BioBIKE Language Syntax
Working with large numbers of items: Mapping and Loops

II. Loops

A. Overview of loops by example

B. Overview of anatomy of the loop

C. Parts of the loop

1. Primary iteration control (top line of FOR-EACH)

2. Additional Control section

3. Initialization section

4. Variable update section

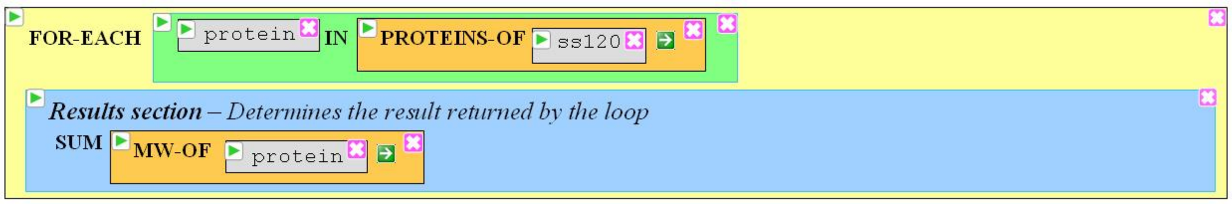
5. Body

6. Results section

7. Final action and the RETURN command

II.A. Overview of loops by example

Implicit mapping is simple: just replace a single item with a set of items. Explicit mapping is not too bad: just define a function f(x) = function and provide a list of x's. In contrast, looping uses what seems like a separate language. A ***loop*** executes one set of instructions repeatedly. Each time through the instructions is called an ***iteration***. Here's a previous example rendered as a loop:[*]



```
FOR-EACH  protein  IN  PROTEINS-OF  ss120
Results section – Determines the result returned by the loop
SUM  MW-OF  protein
```

Translation:

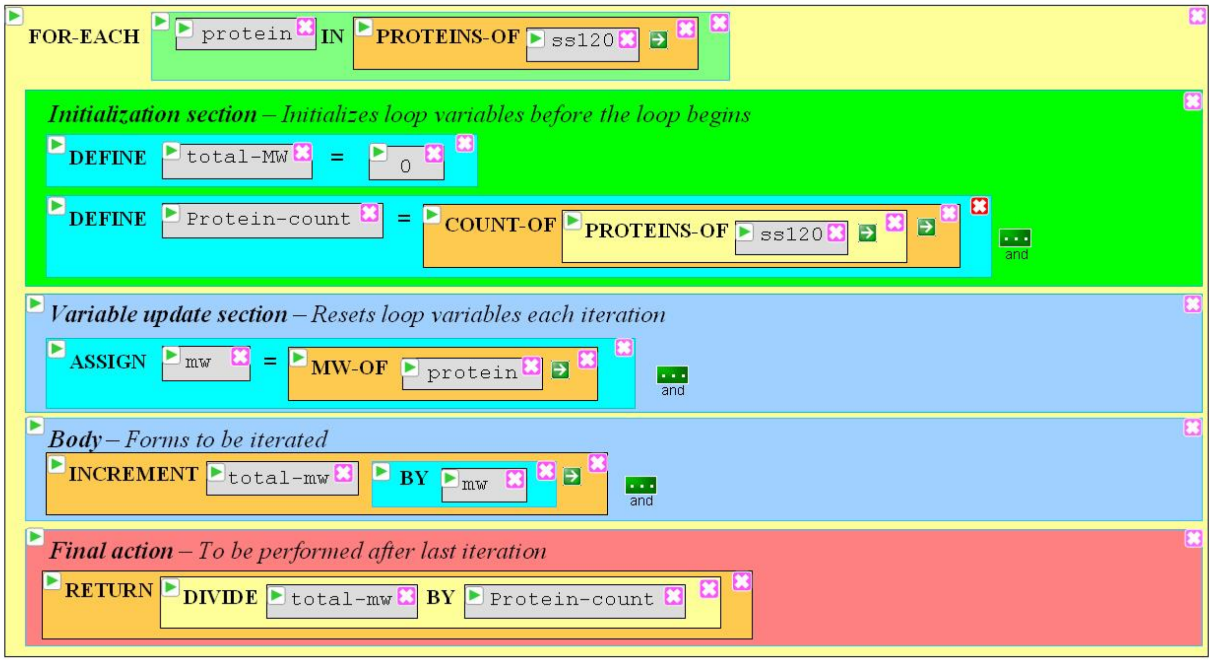
a. Consider each protein in the set of all proteins of ss120, one at a time.

b. Accumulate the molecular weights of each protein.

c. When the last protein has been considered, return the sum

That gets you the total molecular weight. Or this code gets you the entire answer by means of a more complicated loop:

[*] In this and many other examples shown here, I use a close version of the language, not what is currently available.



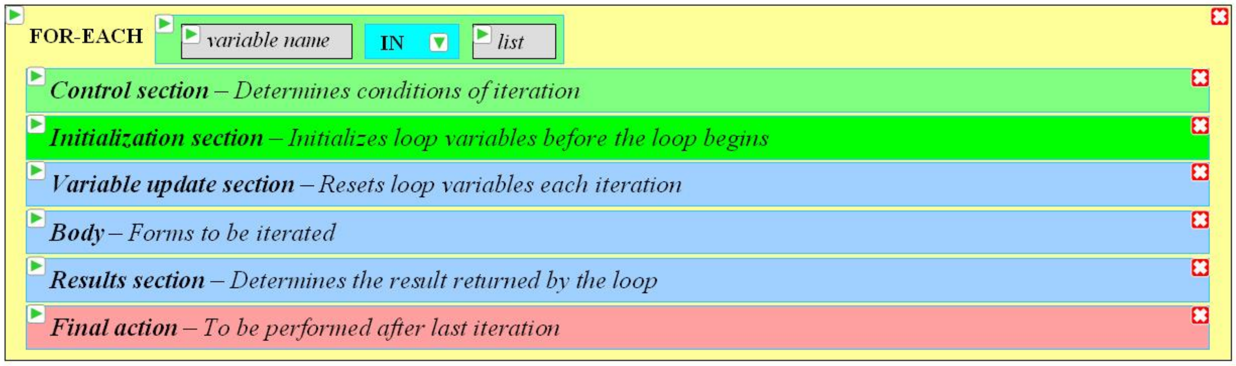
```
FOR-EACH protein IN PROTEINS-OF ss120
  Initialization section – Initializes loop variables before the loop begins
    DEFINE total-MW = 0
    DEFINE Protein-count = COUNT-OF PROTEINS-OF ss120
  Variable update section – Resets loop variables each iteration
    ASSIGN mw = MW-OF protein
  Body – Forms to be iterated
    INCREMENT total-mw BY mw
  Final action – To be performed after last iteration
    RETURN DIVIDE total-mw BY Protein-count
```

