Two Classical Theorems Revisited

R.Morales-Bueno[1], I.Fortes[2], Ll.Mora[1] and F.Triguero[1]
[1]Dpto. Lenguajes y C.Computacion. E.T.S.I.Informática.
Campus de Teatinos. 29071 Málaga, Spain
{morales,llanos}@lcc.uma.es,triguero@ctima.uma.es
[2]Dpto. Matematica aplicada. ifortes@ctima.uma.es

1 Introduction

This paper describes the essential contents of a seminar to enhance the theoretical aspects of a first programming course. The main objective is to show that from a few control structures we can derive the rest. Another benefit for students is the acquisition of a good grasp of two classical theorems from a strictly programming point of view. These two theorems are: the Böhm-Jacopini theorem and the normal form Kleene theorem. A constructive proof for each theorem is given. In more specific terms, we show how unconditional and conditional jumps can be replaced by while statements (Böhm-Jacopini's theorem). This is the basis of structured programming. Also, we show, in a constructive way, that each program has an equivalent program with only one while statement (normal form Kleene's theorem).
In this paper, only the necessary results will be developed. Other results that complete the seminar can be seen in ([3, 4, 8]).
The next section presents the syntax and semantics of the LOOP language and some LOOP programs are given. The following section is dedicated to develop the WHILE language in a similar way. Finally, the constructive proof for each theorem is presented.

2 The LOOP Language

We begin by informally introducing the LOOP language. We study a programming language, whose programs compute certain functions. The LOOP language contains zero, increment, and assignment statements, and, in addition, a *"loop"* statement which functions much like a DO-loop. For this reason, these programs are called LOOP programs. The only data in LOOP programs are natural numbers which are stored in variables.

The syntax of LOOP programs is defined inductively: if `X1` and `X2` are names of variables, then `X1:=X2`, `X1:=X1+1` and `X1:=0` are LOOP programs. We only have the simplest of operations. We cannot add, multiply, divide, or whatever. All these will have to be implemented by programs using only these as primitive functions. Furthermore, if P and Q are LOOP programs, then P;Q is a LOOP program, as is `DO X TIMES` P `OD`. This statement is called a DO-loop (or simply loop). The variable `X` is the loop variable; the sequence P, the body of the loop statement; and `DO X TIMES` the head and `OD` the tail of the loop statement.
Regarding the semantics of LOOP programs, that is, the manner in which they are to be executed, we could say the following: assignment statements are executed in the obvious way so that the variable values are changed in the appropriate fashion. A LOOP program of the form P;Q is executed by executing the program P first, and then (with the values P leaves in the variables remaining intact) executing program Q. A program of the form `DO X TIMES` P `OD` executes program P as many times as the value of X at the beginning of the loop.
In addition to specifying the LOOP program itself, it is necessary to specify which of the variables are to be understood as input variables, and which as output variables. (Typically, there is only one output variable).
The function $f : \mathbb{N}^n \to \mathbb{N}$ computed by a LOOP program (with n input variables) is defined as follows:
$f(a_1, \ldots, a_n)$, with $a_i \in \mathbb{N}$, is the value of the output variable after the program is run with $a_i \quad 1 \leq i \leq n$ in the i-th input variable and 0 in all other variables at the beginning of execution.
In the next sections, both the syntax of LOOP and the semantics of LOOP will be defined.

2.1 Syntax of LOOP programs

In this section we will define the set of all legal LOOP programs. The syntax of LOOP is defined using Backus-Naur form (BNF).
Let $G = (N, T, R, S)$ be a grammar where:
$T = \{$`DO`,`OD`,::=,`TIMES`, $+, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$,`X`,;$\}$
$N = \{program, assignment, sent, sentences, identifier, num\}$
The set of rules R is:

1	*S*=<*program*>	::=	{(n,p,<*sentences*>)|n,1 ≤ p}
2	<*sentences*>	::=	<*sent*>
3		|	<*sentences*>;<*sent*>
4	<*sent*>	::=	<*assignement*>
5		|	`DO`<*identifier*>`TIMES`<*assignment*>`OD`
6	<*assignement*>	::=	<*identifier*>`:=0`
7		|	<*identifier*>`:=`<*identifier*>
8		|	<*identifier*>`:=`<*identifier*>`+1`
9	<*identifier*>	::=	`X`<*num*>
10	<*num*>	::=	(1|2|3|4|5|6|7|8|9){0|1|2|3|4|5|6|7|8|9}

