Turing project: ALGOL 60 Tutorial

Viktor Gsteiger
University of Basel
Matriculation Number: 18-054-700

November 21, 2020
Seminar: 58826-01 - Turing Award Winners and Their Contributions

Abstract

The difficulty of learning a new programming language is inherently great. One may have no previous experience all together, one may have some experience but with another language or one may have some knowledge about the language at hand but may have forgotten large parts of the learn things again. The difficulty of learning a programming language that is not used any more and never had great commercial success is even greater, however, in the case of ALGOL 60, I am convinced, that the effort is not without benefits. ALGOL 60 is one of the grandparents of most modern programming languages and thus a direct predecessor of the tools we use every day. It is thus important to study the roots of our tools, to learn from past experiences and correct past mistakes.

Contents

1 Introduction

This tutorial aims to give the reader an introduction into the ALGOL 60 programming language. The reader should be able to program small to mid size procedures after reading this tutorial and should be able to translate and execute the ALGOL 60 program with the help of the marst translator. This tutorial does not aim to be complete and due to the inherent difficulty of learning a programming language it does not aim to lead to success.

2 Background

ALGOL 60 was the direct successor of the International Algebraic Language (IAL or later called ALGOL 58) and was a joint effort of European as well as American computer scientists in the years 1958 to 1960. With the help of the ALGOL Bulletin, a publication edited by Peter Naur, and several conferences the ALGOL 60 report could be published in 1960. ALGOL 60 did not have great commercial success on its own, however, the concepts introduced by the language can be witnessed in programming languages until nowadays.

The GNU marst translator translates programs written in ALGOL 60 into the ANSI C 89 programming language. It is part of the GNU project and currently maintained by Andrew Makhorin and the last release dates back to 2013.

3 ALGOL 60 Environment Setup

Before we can start programming in ALGOL 6, we will need to install some prerequisites to edit, translate and execute ALGOL 60 programs.

3.1 Text Editor

To edit any kind of text document, one will need a text editor. Examples include Windows Notepad, vim, EMACS, Atom or similar text editors.

The files created with the text editor are source files with ALGOL 60 programs usually having the extension ".alg".

3.2 The C Compiler

The C Compiler translates the human readable source code into executable machine language. In the case of writing ALGOL 60 programs the C Compiler is not directly accessed by the user but rather compiles the translated ALGOL 60 program into machine code.

The C Compiler usually used is the GNU C/C++ compiler. In the following subsection I will discuss on how to install the C compiler on the UNIX based Operating Systems. It is sadly not possible for me to install it on Microsoft Windows and thus I will focus on the UNIX based OS.

3.3 C Compiler Installation

3.3.1 Installation on UNIX

The GNU C/C++ compiler is mostly already installed on UNIX systems. To check whether the compiler is already install type the following into the command line:

```
$ gcc -v
```

If the GNU compiler is already installed then something like the following will be printed out to the command line:

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr .......
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

If the GNU compiler is not installed on your UNIX system you will need to install it from an official GNU distribution. See the documentation on the download page for reference.

3.3.2 Installation on Mac OS

The easiest way to install the GNU compiler on a Mac OS X is to install the Xcode development environment provided by Apple. See the documentation on the download page for reference.

This tutorial has been written based on Mac OS and the examples have been translated and compiled on Catalina.

3.4 ALGOL 60 Translator Installation

The marst ALGOL 60 translator can be downloaded from any gnu mirror under `/gnu/marst/`. We will be using version 2.7 or marst released in 2013. Download the tar directory and uncompress it.

To install marst on your OS type the following into the command line at the location of the `marst-2.7` directory:

```
$ ./configure; make; make install
```

This should configure, build, and install the marst package. For more information see the `README` or the `INSTALL` file.

4 ALGOL 60 Program Structure

Before we introduce the building blocks involved in developing an ALGOL 60 program we will introduce an example ALGOL 60 program and its structure so that we may use it again for reference in the following sections.

4.1 ALGOL 60 Hello World Example

An ALGOL 60 program can be constructed with the following parts:

- Procedures
- Variables
- Statements
- Comments

A simple example to display various parts of an ALGOL 60 program would be the following:

```
1 begin
2     comment A first ALGOL 60 program;
3     procedure main;
4     begin
5         outstring(1, "Hello, world!\n")
6     end;
7     main
8 end
```

Listing 1: hello.alg program

The parts of the above program are the following:

1. The first line declares the block scope of our program.
2. The second line declares the procedure which we called the main procedure. The identifier of the procedure may be changed.
3. The next line is a comment which will be ignored by the translator and is used to comment on the code at hand to make it easier for fellow programmers to understand the intention of the program.
4. The `begin` keyword signifies that a block of the procedure starts here.
5. Following comes an output keyword `outstring` which displays the string given to the first output channel.
6. The seventh line calls the main procedure.

