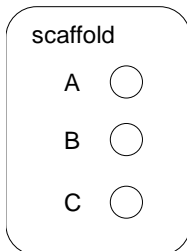


## molecular complexes and reactions in little b

In little b, molecular complexes are represented as graphs of connected components called monomers. Each monomer has one or more *sites* which may be either *bond* sites or *state* sites. Bond sites represent regions of interaction between monomers. State sites represent modifications - different states - of a part of a molecule.

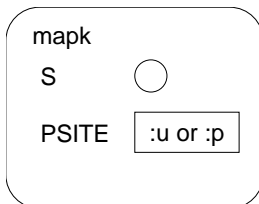
Monomers are defined using the *defmonomer* form. Let's define a molecule called scaffold with three bond sites:

```
(defmonomer scaffold a b c)
```



That's it. Now let's define a kinase called mapk:

```
(defmonomer mapk s (psite :states (member :u :p)))
```



This monomer has 2 sites; a bond site, named *s*, and a state site, named *psite*, which represents a single phosphorylatable epitope. We indicate it is a state site, with the `:states` keyword. This is followed by a Lisp type specifier indicating the valid states for this site. In this case, the P sites may have states `:u` (which we use to mean “unphosphorylated”) or `:p` (“phosphorylated”). We're saying that the value of *psite* should be a member of the set `(:U :P)`.

In addition to the site name, one can specify *tags*, which are additional names which are useful for matching patterns (more on this later). Tags specify additional information about a site. For example, to specify that the *psite* is a serine residue, we might adopt a convention of using the tag `:serine` to indicate

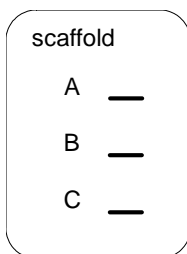
this. In addition, we may wish to record the amino acid identifier. In this case, ser405:

```
(defmonomer mapk
  s
  (psite :states (member :u :p)
        :tags (:serine :ser405)))
```

### Describing complexes using []

Complexes are described using the square brackets. A complex consisting of a single monomer uses only one pair of square brackets. For example,

```
[scaffold _ _ _]
```



indicates a single scaffold monomer where sites A, B, and C are all unconnected, indicated by the \_ symbol. The sites are specified in order, and the values specified are called the bindings. Alternatively, one could refer to the sites by name like so.

```
[scaffold A._ B._ C._]
```

or [scaffold B.\_ C.\_ A.\_] etc.

Here, the unconnected binding \_ appears after the dot following the name of the site. If a binding is not provided for a bond site, it is assumed by default to be unconnected, so the forms above could be written in abbreviated form as:

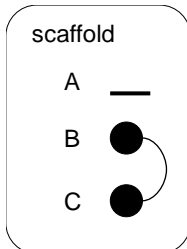
```
[scaffold]
```

Objects like those described above are instantiated inside little b as instances of CLOS (Common Lisp Object System) classes. This type of object is a *complex-species-type*.

*Connecting sites using bond labels*

Bonds connect sites to each other, and are specified with numbers. Bonds are specified when two sites share the same binding *and* the binding is a positive number. Here is a scaffold in which an intramolecular bond connects sites B and C:

```
[scaffold _ 1 1]
```



These numbers (always positive integers) are called *bond labels*. They can appear in any order, need not start at 1, but must be positive integers. This means,

```
[scaffold _ 1 1] = [scaffold _ 2 2] = [scaffold _ 999 999]
```

Note, we also could have specified this bond by referring explicitly to site names, like so:

```
[scaffold C.1 B.1]
```

or so:

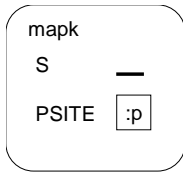
```
[scaffold B.1 C.1 A._]
```

Not a formal specification, but hopefully you get the picture.

### *Specifying state*

Like bonds, states are specified by supplying a value at the position or after the name of a site. For example, to specify the phosphorylated form of one of the kinases, we'd write:

```
[mapk _ :p]
```



or alternatively,

`[mapk psite.p]` (the S site unbound by default)

The default value of a state site is either specified explicitly. For example,

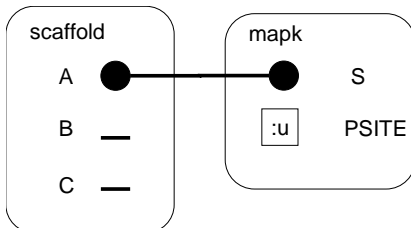
```
(defmonomer mapk s (psite :states (member :u :p) :default :u))
```

If a default is not explicitly specified and the type is a member type specifier (a list beginning with `member`), then the first list element following `member` is used (in this case, `:u`).