Translation:

a. *Consider each protein in the set of all proteins of ss120, one at a time.*
b. *Before the loop begins, set the sum of molecular weights to zero. The initialization occurs only once.*
c. *Before the loop begins, set the number of proteins. This will be a constant.*
d. *Find the molecular weight of the one protein you're considering at the moment. This assignment is repeated each time through the loop for each protein.*
e. *Add that molecular weight to the growing total.*
f. *(Loop) Repeat steps d and e until you've considered each protein in the set.*
g. *When you've finished considering each protein, calculate the average molecular weight and use that as the value returned by the `FOR-EACH` function.*

II.B. Overview of anatomy of the loop

BioBIKE supports two functions to describe loops: FOR-EACH and the very similar but more general LOOP function. This section will discuss only FOR-EACH. Loops can be divided into the following (mostly optional) parts:



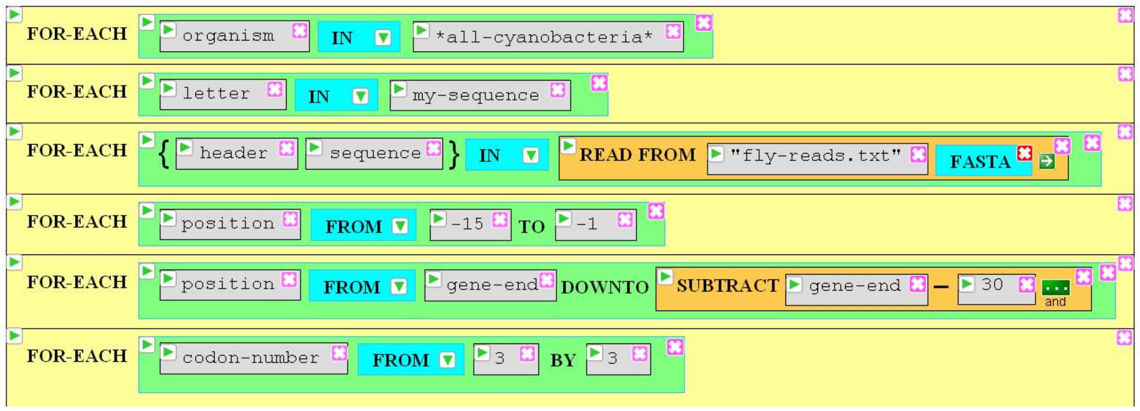
```
FOR-EACH variable name IN list
  Control section – Determines conditions of iteration
  Initialization section – Initializes loop variables before the loop begins
  Variable update section – Resets loop variables each iteration
  Body – Forms to be iterated
  Results section – Determines the result returned by the loop
  Final action – To be performed after last iteration
```

The Initialization section (green) is the first to be executed and is executed only once, before the loop begins. The Primary iteration control (first section) and Control section (light green) are then set up before the loop begins, but they are considered again with each iteration. The next three sections (blue) are executed each iteration. The Final Action section (red) is executed only once, after the loop is finished. In the sections that follow, I'll discuss each part.

II.C. Primary iteration control (top line of FOR-EACH)

The primary iteration consists of two parts: the ***variable*** and the ***values*** that will be assigned to the variable. Each time through the loop, the variable will take only one of the values. The rest of the loop describes how that value will be used.

Most people who have learned other computer programming languages expect loop variables to take only numeric values, counting sequentially from one number to another number. BBL loops can do this, but it's more common in bioinformatics to want to go through a list of things, like a list of genes or organisms or the nucleotides of a sequence. The example at the beginning of this section calculating molecular weight is a typical case. You can change the primary iteration, how the variable is assigned values, to achieve the following forms:

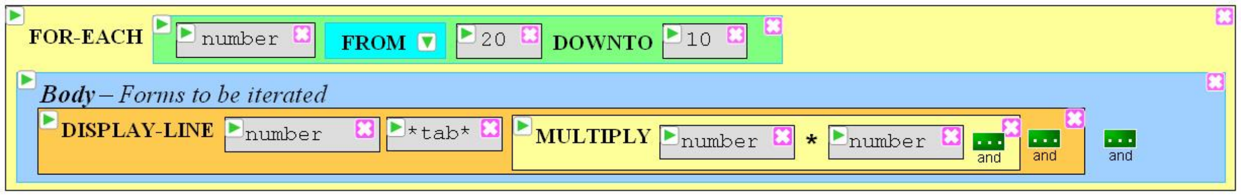


```
A. FOR-EACH organism IN *all-cyanobacteria*
B. FOR-EACH letter IN my-sequence
C. FOR-EACH { header sequence } IN READ FROM "fly-reads.txt" FASTA
D. FOR-EACH position FROM -15 TO -1
E. FOR-EACH position FROM gene-end DOWNTO SUBTRACT gene-end − 30
F, FOR-EACH codon-number FROM 3 BY 3
```

Code fragments showing different formats for primary iteration:

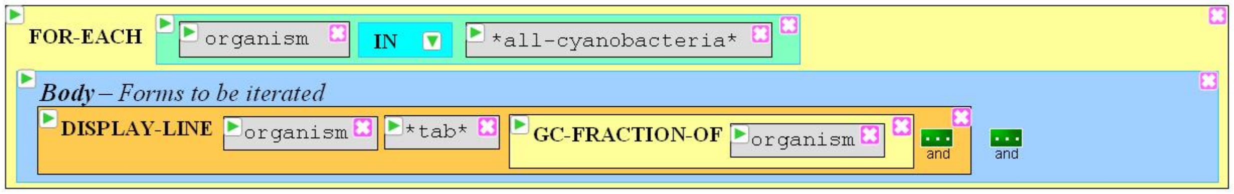
A. Consider each organism within the list of organisms called `*all-cyanobacteria*`
B. Consider each letter within the string called `my-sequence`
C. Consider each header/sequence pair in the list of such pairs read from the FastA file "fly-reads.txt"
D. Consider each position, taking values from -15 up to -1.
E. Consider each position, taking values from the variable called `gene-end` down to the same number less 30.
F. Consider each codon-number, starting from 3 and proceeding by 3 (3, 6, 9,…)

SQ1. What values do you predict will be displayed in the following loop?



```
FOR-EACH number FROM 20 DOWNTO 10
  Body – Forms to be iterated
    DISPLAY-LINE number *tab* MULTIPLY number * number
```

SQ2. What values do you predict will be displayed in the following loop?

