A Smaller Programming Language.

At this point two rather different models of computation have been presented and discussed. One, the *NICE* programming language, has a definite computer science flavor, while the other, Turing machines, comes from mathematical logic. Several questions seem rather germane.

- *which is better?*
- *is one more powerful than the other?*

The programming language is of course more comfortable for us to use and we as computer scientists tend to believe that programs written in similar languages can accomplish any computational task one might wish to perform. Turing machine programs are rather awkward to write and there is a real question about whether they have the power and flexibility of a modern programming language.

In fact, many questions about Turing machines and their power arise. Can they deal with real numbers? arrays? Can they do *while* statements? To discover the answers to our questions we shall take what may seem like a rather strange detour and look carefully at the *NICE* programming language. In this section we will find out that many of the features we hold dear in programming languages are not necessary (convenient, yes, but not necessary) when our aim is to examine the power of a computing system.

To begin, how about numbers? Do we really need all of the numbers we have in the *NICE* language? Maybe we could discard half of them.

Negative numbers could be represented as positive numbers in the following manner. If we represent numbers using *sign plus absolute value* notation, then with a companion variable for each of our initial variables we can keep track of the signs for all values that are used in computation. For example, if the variable x is used, we shall introduce another named signx that will have the value 1 if x is positive and 0 if x is negative. For example:

value	x	signx
19	19	1
-239	239	0

Doing this means that we need not have negative numbers any longer, but we shall need to exercise some caution while doing arithmetic. Employing our new convention for negative numbers, multiplication and division remain much the same (we need to pay attention to signs), but addition and subtraction become a bit more complicated. For example, the assignment statement z = x + y becomes:

```
if signx = signy then
   begin z = x + y; signz = signx end
else if x > y
   then begin z = x - y; signz = signx end
   else begin z = y - x; signz = signy end
```

This may seem a bit barbaric, but it does get the job done. Furthermore, it allows us to state that we need only *nonnegative numbers.*

[NB. An interesting side effect of the above algorithm is that we now have two different zeros. Zero can be positive or negative, exactly like some second-generation computers. But, this will not effect arithmetic as we shall see.]

Now let us rid ourselves of *real* or *floating point* numbers. The standard computer science method is to represent the number as an integer and specify where the decimal point falls. Another companion (which we shall call pointx) for each variable is now needed to specify how many digits are behind the decimal point. Here are three examples.

value	x	signx	pointx
537	537	1	0
0.0025	25	1	4
-6.723	6723	0	3

Multiplication remains rather straightforward, but if we wish to divide, add, or subtract these numbers we need a *scaling* routine that will match the decimal points. In order to do this for x and y, we must decide which is the greater number. If pointx is greater than pointy we scale y with the following code:

```
while pointy < pointx do
   begin
     y = y*10;
     pointy = pointy + 1
   end
```

and then go through the addition routine. Subtraction (x - y) can be accomplished by changing the sign (of y) and adding.

As mentioned above, multiplication is rather simple because it is merely:

```
z = x*y;
pointz = pointx + pointy;
if singx = signy then signz = 1
                 else signz = 0;
```

After scaling, we can formulate division in a similar manner.

> Since numbers are never negative, a new sort of subtraction may be introduced. It is called *proper subtraction* and it is defined as:
>
> x - y = maximum(0, x - y).
>
> Note that the result *never goes below zero.* This will be useful later.

A quick recap is in order. None of our arithmetic operations lead below zero and our only numbers are the *nonnegative integers.* If we wish to use negative or real (floating point) numbers, we must do what folks do at the machine language level and fake them!

Now let's continue with our mutilation of the *NICE* language and destroy expressions! *Boolean expressions* are easy to compute in other ways if we think about it. We do not need $E_1 > E_2$ since it is the same as:

$$E_1 \geq E_2 \text{ and not } E_1 = E_2$$

Likewise for $E_1 < E_2$. With proper subtraction, the remaining simple Boolean arithmetic expressions can be formulated arithmetically. Here is a table of substitutions. (Be sure to remember that we have changed to proper subtraction and so a small number minus a large one is zero.)

