

## The *NICE* Programming Language

As computer scientists, we tend to believe that computation takes place inside computers. So, when pressed to describe the limits of computation we might, after the manner of Archimedes, reply that anything can be computed given a large enough computer! When pressed further as to how this is done we probably would end up by saying that all we need in order to perform all possible computations is the ability to run programs which are written in some marvelous, nonrestrictive programming language.

Since we are going to study computation rather than engage in it, the computer is not required, just the programs. These shall form our model of computation. Thus an ideal programming language for computation seems necessary. Let us call this the *NICE* language and define it without further delay. Then we can go on to describing what is meant by computation.

We first need the raw materials of computation. Several familiar kinds of things come immediately to mind. *Numbers* (integers as well as real or floating point) and Boolean *constants* (*true* and *false*) will obviously be needed. Our programs shall then employ *variables* which take these constants as values. And since it is a good idea to tell folks exactly what we're up to at all times, we shall declare these variables before we use them. For example:

```
var  x, y, z: integer;  
     p, q: Boolean;  
     a, b, c: real;
```

Here several variables have been introduced and defined as to *type*. Since the world is often nonscalar we always seem to want some data structures. *Arrays* fill that need. They may be multidimensional and are declared as to type and dimension. In the declaration:

```
var  d, e: array[ , ] of integer;  
     s: array[ ] of real;  
     h: array[ , , , ] of integer;
```

s is a one dimensional array while h is four dimensional. As usual, elements in arrays are referred to by their position. Thus s[3] is the third element of the array named s.

So far, so good. We have placed several syntactical restrictions upon variables and specified our constants in a rather rigid (precise?) manner. *But we have not placed any limits on the magnitude of numbers or array dimensions.* Fine. Recall that we are dealing with an ideal, gigantic computer which can handle *very large* values. So why limit ourselves? To enumerate:

- a) *Numbers can be of any magnitude.*
- b) *Arrays may be declared to have as many dimensions as we wish.*
- c) *Arrays have no limit on the number of elements they contain.*

In fact, our only restriction will be that at any instant during a computation *everything must be finite*. That is, no numbers or arrays of infinite length. Huge - yes, but not infinite! (This means that the decimal representation 0.333... for one third is not allowed yet several trillion 3's following a decimal point is quite acceptable.) We should also note that even though we have a number type named *real*, these are not real numbers in the mathematical sense, but floating point numbers.

On to the next step - *expressions*. They are built from variables, constants, operators, and parentheses. *Arithmetic expressions* such as:

$$x + y \& (z + 17), \text{ or} \\ a[6] - (z \& b[k, m+2]) / 3$$

may contain the operators for addition, subtraction, multiplication, and division. *Boolean expressions* are formed from arithmetic expressions and *relational operators*. For example:

$$x + 3 = z / y - 17 \\ a[n] > 23$$

*Compound Boolean expressions* also contain the *logical connectives* *and*, *or*, and *not*. They look like:

$$x - y > 3 \text{ and } b[7] = z \text{ and } v \\ (x = 3 \text{ or } x = 5) \text{ and not } z = 6$$

These expressions may be evaluated in any familiar manner (such as by operator precedence or merely from left to right). We do not care how they are evaluated, just that we maintain consistency throughout.

In every programming language computational directives appear as *statements*. Our *NICE* language contains these also. To make things a little less wordy we shall introduce some notation. Here is the master list:

<i>E</i>	arbitrary expressions
<i>AE</i>	arithmetic expressions
<i>BE</i>	Boolean expressions
<i>V</i>	variables
<i>S</i>	arbitrary statements
<i>N</i>	numbers

The various statements in our language follow.

- a) **Assignment**. Values are assigned to variables by statements of the form:  
 $V = E$ .
- b) **Transfer**. This statement takes the form *goto N* where *N* is an integer which is used as a *label*. (A label precedes a statement. For example: 10: *S*).
- c) **Conditional**. The syntax is: *if BE then S1 else S2* where the *else* clause is optional.
- d) **Blocks**. These are groups of statements bracketed by *begin* and *end* Such as: *begin S1; S2; ... ; Sn end*

Figure 1 contains a fragment of code which utilizes all of the statements defined so far. After execution the variable *z* is set to the value of *x* factorial.

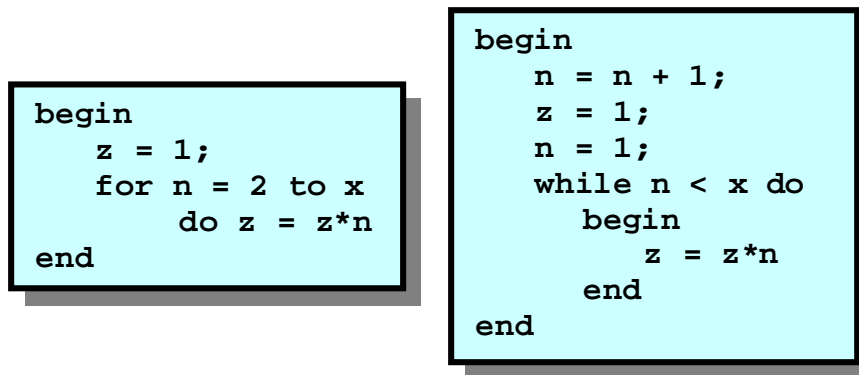
```
begin
  z = 1;
10: z = z*x;
    x = x - 1;
    if not x = 0 then goto 10
end
```

Figure 1- Factorial Computation

- e) **Repetition**. The *while* and *for* statements cause multiple repetition of other statements and have the form:

```
while BE do S
for V = AE to AE do S
```