Then, a LOOP program is defined as (n, p,P) consisting of two natural numbers and a sequence P of statements. The variables occurring in the sequence P are called the program variables. The number p specifies that {`X1,...,Xp`} are the variables used in the sequence P. The number n specifies how many of these variables are the input variables and which are: `X1,...,Xn` The only output variable is `X1`. The general idea is that, given input values, a program determines a computation on these input values, which terminates after some finite time producing a result. Each line of a program (n, p,P) contains: an assignment statement, a DO-head (`DO Xi TIMES`) or a DO-tail (`OD`).
To clarify the notation, we will use `X,Y,Z,V,I` instead of `X1,...,X5` as given variables. If the program has only one input variable, it will be `X`, and if there are two input variables, they will be `X` and `Y`, and so on. Of course, the output variable is `X`. LOOP programs are not designed to ease programming, but to be as simple as possible. LOOP programs are as powerful as any other programming language without indefinite loops.

Example 1
The IF-statement is very intuitive, and will be used frequently. The IF-statement, `IF X`$\neq 0$ `THEN` P `FI`, can be obtained from only DO-statements as it is shown:

```
Y:=0;
DO X TIMES
    Y:=0;
    Y:=Y+1
OD;
DO Y TIMES
    P
OD
```

We propose for the reader to implement the case `IF X`$\neq 0$ `THEN` P `ELSE` P' `FI` using only DO-statements.

Example 2
An example of a function that can be computed by using only LOOP programs is addition. The program (2,2,P) computes the function

$$addition : \mathbb{N}^2 \to \mathbb{N}$$

defined as $addition(x, y) = x + y$, where P is:

```
DO Y TIMES
    X:=X+1
OD
```

Now, we introduce the intuitive semantics of LOOP programs and give some examples of the power of this language.

2.2 Semantics of LOOP programs

The meaning of a program, and in particular of a LOOP program, is the set of all computations described by the program. The set of all pairs (initial value, final value) determined by these computations is the input/output relation computed by the program. This is always a functional relationship, since the computation proceeds deterministically.
The computations associated with a LOOP program $(n, p,$P) consist of the repeated application of operators of the state vector $x \in \mathbb{N}^p$ representing the values of all the p program variables.
Which operator is to be applied will be formally defined now. Let $a \in \mathbb{N}^n$ be input values. Then the state vector x is initialized to $(a_1, a_2, \ldots, a_n, 0, \ldots, 0)$. That is, input variables get their value as determined by the input; all other program variables are set to zero. The actual computation starts on this initialized state vector with the first statement of the program, i.e., initializing is not considered part of the computation. The computation proceeds sequentially in accordance with the program text and the meaning of the individual statements. There are two types of statements: assignment statements and loop statements.
The execution of assignment statements (`X`:= 0, `X`:=`Y`, `X`:=`Y`+1) transforms the current value of the state vector by replacing a component value by 0, the value of the other component and the addition of one to the value of the other component, respectively . This transformation is considered a single step in the computation.
The sequence P in a loop statement `DO X TIMES` P `OD` is to be executed as many times as indicated by `X`.
Through this way of execution, the program terminates after executing the last statement. The output value is the value of the first component in the state vector.

Example 3
In example 2 the LOOP program is (2,2,P) where `X,Y` are the input variables and `X` is the output variable. Also, (`X,Y`) are the program variables. In this example the loop statement is executed `Y` times, hence the output value of `X` is finally $x + y$. Let us assume that `Y` is 0, then the loop is not executed and the content of `X` remains x.

2.3 The Power of the LOOP Language

Now, we show the power of the LOOP language through some programs. Also, these programs and constructions will be useful for the next section where the WHILE language will be introduced. We show that all statements except indefinite loops can be constructed from LOOP programs.

