996

Contents

Chapter 1

Introduction

In the operational suite on Cray computer, the blacklist was basically a list of undesired stations to be excluded from the analysis in operations, and usually in prepan experiments, too, based on monthly monitoring by the Operations Department. The technique for blacklisting has been streamlined as a part of the migration of operational codes from Cray to Fujitsu.

A new blacklist format has been introduced that allows a great deal more flexibility in decision making on the use of observations. The blacklist now consists of two parts: data selection part and monthly monitoring part. Data selection part contains information about which variables will be used in the assimilation, and it should be amended only rarely, except in experimentation. The monthly monitoring part, on the other hand, will be updated fairly frequently as a result of data monitoring. The former automatic ship blacklist is not supported any more.

This guide comprehensively describes the format of the blacklist language developed at ECMWF during the migration project in 1995-96 based on initial idea by *Mats Hamrud*.

Chapter 2

The Blacklist language

The way the blacklisting now works in the IFS context is as follows. One edits a blacklist file which is written in a specific format. That file is then converted into a subroutine (C language) using the blacklist compiler. The subroutine is then compiled and linked into the executable. This external routine is called from the IFS with a list of arguments in the observation screening run. IFS then receives a few flags telling whether to reject or accept this station or variable for assimilation. The following example will clarify the consepts used in blacklisting.

```
if (OBSTYP = synop) then
   if VARIAB in (u10m, v10m)
             and LSMASK = land
             and abs(LAT) < 25 then
        fail(constant);
   endif
endif;
```

There are several patterns in this single blacklisting rule and in the following they will be called:

- variables, like `OBSTYP, VARIAB, LSMASK, LAT` (see 2.1)
- keywords, like `synop, u10m, v10m, land, constant` (see 2.2)

- statements, like `if-then-endif`-block (see 2.3.1)
- operators, like `and`, `in`, `=`, `<` (see 2.3.2)
- built-in functions, like `abs` (see 2.4)
- actions, like `fail` (see 2.5)

Variables get their values from IFS. These are compared against the keywords or values given in the blacklist. If the blacklist rule is true, fail-function takes action activating blacklisting flags and returning back to the calling routine in IFS. Note that the blacklist language is case insensitive and no column orientation is required.

2.1 Variables

A list of variables that are currently defined in IFS is given below. Adding new variables, see for 5.3.

2.1.1 Report characteristics

Variable	Meaning	Possible value
OBSTYP	Observation type	Keyword (as listed below)
STATID	Station id	Right justified 8 character string
CODTYP	Code type	Integer value as defined in IFS
INSTRM	Instrument type	Integer value as defined in IFS
DATE	Date	Packed integer YYMMDD
TIME	Exact time	Packed integer HHMMSS
LAT	Latitude	Real value in degrees ($-90 <= LAT <= 90$)
LON	Longitude	Real value in degrees ($-180 < LON <= 180$)
STALT	Station altitude	Real value in metres

2.1.2 Model/first guess characteristics

Variable	Meaning	Possible value
MODORO	Model orography	Real value in metres
LSMASK	Land/sea -mask	Keyword (as listed below)
MODPS	Model surface pressure	Real value in hectopascals (`hPa`)
MODTS	Model surface temperature	Real value in Kelvins
MODT2M	Model 2 metre temperature	Real value in Kelvins

2.1.3 Observation characteristics

Variable	Meaning	Possible value
VARIAB	Variable name	Keyword (as listed below)
VERT_CO	Type of vertical coordinate	Keyword (as listed below)
PRESS	Value of vertical coordinate	Pressure in hectopascals (`hPa`) or height in metres or satellite channel (integer)
PRESS_RL	Reference level pressure	as in PRESS
PPCODE	Synop pressure code	Keyword (as listed below)
OBS_VALUE	Observed value	Real value
FG_DEPARTURE	First guess departure	Real value
OBS_ERROR	Observation error	Real value
FG_ERROR	First guess error	Real value

2.2 Keywords

Keywords are fixed values against which certain variables are compared. They should be consistent with the IFS definitions. A list of keywords that are currently defined in the blacklist. Adding new keywords is straightforward.