4.2 Compile and Execute an ALGOL 60 Program

We will now save the hello.alg program, translate it, compile it, and run it. The steps to do this are the following:

1. Open your text editor and type in the above program.
2. Save the file as `hello.alg`.

3. Open a command line and navigate to the directory where the above program has been saved.
4. Type `marst hello.alg -o hello.c`.
5. If there are no errors the translator creates the C file `hello.c`.
6. Compile and link the file with the following command `gcc hello.c -lalgol -lm -o hello`.
7. Run your executable `./hello`.
8. If everything worked fine you should see `"Hello, world!"` printed on the command line.

Note: The marst translator and the c compiler are rather verbose when it comes to warnings. However, all programs specified in this tutorial have been tested and work according to their specification.

5 ALGOL 60 Basic Syntax

5.1 Formal Notation

The notation used in this part of the tutorial is the Backus Naur form also used in the original AGLOL 60 report. The notation is best explained with an example:

$$<\text{ ab }>::=(|<\text{ ab }>(|\text{example}|) \qquad (1)$$

Where the sequence of character enclosed in brackets represent meta-linguistic variables which are represented by a sequence of symbols. The ::= and | signify meta-linguistic connectives. | has the meaning or. Any symbol in a formula that is neither a variable nor a connective denotes itself. Variables can be replaced with their own definition. So the example signifies a recursive rule for the formation of values of the variable <ab>. Some values for <ab> are:

example(or)((

5.2 Symbols

The basic symbols of the ALGOL 60 programming language consists of letters, digits, logical values, and delimiters. With the basic symbols of ALGOL 60 every ALGOL 60 program can be created. The basic symbols themselves have no semantic values and are combined together to create every program. The basic symbols are made up as follows:

5.2.1 Letters

$$\begin{aligned}<\text{ digit }>::=&a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|\\ &A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z\end{aligned} \qquad (2)$$

Which are used for forming identifiers and strings.

5.2.2 Digits

$$<\text{ digit }>::=0|1|2|3|4|5|6|7|8|9 \qquad (3)$$

Which are used for forming numbers, identifiers and strings.

5.2.3 Logical values

$$<\text{ logical values }>::=\textbf{true}|\textbf{false} \qquad (4)$$

Which have a set meaning as boolean logical values.

5.2.4 Delimiters

$$\begin{aligned}
<\text{delimiter}>&::=<\text{operator}>\mid<\text{separator}>\mid<\text{bracket}>\mid<\text{declarator}>\mid\\
&<\text{specificator}>\\
<\text{operator}>&::=<\text{arithmetic operator}>\mid<\text{relational operator}>\mid\\
&<\text{logical operator}>\mid<\text{sequential operator}>\\
<\text{arithmetic operator}>&::=+\mid-\mid*\mid/\mid\%\mid\hat{}**\\
<\text{relational operator}>&::=<\mid<=\mid=\mid>=\mid>\mid!=\\
<\text{logical operator}>&::===\mid->\text{(meaning }\supset)\mid\mid\text{(meaning or)}\mid\&\mid!\\
<\text{sequential operator}>&::=\textbf{go to}\mid\textbf{if}\mid\textbf{then}\mid\textbf{else}\mid\textbf{for}\mid\textbf{do}\\
<\text{separator}>&::=,\mid.\mid\#\mid:\mid;\mid:=\mid\textbf{step}\mid\textbf{until}\mid\textbf{while}\mid\textbf{comment}\\
<\text{bracket}>&::=(\mid)\mid[\mid]\mid"\mid\textbf{begin}\mid\textbf{end}\\
<\text{declarator}>&::=\textbf{own}\mid\textbf{Boolean}\mid\textbf{integer}\mid\textbf{real}\mid\textbf{array}\mid\textbf{switch}\mid\textbf{procedure}\\
<\text{specificator}>&::=\textbf{string}\mid\textbf{label}\mid\textbf{value}
\end{aligned}\tag{5}$$

All delimiters have a fixed meaning for which is mostly obvious, or else the meaning will be explained at the appropriate section.
Typographical features such as blank spaces can be inserted between symbols, however, multi-character symbols should contain no blank space.
The separator **comment** has special importance and has the purpose of writing symbols into the code that will not be translated into machine executable code. The commenting symbols are all symbols between the separator **comment** and the separator ;.