$E_1 \leq E_2$	$E_1 - E_2 = 0$
$E_1 \geq E_2$	$E_2 - E_1 = 0$
$E_1 = E_2$	$(E_1 - E_2) + (E_2 - E_1) = 0$

This makes the Boolean expressions found in *while* and *if* statements less complex. We no longer need to use relational operators here since we can we

assign these expressions to variables and then use these variables in *while* or *if* statements. Thus the only simple Boolean expressions required are:

`x = 0` and `not x = 0`

Whenever a variable such as z takes on a value greater than zero, the (proper subtraction) expression 1 - z turns its value into zero. Thus Boolean expressions which employ *logical connectives* may be restated arithmetically. For example, instead of asking if x is not equal to 0 (i.e. *not* x = 0), we just set z to 1 - x and check to see if z is zero. All of these transformations are included in the chart below and are followed by checking z for zero.

not x = 0	**z := 1 - x**
x = 0 and y = 0	**z := x + y**
x = 0 or y = 0	**z := x*y**

If we use this, Boolean expressions other than those of the form x = 0 are no longer needed.

Compound arithmetic expressions are not necessary either, correct? Of course not. We shall just break them into sequences of statements that possess one operator per statement. Now *we no longer need compound expressions of any kind!*

What next? Well, for our finale, let's remove all of the wonderful features from the *NICE* language that took language designers years and years to develop.

a) ***Arrays***. We shall just encode the elements of an array into a simple variable and use this. (And we shall leave this transformation as an exercise!)

b) Although ***while*** statements are almost the most important portion of structured programming, a statement such as:

while x = 0 do S

(recall that only x = 0 exists as a Boolean expression now) is just the same computationally as:

```
10: z = 1 - x;
    if z = 0 then goto 20;
    S;
    goto 10
20: (* next statement *)
```

c) The ***case*** statement is easily translated into a barbaric sequence of tests and transfers. For example, consider the statement:

case E of: N_1: S_1; N_2: S_2; N_3: S_3 endcase

Suppose we have done some computation and set x, y, and z such that the following statements hold true.

if x = 0 then $E = N_1$
if y = 0 then $E = N_2$
if z = 0 then $E = N_3$

Now the following sequence is equivalent to the original *case.*

```
    if x = 0 then goto 10;
    if y = 0 then goto 20;
    if z = 0 then goto 30;
    goto 40;
10: begin S1; goto 40 end;
20: begin S2; goto 40 end;
20: begin S3; goto 40 end;
40: (* next statement *)
```

d) ***if-then-else*** and ***goto*** statements can be simplified in a manner quite similar to our previous deletion of the *case* statement. Unconditional transfers (such as *goto* 10) shall now be a simple *if* statement with a little preprocessing. For example:

```
z = 0;
if z = 0 then goto 10;
```

And, with a little bit of organization we can remove any Boolean expressions except x = 0 from if statements. Also the *else* clauses may be discarded through careful substitution.

e) **Arithmetic.** Let's savage it almost completely! Who needs *multiplication* when we can compute z = x*y iteratively with:

```
z = 0;
for n = 1 to x do z = z + y;
```

Likewise *addition* can be forgotten. The statement z = x + y can be replaced by:

```
z = x;
for n = 1 to y do z = z + 1;
```

The removal of *division* and *subtraction* proceeds in much the same way. All that remains of arithmetic is *successor* (x + 1) and *predecessor* (x - 1).

While we're at it let us drop simple assignments such as x = y by substituting:

```
x = 0;
for i = 1 to y do x = x + 1;
```

f) The ***for*** statement. Two steps are necessary to remove this last vestige of civilization from our previously *NICE* language. In order to compute:

for i = m to n do S

we must initially figure out just how many times S is to be executed. We would like to say that;

t = n - m + 1

but we cannot because we removed subtraction. We must resort to:

```
    z = 0;
    t = n;
    t = t + 1;
    k = m;
10: if k = 0 then goto 20;
    t = t - 1;
    k = k - 1;
    if z = 0 then go to 10;
20: i = m;
```

(Yes, yes, we cheated! But nobody wanted to see the loops that set k and i to m. OK?) Now all we need do is to repeat *S* over and over again t times. Here's how:

```
30: if t = 0 then goto 40;
    S;
    t = t - 1;
    i = i + 1;
    if z = 0 then go to 30;
40: (* next statement *);
```

Loops involving *downto* are similar.

Now it is time to pause and summarize what we have done. We have removed most of the structured commands from the *NICE* language. Our deletion strategy is recorded the table of figure 1. Note that statements and structures used in removing features are not themselves destroyed until later.

