jff-a2c

A simple demonstration compiler for ALGOL 60

User Manual of Version II

Jan van Katwijk
Faculty of Electrical Engineering, Mathematics and Computer Science
TU Delft
The Netherlands
J.vanKatwijk@ewi.tudelft.nl

May 29, 2004

1 Introduction

There is no doubt that nowadays, ALGOL 60 is not one of the mainstream programming languages. Nevertheless, the language contains a number of interesting features for discussing language semantics. Apart from that, ALGOL 60 as a language was a landmark in the programming language world, and is seen by many scientists as a significant improvement to many of its successors.

In discussions with students, it became clear that many of the students hardly appreciate a strictly formal approach to language semantics. Experience shows that a more operational approach to semantics, e.g. semantics expressed in another language well-understood, is often preferred. After showing semantics, that are expressed in terms of 'C', of many constructs in a variety of programming languages, and after explaining some of the semantics of the ALGOL 60 language, it became clear that a simple tool for generating a systematic translation from the various constructs in ALGOL 60 (i.e. by name parameters, formal procedures, value arrays, switches etc) would be helpful and interesting.

About 25 years ago, the author wrote an ALGOL compiler for use on the PDP-11. Recently, the author was looking for a toy project in his spare time, he decided to write a simple ALGOL to C translator, with as particular features:

- the structure of the resulting C program should reflect the structure of the ALGOL source program, i.e. ALGOL structures should be mapped as closely as possible onto equivalent C constructs. Procedures in ALGOL should be mapped upon procedures in C, data structures in ALGOL (arrays and variables) should be mapped upon arrays and variables in C, and control structures in ALGOL 60 should be mapped as faithfully as possible on their C equivalents.

- ALGOL operators should not be hard-wired in the language. A translator that allows students to define the operators, with own chosen priorities is helpful in discussing syntax and semantics.

The implementation was a fun (spare time) project, therefore, the translator is named *jff-a2c*, where *jff* stands for Just For Fun. The current translator fully implements the ALGOL 60 language as described in the "Modified report on the Algorithmic Language ALGOL 60".

This documents acts as user manual and provides a brief description of the mappings by means of examples. In section 2 we will give a brief overview of the Operating Environment, in section 3 we

discuss the installation and running of the translator. In section 4 we describe representation issues of the implemented language, the standard environment and the operator file. In section 5 we discuss a number of mapping examples, and in section 6, we briefly discuss the structure of the compiler.

2 Operating environment and performance

The translator is developed under Linux and all software has been tested under Cygwin as well. The implementation language is C, the target language is plain C.

The translator is shown to be capable of compiling an artifically generated 1 million line ALGOL program, although the resulting C program could not be compiled anymore by the local C compiler due to space constraints. Compilation speed is about 1000 lines a second on a 1.3 GHz machine with sufficient memory.

3 Installing and running the translator

3.1 Installing the translator

The translator sources are distributed as a tarball. Untarring will create a directory *jff-a2c*. Running configure, make and make install will install the compiler. Make install might require root permissions. The following files will be installed

- *jff-algol*, a driver program for the translator;
- *jff-a2c*, the front end, translating ALGOL sources to C sources;
- *lib_jff.a*, the run time support functions;
- *operator*, a file containing the specifications and definitions of the ALGOL operators;
- *prelude*, a file containing the ALGOL source for the environmental block;
- *jff_header.h*, an include file for translating the generated C program.

IMPORTANT: jff_header.h contains some machine dependent macros. Before running *make*, it is best to run *make gc* and *make jff_header.h*.

3.2 Running the translator

After installation, the translator can be run by invoking

```
jff-algol {options} filename [{options} filename]*
```

The filename(s) should end with ".alg", the translator will translate the programs *filename*, and compile and link the resulting C programs into executable files named as the sources, without extension. Also by default the translated ".c" program and a generated ".h" program will be generated and kept. Options are

- '-s filename': the file *filename* will be used as operator file rather than the default one;
- '-p filename': the file *filename* will be used as prelude file rather than the default one;
- '-h filename': the file *filename* will be used rather than the predefined file *jff_header.h*";
- '-f filename': the file *filename* will be used rather than the predefined library file *lib_jff.a*;

- '-o': compilation will be stopped after running the Algol to C front end;
- '-r': the generated ".c' and ".h" file will be deleted;
- '-i': the generated ".c" file will not be processed by *indent*;
- '-v': the system will print the commands with which the front end and c compiler are called during processing.

Since there might be systems where *indent* is not available, the compiler checks on the existence of */usr/bin/indent* and skips indenting if no such program exists.

4 Language Representation, Prelude and Operators

As stated earlier, the translator implements the full language as described in "Modified Report on the algorithmic language ALGOL 60".

Keywords are written with lower case letters and are either stropped or non-stropped. The first non blank character in the file to be translated, determines the mode. If this first character is a single quote ('), stropped mode is assumed, otherwise non-stropped mode. In stropped mode, blanks and control characters are ignored, apart from their use in strings and character denotations. In non-stropped mode, blanks and control characters act as delimiter.

Keywords The predefined keywords are given below.

```
c_procedure
c_proc
array
begin
do
end
else
for
false
true
goto
go
to
if
label
procedure
proc
step
switch
string
then
until
value
while
comment
```

The keyword *c_procedure* or *c_proc* only can be used in the prelude (see section 4.1). The operator file furthermore defines the type names

```
integer
real
boolean
```

These type names act also as reserved words but can be changed by the user.

Identifiers Identifiers may contain both upper and lower case characters, digits and the underscore character ('_'). An identifier will start with a letter. The case of the letters is significant.

Strings Strings are sequences of characters enclosed between '"' characters. Within a string, the '\' character can be used as escape, i.e. '\"' is a character in the string, not terminating the string.

Character denotations Although ALGOL 60 does not support the notion of character denotation, the implemented language does. In general '&x' denotes the character 'x'. Furthermore

- '&\s' denotes a white space;
- '&\t' denotes a tab;
- '&\n' denotes a newline;
- '&\f' denotes a formfeed character.
- '&\0' followed by octal characters (not exceeding the value 255) denotes a character with that value.

Operators The predefined operator symbols are

```
+
-
*
/
div
^
<
>
<=
>=
^= (\=, !=)
and
or
! (not)
equiv
implies
```

4.1 The standard environment

Simple and mathematical functions The simple mathematical functions are according to the MR.

```
real procedure abs (e); value e; real e;
integer procedure iabs (e); value e; integer e;
integer procedure sign (x); value x; real x;
integer procedure entier (e); value e; real e;
real procedure sqrt (e); value e; real e;
real proc sin (a); value a; real a;
real proc cos (a); value a; real a;
real procedure arctan (e); value e; real e;
real procedure ln (e); value e; real e;
real procedure exp (e); value e; real e;
real procedure epsilon;
real procedure random;
real procedure maxreal;
real procedure minreal;
procedure stop;
procedure fault (str, r); value r; string str; real r;
```

Standard input-output procedures The standard input/output is according to the MR.

```
integer procedure length (str); string str;
procedure space (channel); value channel; integer channel;
procedure newline (channel); value channel; integer channel;
procedure outstring (channel, str);
value channel; integer channel; string str;
procedure outterminator (channel); value channel; integer channel;
procedure inchar (channel, str, v); value channel; integer channel, v; string str;
integer procedure outchar (channel, str, v); value channel, v; integer channel, v; string str;
procedure outinteger (channel, v); value channel, v; integer channel, v;
procedure outreal    (channel, v); value channel, v; integer channel; real v;
procedure ininteger (channel, v); value channel; integer channel, v;
procedure inreal (channel, v); value channel; integer channel; real v;
```

Non standard input-output procedures ALGOL uses *channels* to indicate the device or file on which the input-output operation is done. In this implementation, a channel is a number between 0 and 31, that can be associated with a file (or device). The procedure *open_file* binds a file (or device) to a channel, the procedure *close_file* disconnects the channel from file or device. Standard, channel 0 is bound to standard input, channel 1 and 2 to standard output.
Reading from a channel not bound to a file, will bind that channel to standard input. Correspondingly, writing to a channel, not bound to a file or device, will bind that channel to standard output. Operations on non channels outside the range *0 .. 31* will lead to a return value (if appropriate) of *-1*.

```
integer procedure open_file (str, kind); value kind; string str; integer kind;
integer procedure close_file (channel); value channel; integer channel;
```

The *kind*, specified in the *open_file* procedure is

- 'r' for opening the file for reading;
- 'w' for opening the file for writing;
- 'a' for opening the file in append mode.