Variable	Keyword
OBSTYP	synop, airep, satob, dribu, temp, pilot, satem, paob, scatt (or integer values from 1 to 9)
CODTYP	(an integer value as defined in IFS)
INSTRM	(an integer value as defined in IFS)
LSMASK	land sea

Variable	Keyword	Definition
VARIAB	u	Upper air wind u-component
	v	Upper air wind v-component
	z	Geopotential
	dz	Thickness
	pp	Upper air pressure
	t	Upper air temperature
	td	Upper air dew point
	rh	Upper air relative humidity
	q	Specific humidity
	pwc	Precipitable water content
	t2m	2 metre temperature
	td2m	2 metre dew point
	rh2m	2 metre relative humidity
	ps	Surface pressure
	ts	Surface temperature
	rawrad	Raw radiance
	ccrad	Cloud cleared radiance
	scattu	Scatterometer wind u-component
	scattv	Scatterometer wind v-component
		... and many others (see 5.1)

Variable	Keyword	Definition
VERT_CO	pressure	Pressure of observation level
	height	Height of observation level
	tovs_cha	TOVS channel
	scat_cha	SCATT channel
PPCODE	sealev	Sea level
	stalev	Pressure of station level
	p500gpm	Pressure of 500gpm
	p1000gpm	Pressure of 1000gpm
	p2000gpm	Pressure of 2000gpm
	p3000gpm	Pressure of 3000gpm
	p4000gpm	Pressure of 4000gpm
	g1000hpa	Geopotential of 1000hPa
	g900hpa	Geopotential of 900hPa
	g850hpa	Geopotential of 850hPa
	g700hpa	Geopotential of 700hPa
	g500hpa	Geopotential of 500hPa

2.3 Statements and operators

2.3.1 IF-statement syntax

The IF-statement syntax (note the semicolon (;) after each statement):

Syntax	Meaning

`if (condition) then`
  statement_1;
  statement_2;
  etc.
else
  statement_1;
  statement_2;
  etc.
endif;
	IF-test with an optional ELSE-block.
	Nested IF-tests are valid in every statement.
	Every IF-THEN or IF-THEN-ELSE must match an ENDIF
	Condition can be any logical or arithmetic operation

2.3.2 List of the simple operators

A list of operators that are currently defined in the Blacklist-language:

Operator	Meaning
and	Logical AND to be used in the IF-condition
`&`	
or	Logical OR
`|`	
not	Logical NOT
==	Logical EQUAL-sign in the IF-condition only
=	Logical EQUAL-sign in the IF-condition when the types on both side have to match. Alternatively an assignment operator in other statements
`>`	Greater than
`<`	Smaller than
`>=`	Greater than or equal to
`<=`	Smaller than or equal to
`<>`	Not equal to
`/=`	
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`^`	Power
`**`	
`%`	Modulo (`a modulo b = a%b`. Same as `mod(a,b)`)

2.3.3 List of more complex operators

Somewhat more complex operators can also be used to simplify coding. For example the compound `AND`-operators below:

Complex operator	Equivalent meaning
$a < x < b$	$a < x$ AND $x < b$
$a <= x <= b$	$a <= x$ AND $x <= b$
$a <= x < b$	$a <= x$ AND $x < b$
$a < x <= b$	$a < x$ AND $x <= b$
$a > x > b$	$a > x$ AND $x > b$
$a >= x >= b$	$a >= x$ AND $x >= b$
$a >= x > b$	$a >= x$ AND $x > b$
$a > x >= b$	$a > x$ AND $x >= b$
in (*list*)	Checks whether an item appears in a list, where the elements are separated with a comma (,). A list may contain either numbers or strings.
notin (*list*)	Checks whether item is NOT in the list

2.4 Built-in functions

The Blacklist-language also contains some built-in functions. They are listed below:

Function	Meaning
exp(x)	Exponent (e^x)
ln(x)	Natural logarithm; $x > 0$
log10(x)	Base-10 logarithm; $x > 0$
lg(x)	
sqrt(x)	Square root ($\sqrt{x}$); $x \geq 0$
mod(a,b)	a modulo b; same as a%b
max(x_1, x_2, ..., x_n)	Maximum of the elements x_i
min(x_1, x_2, ..., x_n)	Minimum of the elements x_i
sum(x_1, x_2, ..., x_n)	Sum of the elements x_i
prod(x_1, x_2, ..., x_n)	Product of the elements x_i

Function	Meaning
abs(x)	Absolute value
sin(x)	Sine; x in degrees
cos(x)	Cosine; x in degrees
tan(x)	Tangent; x in degrees
asin(x)	Arcussine; returns degrees
acos(x)	Arcuscosine; returns degrees
atan(x)	Arcustangent; returns degrees
atan(x,y)	Arcustangent (two param. version); returns degrees
sinh(x)	Hyberbolic sine; x is a scalar
cosh(x)	Hyberbolic cosine; x is a scalar
tanh(x)	Hyberbolic tangent; x is a scalar
int(x)	Integer part (truncated) of a value
round(x)	Round to the nearest integer
ceil(x)	Ceiling int. value i.e. $x \leq ceil(x)$
floor(x)	Floor int. value i.e. $x \geq floor(x)$
rand()	Return a random number
srand(x)	Supply random seed in x
cputime()	Return CPU-time used

In addition, there is one special function to study whether a point is within a circular area on the Earth (e.g. to blacklist Meteosat SATOBs if they are too far away):

```
if (not (rad (0, 0, 45, LAT, LON))) then fail(monthly); endif;
```

The function is called `rad()` and requires five (5) arguments. It returns one (1) if the observation is within the circle, otherwise zero (0). The usage is

```
rad(reflat, reflon, refdeg, LAT, LON)
```

where the `refdeg` is radius of the circle on the Earth with the `(reflat, reflon)` as a center point of the circle. The `(LAT, LON)` is the position of the observation to be checked, i.e. `LAT` and `LON` of the report. All values are given in degrees. See also picture 2.1.

The following arithmetic is performed in the function `rad()`:

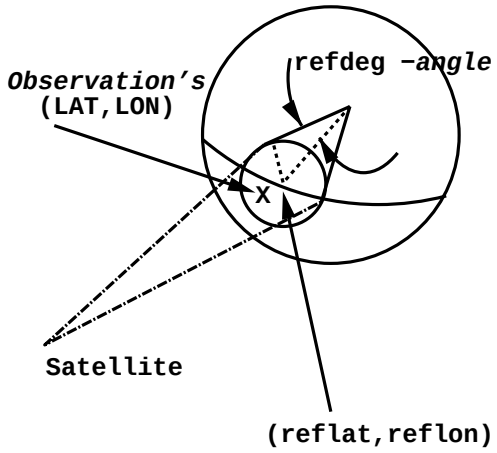

Figure 2.1: Schematic view of the `rad()`-function parameters.

1. Convert all degrees to radians
2. Calculate angle distance (in radians) relative to the center point

```
obsdeg = acos( cos(reflat) cos(LAT) cos(LON-reflon) +
               sin(reflat) sin(LAT) )
```

3. Return one from `rad`, if $obsdeg \leq refdeg$, otherwise zero.

2.5 Actions

Finally, perhaps the most important function `fail()`. It returns information back to the application.

The `fail()`-function is a variable number argument function. If no arguments are given, the first argument is assumed to contain keyword `monthly`, i.e. rejection occurs in the monthly monitoring part of the blacklist-file. If the second argument – seriousness of the blacklisting – is omitted, then seriousness is assumed to be equal to one.