5.3 Identifiers

In ALGOL 60 an identifier has no inherent meaning, but serve the identification of variables, arrays, labels, switches and procedures. An identifier starts with a letter and is followed by zero or more letters or digits. Every combination is allowed except the previously defined delimiters. Some examples for identifiers are as follows:

```
Soup
V17a
MARILYN
```

5.4 Numbers

Numbers are used for arithmetic operations. There are two types of numbers in the ALGOL 60 programming language. Both have the same functionalities, however different ranges of size. The types of numbers are **integer** and **real**. Integers are any positive or negative combination of digits. Reals are any integer including a decimal fraction (denoted by the separator .) and/or an exponent part (denoted by the separator #). More on this in the section ALGOL 60 Data Types. Some examples for numbers are as follows:

```
0
.5384
-.083#-02
```

5.5 Strings

A string is any sequence of basic symbols not containing ". Strings are used as actual parameters of procedure, for example also the procedure `outstring` which has been included into the translation program for output functionalities. Due to the translation to a C program the strings may be coded as usual in C fashion. Escape sequences like \n are allowed. To use double quote in a string use a backslash \". Some examples for strings are as follows:

```
"This is a string"
"This is another \"string\""
```

5.6 Indentation

ALGOL 60 does not require any indentation, however, to make the code more easily readable we will be using an easy sort of indentation where the code within a `begin` and `end` will be indented by four spaces. Furthermore, if there is a label, the indentation will be also four spaces in the to the label belonging code. Additionally, if the execution blocks of for loops are larger than simple blocks, there will also be four spaces indentation on the next line.

6 ALGOL 60 Data Types

As the original ALGOL 60 document was written without a specific hardware implementation in mind the ALGOL 60 data types used in this tutorial will reflect the C language data types in sizes. Data types define how much space will be occupied in storage.

6.1 Integer Types

As mentioned in subsection Numbers integers refer to any positive or negative concatenation of numbers without any exponent or decimal fractions. The values possible are between `-32'768` to `32'768` if int is stored in 2 bytes or between `-2,147,483,648` to `2,147,483,647` if int is stored in 4 bytes. To get the exact possible size of the integer write a short program as follows:

```
inline("printf(\"%lu\", sizeof(int));")
```

The inline procedure is a pseudo procedure implemented by the MARST developers to signify what code will be one to one translated into C code.

Examples for integers in ALGOL 60 are as follows:

```
123
0
-6577
```

6.2 Real Types

Again, as mentioned in subsection Numbers reals refer to any positive concatenation of digits which may include exponents and/or decimal fractions. The values possible are between `1.2E-38` to `3.4E+38` with a precision to 6 decimal places if float is stored in 4 bytes

Examples for reals in ALGOL 60 are as follows:

```
.87
-.03#-04
-56.46
```

6.3 Boolean Types

Boolean types are the simplest types possible and adhere to the formal logic. The possible values for Boolean types are true or false.

Examples for Boolean types in ALGOL 60 are as follows:

```
true
false
```

7 ALGOL 60 Expressions

The main constituents of any ALGOL 60 programs describing algorithmic processes are arithmetic, Boolean, and designational, expressions. These expressions contain logical values, numbers, variables, function designators, and operators.

7.1 Variables

A variable represents a single values. This value can be used in expressions and can be used to form other values and may be changed by means of assignment statements. Array expressions with their identifiers are also considered variables. The identifiers of arrays are enclosed in subscript brackets []. The type of a variable is defined in the declaration of the variable itself (see Type Declaration for more information on types) or for the array identifier (see Array Declaration for more information on arrays). Examples for variables are as follows:

```
beta
Q[7, 2]
a17
```

7.2 Function designators

A function designator represents a single value that can be attained by the application of a given set of rules defined by a procedure declaration (see Procedure Declaration for more information on procedures) to a defined set of parameters which can either be single values or variables. It is important to note that if the function is called without any parameter that the brackets should not be added. Examples for function desigantors are as follows:

```
Compile("Test")Stack:(P)
J(1 + s, n)
Rhesus
```

7.3 Arithmetic expressions

Arithmetic expressions, as the name says, are rules for computing numerical values. Simple arithmetic expressions are the application of the arithmetic operations of the rule upon the actual numerical values of the primaries involved in the expression. The numerical values of primaries are either simply the values given in the case of numbers or in the case of variables the currently assigned values of the variables (see Assignment Statement for more information on assignments). For functional designators it is the value received by executing the corresponding procedure.
The possible arithmetic operators are +, -, *, /, % and **. For more information on the semantics of those operators see Boolean expressions.