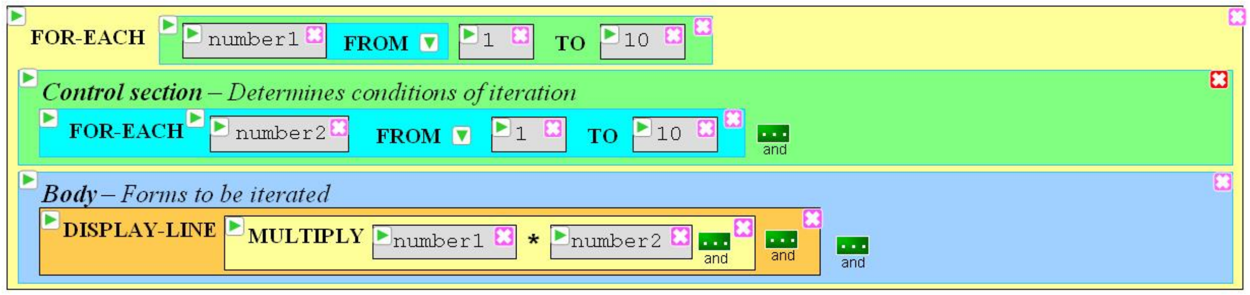


```
FOR-EACH organism IN *all-cyanobacteria*
  Body – Forms to be iterated
    DISPLAY-LINE organism *tab* GC-FRACTION-OF organism
```

II.D. (Additional) Control section

Format E above makes clear that sometimes additional control over the loop is necessary besides that provided by the primary iterater. If the loop were allowed to proceed as shown in that format, it would begin at 3 and continue forever. Through the Control Section, you can impose additional FOR-EACH conditions and conditions of two new types: WHILE and UNTIL.

The additional FOR-EACH conditions proceed in parallel with the first. Thus:



```
FOR-EACH number1 FROM 1 TO 10
  Control section – Determines conditions of iteration
    FOR-EACH number2 FROM 1 TO 10
  Body – Forms to be iterated
    DISPLAY-LINE MULTIPLY number1 * number2
```

…displays only 10 numbers (1, 4, 9, … 100), not the 10 x 10 table you might expect. The loop shown above may be understood as follows:

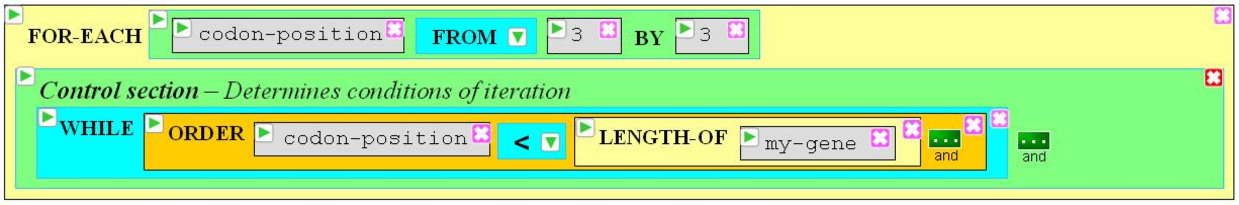
Translation:

a. *Consider each number, one at a time, from 1 to 10. Call it* `number1`.
b. *At the same time, consider each number, one at a time, from 1 to 10. Call it* `number2`.
   (Notice that when explained in this way, the folly of this loop becomes clear. There's no difference between `number1` *and* `number2` *and no sense in inventing both. They will never differ from one another.)*
c. *Display the following on a single, fresh line: the value of* `number1` *multiplied by* `number2`.
d. *(Loop) Repeat steps c for each* `number1` *and* `number2`, *changing in lockstep.*

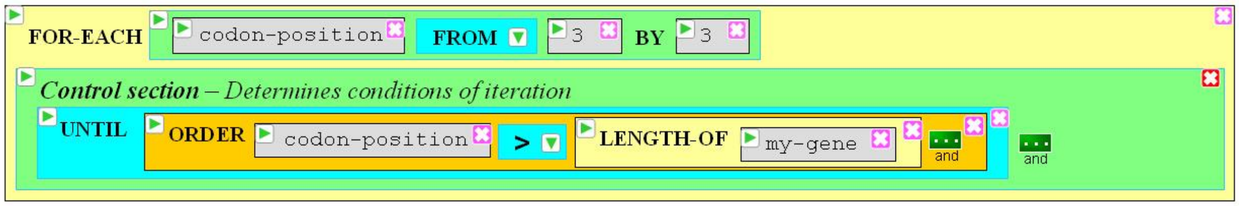
There is a way of breaking the connection between the `number1` and `number2` so that they increase not in lockstep but like columns in a speedometer. We'll cover that in the **Body** section.

In contrast to FOR-EACH, WHILE and UNTIL do not create a new variable. They describe a condition. In the case of WHILE, that condition must be met or the loop is terminated. In the case of UNTIL, if that condition ***then*** the loop is terminated. For example:

Both loop fragments will have the same effect. In the first case, `codon-position` will increase by 3 so long as that position remains less than the full length of some gene. In the second case, `codon-position` will increase by 3 up until the point that position exceeds the length of the gene. Sometimes it is easier to think about continuing the loop WHILE something remains true, and sometimes it is easier think about continuing it UNTIL something becomes true. It is possible to combine any number of controls into a loop. The first loop fragment may be understood as follows:



```
FOR-EACH codon-position FROM 3 BY 3
Control section – Determines conditions of iteration
  WHILE ORDER codon-position < LENGTH-OF my-gene
```



```
FOR-EACH codon-position FROM 3 BY 3
Control section – Determines conditions of iteration
  UNTIL ORDER codon-position > LENGTH-OF my-gene
```

Translation:

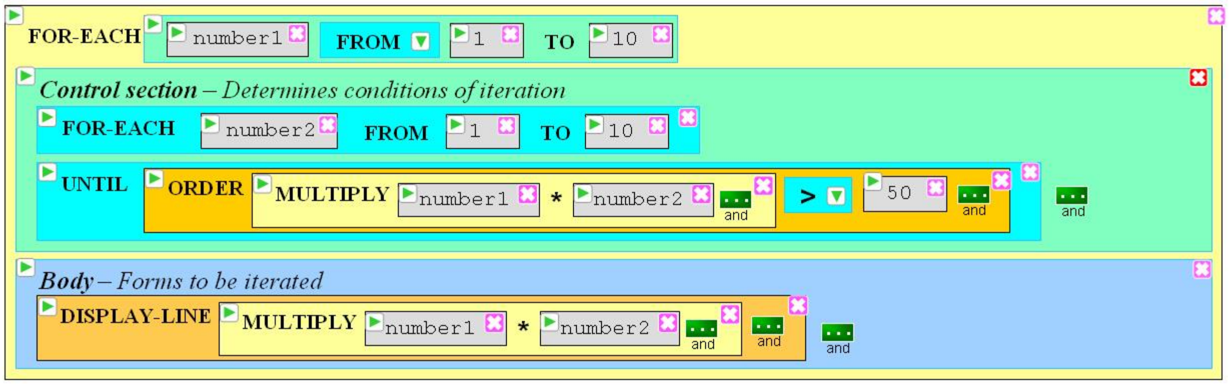
a. *Consider each codon-position, one at a time, starting with the number 3 and proceeding upwards, counting by 3.*

b. *Continue with the loop so long as the codon position is less than the length of* `my-gene` *(This example presumes that you have previously defined* `my-gene` *to be some gene.*

c. *(The loop continues in some way not shown here)*

Using WHILE or UNTIL can be a bit dangerous. If you specify a condition that is always true or that never can be met, you might end up with a loop that goes on forever (or until your time allocation is exceeded,… default = 40 seconds).

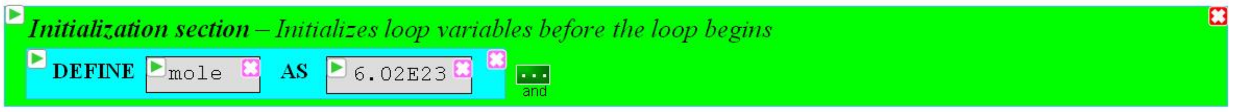
SQ3. How many numbers will the following loop display?



```
FOR-EACH number1 FROM 1 TO 10
Control section – Determines conditions of iteration
  FOR-EACH number2 FROM 1 TO 10
  UNTIL ORDER MULTIPLY number1 * number2 > 50
Body – Forms to be iterated
  DISPLAY-LINE MULTIPLY number1 * number2
```

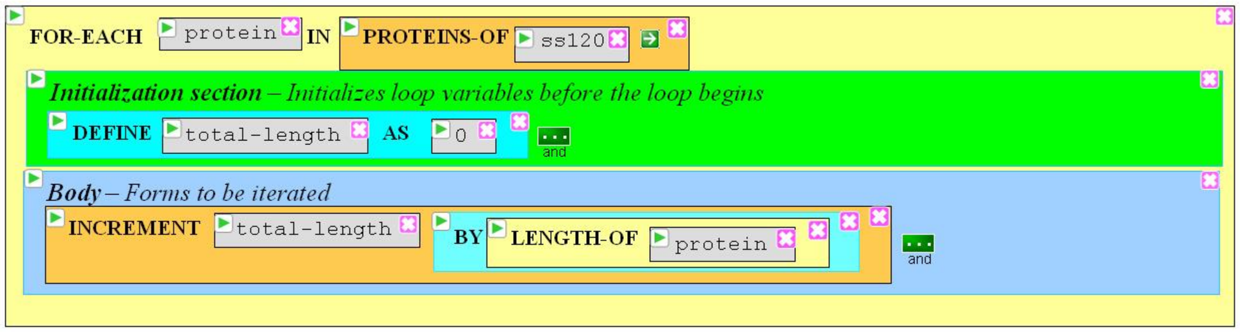
II.E. Initialization section

The first thing a loop does is to initialize variables you specify to be initialized. It does this even before initializing the primary iterater. You often may make loops without this section, but it does have its moments. For example, you might want to initialize variables to set a constant:



```
Initialization section – Initializes loop variables before the loop begins
    DEFINE mole AS 6.02E23
    and
```

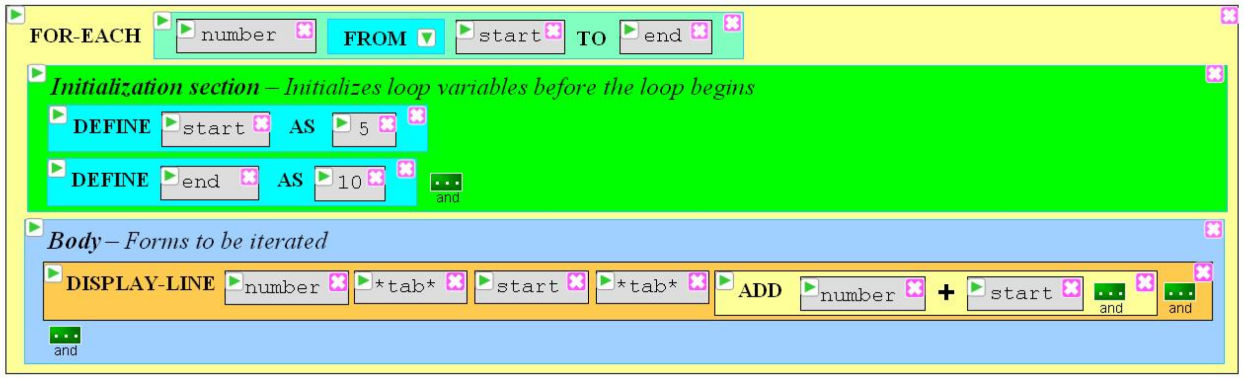
Or, you might want to initialize a variable that will change over the course of the loop:



```
FOR-EACH protein IN PROTEINS-OF ss120
    Initialization section – Initializes loop variables before the loop begins
        DEFINE total-length AS 0
        and
    Body – Forms to be iterated
        INCREMENT total-length BY LENGTH-OF protein
        and
```

It is possible to define any number of variables and constants.