Any other value for *kind* will cause the file to be opened for reading.

- *open_file* will return an integer value in the range of *3 .. 31* if the operation is successful. It will return *-1* if opening the file is unsuccessful, and it will return *-2* if the number of open files exceeds the limit.
- *close_file* will return the value of *channel* if the operation is successful, otherwise, *-1* is returned.

Apart from the standard procedures, two more character input-output procedures are available. *raw_in* will read a character and returns its ASCII value, similarly, *raw_out* will take a character value and output it to the specified device.

```
integer procedure raw_in (channel); value channel; integer channel;
procedure raw_out (channel, val); value channel, val; integer channel, val;
```

Other non-standard procedures In order to ease manipulation of arrays, some procedures are provided with which information on the dimensionality and the bounds can be obtained. In general, a call to *x_lwb* (where *x* stands for either *i*, *r* or *b*) will return the value of the i-th lowerbound of the array. In case the value of the second parameter *i* is out of bounds of the array given as parameter, the procedure returns the number of dimensions of the varray. Similarly for the *x_upb* procedures.

```
integer procedure i_lwb (a, i); value i; integer array a; integer i;
integer procedure b_lwb (a, i); value i; boolean array a; integer i;
integer procedure r_lwb (a, i); value i; real array a; integer i;
integer procedure i_upb (a, i); value i; integer array a; integer i;
integer procedure b_upb (a, i); value i; boolean array a; integer i;
integer procedure r_upb (a, i); value i; real array a; integer i;
```

4.2 Operators and the operator file

As stated earlier, operators are specified and defined in the file *operator*. In this file, the parameter profile, the return type and the translation scheme is specified for each operator, available for translating the ALGOL source. The operator file has three sections

- the *typedef* section;
- the *binary operator* section, and
- the *unary operator* section.

The typedef section The *typedef* section specifies the type names as used in the ALGOL program and, the internal type names of the standard types, and the typenames as used in the resulting C program.

```
typedef integer : A_int : int;
typedef real: A_Real : double;
typedef boolean: A_Bool: char;
end;
```

- The first identifier following *typedef* in each line, is the keyword for the type as appearing in the ALGOL source (including the prelude);
- The second identifier is the translator internal predefined name for the type;
- the third identifier is the name of the type used in the resulting C program.

In principle, a user might want to modify the names of the types as appearing in the ALGOL program and in the resulting C program. Currently, however, the run time system expects the C-types *double*, *int* and *char* to be used.
The translator will read the operator file and expects definitions for *A_int*, *A_Bool* and *A_Real*. If any of these definitions is missing, compilation will halt.

The binary operator section All binary operators, that can be used in the ALGOL program, are described and defined in this binary operator section. The structure of the section is

```
binary binary_operator_specification end
```

binary_operator_specification is defined as a sequence of one or more *binary_operator*s. A *binary_operator* is defined as

```
[ operator_symbol priority_definition {binop_spec}+ ]
```

This means, an operator symbol, an integer value specifying the priority, and one or more times a *binop_spec*.

- The *operator_symbol* is either one of the predefined operator symbols or an identifier. In case it is an identifier, it will be treated as a keyword in the ALGOL program.
- The *priority_definition* is an integer value, indicating the (relative) priority of the operator.
- The *binop_spec* is a description of a parameter profile for the operator, the type of the return value and a C-string, used to translate the operator. It takes the form

  ```
  type type type [constant] string
  ```

- As one might expect, the first occurrence of the *type* indicates the type of the left operand, the second occurrence of the *type* indicates the type of the right operand, and the third occurrence indicates the result type.

- In case *constant* is specified, it is assumed that the operator, when applied on constant operands, yields a constant result, which will be computed during compilation of the resulting C program. It must be realized that this has implications especially for array bounds, treating arrays with constant bounds is different from arrays with non-constant bounds.
- The *string* models the code generated for the operator. It may contain the indicators %L and %R, that are used as place holders for the code generated for the left resp. the right operand.

As an example, consider the definition of a new operator *bit_and*

```
[bit_and 7 {{integer integer integer "((%L) & (%R))"}}]
```

The operator *bit_and* is apparently only defined for integer operand values. It returns an integer value and its implementation is the & operator of C.

Notice that both %L and %R act as *placeholder* for the operand code. In case either of those symbols appears more than once in the string, code for the corresponding operand will be replicated.

The unary operator section The section for the specification of unary operators has a structure similar to that of the binary operators.

```
unary unary_operator_specification end
```

Obviously, where in binary operator specifications two operand types have to be specified, here just one operand type is to be specified. As an example, consider the definition of the operator *fac*

```
[ fac 10 {{integer integer "fac (%L)"}
          {real real "Rfac (%L)"}} ]
```

An operator *fac* is defined, its relative priority is 10. If an *integer* operand is provided for, the return value will be of type *integer* and its implementation is through a call to the C function *fac*. If a *real* operand is provided for, the return value will be of type *real*, and the function *Rfac* is called in run time.