The normal arithmetic rules apply. Thus also the precedence from left to right with the exponent operator having the highest precedence, the multiplication or division operators the second highest and the addition or subtraction operator the third highest precedence. Expressions within parentheses are evaluated on their own and further calculated in the subsequent calculations. Examples for simple arithmetic expressions are as follows:

```
w * u - Q(S + C) ** 2
a * sin(omega * t)
```

There is also the possibility of more complex arithmetic expressions which involve Boolean expressions. In this case the value of the arithmetic expression is calculated from the Boolean expressions that are true. Examples for complex arithmetic expressions are as follows:

```
if q > 0 then U := U + V else if a * b > 17 then U := U / V else U := 0
if s then n := n - 1 else n := n + 1
```

7.3.1 Arithmetic expression types

The types of arithmetic expressions must be integer or real (see ALGOL 60 Data Types for more information on types).
The operators +, - and * have the conventional meaning and will return integer if all operands are integer, else real.
The operation <term> / <factor> and <term> % <factor> both denote division while / is defined for all four combinations of integer and real and will give a result of type real while % is only defined for two operands of type integer and will return an integer. / is defined as the multiplication of the term with the reciprocal of the factor. % is defined as the multiplication of the sign of the / division of the two operands with the largest absolute integer that is smaller than the / division of the two operands.
The operation <factor> ** <primary> means exponential, where the factor is the base and the primary the exponent.

7.4 Boolean expressions

Boolean expressions are rules for calculating logical values where the principles of evaluations are analogous to the principles of arithmetic evaluations. The possible boolean operators are !, &, |, -> or ==.

The relational operators <, <=, =, >, >=, and != are applied on two arithmetic expressions.

The precedence of the operators are first the precedence, then the relational operators, next the !, following the &, and then the |, with -> following and == coming last. The semantics of the operators are defined as follows:

b1	false	false	true	true
b2	false	true	false	true
!b1	true	true	false	false
b1&b2	false	false	false	true
b1|b2	false	true	true	true
b1->b2	true	true	false	true
b1==b2	true	false	false	true

Examples for boolean expressions are as follows:

```
x = -2
a + c > -5 & z - 6 < o
if k < 5 then s > w else h == c
```

7.5 Designational expression

Designational expressions are rules to determine the labels of an ALGOL 60 Statements. Here the evaluation happens analogous to the Arithmetic expressions. Boolean expressions can be used to introduce some conditional logic into the designation of the labels. A switch designator refers to the corresponding switch declaration and by the value given to the switch declaration the switch selects the designational expression listed in the switch declaration. More on Switch Declaration later.
Examples for designational expressions are as follows:

```
p9
Select[k-2]
if b < c then p9 else s[if w <= 0 then 5 else n]
```

8 ALGOL 60 Statements

Statements in ALGOL are the units of operation of every program. Statements will be executed in order as written, however, there are additional control mechanisms as go to statements which allow to let the control flow be directed in a more granular fashion. Statements are separated by the delimiter ;. Labels on statements allow for this dynamic control flow.
Conditional statements may also allow to select certain statements to be executed and others to be skipped.

8.1 Compound statements and blocks

Basic statements do not require the keywords begin or end and are the basic building blocks for the following compound statements and blocks. Basic statements may either be assignment statements, go to statements, conditional statements or for statements. Basic statements may also include labels.

Compound statements and block statements are of the form as follows with S denoting statements which may be again complete compound statements or blocks, L denoting labels and D denoting declarations:

Compound statements:

```
L: L: ... begin S; S; ... S; S end;
```

Blocks:

```
L: L: ... begin D; D; ... D; S; S; ... S; S end;
```

Examples for basic statements are as follows:

```
tmp := 3;
Rome: x := x + 1;
for i := 1 step 1 until len do x := x + 3;
go to Rome;
```

Examples for compound statements are as follows:

```
begin
    x := 0;
    for t := 0 step 5 until 100 do x := x + t;
        if t = x then go to CONTINUE else x := 0;
end;
```

Examples for blocks are as follows:

```
Z:   begin
         integer i, j; real t;
         for i := 0 step 1 until m do
         begin
             for j := 0 step 1 until m do
                 begin
                     z := z + i + j
                 end
         end
     end;
```

On more information on the scope rules of ALGOL 60 and the blocks see ALGOL 60 Scope Rules.