Example 4
Notice that the assignment <*identifier*>`:=`<*identifier*>`-1` is not allowed in our syntax. So, at this moment we cannot compute the subtraction function straightforwardly. However, the function $f(x) = x - 1$ can be computed by a LOOP program (1,2,P) in this way:

```
DO X TIMES
    X:=Y;
    Y:=Y+1
OD
```

Initially `Y` is 0. The difference between `Y` and `X` is constant and equal to one after every execution of the loop statement; the loop statement is executed x times, hence the value of `Y` is finally x, and the value of `X` (output) is $x-1$.

Example 5
The subtraction function $f : \mathbb{N}^2 \to \mathbb{N}$ defined as $f(x, y) = x - y$ can be computed by a LOOP program (2,3,P). First,

```
DO Y TIMES
    X:=X-1
OD
```

and replacing `X:=X-1` with the corresponding code we obtain:

```
DO Y TIMES
    DO X TIMES
        X:=Z;
        Z:=Z+1
    OD
OD
```

In the construction of programs we will use statements corresponding to functions that have been proved to be computed by LOOP programs. Examples of this are `X:=Y-1`,`X:=X+Z`,`X:=Y-Z`.
We encourage the reader to compute the product function with a LOOP

program.
Now, we show how any condition that appears in an IF statement can be obtained from condition X$\neq$ 0. For example, a sentence `IF X`$\neq$ 0 `AND Y`$\neq$ 0 `THEN` P `FI` can be replaced by:

```
Z:=X+Y;
Z:=Z-1
IF Z≠0 THEN P FI
```

where `Z` is the variable associated with the condition "`X`$\neq$ 0 `AND Y`$\neq$ 0", that is, `Z`$\neq$ 0 if and only if "`X`$\neq$ 0 `AND Y`$\neq$ 0".
Also `X`$\neq$`Y` can be computed by `X-Y`$\neq$ 0 `AND Y-X`$\neq$ 0 replacing the corresponding code.
If `Z` is the associated variable to a condition C, then the associated variable to `NOT`C is T, obtained by

```
T:=0;
T:=T+1;
IF Z≠ 0 THEN T:=0 FI
```

Finally, the case `X`=`Y` can by computed by `NOT X`$\neq$`Y`, and `X`=0 by `NOT X`$\neq$ 0.
It is easy to check that the Boolean operator `OR` and all the relational operators $\leq, \geq, <, >, =$ can be computed with LOOP programs. It is known that the functions computable by LOOP programs are precisely primitive recursive functions [3, 8].
Now, let us see the restrictions of the LOOP language. Every LOOP computed function is total. So, LOOP programs cannot compute partial functions. This is a restriction of this language; another restriction is related to functions that grow very quickly.
More precisely, assume the function $f : \mathbb{N} \to \mathbb{N}$

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ \text{diverges} & \text{otherwise} \end{cases}$$

This function is computable and partial. So, no LOOP program can compute it. Examples of functions that grow very quickly (non-LOOP-computable) are the Ackermann function [1] and a version of the busy-beaver function [7].

3 The WHILE Language

In this section, we modify the LOOP language in order to include all computable functions. Let us see now what rules must be replaced and what rules must be added in order to obtain this new language. We replace the loop statement by a while statement (for this reason, we will call this language "the WHILE language") and we add the statement `X:=X-1`.

Example 6
The function $f(x)$:
$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ \text{diverges} & \text{otherwise} \end{cases}$
can be computed by the following program:

```
WHILE X≠0 DO
      X:=X+1
OD;
X:=X+1
```

The grammar for this language is similar to the one described in section 1.1; we must replace rule number 5 by a new rule 5, and add rule number 8', to those grammars:

$5 < sent > \quad ::= \quad$ `WHILE` $< identifier >\neq 0$ `DO` $< sent >$ `OD`.
$8' < assignement > \quad ::= \quad$ `X` := `X-1`

We will now see how the DO-statement, which has been suppressed in the language, can be simulated in the new language.

Example 7
The fragment of program:

```
DO X TIMES
      P
OD
```

is equivalent to:

```
Y:=X
WHILE Y≠0 DO
    P
    Y:=Y-1
OD
```

where `Y` must be a new variable not used until now.
With this example we verify that all programs written in the LOOP language can be written in the WHILE language. Example 6 shows that the class of functions computed by WHILE programs is a proper superclass of the one computed by LOOP. It can be proved that the class of functions computed by WHILE programs and the class of recursive functions are the same [3, 4, 8].