The standard operator file The operator file as used in the distribution reads

```
typedef integer:  A_int:   int;
typedef boolean:  A_bool:  char;
typedef real:     A_real:  double;
typedef void:     void:    void;
end;

binary [ + 7  ((integer integer integer constant "(%L) + (%R)")
                  (integer real real constant "((double)(%L)) +  (%R)")
                  (real real real constant "(%L) + (%R)")
                  (real integer real constant "(%L) + ((double)(%R))"))]
         [ * 8 ((integer integer integer constant "(%L) * (%R)")
                  (integer real real constant "((double)(%L)) * (%R)")
                  (real integer real constant "(%L) * ((double)(%R))")
                  (real real real constant "(%L) * (%R)")) ]
         [ ^ 9 ((integer integer integer "(__ipow(%L,%R))")
                  (real integer real " __npow (%L, %R)")
                  (integer real real "__fpow ((double)%L, %R)")
                  (real real real "__fpow (%L, %R)")) ]
         [ / 8 ((integer integer integer constant "(%L) / (%R)")
                  (integer real real constant "((double)(%L))/(%R)")
                  (real integer real constant "(%L) /((double)(%R))")
                  (real real real constant "(%L)/(%R)")) ]
         [ - 7 ((integer integer integer constant "(%L) - (%R)")
                  (integer real real constant "((double)%L)-(%R)")
                  (real real real constant "(%L) - (%R)")
                  (real integer real constant "%L - (double)%R " ) ) ]
         [ <=  6 ((integer integer boolean "(%L) <= (%R)")
```

```
                (real real boolean "(%L) <= (%R)")
                (integer real boolean "((double)(%L))<= (%R)")
                (real integer boolean "(%L) <= ((double)(%R))")]
        [ >=  6 ((integer integer boolean "(%L) >= (%R)")
                (real real boolean "(%L) >= (%R)")
                (integer real boolean "((double)(%L)) >= (%R)")
                (real integer boolean "(%L) >= ((double)(%R))")]
        [ > 6  ((integer integer boolean "(%L) > (%R) ")
         (real real boolean "(%L) > (%R)")
         (integer real boolean "((double)(%L)) > (%R)")
         (real integer boolean "(%L) > ((double)(%R))")]
        [ < 6  ((integer integer boolean "(%L) < (%R) ")
         (real real boolean "(%L) < (%R)")
         (integer real boolean "((double)(%L)) < (%R)")
         (real integer boolean "(%L) < ((double)(%R))")]
        [ = 6  ((integer integer boolean "(%L) == (%R)")
                (real real boolean "(%L) == (%R)")
                (boolean boolean boolean "(%L) == (%R)")
                (integer real boolean "((double)(%L)) == (%R)")
                (real integer boolean "(%L) == ((double)(%R))" )]
        [ ^= 6 ((integer integer boolean "(%L) != (%R)")
                (real real boolean "(%L) != (%R) ")
                (boolean boolean boolean "(%L) != (%R)")
                (integer real boolean "((double)(%L)) != (%R)")
                (real integer boolean "(%L) != ((double)(%R))" )]
        [ div 4 ((integer integer integer constant  "(%L) /(%R)"))]
        [ equiv 1 ((boolean boolean boolean "(%L) == (%R)")) ]
        [ implies 2 ((boolean boolean boolean "(((%L) & (%R)) || (!(%L) & !(%R)))")) ]
        [ and 4 ((boolean boolean boolean "(%L) && (%R)"))]
        [ or 3 ((boolean boolean boolean "(%L) || (%R)"))]
end;

unary   [ + 10 ((integer integer constant "%L")
                (real real constant "%L")) ]
        [ - 10 ((integer integer constant "-(%L)")
                (real real constant " -(%L)")) ]
        [ not 5 ((boolean boolean constant "!(%L)"))]
end;
```

4.3 Run time support

After successful translation of the ALGOL source program, the default operation is to compile and link the resulting C program to the precompiled run time support functions, and the appropriate C libraries. The installation procedure will translate the source of the runtime system (in the source file *run_time.c* and will create a library *lib_jff.a*.

The runtime support package is NOT completely machine independent, in a number of situations, the parameters are searched in a certain order. In build time, a function is executed that will generate the appropriate definitions as used in the runtime package. It must be realized however that the author does not have had any experience with machines with increasing addresses of subsequent parameters.

The run time support package contains:

- runtime procedures for local management and array (data) management;
- a series of functions to help in parameter transfer for calls to formal parameters;
- an implementation of the procedures that are defined in the standard prelude, as far as they are not directly available in a C library.

It is not advised to modify the procedures from the first two categories category.

Dynamic array management Arrays with non static bounds are dealt with in run time. At run time a stack (implemented through a linked list) is built, each stack element relates to an invocation of a procedure where arrays are allocated dynamically. Both *__deallocate* and *__jff_longjmp* deal with removing entries from this stack (while deallocating space belonging to the activation record, identified through the stack element). The stack orientation is important for

- handling the parameters of *__jff_element_address*. The values of the indices are addressed based on the addres of the first actual parameter. It is assumed that the index values are integer.
- long jumps. The target level of the jump is determined by an address comparison between the link pointer that is stored in the run time simulated stack as identifying element for the arrays belonging to this activation, and the address of the jmp buffer.
- handling the parameters of a call to a formal procedure (see section below).

Handling parameters for calls to formal procedures For a call to a formal procedure, run time procedures verify the correctness of the parameters and extract the required value. It is assumed that all actual parameters to the call to the envelope procedure that are to be transferred to the formal procedure that is being called, are of type *(char *)*.

Adding procedures to the prelude For each procedure, specified in the prelude to be a *c_procedure* or an implementation of a user defined operator, a C procedure with the correct parameter profile has to be provided. Extending the prelude takes two steps:

- the specification of an ALGOL procedure, say *X*, is added to the prelude. The specification is followed by the keyword *c_procedure*;
- the implementation of a C procedure with the same name *X* is added to the file *run_time.c*

Then, invoking *make* and *make install* will compile and install the modified translator.

In case compilation and linking of the C output of the compiler is done manually, one should specify *-lm* in order to include the appropriate C library in the linking process. This library is required for basic numerical operations.

4.4 Known bugs

One bug is known, passing a real parameterless procedure as a by name integer parameter will not give the correct result.

5 Mapping and semantics

As stated in the introduction, one of the objectives of the translator is to translate the various structures in an ALGOL 60 programs into similar structures in a resulting C program. In this section, we provide 12 examples with increasing complexity, each of them emphasizing a particular ALGOL 60 construct, and we show and briefly discuss the resulting C code. The code for each of the examples is taken from the compiler output and only edited for readability[1].

[1]It must be realized that this compiler output is output from an earlier version of the compiler. In details, the code generated by this version of the compiler, may deviate

Example 0 Standard C text books start introducing the language by a program that will print *Hello world*. In ALGOL, this would look like

```
begin
   outstring (1, "Hello world\n");
end;
```

The generated program looks as given below.

```
#
// jff_algol (Jan van Katwijk)
// Version 1.0
// input file: example0.alg
// Compiled at Tue Sep  2 16:42:57 2003

#include <stdio.h>
#include "example0.h"
// The main program
int
main ()
{
  char *LP = (char *) NULL;
  outstring (1, "Hello world\n");
}
```

Example 1 As a first real example, we show a simple program with a well known procedure definition, the procedure *fac*. The ALGOL source reads as follows:

```
begin
   integer i;
   integer procedure fac (n); value n; integer n;
      fac := if n < 1 then 1 else n * fac (n - 1);

   outstring (1, "Example 1\n");
   for i := 1 step 1 until 10 do
   begin
      outinteger (1, i); space (1); outinteger (1, fac (i)); newline (1)
   end;
end;
```

The translated program is given below.

```
#
#include        <stdio.h>
#include "example1.h"
//      Code for the global declarations

int i_40;                       // i declared at line 2
```

The variable i in the ALGOL program was declared in the outermost block, in the C program it is mapped upon a global variable. The name of the variable in the C program is composed by combing its name as it appears in the ALGOL source, with the number of the block in which the variable is declared.

```
// body for C-style function/procedure fac (_fac_0) on level 1
int _fac_0 (int n) {
  int __res_val;
  __res_val = ((n) < (1)) ? 1 : (n) * (_fac_0 ((n) - (1)));
  return __res_val;
}                               //    end of code for fac (_fac_0)
```

For a class of simple procedures, a class to which the procedure *fac* belongs, the compiler generates a straight forward C function. A translated ALGOL *function* procedure contains a local variable *__res_val*, acting as container for the function value. The body of the main program closely resembles the statements of the ALGOL program, it is given without further comment.

```
// The main program
int main () {
  char *LP = (char *) NULL;
  {
    outstring (1, "Example 1\n");
    for (i_40 = 1; (i_40 - (10)) * sign ((double) 1) <= 0; i_40 += 1)
      {
        outinteger (1, i_40);
        space (1);
        outinteger (1, _fac_0 (i_40));
        newline (1);
      }
  }
}
```

Example 2 In the second example, we show how non local variables are accessed.

```
begin
   integer h;
   procedure do_it;
   begin
      integer i, j;
      procedure increment;
      begin
         i := i + 1;
         h := h + 1;
      end;
      i := j := h := 0;
      for j := 1 step 1 until 100 do increment;
   end;
   do_it;
end;
```