8.2 Assignment Statement

Assignment statements allow the programmer to assign values to one or more variables. The assignment operator is :=. In case of subscript variables the subscript expression is evaluated before assignment. So an assignment to a variable in an array gets assigned to only the subscript expression and not the whole array.
All variables on the left side of an assignment must be declared the same type as the values on the right side. So a Boolean value can only be assigned to a Boolean variable. If the variable is of type integer or real, the right side must be an arithmetic expression. A real value can be assigned to an integer value, however, the assignment happens with the following rounding: the largest integer that is not greater than the arithmetic expression E + 0.5.

Examples for assignment statements are as follows:

```
p := a[5] := n + 1;
f := f + 1;
V := Q > G & N;
```

8.3 Go To Statement

As mentioned in the initial text to this section, a go to statement is used to dynamically control the program flow and to interrupt the code at certain locations. The location of the successor of a go to statement is a Designational expression. So the next statement to be executed will have the same label as the designational expression.
Go to statements can not lead outside the scope of a block. For more information see ALGOL 60 Scope Rules.

Examples for go to statements are as follows:

```
go to town;
go to Rome[if p < 0 then S else S + 1];
go to if b < c then q else a[0];
```

8.4 Conditional Statement

Conditional statements are used to execute certain parts of a block or skip certain others based on Boolean expressions. The known conditional statement of ALGOL 60 is the if statement. The conditional statement of an if statement gets executed if the clause of the if statement is true, otherwise the statement gets skipped and if an else statement exists, this statement will get executed. If statements can be chained by applying another if statement within the else statement. It is important to note that between an if and else there should not be a delimiter as it is still the same statement. If one wants to execute more than one statement within an if or else branch one should create another compound statement.

Examples for conditional statements are as follows:

```
if s then n := n + 1;
if p > 0 then V: q := n + m else go to P;
comment The following is one large conditional statement;
if p == true then
begin
    if s < 0 then a := g / s else y := 5;
end
else if g != 0.0 then a := 5
else go to S;
```

8.5 For Statement

For statements execute the statement S declared right after the for statement 0 or more times. The for statement also includes a controlled variable upon which a sequence of assignments may take place. So the control flow of a for statement is to initially assign a value to the controlled variable, execute the statement within the for statement, test if the breaking condition has already been reached and then either execute the assignment and go to the statement within again or leave the for statement.
For statements could also be achieved by go to statements, however, they are much easier to read for their purpose.

Examples for for statements are as follows:

```
for q := 0 step s until n do B[q] = A[q];
for i := 1 step 10 until 100 do s := s + i;
```

While statements are technically not part of ALGOL 60, however, it is still possible to create while statements with the help of go to statements. An example is the following:

```
R:   V := V + E;
     if V > 100 then go to EXIT;
     go to R;
EXIT: V := V + 10;
```

8.6 Procedure Statement

Procedure statements exist to invoke procedures written in ALGOL 60. Procedure Declaration will be discussed later. The parameter amount upon calling a procedure must be the same as defined in the procedure declaration. The call of the procedure can have several effects, depending on the parameters passed when invoked:

- Call by value:
  The local parameters of the procedure receive the values of the parameters given at the call of the procedure. The local parameters of the procedure are assigned these values explicitly before entering the procedure body. The local parameters are treated as local variables inside the procedure. Switch identifiers or procedure identifiers can not be passed by value as they do not have an inherent value. To define a formal parameter to be passed by value the keyword `value` has to be defined at the parameter declaration of the procedure.

- Call by name:
  The local parameters of the procedure are replaced by the actual parameter. Interesting is that the parameters given at the call are not evaluated upon entering the procedure. This has interesting side effects when for example passing procedures or arrays as parameters. If arrays or procedures are given as parameters, they must have the same dimensions as the arrays used within the procedure.

Examples for procedure statements are as follows:

```
Transpose(W, i + 1);
Spur(A) Order: (7) Result to: (V);
```

9 ALGOL 60 Declarations

Declarations in ALGOL 60 serve the purpose to define properties for identifiers within a block. The statement of Compound statements and blocks was discussed previously. More on ALGOL 60 Scope Rules later. Some sort of static variable declaration is possible with the keyword `own`. Own declares a variable within a block to be the same every time the program enters this block. All other declared values are undefined. It is important that all identifiers must be declared within a program with the exception of labels and formal parameters of procedures. An identifier can not be re-declared within a block head.