Arguments in the `fail(arg1, arg2)`-function are:

Argument#1 (`arg1`)	**Meaning**
monthly	monthly monitoring (*default*)
constant	constant blacklisting
experimental	experimental blacklisting
whitelist	whitelisting, i.e. enforcing to use this report or variable
Argument#2 (`arg2`)	**Meaning**
level	level of seriousness of blacklisting Range is between [0 . . . 1]. *Default=1*

When a call to the `fail()`-function occurs, the control is returned immeadiately to the calling application. Normally the application is the IFS, which will get the following (Fortran) variables updated:

Variable	**Type**	**Meaning**
NCMBLI	Integer	Blacklisting indicator 0 = not blacklisted (*default*) 1 = monthly monitoring 2 = constant blacklisting 3 = experimental 4 = whitelist
ZCMCCC	Real	Seriousness of the blacklisting 0 = Default if not blacklisted 1 = Default if blacklisted (i.e. `NCMBLI > 0`) [0.01 . . . 0.99] for non-complete blacklisting (optional)
FEEDBACK	Integer	Feedback vector telling which variable(s) caused the blacklisting to occur 0 = Blacklist line number where the `fail()`-function took action 1-N = Pointers to the variable indices to help to locate the the responsible variables

There is a range of values for ZCMCCC, and together with other information in the quality control, and a value less than one may still lead to the use of this variable in the assimilation. The inclusion of this option of non-strict blacklisting increases flexibility of the use of observations.

If one wants to use whitelisting, these rules have to be placed right to the beginning of the blacklist file.

Examples of non-standard cases:

1. whilelist: place these rules right to the beginning of the blacklist. The search will be terminated at the first fulfilled whitelist rule.
2. blacklist seriousness is between 0.01 ... 0.99, the search continues and returns with the highest seriousness found.

2.6 Variable declaration

Variable declaration has to be performed, if data will be passed from an application (like IFS) into the blacklist. This is normally done through `external`-declaration (see for 4.2 or 5.1). Also, selected variables can be protected by defining them as constants.

Additional or local variables can be defined everywhere in the code, even within the IF-THEN-ELSE-ENDIF -block (except in IF-condition). However, any attempt to use undeclared or uninitialized variables will cause the Blacklist-compilation to fail.

The simplest variable declaration is an assignment operation.

Variable declation	**Meaning**

a = 10;
b = a ** 2;
const c = b;
 	
external d;
 	
external s is special;
	Variable **a** was made local and modifiable with value **10**
	Variable **b** was made local and modifiable
	Variable **c** was made constant with the current value of **b**.
	Only this particular assignment is possible.
	Variable **d** gets its value from outside (from application).
	The value cannot be changed without causing
	compilation to fail.
	The attribute special guarantees that variable **s** is external
	and belongs to the observation report body entry part.
	Unmodifiable as well.

a = "12345678"; const_char c = a; external_char d; external_char s is special;	Variable **a** was made local and modifiable with a character value of **"12345678"**. Unless otherwise stated a character string MUST have exactly 8 chars. If a dot (.) is found from a string during the character string comparison, it is treated as a wildcard i.e. any character. Variable **c** was made constant with the current value of **a**. Character variable **d** gets its value from outside. The attribute special guarantees that variable **s** is external_char and belongs to the observation report body entry part.

Chapter 3

Operational and experimental use of blacklist

3.1 Location of blacklist files

3.2 An emergency rebuild

There is a copy of all building blocks needed to convert from old blacklist(s) to the new format, and compile them into a linkable object with the IFS. All needed files are currently found on FUJITSU directory `~daj/bin`. To bypass default compilation sequence. To use that sequence, run the following script (Korn-shell `/bin/ksh`):

```
# Define user
USER=daj
# Define new path for old-to-new conversion routine
export BL_OLD2NEW=~${USER}/bin/bl_old2new.x
# Run the conversion using blcomp
~${USER}/bin/blcomp -o old_list new_list.B
# Re-create "new_list.B" to contain the data selection part
cat > new_list.B << EOF
#include "external_new_list.b"
#include "data_sel_part"
```

```
#include "monthly_new_list.b"
EOF
# Create C_code.o from the modified new_list.B
# Don't forget to use the new compiler
# from directory (the -x option)
~${USER}/bin/blcomp -x ~${USER}/bin/bl95.x -c new_list.B
```

3.3 Remaking the blacklist compiler

See for documentation under ClearCase. Run the following commands on a workstation:

```
% selview -p bl -s CY
```

After that pick the latest release and look for README-file (see also for appendix A).

