Algol 60 Interpreter
NASE A60

Erik Schönfelder

last updated April 2005

for Version 0.22a

1 The Goal of the Interpreter

This Algol 60 interpreter is based upon the “Revised Report on the Algorithmic Language Algol 60” [RRA60].

At school, a long time ago, I learned Algol 60 in a completely theoretical manner. Later I learned Algol 68 and C (and more ...).

The concept of call-by-name never left my mind, and so I started to write this Algol 60 interpreter: Made for fun and a call-by-name.

Here is an example:

```
'begin'
        'integer' 'procedure' one;
        'begin'
                write (‘one called \n’);
                one := 1
        'end';

        'procedure' foo (n);
        'integer' n;
        'if' n > 0 'then'
                foo ( n - one );

        foo (5)
'end'
```

The parameter ‘n’ in ‘foo (n)’ is called by name. Every time ‘n - one’ is evaluated, ‘n’ is evaluated by name. Guess how many times ‘one’ is called: 5, 10, 15 ?

Guess or prove ? – I want to run the example and see the result. And now you can do like me.

This was the main goal: call-by-name.

Many things were later added, and now the defining description of the “Revised Report on the Algorithmic Language Algol 60” is nearly (hopefully) fulfilled.

2 Installation of the Algol 60 Interpreter

A60 now runs on Un*x machines and PC's.

2.1 Installation for Un*x

Since version v0.18 a configure script is provided.

Simply run `./configure` followed by `make`.

For your convenience the old Makefile is still avail as Makefile.unx.

If configure does not work for you, follow this old instructions:

Glance through the Makefile and change the FLAGS as appropriate:

`_POSIX_SOURCE`
: define this when compiling for a Posix compliant System. This should work and suffice for many Systems including SunOS and Linux.

`USG`
: define this when compiling for a System V Un*x. For a BSD system define nothing; this is the default.

`VPRINTF_MISSING`
: define this if your system does not provide the vprintf () function. This is used in err.c.

`REALLOC_MISSING`
: define this if your system does not provide the realloc () function. This is used in util.c.

`ALLOCA_MISSING`
: define this if your system does not provide the alloca () function. This is only used by bison. If you are not using the bison generated parser, this define is not used.

`NO_LIMITS_H`
: define this if your system has no header limits.h, defining `LONG_MIN` and `LONG_MAX`. (don't care: set it if you're in doubt)

`NO_ENUMS`
: define this if your compiler bombs on enums and you have changed the enum declarations in the header files. Look to ENUM.README for more about this (normally you will not).

`DEBUG`
: define this if you would like to include general debug code (normally you will not).

`PARSEDEBUG`
: define this if you would like to include the debug code for the parser (normally you will not).

`MEMORY_STATISTICS`
: define this if you would like to include some code for computing statistics about the amount of heap and stack used (normally you will not).

For installation adjust BINDIR to point to the destination for the “a60” binary, and LIBDIR to point to the destination of the “a60-mkc.inc” file. If you don’t want this, set them to ‘/tmp’; they are only used, if C output is being compiled. MANDIR and MANSUFF are used to install the “a60.man” manual page.

Ah, we are back to normality:

Say `make` to compile.

If you would like to make the simple edit-and-go xa60 application, say `make xa60`.

If you would like to run the test suite, say `make test`, and hopefully no differences between the expected output and the actual output will be found.

Say `make install` to install the binary, the manpage and the include-file.

Say `make xa60-install` to install the xa60 binary and the xa60 manpage.

2.2 Installation for PC’s

I’ve compiled the sources with QuickC v2.0 using qc-makeit.bat. The project file is qc-a60.mak. The compiler itself runs short of memory when running the optimiser, so the a60-ptab.c module had better be compiled without it.

C code generation is possible, but I’ve tried it only with few examples, because the large generated macros cannot be compiled properly.

3 Algol 60 Command Line Options

When you invoke Algol 60 ...

Without arguments, the program text is read from standard input, and executed upon reaching EOF.

The available options:

'`-h`' Print the usage message and exit.

'`-V`' Print the Version string and exit.

'`-v`' Be verbose processing the input. The version string is displayed too.

'`-n`' Don't run the input; only parse and check.

'`-i`' Do not check or execute the input; parse only. (This was useful for debugging the interpreter.)

'`-t`' Trace line numbers when running the input.