The source program contains a procedure *do_it*, with contained in it, a local procedure *increment*. The latter modifies the values of the global variable h and the variable i that is local to the procedure *do_it*. The translated version of the program is given below.

```
#
#include        <stdio.h>
#include "programma.h"
//      Code for the global declarations

int h_40;                          // h declared at line 2
// code for function/procedure do_it (_do_it_0) on level 1
void _do_it_0 () {
  struct __do_it_0_rec data_for_do_it;
  struct __do_it_0_rec *LP = &data_for_do_it;
  {                                  // code for  block at line 5
    (LP)->i_43 = (LP)->j_43 = h_40 = 0;
// translation of step until element
    for ((LP)->j_43 = 1;
         ((LP)->j_43 - (100)) * sign ((double) 1) <= 0;
         (LP)->j_43 += 1)
      _increment_1 (LP);
  }                                  // end of code for block starting at 5
}                                  //    end of code for do_it (_do_it_0)
```

For the translation of the procedure *do_it* a *record* is defined *__do_it_0_rec* that contains the data local to the invocations of the procedure. The specification of the record (in fact a somewhat loose interpretation of the term activation record) is contained in the include file and given below

```
struct __do_it_0_rec {
   char *l_field;
   int i_43;// i declared at line 5
   int j_43;// j declared at line 5
};
```

The so-called link pointer *LP*, a variable local to the procedure, will point to the record on invocation of the procedure. Since the ALGOL procedure *do_it* has two local variables, *i* and *j*, the record contains fields implementing these variables.

For each procedure the translator will generate a specification of such an activation record, that will contain all data local to the procedure, except for the variable *LP* which is used as the link pointer. Since parameters also should be addressable through the link pointer, the record contains a field for each parameter, and on initialization of the procedure, these fields are initialized.

Two fields are particular to the record

- a field *l_field*, that will contain the static link (i.e. a pointer to the "activation record" of the statically enclosing procedure);
- a field *res_val* that will contain the function value. Notice that in example 1, the procedure *fac* was translated to a recordless function, and a local variable *__res_val* was declared for that purpose.

The variable *i* and *j* are local to *do_it* and therefore stored in this record. Access to these variables is through the *LP* pointer to the reord. Since the procedure *increment* has to access the record of the activation of *do_it*, the link pointer *LP* in *_do_it_0* is passed as parameter to *_increment_1*, the translation of the procedure *increment*.

```
// code for function/procedure increment (_increment_1) on level 2
void _increment_1 (struct __do_it_0_rec *ELP) {
   struct __increment_1_rec data_for_increment;
   struct __increment_1_rec *LP = &data_for_increment;
   LP->l_field = ELP;
   ((struct __do_it_0_rec *) (LP->l_field))->i_43 =
            ((struct __do_it_0_rec *) (LP->l_field))->i_43 + 1;
   h_40 = h_40 + 1;
}                                //    end of code for increment (_increment_1)
```

Access to the variable *i* in the procedure *do_it* is now through the link field. The main program remains (almost) trivial as can be seen below.

```
// The main program
int main () {
   char *LP = (char *) NULL;
   {                                // code for  block at line 2
      _do_it_0 ();
   }                                // end of code for block starting at 2
}
```

Example 3 In the third example, we demonstrate the mapping of arrays.

```
begin
   integer array A [1:10];
   integer i;
   outstring (1, "Example 3: simple static arrays\n");
   for i := 1 step 1 until 10 do A [i] := i * i;
   for i := 1 step 1 until 10 do
   begin
      outinteger (1, i); space (1); outinteger (1, A [i]); newline (1);
   end;
end;
```

The array *A* is an array with static bounds, and therefore is mapped onto a C array, as shown below

```
#
#include        <stdio.h>
#include "programma.h"

//      Code for the global declarations
int A_40[10 - 1 + 1];                // A declared at line 2
int i_40;                        // i declared at line 3
```

```
// The main program
int main ()
{  char *LP = (char *) NULL;
   {                                // code for  block at line 2
    outstring (1, "Example 3: simple static arrays\n");
//  translation of step until element
    for (i_40 = 1; (i_40 - (10)) * sign ((double) 1) <= 0; i_40 += 1)
      A_40[i_40 - 1] = (i_40) * (i_40);
// translation of step until element
    for (i_40 = 1; (i_40 - (10)) * sign ((double) 1) <= 0; i_40 += 1)
      {
        outinteger (1, i_40);
        space (1);
        outinteger (1, A_40[i_40 - 1]);
        newline (1);
      }
  }                                // end of code for block starting at 2
}
```

Example 4 In this example we show the use of arrays as (by name) parameter.

```
begin
   integer array A [1:10];
   integer i;
   procedure init_array (b); integer array b;
   begin
      integer j;
      for j := i_lwb (b, 1) step 1 until i_upb (b, 1) do
         b [i] := i * i;
   end;
   outstring (1, "Example 4: arrays by name\n");
   init_array (A);
   for i := 1 step 1 until 10 do
   begin
      outinteger (1, i); space (1); outinteger (1, A [i]); newline (1);
   end;
end;
```

Obviously, the called procedure has to get some information on the array that is passed as parameter (its dimensions, the bounds). The specification of an array in an ALGOL procedure does not provide this information, we therefore generate a *descriptor* for the array, and pass that as parameter as well

```
#
#include        <stdio.h>
#include "programma.h"
//      Code for the global declarations
int __dv0[2 * 1 + DOPE_BASE];
int A_40[10 - 1 + 1];                // A declared at line 2
int i_40;                          // i declared at line 3
```

The descriptor *__dv0* is declared as global,

```
// body for C-style function/procedure init_array (_init_array_1) on level 1
void _init_array_1 (int *Db, int *b) {
  char *LP = (char *) 0;         // dummy
  int j_43;
  {                                // code for  block at line 6
// translation of step until element
    for (j_43 = i_lwb (Db, b, 1);
         (j_43 - (i_upb (Db, b, 1))) * sign ((double) 1) <= 0; j_43 += 1)
      (*(int *) __jff_element_address (b, Db, 1, i_40)) = (i_40) * (i_40);
  }                                // end of code for block starting at 6

}                                //    end of code for init_array (_init_array_1)
```

Again, the procedure *init_array* is such that it does not require an explicitly specified activation record. The translation, i.e. *_init_array_1*, has two parameters:

- the parameter *Db* is a pointer to the descriptor of the array;
- the parameter *b* is a pointer to the array itself.

Access to an array element now is through a run time function *__jff_element_address*. This function takes as parameters

- the address of the array;
- the address of the descriptor;
- the number of indices given;
- and the expression(s) with which indexing is performed.

The main program, now, has to initialize the descriptor. It does so by setting some values and calling *__dv_init*.

```
// The main program
int main () {
  char *LP = (char *) NULL;
  {                                 // code for  block at line 2
    __dv0[0] = (1 * 256) + sizeof (int);
    __dv0[2] = 1;
    __dv0[3] = 10;
    __dv_init (__dv0);
    outstring (1, "Example 4: arrays by name\n");
    _init_array_1 (__dv0, A_40);
// translation of step until element
    for (i_40 = 1; (i_40 - (10)) * sign ((double) 1) <= 0; i_40 += 1)
      {
        outinteger (1, i_40);
        space (1);
        outinteger (1, A_40[i_40 - 1]);
        newline (1);
      }
  }                                 // end of code for block starting at 2
}
```

Notice that the run time function is only used within the procedure *_init_array_1*.