Category	***Item Deleted***	***Items Used***
Constants	**negative numbers** **real numbers**	**extra variables** ***case*** **extra variables** ***while***
Boolean	**arithmetic operations** **logical connectives**	**arithmetic**
Arrays	**arrays**	**arithmetic**
Repetition	***while***	***goto, if-then-else***
Selection	***case, else***	***if-then***
Transfer	**unconditional *goto***	***if-then***
Arithmetic	**multiplication** **addition** **division** **subtraction** **simple assignment**	**addition, *for*** **successor, *for*** **subtraction, *for*** **predecessor, *for*** **successor, *for, if-then***
Iteration	***for***	***If-then*, successor, predecessor**

Figure 1 - Language Destruction

We have built a smaller programming language that seems equivalent to our original *NICE* language. Let us call it the *SMALL* programming language and now precisely define it. In fact we will start from scratch. A ***variable*** is a string of lower case Roman letters and if x and y are arbitrary variables then an (*unlabeled)* ***statement*** takes one of the following forms.

```
x = 0
x = x + 1
x = x - 1
if x = 0 then goto 10
halt(x)
```

In order to stifle individuality, we mandate that statements *must have* ***labels*** that are just integers followed by colons and are attached to the beginnings of statements. A ***title*** is again the original input statement and program heading. As before, it looks like:

```
program name(x, y, z).
```

> **Definition.** *A* ***program*** *consists of a title followed by a sequence of consecutively labeled instructions separated by semicolons.*

An example of a program in our new, unimproved *SMALL* programming language is the following bit of code. We know that *it is a program* because it conforms to the syntax definitions outlined above. (It does addition, but we do not know this yet since the semantics of our language have not been defined.)

```
program add(x, y);
1: z = 0
2: if y = 0 then goto 6;
3: x = x + 1;
4: y = y - 1;
5: if z = 0 then go to 2;
6: halt(x)
```

On to semantics! We shall describe computation in much the same way that we did for Turing machines. This shall be carried out in an informal manner, but the formal definitions are quite similar to those presented in the last section.

First, variables and their values during computation. Variables take on values while a program is running, so we shall represent a variable and its value at any time during the execution of a program by the pair:

$$\langle x_i, v_i \rangle$$

All we really need to know at any point during the execution of a program is the environment, or the contents of memory. Thus knowing what instruction we are about to execute and the values of all the variables used in the program tells us all we need know about a program currently executing.

Just as we did for Turing machines, we define a *configuration* to be the string:

$$k \langle x_1, v_1\rangle\langle x_2, v_2\rangle \ldots \langle x_n, v_n\rangle$$

where k is an instruction number (of the instruction about to be executed), and the variable-value pairs are the current values of all variables in the program.

The manner in which one configuration *yields* another should be rather obvious. One merely applies the instruction mentioned in the configuration to the proper parts of the configuration (that is, those variables in the instruction). The only minor bit of defining we need to do is for the halt instruction. As an example, let instruction five be *halt*(z). Then if x, y, and z are all of the variables in the program, we say that:

$$5 \langle x, 54\rangle \langle y, 23\rangle \langle z, 7\rangle \rightarrow 7$$

(Note that a configuration may be either an integer followed by a sequence of variable-value pairs or an integer. Also think about why a configuration is an integer *if and only if a program has halted*.) We may now reuse the Turing machine system definition for *eventually yielding* and computation has almost been completely defined.

Initially, input variables are set to their input values and all other variables are set to zero. Execution always begins with instruction number one. Now we know what an *initial configuration* looks like. *Halting configurations* were defined above to be merely numbers. *Terminal configurations* are defined in a manner almost exactly the same as for Turing machines. (We should note that terminal configurations can involve undefined variables and non-existent instructions. This is the stuff detected by compilers. And here is a point to ponder: are there any more things that might pop up and stop a program?)

We will now claim that *programs compute functions* and that all of the remaining definitions are merely those we used in the section about Turing machines. (The formal statement of this is left as an exercise.)

At this point we should believe that any program written in the *NICE* language can be rewritten as a *SMALL* program. After all, we went through a lot of work to produce the *SMALL* language! This leads to a characterization of the computable functions.

> **Definition.** *The* ***computable functions*** *are exactly those computed by programs (written in the SMALL programming language).*