Introduction to Bioinformatics (2003)

Scenario 4: Metabolic Modeling (`Glycolysis.pl`)

Outline:

I. Overview of `glycolysis.pl`
II. Output of `glycolysis.pl`
III. Arrays and arguments to subroutines

I. Overview of `Glycolysis.pl`

Our goal is to capture the essential features of glycolysis in trypanosomes so that we can manipulate a model of the pathway to determine which step may be most effectively inhibited to lower ATP concentration. The basic tool of modeling we're considering is the use of differential equations to compute the change in metabolite concentrations over a short time interval. The work cycle in the program should look something like this:

1. Initialize constants for each reaction
2. Initialize concentrations for each metabolite
3. Calculate the rate of change of each metabolite, given the current concentrations (each rate of change is analogous to a velocity)
4. Multiply each rate of change by a time increment, giving an increment for each metabolite (analogous to multiplying a velocity by a duration to give the distance traveled)
5. For each metabolite, add the increment to the original concentration to a new concentration (analogous to finding a new location by adding the distance traveled to the old location)
6. Repeat steps 3 through 5 as many times as you like

Now download `Glycolysis.pl` (go to the main scenario page and scroll down to the list of programs).

SQ1. Identify segments of the program that accomplish each of the processes identified above.

This program is in some ways more complicated than any we've encountered thus far in the course, owing to the large number of constants and variables. On the other hand, those constants and variables should seem pretty familiar. And the Main Program isn't too complicated.

SQ2. Identify the source of the constants in the program segment entitled "Rate constants and equilibrium ratios".

SQ3. In the subroutine Model, there is the following line:

```
my $dFruP2_dt = +$v_Rxn4 - $v_Rxn5;
```

What does `$dFruP2_dt` represent? `$v_Rxn4`? `$v_Rxn5`?
Why is `$dFruP2_dt` defined as the sum of `+$v_Rxn4` and `-$v_Rxn5`?

SQ4. How is `$v_Rxn4` calculated?

II. Output of `Glycolysis.pl`

The time has arrived. Run the program (on my computer it takes about 20-30 sec). Other than filling the screen with lots of numbers what did it do?

SQ5. Examine the documentation of the program and the program itself to determine what of use the program actually produces.

SQ6. Examine the output in Excel (block out all the data and use a Line chart).

SQ7. Identify on the chart the metabolite that most conspicuously drops in concentration throughout the duration of the simulation and the metabolite that most conspicuously increases in concentration. Do these behaviors make intuitive sense?

It seems pretty absurd to think that a population of trypanosomes can suck up a significant amount of blood glucose, yet that is what this simulation seems to be saying, because the model doesn't know enough to consider blood glucose a nearly inexhaustible pool. Let's help it out. There are several ways one could change the program to capture the idea that blood glucose remains constant. One way is to put in the total amount of glucose in the blood and the total number of trypanosomes… too much work for now. Let's just hard wire in constant blood glucose.

SQ8. Change the subroutine Model to ensure that the concentration of external glucose (external to the trypanosome) doesn't change. (I chose this subroutine as the best place to make the change, because it seems to me that the subroutine should contain all our thinking of what affects metabolite concentrations.) Run the program with this change and display the results in Excel. Use this version henceforth.

SQ9. What is the behavior now of the metabolites you identified in SQ7?

SQ10. Which metabolites achieve, after an initial sorting out period, near steady state (a nearly constant concentration over time)? You may have to adjust the scale of the Y-axis to see the behavior of some of the metabolites.

III. Arrays and arguments to subroutines

[Those with little computing experience may wish to go lightly through this section, particularly the last part]

By now you may be used to seeing something like the following lines of code beginning a subroutine (these were taken from Plot_function.pl, available from the main scenario page for Scenario 4):

```
sub f {
    my ($x) = @_;          # Concentrations of substrate and product
```

`$x` is the argument passed to the function, in a function call that might look like this:

```
$y = f(47);
```

The number 47 is passed to the subroutine `f` and eventually gets to the subroutine variable called `$x`. It may seem peculiar to you, however, that `$x` must appear within parentheses. And what is the mysterious `@_` all about? The relevant secrets will now be revealed.