Steps in a *for* statement are assumed to be one unless *downto* (instead of *to*) is employed. Then the steps are minus one as decrementation rather than incrementation takes place. It is no surprise that repetition provides us with a more tasteful method of computing factorials. Two methods appear in figure 2.



```

begin
  z = 1;
  for n = 2 to x
    do z = z*n
  end

```

```

begin
  n = n + 1;
  z = 1;
  n = 1;
  while n < x do
    begin
      z = z*n
    end
  end
end

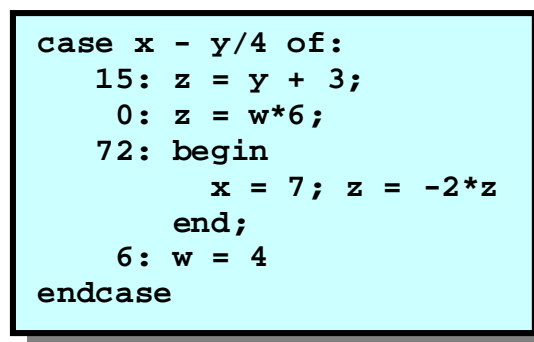
```

Figure 2 - Tasteful Factorial Computation

- f) **Computation by cases.** The *case* statement is a multiway, generalized *if* statement and is written:

```
case AE of N1: S1; N2: S2; ... ; Nk: Sk endcase
```

where the  $N_k$  are numerical constants. It works in a rather straightforward manner. The expression is evaluated and if its value is one of the  $N_k$ , then the corresponding statement  $S_k$  is executed. A simple table lookup is provided in figure 3. (Note that the cases need not be in order nor must they include all possible cases.)



```

case x - y/4 of:
  15: z = y + 3;
  0: z = w*6;
  72: begin
      x = 7; z = -2*z
    end;
  6: w = 4
endcase

```

Figure 3 - Table Lookup

- g) **Termination.** A *halt(V)* statement brings the program to a stop with the value of  $V$  as output.  $V$  may be a simple variable or an array.

Now that we know all about statements and their components it is time to define programs. We shall say that a *program* consists of a *heading*, a *declaration section* and a *statement* (which is usually a block). The heading looks like:

$$\text{program name}(V_1, V_2, \dots, V_n)$$

and contains the name of the program as well as its *input parameters*. These parameters may be variables or arrays. Then come all of the declarations followed by a statement. Figure 4 contains a complete program.

```
program expo(x, y)
var n, x, y, z: integer;
begin
  z = 1;
  for n = 1 to y do z = z*x;
  halt(z)
end
```

Figure 4 - Exponentiation

The only thing remaining is to come to an agreement about exactly what programs do. Let's accomplish this by examining several. It should be rather obvious that the program of figure 4 raises  $x$  to the  $y$ -th power and outputs this value. So we shall say that *programs compute functions*.

Our next program (in figure 5) is slightly different in that it does not return a numerical value. Examine it.

```
program square(x)
var x, y: integer;
begin
  y = 0;
  while y*y < x do y = y + 1;
  if y*y = x then halt(true) else halt(false)
end
```

Figure 5 - Boolean Function

This program does return an answer, so it also computes a function. But it is a *Boolean function* since it takes the values *true* or *false*. We might depict this one as:

$\text{square}(x) = \text{true}$  if  $x$  is a square and *false* otherwise

or we can say that the program named 'square' decides whether or not an integer is a perfect square. In fact, we state that this program *decides membership* for the set of squares.

Let us sum up what we have determined that programs accomplish when executed. They may compute functions or decide membership in sets. That's it.

So far, so good. But what if we write down something rather silly such as:

```
program nada(x)
var x, y: integer;
x = 6
```

This has no *halt* statement and thus has no output. So, what does it do? Well, not much that we can see! Let's try another in the same vein. Consider the well-known and elegant:

```
program loop(x)
var x: integer;
while x = x do x = 17
```

which does something, but alas, nothing too useful. In fact, programs which either do not execute a *halt* or even contain a *halt* statement are programs, but accomplish very little that is evident to an observer. We shall say that *these compute functions which are undefined* (one might say that  $f(x) = ?$ ) since we do not know how else to precisely describe the results attained by running them.

Let us examine one which is sort of a cross between the two kinds of programs we have seen thus far. This, our last strange example, sometimes halts, sometimes loops and is included as figure 6.

```
program fu(x)
var n, x: integer;
begin
  n = 0;
  while not x = n do n = n - 2;
  halt(x)
end
```

Figure 6 - Partially Defined Function

This halts only for even, positive integers and computes the function described as:

$fu(x) = x$  if  $x$  is even and positive, otherwise undefined

Since it halts at times (and thus is defined on the even, positive integers) we will compromise and maintain that it is *partially defined*.

To recap, we have agreed that computation takes place whenever programs are run on some ideal machine and that programs are written in our *NICE* language.

We have depended upon our computing background for the definition of exactly what occurs when computation takes place. (We could go into some detail of how statements in our language modify the values of variables and so forth, but have agreed not to at this time.) So, given that we all understand program execution, we can say that based upon a belief in our definition:

- *programs compute functions and*
- *any computable function can be computed by a program.*

The functions we often compute possess values for all of their input sets and are called *defined*, but some functions do not. Those functions that never have output are known as *undefined functions*. And finally some functions possess output values for some inputs and no output for others. These are the *partially defined functions*.

At last we have defined computation. It is merely the process of running *NICE* programs.