Example 5 In this example, we demonstrate handling arrays with non constant bounds.

```
begin
   procedure do_it (n); value n; integer n;
   begin
      integer array A [1 : n];
      integer i;
      for i := 1 step 1 until n do A [i] := i * i;
      for i := 1 step 1 until 10 do
      begin
         outinteger (1, i); space (1); outinteger (1, A [i]); newline (1);
      end;
   end;
   outstring (1, "Example 5: arrays with non constant bounds\n");
   do_it (10);
end;
```

The translated C text is given below

```
#
#include        <stdio.h>
#include "programma.h"
//      Code for the global declarations

// code for function/procedure do_it (_do_it_0) on level 1
void _do_it_0 (int n) {
  struct __do_it_0_rec data_for_do_it;
  struct __do_it_0_rec *LP = &data_for_do_it;
  LP->n = n;
  {                               // code for  block at line 4
    (LP)->__dv1[0] = (1 * 256) + sizeof (int);
    (LP)->__dv1[2] = 1;
    (LP)->__dv1[3] = (LP)->n;
    __dv_init ((LP)->__dv1);
    ((LP)->A_43) = (int *) __jff_allocate_array (&(LP)->__dv1, LP);
// translation of step until element
    for ((LP)->i_43 = 1; ((LP)->i_43 - ((LP)->n)) * sign ((double) 1) <= 0;
         (LP)->i_43 += 1)
      (*(int *)
       __jff_element_address (((LP)->A_43), &(LP)->__dv1, 1, (LP)->i_43)) =
         ((LP)->i_43) * ((LP)->i_43);
// translation of step until element
    for ((LP)->i_43 = 1; ((LP)->i_43 - (10)) * sign ((double) 1) <= 0;
         (LP)->i_43 += 1)
      {
        outinteger (1, (LP)->i_43);
        space (1);
        outinteger (1,
                    (*(int *)
                     __jff_element_address (((LP)->A_43), &(LP)->__dv1, 1, (LP)->i_43)));
        newline (1);
      }
  }                               // end of code for block starting at 4
  __deallocate (LP);
}                                //    end of code for do_it (_do_it_0)
```

In the translated procedure *_do_it_0* the descriptor for the array A, local to the procedure *do_it* is initiated. The descriptor is named *__dv1*, and initialization takes place through a call to *__dv_init*. Space for the (local) array is obtained by a call to *__jff_allocate_array*. The function has two parameters

- a pointer to the descriptor. This descriptor contains sufficient information for computing the required amount of space.
- the LP pointer. Its value is used for administrative purposes.

_do_it_0 also contains a call to a run time function

The space allocated for the array will be deallocated by exit from the procedure. By normal exit of the procedure, the run time procedure *__deallocate* will take care of this. In case of a goto to outside the procedure, the run time system will ensure that the space is deallocated.

```
// The main program
int main () {
  char *LP = (char *) NULL;
  {                               // code for  block at line 2
    outstring (1, "Example 5: arrays with non constant bounds\n");
    _do_it_0 (10);
  }                               // end of code for block starting at 2
}
```

Example 6 According to the Modified report, arrays may be passed by value which is demonstrated in this example.

```
begin
   integer array A, B [1: 10];
   integer i;
   procedure dummy (b); value b; array b;
   begin
      integer j;
      for j := r_lwb (b, 1) step 1 until r_upb (b, 1) do
      begin
         b [i] := b [i] + 1;
         B [i] := b [i];
      end;
   end;
   outstring (1, "Example 6: arrays by value\n");
   for i := 1 step 1 until 10 do A [i] := i * i;
   dummy (A);
   for i := 1 step 1 until 10 do
   begin
      outinteger (1, i); space (1); outinteger (1, B [i]); newline (1);
   end;
end;
```

In this example, an integer array is passed to the procedure *dummy* that expects a real array. ALGOL 60 allows this, which has some consequences for the translation. In particular, it means that we create a new descriptor for the *by value* array.

```
#
#include        <stdio.h>
#include "programma.h"
//      Code for the global declarations
int __dv0[2 * 1 + DOPE_BASE];
int A_40[10 - 1 + 1];                // A declared at line 2
int B_40[10 - 1 + 1];                // B declared at line 2
int i_40;                          // i declared at line 3
```

Both arrays, *A* and *B* make use of the same descriptor *__dv0*.

```
// code for function/procedure dummy (_dummy_1) on level 1
void _dummy_1 (int *Db, double *b) {
  struct __dummy_1_rec data_for_dummy;
  struct __dummy_1_rec *LP = &data_for_dummy;
  LP->Db = __jff_descriptor_for_value (Db, sizeof (double), LP);
  LP->b = __jff_allocate_array (LP->Db, LP);
  __typed_copy (Db, LP->Db, b, LP->b);
  {                                // code for  block at line 6
// translation of step until element
    for ((LP)->j_43 = r_lwb (LP->Db, ((LP)->b), 1);
         ((LP)->j_43 - (r_upb (LP->Db, ((LP)->b), 1))) * sign ((double) 1) <=
         0; (LP)->j_43 += 1)
      {
        (*(double *) __jff_element_address (((LP)->b), LP->Db, 1, i_40)) =
          ((*(double *) __jff_element_address (((LP)->b), LP->Db, 1, i_40))) +
          ((double) (1));
        B_40[i_40 - 1] =
          (int) ((*(double *)
                  __jff_element_address (((LP)->b), LP->Db, 1, i_40)));
      }
  }                                // end of code for block starting at 6
  __deallocate (LP);
}                                //    end of code for dummy (_dummy_1)
```

What can be seen is that in the initialization of the procedure, a descriptor is created for the array passed by value by a call to *__jff_descriptor_for_value*. This descriptor is (almost) a copy of the descriptor passed as parameter, the possible difference being the element type of the array. Space for the *by value* array is allocated using *__jff_allocate_array*. The value of the actual parameter is copied into the value array by a call to *__typed_copy*. This function will take care of conversions if needed.

```
// The main program
int main () {
  char *LP = (char *) NULL;
  {                                // code for  block at line 2
    __dv0[0] = (1 * 256) + sizeof (int);
    __dv0[2] = 1;
    __dv0[3] = 10;
    __dv_init (__dv0);
    outstring (1, "Example 6: arrays by value\n");
// translation of step until element
    for (i_40 = 1; (i_40 - (10)) * sign ((double) 1) <= 0; i_40 += 1)
      A_40[i_40 - 1] = (i_40) * (i_40);
    _dummy_1 (__dv0, A_40);
// translation of step until element
    for (i_40 = 1; (i_40 - (10)) * sign ((double) 1) <= 0; i_40 += 1)
      {
        outinteger (1, i_40);
        space (1);
        outinteger (1, B_40[i_40 - 1]);
        newline (1);
      }
  }                                // end of code for block starting at 2
}
```

Example 7 ALGOL 60 is, may be, most famous for its by name parameter passing mechanism. Its usage is best demonstrated by an example

```
begin
   integer array A [1 : 10];
   integer i;
   integer procedure sum (a, l, u, b);
   value l, u;
   integer a, l, u, b;
   begin
      integer h;
      h := 0;
      for a := l step 1 until u do h := h + b;
      sum := h;
   end;
   for i := 1 step 1 until 10 do A [i] := i * i;
   outstring (1, "Example 7: by name parameters\n");
   outinteger (1, sum (i, i_lwb (A, 1), i_upb (A, 1), A [i]));
   newline (1);
end;
```

During the execution of a procedure call, each by name parameter can be thought of as being replaced by the expression, appearing as actual parameter. In the example above, the procedure *sum* has two by name parameter, the a appears as control variable in the for loop, the b is only accessed in this loop.