Any time a subroutine is called with an argument list, the special list variable @_ springs into being within the subroutine, initialized with the values of the argument list. So calling a subroutine, like:

```
Calc_area_of_rectangle($height, $width);
```

creates a list ($height, $width) and gives that list to `@_`, an array defined within the subroutine. `@_` acts like any other array variable, except that it cannot be declared (`my @_` is illegal). So you could gain access to the arguments with statements like:

```
my $h = $_[0];
my $w = $_[1];
```

(recall that the scalar `$_[0]` accesses the first element of the array `@_`). It's more usual, however to extract the values as a complete list:

```
my ($h, $w) = @_;
```

Note that `@_` is ***the same entity*** as `($height, $width)`. If you assign `47` to `$_[0]`, `$height` will get the value `47`. (For those who have learned other languages, all Perl subroutines are passed arguments by reference.). However, `($h, $w)` is ***not*** the same entity as `@_` or `($height, $width)`. If you assign `47` to `$h`, `$height` remains unchanged. Since one usually begins a subroutine by assigning the contents of `@_` to a list of local variables, Perl subroutines ***in practice*** generally take arguments by value, not reference. So, you can do it either way.

What happens if there is only one argument, as in the subroutine call `y = f(47);` ? Nothing has changed. You still need to extract that one value from an array (consisting of one value).

SQ11. What happens if you try to extract the argument in this way?

```
Sub f {
   my $x = @_;
```

Make a simple test program and find out (print out $x). Why do you get the number you get?

Problems occur when you try to pass arrays into a subroutine. Every subroutine gets one and only one array, i.e. `@_`. If you pass it 29 arguments, they're all in `@_`. If you pass it one array, the elements of the array are all in `@_`. If you pass it ***two*** arrays, the elements of both are still in one array, `@_`, all squashed together.

Take a look at the subroutine `Multiply` within the program `Glycolysis.pl`. The purpose of this subroutine is to multiply a given array by a given scalar. It clearly requires two arguments, one a scalar value and the other an array. The subroutine extracts these values readily enough:

```
sub multiply {                  # Argument 1: scalar, Argument 2: array
   my ($scalar, @array) = @_;
```

The first element of @_ is given to $scalar, and the remaining elements (however many there may be) are given to @array, which is precisely what you want to happen.

SQ11. What problem arises in the following similar subroutine?

```
sub multiply {                    # Argument 1: array, Argument 2: scalar
   my (@array, $scalar) = @_;
```

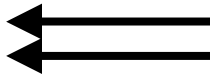
Now let's consider the worst case, a subroutine into which you want to pass more than one array. An example of this is the subroutine `Add` in `Glycolysis.pl`. This subroutine adds each element in one array to the corresponding element of another and returns the resulting array. It would be natural to write:

```
@sum_array = Add(@array1, @array2);
...
sub Add {                         # Argument 1: scalar, Argument 2: array
   my (@a1, @a2) = @_;
```

… but it won't work! Perl mashes @array1 and @array2 into a single array, and as soon as that happens, there's no longer any way of knowing how to separate the numbers back into two arrays.

SQ12. Try it! Download Argument_test.pl (from the Scenario web site). Note that the main program tries to pass the subroutine Add two arrays, @array1 and @array2. Run the program. How does it fail? If you don't see why, add the following diagnostic lines to the subroutine:

```
sub Add {
    my (@a1, @a2) = @_;
    print "array 1: ", join(" ", @a1), $LF;
    print "array 2: ", join(" ", @a2), $LF;
```



Run the program again. @a1 should have gained the values of @array1, and @a2 should have gained the values of @array2. Did that happen?

The solution to this problem[1] is to pass not the ***values*** of the two arrays but references to them:

```
@sum_array = Add(\@array1, \@array2);
```

Read literally, this means, "Pass to the subroutine `Add` the places in memory where `@array1` and `@array2` reside. Take what `Add` returns and put it in `@sum_array`." @_ thus gets just ***two*** values, i.e. the references to the two arrays. The subroutine accesses these locations as follows (lines taken with some modification from `Add` within Glycolysis.pl):

```
sub Add {
  my ($a1ref,$a2ref) = @_;
  my @a1 = @$a1ref;
  my @a2 = @$a2ref;
```

The weird looking `@$a1ref` is the same array as `@array1`, but `@a1` is a completely different entity, though it has the same values as `@array1`.

[1] I don't believe true Perl aficionados consider this a problem but rather a useful feature.