9.1 Type Declaration

To declare certain identifiers to represent simple variables of a given type, one has to declare this type at the block head. As mentioned at the beginning of the section, those declarations may also be own. The variable data types possible for the type declaration are real, integer or Boolean. More on the ALGOL 60 Data Types was discussed earlier.

Examples for type declarations are as follows:

```
real s;
own integer n;
Boolean exists;
```

9.2 Array Declaration

The declaration of an array define one or more identifiers to represent a one or multi-dimensional array. The allowed types . It can furthermore be define of which type the subscripted variables are, which bounds the subscripted variables have and what the dimensions of the array are. The bound of the variables can be defined by giving an upper and a lower bound separated by :, also called bound pair list. The dimension of the array are defined by the number of entries in the bound pair list. The type of the subscripted are all of the same type if one is defined or real otherwise. The subscripted values have no connection to the identity of the variables used in the bound pair lists. Arrays are initiated with zeros or false when declared.

Examples for array declarations are as follows:

```
array value, value2 [-5:5, 10:20], s[0:100];
own integer array C[if b > 1 then 0:4 else 0:100];
Boolean array s[0:10];
```

9.3 Switch Declaration

With a switch declaration one assigns values corresponding to a switch identifier. A switch can be composed of one or more designational expressions which are associated from left to right by positive integers starting at 1. To access one of those designational expressions one has to pass the corresponding integer to access the value. The designational expressions get evaluated each time the values are accessed with the values of the variables at access time. Variables defined outside the scope within the switch is can not be accessed by the switch.

Examples for switch declarations are as follows:

```
switch S := p1, p2, p3;
switch Q := M1, M2, M3, if s > 4 then M4, else M5;
```

9.4 Procedure Declaration

To define the procedure associated with a procedure identifier, one has to declare this procedure. A procedure consists of a block of code and a heading including formal parameters of the procedure. The formal parameters will be, upon calling the procedure, either be replaced or assigned with the parameters given by the caller. For more information on Procedure Statement see the previous definitions. The procedure identifier must be named first after the keyword `procedure`. The types of the formal parameters should be named at the beginning of a procedure. It is customary to also include a commentary at the beginning of a procedure. A comment is placed between the keyword `comment` and the separator ;.

An example for a procedure declaration is as follows:

```
procedure transpose(a) Order: (n); value n;
array a; integer n;
comment Procedure to transpose a two dimensional matrix;
begin
    real w; integer i, k;
    for i := 1 step 1 until n do
        for k := i + 1 step 1 until n do
            begin
                w := a[i, k];
                a[i, k] := a[k, i];
                a[k, i] := w
            end
end
```

It is important to note that procedures may also be typed where the return type will be specified before the keyword procedure and the value to be returned should have the exact same identifier as the procedure itself.

More examples will be shown in the section Procedures.

10 Procedures

As defined previously, procedures are the basic building blocks of ALGOL 60 and can be best compared with functions in modern programming languages. As we have already heard, the parameter passing to procedures can be done in two ways, call by value and call by name. This has ramifications for the behavior of algorithms as one may easily also pass procedures to procedures to be executed and so forth. In the following we will look at some interesting procedures which should highlight some interesting aspects of ALGOL 60. The procedures introduced will get from simple to more complex.

If one would like to test out the procedures following one could call them as specified and print out the result with the dummy function `print` provided by marst.

A very simple but illustrative example of a procedure is the following. It is a procedure without any parameters that has one statement as its body. It could be called by the statement `simple` from a block within the scope of the program or by the procedure itself:

```
procedure simple;
begin
    Num := Num + 1;
end
```

The procedure can be called in the following way:

```
simple;
```

A bit more complex example would be the following. It is a typed procedure which compares two strings and returns a Boolean value:

```
Boolean procedure match(int1, int2);
value int1,int2;
integer int1,int2;
begin
    match := int1 = int2;
end
```

The procedure can be called in the following way:

```
Boolean b;
b := match(10,10);
```

Now, instead of typed procedures one could pass a parameter by name to achieve the same result:

```
Boolean procedure match(int1, int2, res);
value int1,int2;
integer int1,int2;
```

```
Boolean res;
begin
    res := int1 = int2
end
```

The procedure can then be called in the following way:

```
Boolean b;
match(10,10,b);
```

Now to something a bit more complicated we will use different things we have learned so far. We will introduce a procedure which involves an array, call by name and a for loop:

```
procedure spur(a)order:(n)result:(s);
value n;
array a; integer n; real s;
begin
    integer k;
    s := 0;
    for k := 1 step 1 until n do s := s + a[k,k]
end
```