The call is interesting, since the *by name* actuals are i and A *[i]*, These two parameters are related, the call effectively computes the sum of the values of the array elements of the array A.

```
   for i := 1 step l step 1 until u do h := h + A [i]
```

By name parameters are implemented using small procedures, often called *thunks* in the literature. Esssentially, each expression appearing as by name parameter is translated into two procedures

- one to compute the value of the expression the *value thunk*;
- one to compute the address of the entity passed as parameter, the *address thunk*. Obviously, if the entity entity does not have a valid left hand side evaluation, the thunk generates an error message.

```
#
#include        <stdio.h>
#include "programma.h"
//      Code for the global declarations

int __dv0[2 * 1 + DOPE_BASE];
int A_40[10 - 1 + 1];                // A declared at line 2
int i_40;                          // i declared at line 3
// code for function/procedure sum (_sum_1) on level 1
int _sum_1 (char *La, int (*Aa) (char *, int), int (*Va) (char *, int), int l,
        int u, char *Lb, int (*Ab) (char *, int), int (*Vb) (char *, int))
{
  struct __sum_1_rec data_for_sum;
  struct __sum_1_rec *LP = &data_for_sum;
  LP->La = La;
  LP->Aa = Aa;
  LP->Va = Va;
  LP->l = l;
  LP->u = u;
  LP->Lb = Lb;
  LP->Ab = Ab;
  LP->Vb = Vb;
  {                                // code for  block at line 8
    (LP)->h_43 = 0;
// translation of step until element
    for (((LP)->Aa) ((LP)->La, (LP)->l);
         (((LP)->Va) ((LP)->La, 0) - ((LP)->u)) * sign ((double) 1) <= 0;
         ((LP)->Aa) ((LP)->La, ((LP)->Va) ((LP)->La, 0) + 1))
      {
        (LP)->h_43 = ((LP)->h_43) + (((LP)->Vb) ((LP)->Lb, 0));
        outinteger (1, (LP)->h_43);
      }
    LP->res_val = (LP)->h_43;
  }                                // end of code for block starting at 8
  return LP->res_val;
}                                //    end of code for sum (_sum_1)
```

The translation of the procedure *sum*, i.e. (*_sum_1*), has 8 parameters, while the ALGOL procedure only had 4. For each by name parameter, *three* parameters are supplied in the resulting C procedure.

- a parameter (here both *La* and *Lb*) that acts as a pointer to the environment in which thunks are to be evaluated.
- a pointer to the address of the address thunk, (here *Aa* and *Ab*);
- a pointer to the address of the value thunk, (here *Va* and *Vb*).

The initialization of the for loop, i.e. $i := l$, now is translated into

```
(((LP)->Aa) ((LP)->La, (LP)->l);
```

The thunk (addressed indirectly), takes two parameters

- the link to the environment in which the code of the thunk is to be evaluated ($LP \Rightarrow La$);
- the value to be assigned, since eventually an assignment should take place to the address evaluated in the address thunk, we have chosen for the approach that assignment will take place in the address thunk.

Other calls to the implicit procedures follow the same structure.
The actual address thunk, implementing access to i, the actual parameter for which the thunk is created is straightforward, as can be seen below.

```
int A_jff_0A (char *LP, int V) {
  return i_40 = V;
}
```

The actual value thunk is also straight forward. It is beyond the scope of this manual to explain the necessity of the second parameter here.

```
int _jff_0A (char *LP, int d) {
  return i_40;
}
```

Thunks with which the actual parameter *A [i]* are implemented are given below.

```
int A_jff_1A (char *LP, int V) {
  return A_40[i_40 - 1] = V;
}

int _jff_1A (char *LP, int d) {
  return A_40[i_40 - 1];
}
```

The main program now is straight forward, as can be seen below.

```
// The main program
int main () {
  char *LP = (char *) NULL;
  {                                 // code for  block at line 2
    __dv0[0] = (1 * 256) + sizeof (int);
    __dv0[2] = 1;
    __dv0[3] = 10;
    __dv_init (__dv0);
// translation of step until element
    for (i_40 = 1; (i_40 - (10)) * sign ((double) 1) <= 0; i_40 += 1)
      A_40[i_40 - 1] = (i_40) * (i_40);

    outstring (1, "Example 7: by name parameters\n");
    outinteger (1,
                _sum_1 (LP, &A_jff_0A, &_jff_0A, i_lwb (__dv0, A_40, 1),
                        i_upb (__dv0, A_40, i_40), LP, &A_jff_1A, &_jff_1A));
    newline (1);
  }                                 // end of code for block starting at 2
}
```

Example 8 ALGOL 60 was defined in an era where goto's where not yet considered harmful. For ALGOL 60 programs, goto's, are sometimes an omportant mechanism for the control flow in the program. Consider the following program

```
begin
   procedure dummy (n); value n; integer n;
     if n <= 1 then goto L1 else dummy (n - 1);

   outstring (1, "Example 8: goto's\n");
   goto L2;
L1: outstring (1, "We terminated dummy\n");
    goto L_end;
L2: dummy (100);
L_end:
end;
```

Apart from local goto's in the main program, the body of the procedure *dummy* contains a non local goto. Local goto's are directly mapped upon local goto's in C. For non-local goto's essentially the *longjmp* mechanism from C is used. Since in non local goto's also some cleanup has to be done on data that might be allocated (e.g. arrays with non constant bounds), an extended version of the *longjmp* function is made *__jff_longjmp*.

```
#
#include        <stdio.h>
#include "programma.h"
//      Code for the global declarations

jmp_buf _L1_40;
```

Since the label *L1* is used as target for a non local goto, we create a C *jmp_buf* (here named *_L1_40*).

```
// The main program
int main () {
  char *LP = (char *) NULL;
  {                               // code for  block at line 2
    if (setjmp (_L1_40))
      goto L_L1_40;
    outstring (1, "Example 8: goto's\n");
    goto L_L2_40;
  L_L1_40:outstring (1, "We terminated dummy\n");
    goto L_L_end_40;
  L_L2_40:_dummy_0 (100);
L_L_end_40:}                       // end of code for block starting at 2
}
```

In the main program (the context of the label), the *jmp_buf* is initialized through a call to *setjmp*. The code for *_dummy_0*, implementing the ALGOL procedure *dummy* now reads

```
// body for C-style function/procedure dummy (_dummy_0) on level 1
void _dummy_0 (int n) {
  if ((n) <= (1))
    __jff_longjmp (_L1_40, 1);
  else
    _dummy_0 ((n) - (1));
}                                //    end of code for dummy (_dummy_0)
```

Example 9 This example resembles the previous one, the difference being that the label to which a goto will be made in the body of *dummy* is now passed as a parameter.

```
begin
   procedure dummy (n, L); value n, L; integer n; label L;
     if n <= 1 then goto L else dummy (n - 1, L);

   outstring (1, "Example 9: label as parameter\n");
   goto L2;
L1: outstring (1, "We terminated dummy\n");
    goto L_end;
L2: dummy (100, L1);
L_end:
end;
```

Its implementation is given without further comment.

```
#
#include        <stdio.h>
#include "programma.h"
//      Code for the global declarations

jmp_buf _L1_40;
// body for C-style function/procedure dummy (_dummy_0) on level 1
void _dummy_0 (int n, jmp_buf * L) {
  if ((n) <= (1))
    __jff_longjmp (L, 1);
  else
    _dummy_0 ((n) - (1), L);
}                                //    end of code for dummy (_dummy_0)
```

```
// The main program
int main () {
  char *LP = (char *) NULL;
  {                                // code for  block at line 2
    if (setjmp (_L1_40))
      goto L_L1_40;
    outstring (1, "Example 9: label as parameter\n");
    goto L_L2_40;
  L_L1_40:outstring (1, "We terminated dummy\n");
    goto L_L_end_40;
  L_L2_40:_dummy_0 (100, _L1_40);
  L_L_end_40:outstring (1, "We are at the end\n");
  }                                // end of code for block starting at 2
}
```

As can be seen, a label as parameter is just a single value, the address of the corresponding *jmp_buf*. Notice furthermore, that in case the label would have been passed *by name*, the earlier introduced thunks would be used to implement the parameter.