### *Specifying complexes composed of multiple monomers*

Ok, now that we've seen the basic components of the syntax, let's connect two monomers together. All that's required is an outer set of square brackets. A bond label bound to sites in different monomers indicates an intermolecular bond. Here, the scaffold A site is connected to the mapk S site, and the mapk monomer is unphosphorylated (Psite state= :U):

```
[[scaffold 1 _ _][mapk 1 :u]]
```



alternatively:

```
[[scaffold 1][mapk 1]]
[[scaffold A.1][mapk S.1 Psite.u]]
```

etc.

Complexes containing multiple monomers are no different (except for the outer set of brackets) than ones containing a single monomer. Shared bond labels specify connections between bond sites. Single-bracket expressions describing complexes of one monomer and double bracket expressions in which several monomers are involved both create instances of *complex-species-type*.

## Patterns

Pattern matching is specified by means of the asterisk. It is used in place of a bond label (or the unconnected label, \_) for a bond site and in place of the state for a state site.

For example, to specify all complexes in which the scaffold is bound to mapk via the A and S sites, we write:

```
[[scaffold 1 * *][mapk 1 *]]
```

The asterisks specify that the B and C sites of the scaffold may be bound to other complexes (or they may be unbound, \_). The asterisk at the PSITE position of mapk indicates that any state (:U or :P) is matched.

Objects containing wildcards are instances of the CLOS complex-pattern class. When b/biochem/ode is loaded (usually by including b-user/ode-biochem), an accessor field T0 is available which allows you to set the initial conditions of all complex-species-types matching the pattern. For example, this sets the T0 initial condition of all complex-species-types containing an A/S bond between a scaffold and mapk:

```
{[[scaffold 1 * *][mapk 1 *]].t0 := 5}
```

### *The Double Wildcard (\*\*)*

As we've seen, bond sites and state sites may be left unspecified, in which case a default is used. For bond sites, this is normally \_. It would be nice to be able to say that the wildcard is the default. This is what the double wildcard (\*\*) does; any unspecified sites are treated as if \* were used.

For example,

```
[scaffold _ * *] is the same as [scaffold _ **]
```

```
[scaffold B._ A.* C.*] is the same as [scaffold B._ **]
```

### *Wildcard monomer patterns*

Wildcard monomer patterns start with an asterisk, and allow matching of any monomer. They look like this:

```
[* site-patterns]
```

Each site pattern has the following form:

*label-pattern.binding*

The label pattern is either a symbol or a list which should match the site name or one of its tags. By default, the list is interpreted as an “and” requirement. That is, the site’s labels (tags + name) should include all of the specified labels. For example,

```
[* (serine psite)]
```

matches all complexes containing a serine phosphorylation site. More complex logical pattern matches can be built by explicitly specifying and, or and not prefix expressions. E.g.,

```
[* (and (or :tyrosine :serine) (not :psite))]
```

matches any monomer with the tags :tyrosine or :serine, but not with the tag :psite.

The *binding* is specified after the dot.

To specify site state, use the dot operator:

```
[* (serine psite).[ :u :p]]
```

The `.[]` indicates that the site is a state site. The values inside the `.[]` are treated by default as a logical OR operation - since it makes no sense to be in multiple states at once. However, one might wish to exclude a specific state:

```
[* (serine psite).[not :p]]
```

Bond sites are similarly specified. The binding appears after the dot:

```
[* (A B).*]
```

 will match any complex containing a site with A *and* B labels. This is the same as 

```
[* (AND A B).*]
```

```
[* (OR A B)._] ]
```

 will match any complex containing an unbound A site or an unbound B site.

```
[* A._]
```

 will match any complex containing an unbound A site. Note how we don’t need to use a list: `A._` is treated the same as `(A)._`.

```
[[scaffold A.1][* X.1]]
```

 will match any complex in which a scaffold is bound to an X site via its A site.

To match any set of labels, either use the empty list or \*. For example, both [`* ( )._`] and [`* *_._`] are patterns which match monomers with at least one empty site.

## Reactions

Reactions are specified using the `->>` infix operator, as follows:

```
{lhs-pattern ->> rhs-pattern}
```

They specify the transformation of complexes matching the left hand side pattern to those on the right hand side. Patterns are written separated by the `+` symbol. For example, to write the reaction for scaffold binding to mapk via the A/S sites, we write:

```
{[scaffold _ * *] + [mapk _ *] ->> [[scaffold 1 * *][mapk 1*]]}
```

If multiple monomers of the same kind are combined in a complex, some ambiguity may exist in specifying a reaction. In this case, it is possible to label each monomer using the dot operator. Here is the equivalent reaction where we have explicitly labeled the monomers:

```
{[scaffold.a _ * *] + [mapk.b _ *] ->> [[scaffold.a 1 * *][mapk.b 1*]]}
```

To be precise, objects generated by the `->>` infix operator are instances of the CLOS class, *complex-reaction-type*, and I should probably refer to them as “complex reaction type objects” - a bit clumsy, so I’ll refer to them simply as “reactions” here.