'`-strict`' Follow strict a60 convention. Skip whitespace in entire input, except in strings. Keywords must be enclosed in single quotes.

'`-c`' Create C output from the input. This is an experimental option which changes a60 into something like a60-to-c.

'`-C`' Create C output from the input, like the option -c, but then invokes the C compiler and creates an executable (hopefully).

'`-o` *file*' Place the output in file *file*. This is used, if C code is created (via the -c option) or if the input is compiled (via the -C option).

4 Representation of Algol 60 Code

There is a strict form of the input which conforms to RRA60 and also a simple form.

The strict form:

Keywords are expected to be enclosed in single quotes: '. For example: 'begin', 'for', 'if', 'end'.

The case of letters is insignificant in keywords. For example: 'begin' is the same as 'Begin', 'integer' loopvar is the same as 'INTEGER' loopvar.

Whitespace characters are skipped in the input, except in strings. For example: 'integer' greatnumber is the same as 'integer' great number, and the same as ' i n t e g e r' great n u m b e r.

Strings are expected to be enclosed in double quotes, or in a backquote and a quote. For example: `"This is a string"`, ‘This is a string’. The '\' is recognized as a escape character (like C syntax). `"\n"` is a linefeed, `"\""` is a double-quote and `"\\"` is a backslash.

The simple form:

Keywords are written like identifiers. For example: begin, for, if, end. White spaces are recognized to separate tokens. Therefore, it is illegal to use: integer great number;

The simple form is used if no quoted keyword is scanned. RRA60 conformance can be forced with the ‘-strict’ option.

5 Builtin Functions

5.1 Mathematical and conversion functions

entier, abs, sign, sqrt, sin, cos, arctan, exp: implemented as described in RRA60.

rand, pi: random number generation and the constant “pi”:

```
'real' 'procedure' rand;
        'code'
```

returns a random number between 0.0 (inclusive) and 1.0 (exclusive). The randomness of “rand” is not very robust.

```
'real' 'procedure' pi;
        'code'
```

returns the constant “pi”.

5.2 Input / Output via Channels

The input and output functions use channel numbers to read from or to write to. The range of the channel numbers is from 0 to 15 included. The channel numbers 0, 1 and 2 are taken from the standard channel numbering known als 0 = stdin (standard input), 1 = stdout (standard output) and 2 = stderr (standard error).

The channels 2 to 15 were mapped to files. The first use of a channel determines the direction: If it is an output function, the file is opened in write mode, if it is a input function, the file is opened in read mode.

The filename is read from the environment variable “FILE_n” where n is the channel number. If the environment variable is not set the name “FILE_n” is used with n set to the channel number. So if the environment variable FILE_3 is set to data.txt writing to channel 3 will write to the file data.txt. If this environment variable is not set writing to channel 3 will write to the file FILE_3.

5.3 String related functions

length, outstring, insymbol, outsymbol

```
'integer' 'procedure' length (string);
'string' string;
        'code';
```

returns the length of the string string.

```
'procedure' outstring (channel, value);
'value' channel;
'integer' channel;
'string' value;
        'code';
```

send the string value to the channel channel. Channel 1 is stdout (standard output) and channel 2 is stderr (standard error).

```
'procedure' write (string);
'string' string;
        'code';
```

Prints the string string to standard output. This is the same behavior as outstring (1, string).

```
'procedure' insymbol (channel, string, value);
'value' channel;
'integer' channel, value;
'string' string;
        'code';
```

A character is read from channel channel. If the character is found in string, the index is assigned to value with a starting index of 0. If the character is not found, the negative character code is assigned to value. Channel 0 is stdin (standard input).

```
'procedure' outsymbol (channel, string, source);
'value' channel, source;
'integer' channel, source;
'string' string;
        'code';
```

Prints the character at the source position of string string to channel channel. The posistion is counted from 0. If source is a negative value, -source is sent to the channel and the string is ignored. Channel 1 is stdout (standard output) and channel 2 is stderr (standard error).