Example 10 One of the more complex issues in translating ALGOL is handling procedures that are passed as parameter. Complexity is caused by the lack of specification of the parameters as is done in later programming languages.

In the following program, a (useless) procedure *go_to* is defined to implement a goto. This procedure is passed as parameter to a procedure *dummy*, that will repeatedly call itself until the value of the first parameter is *1* or smaller.

```
begin
   procedure do_it;
   begin
      procedure go_to (L); label L; goto L;
      procedure X (p, L); label L; procedure p; p (L);
      procedure dummy (n, p, L); value n;
      integer n; procedure p; label L;
         if n <= 1 then X (p, L) else dummy (n - 1, p, L);

try_again:
      dummy (5, go_to, L2);
      outstring (1, "This is not right\n");
      goto try_again;
   end;
   outstring (1, "Example 10: Useless use of formal procedures\n");
   do_it;
L2:
   outstring (1, "Yes, we made it\n");
end;
```

The resulting C code is presented below, the translation of *do_it* does not contain surprises.

```
#
#include        <stdio.h>
#include "programma.h"
//      Code for the global declarations

jmp_buf _L2_40;
// code for function/procedure do_it (_do_it_0) on level 1
void _do_it_0 () {
  struct __do_it_0_rec data_for_do_it;
  struct __do_it_0_rec *LP = &data_for_do_it;
  {                                // code for  block at line 4
L_try_again_43:
        dummy_3 (LP, 5, LP, D_go_to_1, LP, &A_jff_0A, &_jff_0A);
        outstring (1, "This is not right\n");
        goto L_try_again_43;
```

```
  }                                 // end of code for block starting at 4
}
```

For each procedure, that is passed as parameter to another procedure, an *envelope* function is generated. This envelope then is passed as the actual parameter, and its main task is to transfer the actual parameters, given to the envelope, to the procedure that is really being passed as parameter.

A series of run time functions exist that will interpret the actual parameter information and extract the right data for the call. In this example, three functions are used

- *__get_thunk_link*, that will extract the link, needed for the evaluation of the thunks from the actuals;
- *__get_address_thunk*, that will extract the address of the address thunk of the parameter implementation;
- *__get_value_thunk*, that will extract the address of the value thunk of the parameter implementation.

```
void D_go_to_1 (char *link, int n, char *p1) {  // envelope for go_to
  if (n != 1)
    fault ("wrong number of parameters for go_to", 4);
  _go_to_1 ((char *) (__get_thunk_link (&p1, 1)),
            (char *) (__get_address_thunk (&p1, 1, 'j')),
            (char *) (__get_value_thunk (&p1, 1, 'j')));
}
// end of envelope for _go_to_1
// code for function/procedure go_to (_go_to_1) on level 2
void _go_to_1 (char *LL, jmp_buf * (*AL) (char *, jmp_buf *), jmp_buf * (*VL) (char *, int)) {
  struct __go_to_1_rec data_for_go_to;
  struct __go_to_1_rec *LP = &data_for_go_to;
  LP->LL = LL;
  LP->AL = AL;
  LP->VL = VL;
  __jff_longjmp (((LP)->VL) ((LP)->LL, 0), LP);
}                                 //    end of code for go_to (_go_to_1)
```

The call to the formal parameter as appearing in the code for the procedure X, is translated into

```
  ((LP)->p) ((LP)->Dp, 1, ('j' << 8) + 'T', (LP)->LL, ((LP)->AL), (LP->VL));
```

The *address* of the function to be called is found through the p parameter.

- The first parameter passed to this call is the link pointer, that was passed together with the address of the procedure to be called;
- The second parameter is the number of actuals given;
- the third, fourth, fifth and sixth parameter describe the parameter to be passed to the called formal one
  - descriptor information is given, telling that the parameter is of type label ('j'), and is itself a thunk ('T');
  - link information to the thunk;
  - the address of the address thunk;
  - the address of the value thunk

```
// code for function/procedure X (_X_2) on level 2
void _X_2 (char *Dp, void (*p) (), char *LL, jmp_buf * (*AL) (char *, jmp_buf *),
      jmp_buf * (*VL) (char *, int))
{
  struct __X_2_rec data_for_X;
```

```
  struct __X_2_rec *LP = &data_for_X;
  LP->Dp = Dp;
  LP->p = p;
  LP->LL = LL;
  LP->AL = AL;
  LP->VL = VL;
  ((LP)->p) ((LP)->Dp, 1, ('j' << 8) + 'T', (LP)->LL, ((LP)->AL), (LP->VL));
}                                 //    end of code for X (_X_2)
```

The translation of dummy now is again straightforward

```
// code for function/procedure dummy (_dummy_3) on level 2
void _dummy_3 (struct __do_it_0_rec *ELP, int n, char *Dp, void (*p) (), char *LL,
          jmp_buf * (*AL) (char *, jmp_buf *), jmp_buf * (*VL) (char *, int))
{
  struct __dummy_3_rec data_for_dummy;
  struct __dummy_3_rec *LP = &data_for_dummy;
  LP->l_field = ELP;
  LP->n = n;
  LP->Dp = Dp;
  LP->p = p;
  LP->LL = LL;
  LP->AL = AL;
  LP->VL = VL;
  if (((LP)->n) <= (1))
    _X_2 ((LP)->Dp, (LP)->p, (LP)->LL, ((LP)->AL), (LP->VL));
  else
    _dummy_3 ((struct __do_it_0_rec *) (LP->l_field), ((LP)->n) - (1),
              (LP)->Dp, (LP)->p, (LP)->LL, ((LP)->AL), (LP->VL));
}                                 //    end of code for dummy (_dummy_3)
```

The same applies to the thunks, generated for the label as parameter

```
jmp_buf * A_jff_0A (char *LP, jmp_buf * V) {
  fault ("no assignable object", 11);
}

jmp_buf * _jff_0A (char *LP, int d) {
  return _L2_40;
}
```

As usual, the main program is simple

```
// The main program
int main () {
  char *LP = (char *) NULL;
  {                               // code for  block at line 2
    if (setjmp (_L2_40))
      goto L_L2_40;
    outstring (1, "Example 10: Useless use of formal procedures\n");
    _do_it_0 ();
  L_L2_40:outstring (1, "Yes, we made it\n");
  }                               // end of code for block starting at 2
}
```

Example 11 The last example deals with switches. As stated earlier, ALGOL was defined in an era where goto's where not yet considered harmful, and the language provides switches as a fairly dynamic form of jump table. The following example uses switches

```
begin
   procedure go_to (L); value L; label L; goto L;
   procedure dummy (p, L); procedure p; label L;
      p(L);
   procedure do_it;
   begin
```

```
      switch L := L1, if b then L2 else L1;
      dummy (go_to, L [n]);
   end;
   integer n;
   boolean b;
   outstring (1, "Example 11: use of switches\n");
   n := 2; b := true;
   do_it;
L1: outstring (1, "Incorrect\n");
   goto L_end;
L2: outstring(1, "Correct\n");
L_end:
end;
```

Switches are translated into functions. The switch L in the example above is translated into a function

```
jmp_buf * __switch_L_47 (char *LP, int n) {  // switch L declared at line 7
  switch (n) {
    default:
    case 1:
      return _L1_40;
    case 2:
      return (b_40) ? _L2_40 : _L1_40;
    };                                  // end of switch list
}
```

According to the modified report, accessing a switch with a value out of bounds is not defined. Our interpretation is to equate an out of bounds value to *1*.