## Locations

When reactions involve multiple different locations, we use the `@` infix operator to indicate the relative localizations of components. Let’s define a ligand and receptor, with cognate receptor (R) and ligand (L) binding sites.

```
(defmonomer ligand R)
(defmonomer (receptor membrane) L)
```

Take a look at the definition of the receptor monomer. The first argument is a list which specifies the name and the location-class (`membrane`), which is highlighted. Until now we have left the location-class unspecified (little `b` assumes it is `compartment` by default).

To specify the binding of ligand to receptor, we write:

```
{[receptor _] + [ligand _] @ :outer ->> [[receptor 1][ligand 1]]}
```

The ligand is *localized* to the :outer compartment of the membrane. Note that the @ operator binds less tightly than the + operator, meaning that this expression is equivalent to the following (note the highlighted braces):

```
{[receptor _] + {[ligand _] @ :outer} ->> [[receptor 1][ligand 1]]}
```

Membranes have two sublocations, which are both compartments: :inner and :outer. The location-class of the reaction is assumed to be the same as that of any component for which a sublocation is not specified. The receptor's location class is membrane; since it is not part of an @ expression, little b infers that the reaction's location-class is membrane. In fact, it is an error to have mismatching location classes in a reaction. For example,

```
{[receptor _] + [ligand _] ->> [[receptor 1][ligand 1]]}
```

causes an error because receptor's location class is membrane and ligand's location-class is compartment.

In some case, we must explicitly specify the location-class of the reaction because each of the complexes exists in a sublocation of the location-class. For example, imagine the reaction in which an ion is transported from the inner to the outer compartment of a membrane.

If we write, {[ion] @ :inner ->> [ion] @ :outer}, it's not clear what location class :inner and :outer refer to. There may be other location classes with :inner and :outer fields - which one is it? When this problem arises, we specify the location class explicitly:

```
{[ion] @ :inner @ membrane ->> [ion] @ :outer}
```

The @ operator is evaluated left to right. So this is equivalent to:

```
{{[ion] @ :inner} @ membrane ->> [ion] @ :outer}
```

The ligand-receptor reaction could be written explicitly as:

```
{{[receptor _] + [ligand _] @ :outer} @ membrane  
->> [[receptor 1][ligand 1]]}
```

Because @ binds less tightly than +, in this case, we *must* write the brackets to the left of @ membrane. We cannot write [receptor \_] + [ligand \_] @ :outer @ membrane, since this is equivalent to [receptor \_] + {[ligand \_] @ :outer} @ membrane} which causes an error (note the position of the highlighted brackets).

## Kinetics

Kinetics are specified using the .SET-RATE-FUNCTION field:

```
(define r {[scaffold _ * *] + [mapk _ *]  
          -> [[scaffold 1 * *][mapk 1 *]])  
  
r.(set-rate-function 'mass-action 2)
```

All reactions matching this pattern will have mass-action kinetics with a rate constant of 2.

### *Custom Kinetics*

Custom rate functions can be specified by passing the CUSTOM-RATE function as the first argument to SET-RATE-FUNCTION. The second argument should be a symbolic mathematical expression, and the rest of the arguments are optional and should be an alternating sequence of keywords representing variable names.

Here's an example which shows how to manually define kinetics based on the hill equation for a binding reaction:

```
{[scaffold _ * *] + [mapk _ *]  
 -> [[scaffold 1][mapk 1]].(set-rate-function  
   'custom-rate  
   {[scaffold _ * *] ^ :hill /  
    {[scaffold _ * *] ^ :hill + :kd ^ :hill}}  
   :kd {4 micromolar}  
   :hill 1.3)
```

This mathematical template will be applied to all reactions generated by this pattern. To derive the rate function for a reaction involving specific species and locations, little b code in b/biochem/complex/ode and b/biochem/ode makes 3 substitutions:

- Complex patterns (such as [scaffold \_ \* \*] in the expression above) are substituted with variables representing the concentrations of the complex-species-types matching those patterns.
- Any keywords stored in the .k dictionary associated with the complex-reaction-type are replaced with variable objects which hold parameter values.
- Every function object is called passing the reaction as an argument, and is substituted with the result.

*A few notes on custom-rate substitutions:*

The first substitution step is obviously necessary: the final rate equations in an ODE system must contain variables which represent the concentrations of *specific* molecular species, not patterns.