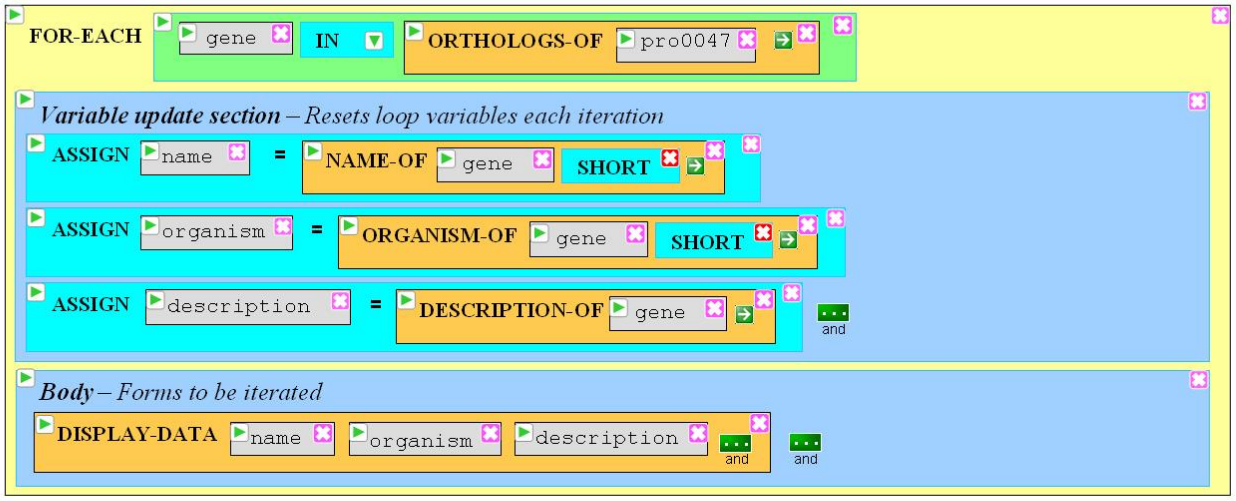
SQ4. What will the following loop display?



```
FOR-EACH number FROM start TO end
    Initialization section – Initializes loop variables before the loop begins
        DEFINE start AS 5
        DEFINE end AS 10
        and
    Body – Forms to be iterated
        DISPLAY-LINE number *tab* start *tab* ADD number + start and
        and
        and
```

II.F. Variable update section

The Initialization Section initializes variables only once, at the beginning. In the Variable Update section, variables are modified every iteration through the loop. This is most commonly employed to set up quantities that will be used within the loop but depend on the value of the iteration variable. For example:



```
FOR-EACH gene IN ORTHOLOGS-OF pro0047
  Variable update section – Resets loop variables each iteration
    ASSIGN name = NAME-OF gene SHORT
    ASSIGN organism = ORGANISM-OF gene SHORT
    ASSIGN description = DESCRIPTION-OF gene
  Body – Forms to be iterated
    DISPLAY-DATA name organism description
```

… which may be understood as follows:

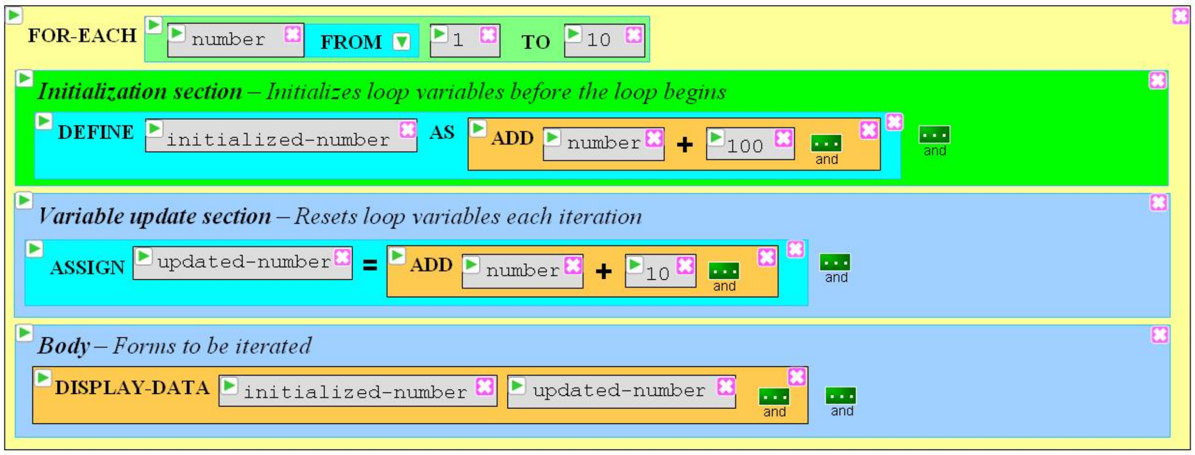
Translation:

a. *Consider each gene amongst those with shared evolutionary antecedents as pro0047*
b. *Retrieve the short form of the name of the gene under consideration and assign that value to* `name`
c. *Retrieve the short name of the organism of the gene under consideration and assign that value to* `organism`.
d. *Retrieve the description of the gene under consideration and assign that value to* `description`.
e. *Display the following, listing on a new line the name of the variable and its value for each of:* `name`, `organism`, *and* `description`.
f. *(Loop) Repeat steps b through e until the set of genes has been exhausted.* ***Note that the assignments (steps b through d) are redone each iteration.***

Each time through the loop, `name`, `organism`, and `description` will take on different values, because they are derived from the primary iterator, `gene`, which takes on a different value each iteration. Variables may also be updated using values that have been defined in the Initialization Section or updated earlier in the Variable Update Section. You ***could*** use this section to assign values to variables that don't change, but that's best left to the Initialization Section.

Sometimes the entire loop is calculating variables and there's no need for the Body Section. It is enough to collect one or more of the variables.

SQ5. Consider the example below of how initialized variables differ from updated variables. Predict what will be displayed.