4 Böhm-Jacopini's theorem

In this section we will prove, in an intuitive manner, the Böhm-Jacopini theorem, [2]. This theorem in its original version says "...*every Turing machine is reducible into, or in a determined sense is equivalent to, a program*

written in a language which admits as formation rules only composition and iteration". In current programming terminology, this statement can be restated as follows: for every program written in an unstructured language, there is an equivalent program in a structured language.
Let us consider programs with jump instructions and, hence, labeled instructions. We assume that no other structure of iteration is allowed. First, we show with an example how to obtain an equivalent program, and then we will present the transformation algorithm.

Example 8
Assume the following program with conditional and unconditional jumps:

```
        S1
L1:      S2
        IF X≠0 THEN GOTO L3
        GOTO L4
L2:      S3
        GOTO L1
        S4
        S5
L3:      S6
        GOTO L2
L4:      S7
```

For this program, we write an equivalent program with only one while statement and without jumps. The idea is to use a counter to indicate the number of the line to be computed. If there is an assignment in the line, this assignment is performed, and then the program passes to the following line (the value of the counter increases by one unit); if there is a jump instruction in the line, we decide whether to pass to the line indicated by the jump, depending on the condition. We will use the variable name `I`, instead `Y` or `Z`, to be nearer to the nomenclature used in programming for counters. Moreover, in order to compact the program, we will write every IF statement in a single line. The equivalent program is:

```
I:=0;
I:=I+1;
WHILE I<=11 THEN
    IF I=1 THEN S1; I:=I+1 FI;
    IF I=2 THEN S2; I:=I+1 FI;
    IF I=3 AND X≠0 THEN I:=9 FI;
    IF I=3 AND X=0 THEN I:=I+1 FI;
    IF I=4 THEN I:=11 FI;
    IF I=5 THEN S3; I:=I+1 FI;
    IF I=6 THEN I:=2 FI;
    IF I=7 THEN S4; I:=I+1 FI;
    IF I=8 THEN S5; I:=I+1 FI;
```

```
    IF I=9 THEN S6; I:=I+1 FI;
    IF I=10 THEN I:=5 FI;
    IF I=11 THEN S7; I:=I+1 FI;
OD
```

We can see that constants have been assigned to variables: `I:=k` is equivalent to `I:=0; I:=I+1;...(k times)...; I:=I+1`. Also, variables can be compared with constants: `I=k` is equivalent to `Z:=k` and we use the comparison `I=Z`. The same for `I`$\leq$`k`.

Now, we will prove the Böhm-Jacopini theorem in a constructive way, as suggested at the beginning. We describe a general method to convert an unstructured program into a structured program. The algorithm is as follows:

- To number the lines from 1 upwards
- Let there be the variable `I`. (In every moment of the execution, this variable contains the number of the line that must be executed.)
- To initialize the variable `I` to one.
- To include a while statement with the following ending condition: the value of `I` is greater than the number of lines of the program being transformed.
- For each line of the program numbered with N, the original text is replaced by one of the following fragments:
  - if the line is an assignment, i.e., S:

    ```
    IF I=N THEN
        S
        I:=I+1
    FI
    ```

  - if the line is an unconditional jump, i.e., `GOTO L`, and the number of the line with label L has number N1:

    ```
    IF I=N THEN
        I:=N1
    FI
    ```

  - if the line is a conditional jump, i.e., `IF X` $\neq$ `0 THEN GOTO L`, and the line with label L has number N1:

    ```
    IF I=N AND X ≠ 0 THEN
        I:=N1
    ```

```
FI;
IF I=N AND X = 0 THEN
    I:=I+1
FI
```

- To close the while statement (`OD`).

With these steps, the reader can check that an equivalent program without jumps can be obtained for each program with jumps.