3.4 Some guidelines

Please do not place any station identifiers into the data selection part of the blacklist. Instead, have them in the monthlt monitoring part. By this way we can have as few changes as possible in the data selection part and make e.g. re-analysis much easier.

After any modifications to the blacklist, please remember to recompile (preferably on a workstation) to check for syntax errors.

Chapter 4

Creating new blacklist file

Blacklist compilation is fully controlled by the script called `blcomp`. It has the following capabilities:

- Optionally convert from an old ASCII blacklist format to a new format
- Check the syntax of a given blacklist
- Create C-language file (`C_code.c`) catered for observation processing
- C-compile the C-file to create linkable object

4.1 Usage of the `blcomp`

The `blcomp`-script has the following usage:

```
blcomp [-aAcCdDefiILmMnoOpSx8] blacklist_file.b (or blacklist_file.B)
```

where the flags are as follows:

Flag	Meaning
-c	Generate C-code file `C_code.c` from BLACKLIST
-o old_blacklist	Converts old BLACKLIST to new before compilation
-x /path/bl95	BL-compiler executable name (overrides BL95 definition)
-i	Ignore "the user interface" in case of errors
-a "arguments"	Arguments to be passed to the BL-compiler
-A	Compile whole BLACKLIST despite exit/return stmts
-C "compiler_name"	C-compiler to be used (if other than `cc` or BL_CC)
-d	Turn debugging on while compiling the BLACKLIST
-D "display:0.0"	(Re-)define DISPLAY environment variable
-e editor-name	Preferred EDITOR ("the user interface", if errors found)
-f "C-flags"	Additional C-compiler flags
-I "pathname"	Search path(s) for `#include`-files
-L	Display the default library (`libbl95.a`) name only and exit
-m error_count	Maximum error count before aborting the BL-compiler
-M	Generates pseudo MAIN-program
-n times	Number of times to loop over the BL-compiled instructions
-O	Optimize BLACKLIST-code
-p	Further debugging: print table(s) of used symbols
-S computer_arch	Computer system architecture (overrides ARCH def.)
-8	Ignore the 8 character limit in strings

The new BLACKLIST-file must have either suffix ".b" or ".B". In the latter case the C-preprocessor `/lib/cpp` will be run in the front of BL-compiler mainly to resolve any possible `#include`-statements.

For pure syntax checking of the new BLACKLIST-file, give:

```
blcomp blacklist_file.b
  or
blcomp blacklist_file.B
```

By giving `blcomp` without arguments you will get the usage. If you fail to do this, check for your setting of the `PATH`-environment variable.

4.2 Conversion from old to new blacklist

Conversion from old to new and syntax checking of the new BLACKLIST-file can be accomplish in the following way:

```
blcomp -o old_text_blacklist_file newfile.b
  or
blcomp -o old_text_blacklist_file newfile.B
```

Here, the *input* file is `old_text_blacklist_file`, and output file is `newfile.b` (or `newfile.B`) in the new blacklist format.

While converting from old to new format, the used suffix `.b` or `.B` of the new blacklist file plays an important role. First of all, there MUST always be one suffix. When the suffix is `.b`, then a single blacklist file (here: `newfile.b`) will be created with all external (e.g. variable declarations) and monthly monitoring rules (a portion of blacklist that normally does not change during one month period) inlined.