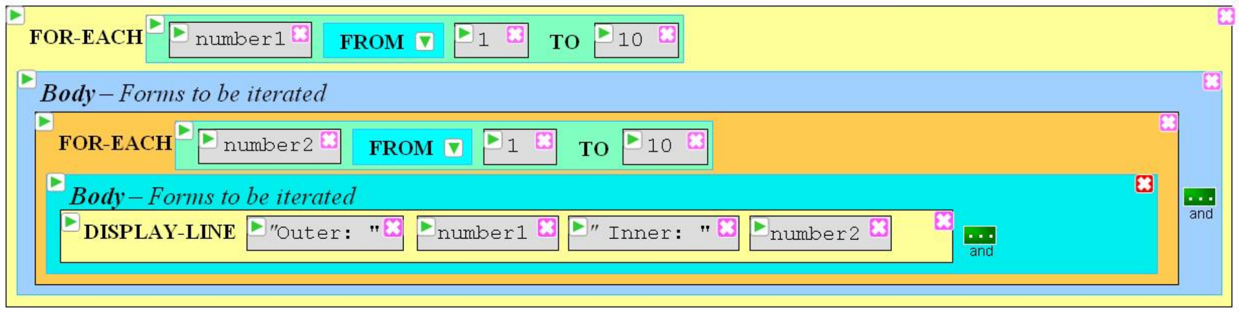
```
FOR-EACH number FROM 1 TO 10
  Initialization section – Initializes loop variables before the loop begins
    DEFINE initialized-number AS ADD number + 100
  Variable update section – Resets loop variables each iteration
    ASSIGN updated-number = ADD number + 10
  Body – Forms to be iterated
    DISPLAY-DATA initialized-number updated-number
```

II.G. Body

In all the sections discussed thus far, there are constraints as to what kind of actions may be taken (e.g. definitions in the Initialization section). In the Body section, you have almost free rein. It is important to realize, however, that actions in the Body section do not cause a result to be returned by the loop, unless there is an explicit RETURN statement. You may multiply a one variable by another, but that multiplication will not necessarily find its way into the results.

Sometimes you would like to have two loop variables running separately from one another, unlike the two variables locked together in the example shown in **SQ3**. You can do this by having one loop nested within another. Nested loops work like a speedometer, with the variable of the inner nested loop like the 1's-digit and the variable of the outer nested loop like the 10's-digit. The inner loop runs to completion for each iteration of the outer loop. Here's an example:

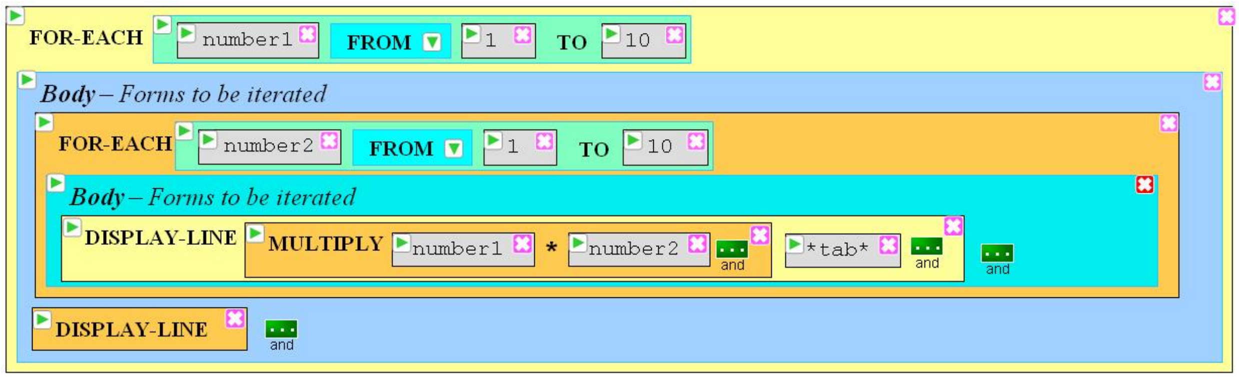
SQ6. What does the function below display?



```
FOR-EACH number1 FROM 1 TO 10
  Body – Forms to be iterated
    FOR-EACH number2 FROM 1 TO 10
      Body – Forms to be iterated
        DISPLAY-LINE "Outer: " number1 " Inner: " number2
```

Or, more interesting:

SQ7. What does the function below display?



```
FOR-EACH number1 FROM 1 TO 10
  Body – Forms to be iterated
    FOR-EACH number2 FROM 1 TO 10
      Body – Forms to be iterated
        DISPLAY-LINE MULTIPLY number1 * number2 and   *tab* and   and
    DISPLAY-LINE   and
```

II.H. Results section

Like all BioBIKE functions, loops return a value. If you pay no attention to what it returns -- for example if you're concerned only what the loop ***displays*** (as in the last example) -- then `NIL` will be returned. Most of the loops used as examples thus far return NIL. They just display something as output. In the real world, however you'll usually want the loop to return a value or a list of values. On the first page of the notes there are examples of loops that ***do*** return values, one because of the SUM keyword and one because of the RETURN function.

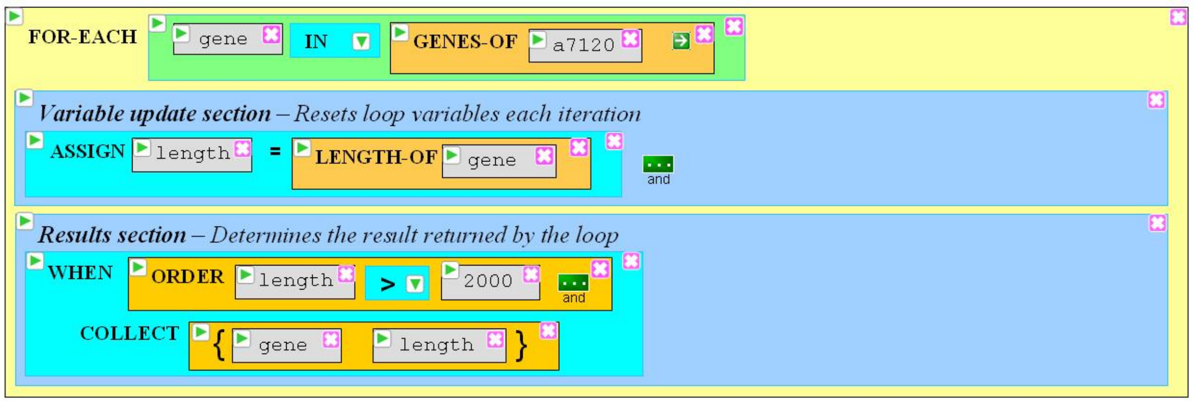
Here are five ways of returning values:

- `COLLECT`, returns a list of values saved over the course of the loop
- `COUNT`, returns the number of times the clause is invoked
- `SUM`, returns the sum of a number of items.
- `MAX`, returns the largest number considered in the loop
- `MIN`, returns the smallest number considered in the loop

`(RETURN` is another way, but that will be discussed in the next section, **Final Actions**).