5 Kleene's theorem

Finally, we pose the following question: how many while statements are necessary to write any program? Generally, in order to reach clear programs, the number of while statements is not restricted. However, from a strictly formal point of view, would it be sufficient to use four, five or some other number? This question was long ago answered by Kleene from a theoretical point of view, stating the theorem named after him. This theorem in its original version says: *Every recursive function is expressible in the form* $\psi(\varepsilon y[R(x,y)])$*, where* $\psi(y)$ *is a primitive recursive function and* $R(x,y)$ *a primitive recursive relation and*$(x)(Ey)R(x,Y)$", [5]. ε represents the minimization and it can be proved that it is equivalent to a while statement; hence, Kleene´s theorem states that each program admits an equivalent program with only one while statement [4, 8]. We will here present a constructive and direct proof of this result. It is constructive because we develop the equivalent program, and direct because we do not use any other model of computation. The algorithm to obtain a program with only one while statement, independently of the number of while statements in the original program, is the following:

- To number the lines from 1 upwards
- Let there be the variable `I`. (In every moment of the execution this variable contains the number of the line that must be executed.)
- To initialize the variable `I` to one.
- To include a while statement with the following ending condition: the value of `I` is greater than the number of lines of the program that is being transformed.
- For each line of the program numbered with N, the original text is replaced by one of the following fragments:
  - If the line is an assignment (i.e., S):

```
IF I=N THEN
    S;
    I:=I+1
FI
```

– If the line is a while-head (`WHILE X≠0 DO`), and N1 is the line number of corresponding while-tail (`OD`):

```
IF I=N AND X≠0 THEN
    I:=I+1
FI;
IF I=N AND X=0 THEN
    I:=N1+1
FI
```

– If the line is a DO-tail (`OD`), and N1 is the line number of its corresponding DO-head (`WHILE`):

```
IF I=N THEN
    I:=N1
FI
```

- To close the while statement (`OD`).

Example 9

```
 1.WHILE X≠0 DO
 2.   S1
 3.     WHILE Y≠0 DO
 4.         S2
 5.         WHILE Z≠0 DO
 6.             S3
 7.         OD
 8.     OD
 9.     WHILE V≠0 DO
10.         S4
11.     OD
12.OD
13.S5
```

For this program according to the previous steps we obtain the following equivalent program, in which there only appears one `WHILE`:

```
I:=0; I:=I+1;
WHILE I≤13 DO
    IF I=1 AND X≠0 THEN I:=I+1 FI;
    IF I=1 AND X=0 THEN I:=13 FI;
    IF I=2 THEN S1; I:=I+1 FI;
```

```
    IF I=3 AND Y≠0 THEN I:=I+1 FI;
    IF I=3 AND Y=0 THEN I:=9 FI;
    IF I=4 THEN S2; I:=I+1 FI;
    IF I=5 AND Z≠0 THEN I:=I+1;
    IF I=5 AND Z=0 THEN I:=8 FI;
    IF I=6 THEN S3; I:=I+1 FI;
    IF I=7 THEN I:=5 FI;
    IF I=8 THEN I:=3 FI;
    IF I=9 AND V≠0 THEN I:=I+1;
    IF I=9 AND V=0 THEN I:=12 FI;
    IF I=10 THEN S4; I:=I+1 FI;
    IF I=11 THEN I:=9 FI;
    IF I=12 THEN I:=1 FI;
    IF I=13 THEN S5; I:=I+1 FI
OD
```

References

[1] Ackermann W. Zum Hilbertschen Aufbau der reelen Zahlen. *Math. Ann.* 99, pp. 118-133. 1928.

[2] Böhm C., Jacopini G.: Flow Diagrams, Turing Machines and Languages with only two Formation Rules. *Communications of the ACM*, Vol. 9, 5, pp.366-371. 1966.

[3] Davis M., Weyuker E. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science* Academic Press 1983.

[4] Kfoury A. J., Moll R. N., Arbib M.A. *A Programming Approach to Computability* Springer-Verlag. 1982

[5] Kleene S.C. General Recursive Functions of Natural Numbers. *Mathematicsche Annalen* Band 112, Heft 5 (1936) pp. 727-742.

[6] Meyer A. R., Ritchie D.M. The complexity of loop programs. *Proc. ACM Nat. Meeting* 1976,465-469.

[7] Morales-Bueno R., Perez-de-la-Cruz J.L., Fortes I., Mora L.: A new total computable function, non-computable with definite loops. Technical Report Dept. 2000/2. Languages and Computer Science. University of Malaga 2000.

[8] Sommerhalder R., van Westrhenen S.C. *The Theory of Computability: Programs, Machines, Effectiveness and Feasibility.* Addison- Wesley 1988.