If the suffix `.B` was used, then the following three (3) files are generated:

- master file (`newfile.B`)
- include-file no. 1 for externals (`external_newfile.b`)
- include-file no. 2 for monthly part (`monthly_newfile.b`)

The contents of the master file is simply the following two lines:

```
#include "external_newfile.b"
#include "monthly_newfile.b"
```

One way to bring in your own modifications, is to create a new master-file, for example:

```
#include "external_newfile.b"
#include "my_own_file"
#include "monthly_newfile.b"
```

This is exactly how the data selection part comes in in the production run, where instead of `my_own_file` is data selection part.

4.3 C-code generation

Enabling fast blacklist handling the blacklist file is always converted into an object file (`.o`) meant to be linked with the (Fortran-)application (like IFS) in conjunction with the blacklist object library (normally `libbl95.a`).

Once a blacklist file (either with `.b` or `.B` suffix) is available, it can be converted to C-language file `C_code.c` and compiled to an object for maximum performance. This can be done as follows:

```
blcomp -c blacklist_file.b
  or
blcomp -c blacklist_file.B
```

4.4 Linking with an application

A Fortran-application (IFS) interfaces the blacklist via two subroutines:

- BLACKBOX_INIT
- BLACKBOX

The former one is responsible for initiating the variable list active by the application. And the latter one handles all burden of interfacing the blacklist file.

To link application with the blacklist software, one needs not only the `C_code.o`-object file, but also the blacklist library `libbl95.a`. Linking command is normally:

```
linker application.o C_code.o /bl95path/libbl95.a other_libs
```

The exact location of the blacklist library can be found via command:

```
blcomp -L
```

4.5 Combining conversion and object generation

If no data selection part is needed, one can combine conversion from old to new blacklist and object code generation described above:

```
blcomp -c -o old_text_blacklist_file newfile.b
  or
blcomp -c -o old_text_blacklist_file newfile.B
```

4.6 User interface

It is always recommended to (cold-)compile a modified blacklist on a workstation to check for syntax errors. If any errors are detected, the `blcomp`-command attempts to open an editor session and jump directly to the line where the (first) error occurred.

Sometimes this facility is not desirable and can be disabled by using `-i` flag in the `blcomp`-command.

Chapter 5

Examples

The blacklist file is normally about 1000 lines long. In order not to confuse readers, we will explain here with very short examples what can be done with the blacklist

5.1 A simple example

A fraction of an old blacklist (`old`) looks like as follows:

```
   3ELC  1 3
  ELBX3  1 333
N503US  2 00030
UAL...  2 00030
   024  3 33000000 033333
   0//  3 33000000 033333
 46527  4 33300
  ERES  5          000003
 08221  6          0330
   201  7 33300000 00333
```

When compiled with `blcomp -o old new.b`, we get a new file `new.b`. The local constant variable declaration section looks as follows:

```
!
!  Written by an automatic conversion program, version 3
!
!
!  File converted from the file "old"
!

! FAILCODE :
const monthly = 1;
const constant = 2;
const experimental = 3;
const whitelist = 4;

! OBSTYP :
const synop = 1;
const airep = 2;
const satob = 3;
const dribu = 4;
const temp = 5;
const pilot = 6;
const satem = 7;
const paob = 8;
const scatt = 9;

! CODTYP : none

! INSTRM : none

! VARIAB :
const u = 3;
const v = 4;
const z = 1;
const dz = 57;
const rh = 29;
const q = 7;
const pwc = 9;
const rh2m = 58;
const t = 2;
const td = 59;
const t2m = 39;
const td2m = 40;
const ts = 11;
const ptend = 30;
const w = 60;
```

```
const ww = 61;
const vv = 62;
const ch = 63;
const cm = 64;
const cl = 65;
const nh = 66;
const nn = 67;
const hshs = 68;
const c = 69;
const ns = 70;
const s = 71;
const e = 72;
const tgtg = 73;
const spsp1 = 74;
const spsp2 = 75;
const rs = 76;
const eses = 77;
const is = 78;
const trtr = 79;
const rr = 80;
const jj = 81;
const vs = 82;
const ds = 83;
const hwhw = 84;
const pwpw = 85;
const dwdw = 86;
const gclg = 87;
const rhlc = 88;
const rhmc = 89;
const rhhc = 90;
const n = 91;
const snra = 92;
const ps = 110;
const dd = 111;
const ff = 112;
const rawbt = 119;
const rawra = 120;
const satcl = 121;
const scatss = 122;
const du = 5;
const dv = 6;
const u10m = 41;
const v10m = 42;
const rhlay = 19;
```

```
const auxil = 200;
const cllqw = 123;
const scatdd = 124;
const scatff = 125;

! LSMASK :
const sea = 0;
const land = 1;

! PPCODE :
const psealev = 0;
const pstalev = 1;
const g850hpa = 2;
const g700hpa = 3;
const p500gpm = 4;
const p1000gpm = 5;
const p2000gpm = 6;
const p3000gpm = 7;
const p4000gpm = 8;
const g900hpa = 9;
const g1000hpa = 10;
const g500hpa = 11;

! VERT_CO:
const pressure = 1;
const height = 2;
const tovs_cha = 3;
const scat_cha = 4;
```