5.4 Output and Input of numbers

```
'procedure' print (value, f1, f2);
'value' value, f1, f2;
'real' value;
'integer' f1, f2;
        'code';
```

The value value is printed with f1 and f2 used as format. [still missing: *** describe f1 and f2 ***] The output is printed to standard output.

```
'procedure' inreal (channel, value);
'value' channel;
'integer' channel;
'real' value;
        'code';
```

Reads a real number from channel channel and assigns it to value. Channel 0 is stdin (standard input).

```
'procedure' ininteger (channel, value);
'value' channel;
'integer' channel;
'integer value;
        'code';
```

Reads a integer type number from channel channel and assigns it to value. Channel 0 is stdin (standard input).

```
'procedure' outreal (channel, value);
'value' channel, value;
'integer' channel;
'real' value;
        'code';
```

Prints the value value to channel channel. Channel 1 is stdout (standard output) and channel 2 is stderr (standard error).

```
'procedure' outinteger (channel, value);
'value' channel, value;
'integer' channel, value;
        'code';
```

Prints the value value to channel channel. Channel 1 is stdout (standard output) and channel 2 is stderr (standard error).

5.5 Variable formatted output

```
'procedure' vprint (...);
        'code';
```

Vprint prints the variable arguments to the standard output. The output is terminated with a newline-character. Numbers are printed width a fixed with (about 14 characters). For example: vprint ("Foo: ", 12, 99.9).

6 C-code creation

[** Still not finished **]

C-code creation for less complex programs is now possible. The resulting code is somewhat faster (example whetstones: about a factor of 50).

Call-by-name procedures must be expandable into C macros. The other procedures are translated into C functions.

Problems / Restrictions:

- Run-time checks are simplified or ignored.
- Labels aren't handled correctly in procedures expanded into C macros.
- Switches cannot be translated.
- To be usable, many things will have to be added (or changed).

7 Some Examples

Example 1:

```
'begin'
        write ("Hi!\n")
'end'
```

Assume these three lines are in a file named ‘hi.a60’. Run it with the call ‘a60 hi’: It produces the output:

```
Hi!
```

Example 2:

```
'begin'
        'integer' 'procedure' fakul (n);
        'value' n;
        'integer' n;
        'begin'
                'if' n < 1 'then'
                        fakul := 1
                'else'
                        fakul := n * fakul (n - 1)
        'end';

        'integer' result;

        outstring (1, "See fakul (5): ");
        result := fakul (5);
        outinteger (1, result);
        outstring (1, "\n");
'end'
```

This will produce the output:

```
See fakul (5):  120
```

Example 3:

The classic call-by-name example: The “Jensen Device”:

[Note: Here the keywords are not quoted; this is not RRA60 compliant, but usable as an extension of NASE A60.]

```
begin
        procedure jdev ( i, n, s, x );
        begin
                s := 0;
                for i := 1 step 1 until n do
                        s := s + x;
        end;

        integer NN;

        NN := 100;
```

```
                    begin
                            integer i;
                            real sum;
                            integer array arr [1 : NN];

                            for i := 1 step 1 until NN do
                                    arr[i] := i;

                            jdev (i, NN, sum, arr [i]);

                            outstring (1, ‘See the sum: ’);
                            outreal (1, sum);
                            outstring (1, ‘\n’)

                    end
            end
```

This will produce the output:

```
See the sum:  5050
```

The clever part is the loop-variable used in jdev which is passed by name and used as index in the array “arr [i]”.

8 Parser and Runtime Messages

[*** not yet - sorry ***]

9 About Bugs and Bug Reports

Surely there are many bugs. Of interest are any core dumps: regardless of correct input or not and compile-time and run-time misbehavior, this should never happen. Secondary are the elegance and efficiency of the implementation.

Please report bugs to Erik Schoenfelder (schoenfr@web.de). Hopefully I will have enough time to reply.

10 Bibliography and References

[RRA60] Revised Report on the Algorithmic Language Algol 60.
Communications of the ACM

Table of Contents

1 The Goal of the Interpreter 1

2 Installation of the Algol 60 Interpreter 2

2.1 Installation for Un*x 2

2.2 Installation for PC's 3

3 Algol 60 Command Line Options 4

4 Representation of Algol 60 Code 5

5 Builtin Functions 6

5.1 Mathematical and conversion functions 6

5.2 Input / Output via Channels 6

5.3 String related functions 6

5.4 Output and Input of numbers 7

5.5 Variable formatted output 8

6 C-code creation 9

7 Some Examples 10

8 Parser and Runtime Messages 12

9 About Bugs and Bug Reports 13

10 Bibliography and References 14