The procedure can then be called (here with an identity matrix) in the following way:

```
array a[0:5,0:5];
integer n, i; real s;
n := 5;
for i := 0 step 1 until n do a[i,i] := 1;
spur(a)order:(n)result:(s);
```

To put additional knowledge to the test we will now look at a procedure that includes conditional statements, go to statements and the outstring procedure:

```
procedure checkEven(num);
value num;
integer num;
begin
    integer mod;
    mod := num - ((num%2)*2); comment This represents the mod operator;
    if mod = 0 then go to even else go to odd;

    even: begin outstring(1, "Is even"); go to exit end;
    odd: begin outstring(1, "Is odd"); go to exit end;
    exit:
end;
```

The procedure can then be called (here with an identity matrix) in the following way:

```
integer n;
n := -10;

checkEven(n)
```

Many more examples could be given, however, I am content that you are able to produce any number of procedures from here on with the help of this tutorial.

11 ALGOL 60 Scope Rules

ALGOL 60 was one of the first, if not the first, to introduce block scopes into a programming language. Before ALGOL 60 most languages had only the ability to introduce some kind of block rules by go to statements, however, those proved to be insufficient and the block rules were introduced. In AGLOL 60 a block is defined as follows:

```
L: L: ... begin D;D;...D;S;S;...S;S end
```

With L denoting labels, D denoting declarations and S denoting statements. The number of each is arbitrary. Now the interesting part is that the identifiers declared within a block are local to this block and will have their own space in memory. So this identifier and its value will have no existence outside of this block and any entity presented by the same identifier declared outside this block will not be accessible within the block. However, variables declares outside the block that are not re-declared within the block are accessible to the block.

Labels are the exception to this rule because labels are always local to the block within they occur. Since labels are inherently local to a block, a go to statement can not lead outside a block.

To reiterate, declarations within a block are local to this block and have the significance declared within this block and are not accessible from the outside. This is recursively true for blocks within blocks. Upon exit from a block, all variables declared within a block loose their significance.

An example to illustrate this is the following:

```
begin
    integer i, k;
    Boolean b;
    real v;
    i := 0;
    b := true;
    comment v is real declared but has no value, i is
    integer with value 0, b is Boolean with value false;
    begin
        integer i;
        v := 0.0;
        i := 10;
        comment i is integer and local to this block with
        value 10, v is real and same as in outer block but
        now with value 0.0, boolean b is still true and
        same as in outer scope;
        begin
            integer b;
            i := i + 10;
            v := 5.0;
            comment i is the same as in the previous block
            and has now value 20, v is the same as in the
            outermost block and has value 5.0, b is now of
            type int with no value;
        end
        comment i currently has value 20, b is boolean
        with value true, v is real with value 5.0
    end
    comment i has value 0, b is boolean with value true,
    v is real with value 5.0
end
```

12 Recursion

Recursion is supported in ALGOL 60 and can be achieved by procedure calls. An example is best illustrated by the well known merge sort algorithm. The following procedures are also good examples on how to implement while loops with go to statements and more:

```
procedure merge(arr, l, m, r);
value l,m,r;
array arr; integer l,m,r;
begin
    integer n1, n2, i, j, k;
    array L[0:m-l+1], R[0:r-m];

    n1 := m-l+1;
    n2 := r-m;

    for i := 0 step 1 until n1 - 1 do L[i] := arr[l + i];
    for j := 0 step 1 until n2 - 1 do R[j] := arr[m + 1 + j];

    i := 0;
    j := 0;

    k := l;

    WHILE:
        if i >= n1 | j >= n2 then go to EXIT;

        if L[i] <= R[j] then
        begin
            arr[k] := L[i];
            i := i + 1
        end
        else
        begin
            arr[k] := R[j];
            j := j + 1
        end;
        k := k + 1;

        go to WHILE;
    EXIT:

    WHILE2:
        if i >= n1 then go to EXIT2;

```

```
            arr[k] := L[i];
            i := i + 1;
            k := k + 1;

            go to WHILE2;
        EXIT2:

        WHILE3:
            if j >= n2 then go to EXIT3;

            arr[k] := R[j];
            j := j + 1;
            k := k + 1;

            go to WHILE3;
        EXIT3:

end;

procedure mergeSort(arr, l, r);
value l, r;
array arr; integer l, r;
begin
    integer m;
    m := (l+r-1)%2;
    if l >= r then go to EXIT;

    begin
        mergeSort(arr,l,m);
        mergeSort(arr,m+1,r);
        merge(arr,l,m,r)
    end;
    EXIT:

end;
```

You can also find a program to run an instance of this merge-sort algorithm in the provided materials.