The external variable definition section looks as follows:

```
! External variables (non-special):
external obstyp;
external_CHAR statid;
external codtyp;
external instrm;
external date;
external time;
external lat;
external lon;
external stalt;
```

```
external modoro;
external lsmask;
external rad;

! External variables (SPECIAL):
external variab is SPECIAL;
external vert_co is SPECIAL;
external press is SPECIAL;
external press_rl is SPECIAL;
external ppcode is SPECIAL;
external obs_value is SPECIAL;
external obs_departure is SPECIAL;
external modps is SPECIAL;
```

And finally the actual monthly monitoring rules in a new blacklist format:

```
if ( OBSTYP = synop ) then
   if   VARIAB in ( z, ps )
   and  STATID = "    3ELC"
   then fail(); endif;

   if   VARIAB in ( z, ps, u10m, v10m )
   and  STATID = "   ELBX3"
   then fail(); endif;

return; endif;

if ( OBSTYP = airep ) then
   if  (VARIAB = t)
   and  STATID in ( "  N503US", "  UAL...")
   then fail(); endif;

return; endif;

if ( OBSTYP = satob ) then
   if   STATID in ( "     0//", "     024")
   then fail(); endif;

return; endif;

if ( OBSTYP = dribu ) then
```

```
   if   VARIAB in ( z, ps, u, v )
   and  STATID = "   46527"
   then fail(); endif;

return; endif;

if ( OBSTYP = temp ) then
   if  (VARIAB = z)
   and  STATID = "    ERES"
   then fail(); endif;

return; endif;

if ( OBSTYP = pilot ) then
   if   VARIAB in ( u, v )
   and  STATID = "   08221"
   then fail(); endif;

return; endif;

if ( OBSTYP = satem ) then
   if       STATID = "     201"
   then fail(); endif;

return; endif;
```

5.2 A more complex example

The Blacklist compiler will generate quite a compact and readable code from the following excerpt:

```
     ATQM  1 3
     ATRK  1 3
     ATSR  1 3
     C6BB  1 3
     C6QK  1 3
    AN...  2 33333                                    50   10
    NWA74  2 33333              -90   90  -40  -80
      035  3 33000000 033333     -50   50  -50   50  1000 401
      104  3 33000000 033333     -50   50   90 -170
```

```
    20674  5          000003                          100   10  11 13
    40179  5          033000                                  05 07
    40179  6          0330                                    05 07
```