You may use no more than one of these methods in a specific loop. Each of these five ways of developing a result can be activated conditionally, using the very useful WHEN option. Here are some examples, some using WHEN, some not.

How to make a list of large genes:

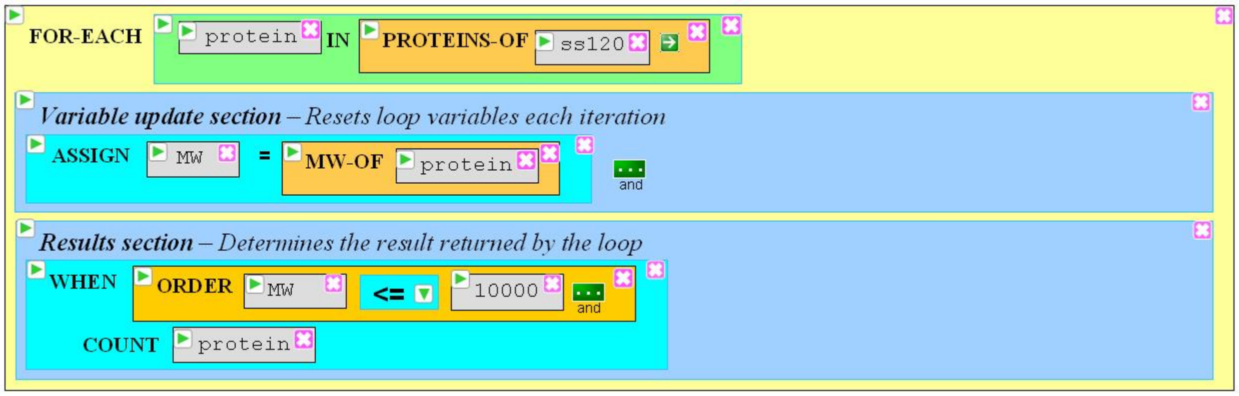


```
FOR-EACH gene IN GENES-OF a7120
  Variable update section – Resets loop variables each iteration
    ASSIGN length = LENGTH-OF gene   and
  Results section – Determines the result returned by the loop
    WHEN ORDER length > 2000 and
    COLLECT { gene length }
```

Translation:

a. *Consider, one at a time, each gene in the set of genes of Anabaena PCC 7120.*
b. *Retrieve the length of the gene and assign that value to a variable called* `name`.
c. *When the length is greater than 2000, put the gene and its length into a collection that will eventually be the result of the completed loop.*
d. *(Loop) Repeat steps b and c until the set of genes has been exhausted.*

How to count the number of small proteins:

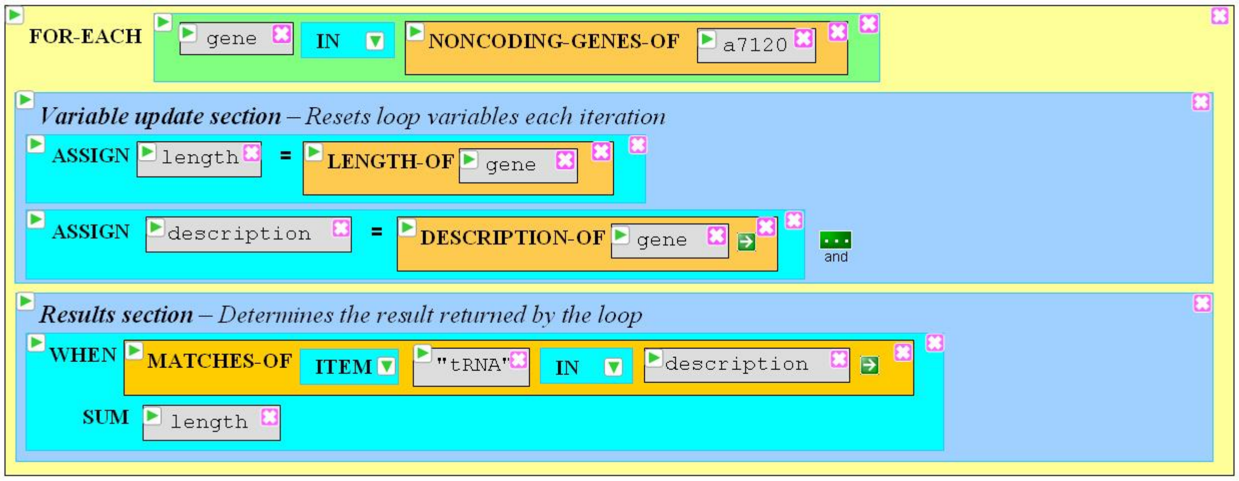


```
FOR-EACH protein IN PROTEINS-OF ss120
Variable update section – Resets loop variables each iteration
  ASSIGN MW = MW-OF protein
Results section – Determines the result returned by the loop
  WHEN ORDER MW <= 10000
    COUNT protein
```

Translation:

a. *Consider, one at a time, each protein in the set of proteins of Prochlorococcus marinus ss120.*
b. *Retrieve the molecular weight of the protein and assign that value to a variable called* `MW`.
c. *When the MW is less than or equal to 10000, count the protein, i.e., add one to a count, which will eventually be the result of the completed loop.*
d. *(Loop) Repeat steps b and c until the set of proteins has been exhausted.*

How to get a sum of the number of nucleotides devoted to tRNA:

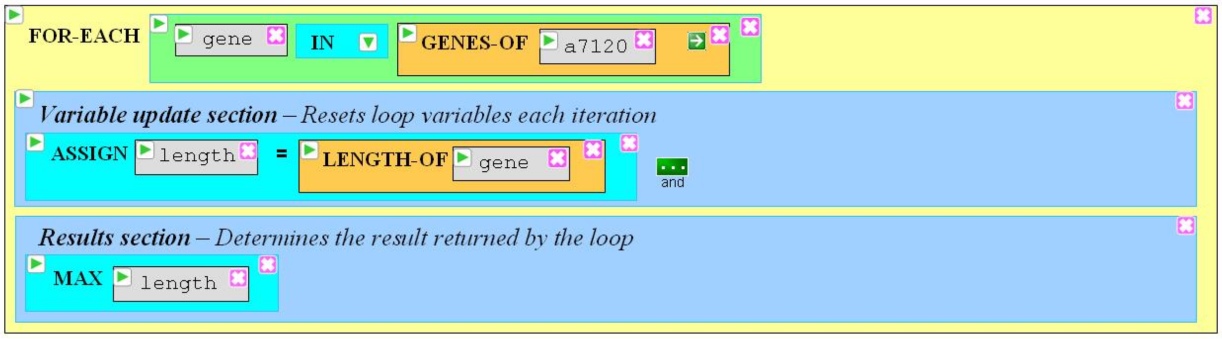


```
FOR-EACH gene IN NONCODING-GENES-OF a7120
Variable update section – Resets loop variables each iteration
  ASSIGN length = LENGTH-OF gene
  ASSIGN description = DESCRIPTION-OF gene
Results section – Determines the result returned by the loop
  WHEN MATCHES-OF ITEM "tRNA" IN description
    SUM length
```