As can be seen in the code below, the procedure *go_to* is again packed into an envelope procedure for its use as parameter. Since now the parameter of *go_to* is specified as *by value*, another run time function is called that will evaluate the thunk passed as parameter to the envelope.

```
#
#include        <stdio.h>
#include "programma.h"
//      Code for the global declarations

jmp_buf _L2_40;
jmp_buf _L1_40;
void D_go_to_0 (char *link, int n, char *p1) {        // envelope for go_to
  if (n != 1)
    fault ("wrong number of parameters for go_to", 2);
  _go_to_0 (*(jmp_buf * *)(__eval_value_thunk (&p1, 1, 'j')));
}
// end of envelope for _go_to_0

// body for C-style function/procedure go_to (_go_to_0) on level 1
void _go_to_0 (jmp_buf * L) {
  __jff_longjmp (L, 1);
}                                //    end of code for go_to (_go_to_0)
```

The ALGOL procedure *dummy* has a *function* and a *by name* label as parameter, which is reflected in the declaration of *_dummy_1*, the implementation of *dummy*. *Dp* and *p* together form the values for the function, *LL*, *AL* and *VL* form the values for the thunks implementing access to the label.

```
// code for function/procedure dummy (_dummy_1) on level 1
void _dummy_1 (char *Dp, void (*p) (), char *LL,
          jmp_buf * (*AL) (char *, jmp_buf *), jmp_buf * (*VL) (char *, int))
{
  struct __dummy_1_rec data_for_dummy;
  struct __dummy_1_rec *LP = &data_for_dummy;
  LP->Dp = Dp;
  LP->p = p;
  LP->LL = LL;
```

```
  LP->AL = AL;
  LP->VL = VL;
  ((LP)->p) ((LP)->Dp, 1, ('j' << 8) + 'T', (LP)->LL, ((LP)->AL), (LP->VL));

}                                //    end of code for dummy (_dummy_1)
```

The implementation of *do_it* (*_do_it_2*) is straight forward, just a call to *_dummy_1*

```
// code for function/procedure do_it (_do_it_2) on level 1
void
_do_it_2 ()
{
  struct __do_it_2_rec data_for_do_it;
  struct __do_it_2_rec *LP = &data_for_do_it;

  { // code for  block at line 7
    _dummy_1 (NULL, D_go_to_0, LP, &A_jff_0A, &_jff_0A);
  } // end of code for block starting at 7

} //    end of code for do_it (_do_it_2)
```

Obviously, for *L [n]* as non-value parameter, thunks are defined.

```
jmp_buf * A_jff_0A (char *LP, jmp_buf * V) {
  fault ("no assignable object", 8);
}

jmp_buf * _jff_0A (char *LP, int d) {
  return __switch_L_47 (((struct __do_it_2_rec *) LP), n_40);
}
```

What remains is the main program and some global declarations, they are given below

```
int n_40;                        // n declared at line 10
char b_40;                        // b declared at line 11
// The main program
int main () {
  char *LP = (char *) NULL;
  {                                 // code for  block at line 2
    if (setjmp (_L2_40))
      goto L_L2_40;
    if (setjmp (_L1_40))
      goto L_L1_40;
    outstring (1, "Example 11: use of switches\n");
    n_40 = 2;
    b_40 = true;
    _do_it_2 ();
  L_L1_40:outstring (1, "Incorrect\n");
    goto L_L_end_40;
  L_L2_40:outstring (1, "Correct\n");
L_L_end_40:}                       // end of code for block starting at 2
}
```

6 Structure of the compiler

The compiler is a fairly straightforward one. Essentially, the parts are:

- a scanner/parser that builds a tree structured representation of the source program, together with a mapping table for the declared entities;
- a treewalker that verifies the context conditions and gathers some general information on (parts of) the program;

- a generator for the specifications (prototypes) for the C code to be generated;
- a generator for the code for declarations and functions.

6.1 The scanner/parser

In spite of the fact that tools like "lex" and "yacc" are available, we have chosen for a simple recursive descent parser. The parser is based on a weakly covering grammar, sufficiently strong though to build a tree structure of the source program. The scanner is a simple handwriten scanner with a possibility for an additional *look ahead* of one symbol. The main functions of the parser/scanner are:

- to identify the symbols appearing in the program;
- to store all symbol representations in a (binary sorted) string table;
- to create a mapping table, relating all defined occurrences of identifiers to the block in which they are defined. The mapping table is simpler than a common symbol table in that does not contain any property information of the defining occurrences. For each block a mapping table is built, (identifier, defining occurrence). The size of the table of mapping tables is determined dynamically.
- to parse the individual constructs in the source program and build parts of the tree.
- to identify syntactic errors and recover from them;
- to add some constructs to the tree. In particular, the body of a for statement with a multiple for list is made into a procedure call, while an appropriate procedure definition is added to the environment in which the for statement appears.

6.2 static semantics treewalker

After the tree is built, static semantics are verified by a recursive treewalker. Main functions of the treewalker are:

- look up the definition for the applied ocurrences of identifiers and verify that there use follows the elaboration of their declaration;
- identify the defining occurrences of the operator symbols;
- add the types (found and inferred) to the nodes of the tree;
- determine whether or not the left hand side of an assignment can be assigned to;
- determine the correctness of actual parameters;
- add some structural information to the tree, in particular, determine whether or not a parameter will need to be translated into a thunk, a set of impliciet functions for the run time evaluation of actual by name parameters;
- mark those procedures that appear as actual parameter in a procedure call, for those procedures, an envelope procedure will have to be generated.
- mark the nodes of the tree, based on predicates over the source program:
  - mark whether or not a declaration is referred to. The compiler will not generate code for entities not accessed during execution time. This marking can be improved. If a level n procedure refers to another level n procedure, but the first procedure is not referred to from outside, the second procedure is marked as being accessed.

  - mark whether or not a label is non-locally referred to. For non locally referred labels, a longjmp/setjmp construct will be constructed, labels that are only accessed locally, are just translated into labels in C;
  - mark whether or not an array needs a descriptor (a describing dope vector). Arrays with constant bounds that are not passed as parameter do not need such a descriptor.
  - mark whether or not explicit deallocation has to be done in run time for locally allocated arrays. This holds for arrays with non constant bounds.
  - mark whether or not a translated ALGOL procedure will need a static link to its environmental procedure. If for a procedure, there is no access to variables that are contained in activation records of an enclosing procedure (not the main program), there is no need for a static link.
  - mark whether or not an ALGOL procedure is sufficiently simple to allow it to be translated into a C procedure with only local variables. In those cases no explicit "activation record" will be constructed.

- to link declarations to either a global list (for those entities that will be mapped onto global entities in the resulting C program) or to a local list of declarations belonging to a procedure. Declarations will therefore appear into two lists:
  - the "normal" list of declarations belonging to the entity in which the declaration was found;
  - a second list, used to construct prototypes and activation record specifications in the third phase.

6.3 Code generation

Given that during static analysis the order of the global declarations is settled, code generation is fairly straightforward. During the first scan the prototypes are created, these prototypes are written into a ".h" file. The second scan generates the code:

- for each procedure (including thunks, switch declarations and bodies for for statements) a C procedure is output;
- Algol control structures are translated into their C counterparts as was demonstrated in the mapping section.