The constant definition is not different from the previous example. For the monthly monitoring rules in a new blacklist format becomes:

```
if ( OBSTYP = synop ) then
   if   VARIAB in ( z, ps )
   and  STATID in ( "    ATQM", "    ATRK", "    ATSR", "    C6BB", "    C6QK")
   then fail(); endif;

return; endif;

if ( OBSTYP = airep ) then
   if      ( 50 >= PRESS >= 10 )
   and  STATID = "   AN..."
   then fail(); endif;

   if     (  ( LAT < -90 or LAT > 90 )   or ( -80 < LON < -40 ) )
   and  STATID = "   NWA74"
   then fail(); endif;

return; endif;

if ( OBSTYP = satob ) then
   if     (  ( LAT < -50 or LAT > 50 )   or ( -170 < LON < 90 ) )
   and  STATID = "     104"
   then fail(); endif;

   if     (  ( LAT < -50 or LAT > 50 )   or ( LON < -50 or LON > 50 ) )
   and ( 1000 >= PRESS >= 401 )
   and  STATID = "     035"
   then fail(); endif;

return; endif;

if ( OBSTYP = temp ) then
   if  (VARIAB = z)
   and ( 100 >= PRESS >= 10 )
   and ( 110000 <= TIME <= 130000 )
   and  STATID = "   20674"
```

```
   then fail(); endif;

   if   VARIAB in ( u, v )
   and ( 50000 <= TIME <= 70000 )
   and  STATID = "   40179"
   then fail(); endif;

return; endif;

if ( OBSTYP = pilot ) then
   if   VARIAB in ( u, v )
   and ( 50000 <= TIME <= 70000 )
   and  STATID = "   40179"
   then fail(); endif;

return; endif;
```

5.3 Adding completely new variable to the system

The current definition of variables can be checked from IFS source code in `obs_preproc/blinit.F`. Adding new variables requires:

1. Never remove or redifine existing variables. That will make re-running earlier cases virtually impossible.
2. Add a variable to the IFS source code in `obs_preproc/blinit.F`.
3. Increase the number of defined variables in `obs_preproc/blinit.F`.
4. External declaration must be done into the `external`-file.
5. Before starting to use the new variable, initialize it properly in `obs_preproc/black.F`.
6. Make sure that the blacklist event flags are fed correctly back to CMA-file in routines `obs_preproc/feblre.F` and `obs_preproc/feblda.F`.
7. The new variable can new be added into the blacklist. If keywords are associated with, declare them in the `external`-file as well.

Appendix A

README-file from ClearCase

```
Instructions for building the new BLACKLIST compiler
====================================================

        (2-SEP-1996 by Sami Saarinen)

(1) Selview to the proper "bl"-project CC-branch, e.g.:

        selview -p bl -s CY15R4

(2) Copy files under view /cc/rd/bl into your local destination:

        [ -d $TMPDIR/bl ] && rm -rf $TMPDIR/bl
        cp -pr /cc/rd/bl $TMPDIR

(3) Go to your local  bl-directory and enter make:

        cd $TMPDIR/bl
        make

    Default settings should be ok for FUJITSU.
    (If necessary, then edit the file "Makefile").
```

(3a) Use following options when compiling for SGI (in C-shell):

```
use epcf90
make ARCH=SGI CC=cc FC=epcf90 FCOPTS="-q -r8" LD=cc LIBS=
```

(3b) Use following options when compiling for CRAY C90:

```
make ARCH=CRAY CPPFLAGS="-N -DCRAY" CC=cc FC=f90 FCOPTS="-dp" \
     AR="bld q" RANLIB="bld tv" LD=cc LIBS=
```

(3c) Use following options when compiling for CRAY T3D:

```
make ARCH=T3D \
     CPPFLAGS="-N -DCRAY -DT3D" \
     CC="env TARGET=cray-T3D /bin/cc" \
     CCOPTS="-DT3D -DCRAY" \
     FC="env TARGET=cray-T3D /mpp/bin/f90" \
     FCOPTS="-dp" \
     LD="env TARGET=cray-T3D /bin/cc" \
     LDFLAGS="-X 1" \
     LIBS=
```

(4) To clean-up from old rubbish, enter: make clean

(5) Master copy of the blcomp-script is kept under
the scripts-directory.