13 Simple Programs

In this section I have assembled a few simple executable programs written in ALGOL 60. All programs can also be found in the additional materials.

13.1 Coin flip

In the following we will introduce a simple program that can generate random numbers with a procedure `random`, evaluate the random number by putting it into the two categories head or tail and do this 10'000 times to see if the random number generator is truly random:

```
begin
    integer procedure mod(a, n);
    value a, n;
    integer a, n;
    begin
         mod := a - ((a%n)*n)
    end;

    procedure random(lim, res);
    value lim;
    integer lim, res;
    begin
        own integer a; own Boolean b;

        if !b then
        begin
            a := 100001;
            b := true
        end;
        a := mod((a * 125), 2796203);
        res := mod(a, lim) + 1
    end;

    procedure testCoinflip(res, max);
    value max;
    integer res, max;
    begin
        integer i, tmp;

        res := 0;
        for i := 0 step 1 until max do
        begin
            tmp := 0;
            random(100, tmp);
            if tmp < 50 then res := res + 1
        end
    end;

    integer res, max; real percentage;

    res := 0; percentage := 0; max := 10000;

    testCoinflip(res, max);

    percentage := (res / max) * 100;
    print(percentage)
end
```

13.2 Greatest number

In the following program we will declare an integer array and get from the procedure `greatestNum` the largest item from the array. The algorithm described in the procedure `greatestNum` was also mentioned in the paper *Proof of Algorithms by General Snapshots* by Peter Naur. We will reuse the random procedure defined before to fill the array with random numbers.

```
begin
    integer procedure mod(a, n);
    value a, n;
    integer a, n;
    begin
        mod := a - ((a%n)*n)
    end;

    integer procedure random(lim);
    value lim;
    integer lim;
    begin
        own integer a; own Boolean b;

        if !b then
        begin
            a := 100001;
            b := true
        end;
        a := mod((a * 125), 2796203);
        random := mod(a, lim) + 1
    end;

    procedure greatestNum(A, R, N);
    value N;
    array A; integer R, N;
    begin
        integer r, i;

        r := 0;
        for i := 1 step 1 until N do
            if A[i] > A[r] then r := i;

        R := A[r]
    end greatestNum;

    array A[0:20];
    integer N, i, R;

    N := 20;

    for i := 0 step 1 until N do A[i] := random(100);

    greatestNum(A, R, N);

    print(A);
    print(R)
end
```

13.3 Fisher–Yates shuffle

In the following program we will declare an integer array and shuffle the array with the help of the procedure `shuffle`. The algorithm described in the procedure `shuffle` is named after Ronald Fisher and Frank Yates, who first described it, and is also known as the Knuth shuffle after Donald Knuth. We will reuse the random procedures defined before to get random numbers.

```
begin
    integer procedure mod(a, n);
    value a, n;
    integer a, n;
    begin
        mod := a - ((a%n)*n)
    end;

    integer procedure random(lim);
    value lim;
    integer lim;
    begin
        own integer a; own Boolean b;

        if !b then
        begin
            a := 100001;
            b := true
        end;
        a := mod((a * 125), 2796203);
        random := mod(a, lim) + 1
    end;

    procedure swap(arr, i, j);
    value i, j;
    array arr; integer i, j;
    begin
        integer tmp;

        tmp := arr[i];
        arr[i] := arr[j];
        arr[j] := tmp
    end;

    procedure shuffle(arr, h);
    comment This is an implementation of the
    Fisher-Yates shuffle;
    value h;
    array arr; integer h;
    begin
        integer j, i;
        for i := h step -1 until 1 do
        begin
            j := random(i);
            swap(arr, i, j)
        end
    end;

    array arr[0:10];
    integer h, i;

    h := 10;
    for i := 0 step 1 until h do arr[i] := i;

    print(arr);
    shuffle(arr, h);
    print(arr)
```

```

end
```

14 Conclusion

This concludes the ALGOL 60 Tutorial. I hope you could learn a thing or two in these pages and I hope that some interesting ALGOL 60 programs will come out of people reading this tutorial. It has been interesting to closely read the specifications of a programming language, to translate it into workable code and then to write a tutorial on how to write proper ALGOL 60 programs.