Translation:

a. *Consider, one at a time, each gene in the set of noncoding genes of Anabaena PCC 7120.*
b. *Retrieve the length of the gene and assign that value to a variable called* `length`.
c. *Retrieve the description of the gene and assign that value to a variable called* `description`.
d. *When a match is found for the text "tRNA" within the description, add the length of the gene to a running sum, which will eventually be the result of the completed loop.*
e. *(Loop) Repeat steps b through d until the set of genes has been exhausted.*

What is the size of the largest gene?

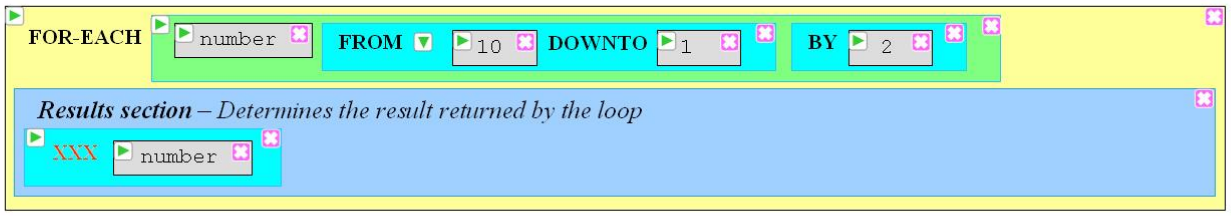


```
FOR-EACH  gene  IN  GENES-OF  a7120
  Variable update section – Resets loop variables each iteration
    ASSIGN  length  =  LENGTH-OF  gene
    and
  Results section – Determines the result returned by the loop
    MAX  length
```

Translation:

a. *Consider, one at a time, each gene in the set of genes of Anabaena PCC 7120.*
b. *Retrieve the length of the gene and assign that value to a variable called* `length`.
c. *If the length is bigger than any previously considered in this loop, remember it, and eventually this largest length as the result of the completed loop.*
d. *(Loop) Repeat steps b and c until the set of genes has been exhausted.*

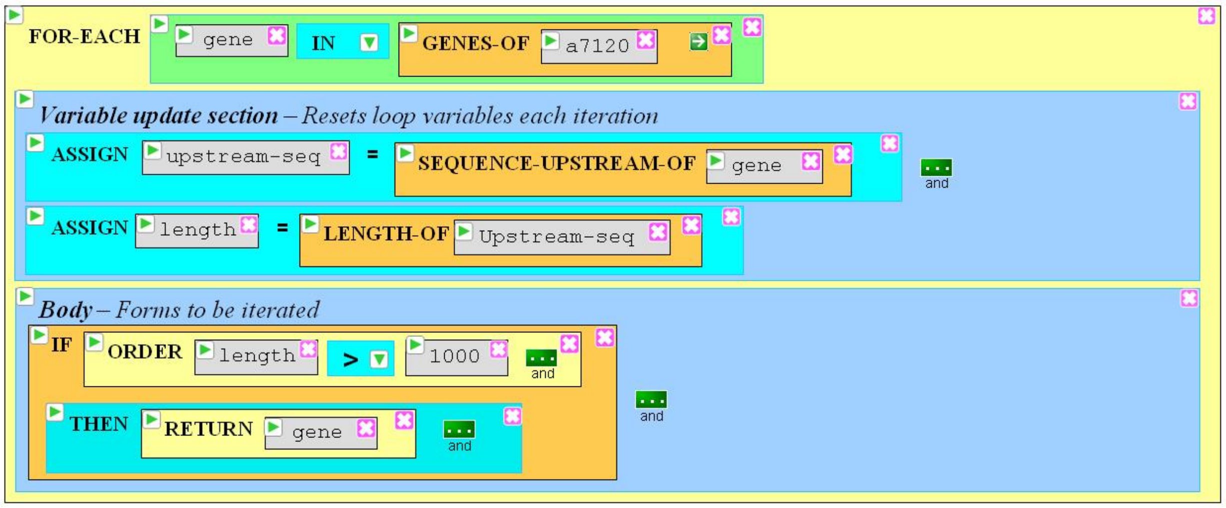
SQ8. Predict the result you would get by replacing XXX in the example below with each of the five Result Section options.



```
FOR-EACH  number  FROM  10  DOWNTO  1  BY  2
  Results section – Determines the result returned by the loop
    XXX  number
```

II.I. Final action and the RETURN command (*incomplete*)

Sometimes the ways of constructing a result offered by the Result Section are insufficient. The RETURN command makes it possible to return a result of any form, at any time. It is possible to stop the loop during an iteration and return a value by placing RETURN within the Body Section. For example, if you want to find the first instance of a long upstream sequence in a genome, you might do something like this:



```
FOR-EACH gene IN GENES-OF a7120
  Variable update section – Resets loop variables each iteration
    ASSIGN upstream-seq = SEQUENCE-UPSTREAM-OF gene
    ASSIGN length = LENGTH-OF Upstream-seq
    and
  Body – Forms to be iterated
    IF ORDER length > 1000 and
      THEN RETURN gene and
    and
```

Translation:

a. *Consider, one at a time, each gene in the set of genes of Anabaena PCC 7120.*
b. *Retrieve the sequence upstream of the gene (the sequence extending from the beginning of the gene backwards until it encounters the previous gene) and assign that value to a variable called* `upstream-seq`*.*
c. *Retrieve the length of the gene and assign that value to a variable called* `length`*.*
d. *If the length of the upstream region is greater than 1000 nucleotides, then return the current gene and exit the loop.*
e. *(Loop) Repeat steps b through d until the above condition is met or until the set of genes has been exhausted.*

In some loops, you may want to perform some actions after all the iterations have been completed. Most commonly, the action to perform is to return a value using quantities developed by the loop. An example of this type is given on the first page of these notes.