The need for the second substitution step is less obvious, and relates to issues of dimensional mathematics. The pattern objects are dimensionless; however, the parameters have dimensions. In the hill function above, directly substituting in the parameter variable would lead to a dimension error. If the reaction is named *r*, then the parameter variables are *r.k.hill* (which is non-dimensional) and *r.k.kd* (which is in millimolar). The denominator of the hill function -  $\{[\text{scaffold\_} * *]^{\text{r.k.hill}} + \text{r.k.kd}^{\text{r.k.hill}}\}$  - then entails addition of a non-dimensional and a dimensional term, which would trigger an error. For this reason, only rate-functions are expressed using non-dimensional objects, and substituted for dimensional quantities when all of the terms are known.

The need for the third kind of substitution provides a means for functions to perform some calculation involving information available only from the localized reaction object. For example, it may be necessary to write a term which sums the concentrations of a particular species in membrane segments of a polygonal cell. In fact, this situation arises when encoding the Von Dassow/Odell segment polarity network.

### *Defining Custom Rate Functions*

Providing rate functions on a per-reaction basis is useful, however, it would be better if we could refer to these custom rate functions by name and use them without having to write the mathematical templates over and over again. Fortunately, little *b* provides this capability in the *define-custom-rate* macro. We could use it to define a Hill equation generator as follows:

```
(define-custom-rate hill (&key (cooperativity 1)
                              (kd 1)
                              species) ())
  (store-parameter :cooperativity cooperativity)
  (store-parameter :kd kd)
  {species ^ :hill /
   {species ^ :hill +
    :kd ^ :hill}})
```

To set the rate function of a reaction *R* with cooperativity of 1.5 and *kd* of 3 millimolar, the user would then write:

```
r.(set-rate-function 'hill
                    :cooperativity 1.5
```

```
:kd {3 millimolar})
```

Not all rate functions can be generated by simple substitution. Some may require examining the list of entities participating in a reaction, their stoichiometries or some other properties, and computing a function according to some algorithm.

A good example is mass-action kinetics. A predefined mathematical template into which values can be substituted will not work in this case. Instead, an expression must be built by multiplying the concentrations of every element that participates in the reaction.

Fortunately, the third argument to the `define-custom-rate` macro provides a solution. It is a list of optional keyword parameters which allow access to information about the reaction:

```
(define-custom-rate name
                    user-lambda-list
                    (&key entities
                        stoichiometries
                        dimensions
                        rate-dimension)
                    &body rate-calculation-code)
```

Let's define a quick-and-dirty implementation of mass-action kinetics:

```
(define-custom-rate mass-action
                    (mass-action-const)
                    (:entities bases
                       :stoichiometries powers)
                    ...)
```

The first thing we do is tell the custom-rate macro to bind values to `BASES` and `POWERS`. The `bases` will contain a list of entities on the left hand side of the reaction-type or complex-reaction-type object. For example, in the case of our scaffold-binding reaction, `BASES` would be a list of the two patterns:

```
([scaffold.a * _ _] [mapk.b _ *])
```

Where did those dotted symbols come from? Little `b` automatically assigns reference symbols in order if the user has not provided them. The `STOICHIOMETRIES`, in this case, and in every case involving complex-patterns, is always a list of 1s:

```
(1 1)
```

Now that we have this information, we can construct the mass-action term. First, let's store the mass-action constant. This is accomplished with a call to STORE-PARAMETER, as follows:

```
(store-parameter :mass-action mass-action-const)
```

Next, we construct the mass-action term, using Lisp's loop facility:

```
(loop with term = 1
      for base in bases
      for power in powers
      do {term := term * base ^ power}
      finally return {term * :mass-action})
```

The loop iterates through all the bases and powers and builds up a term consisting of the product of each base raised to each power, and finally multiplied by a symbol which refers to the mass-action constant stored in the parameter dictionary.

The final macro looks like this:

```
(define-custom-rate mass-action
  (mass-action-const)
  (:entities bases
   :stoichiometries powers)

  (store-parameter :mass-action mass-action-const)

  (loop with term = 1
        for base in bases
        for power in powers
        do {term := term * base ^ power}
        finally return {term * :mass-action})))
```

We could knock off at this point and call it a day, but we haven't really dealt with issues of dimensionality. This is where the other keyword arguments become useful. The user might pass in a rate constant of incorrect dimensionality. Ideally, the rate calculator should deal with this, and either throw an error, or gracefully correct the mistake, if that's possible.

The other keyword arguments are DIMENSIONS and RATE-DIMENSION. DIMENSIONS is a list of concentration dimensions corresponding to each pattern. RATE-DIMENSION is the dimension of the rate of the reaction. When the patterns are finally substituted for concentrations and the :mass-action symbol is substituted for the parameter variable when the reaction is generated, the resulting term should have the dimensions of RATE-DIMENSION.

Because we also know the concentration dimensions (in DIMENSIONS), the stoichiometries, we can calculate the correct dimension of the rate constant, since:

$$\text{RATE} = \text{TERM} * \text{CONSTANT}$$

$$\text{where TERM} = \prod C_i^{S_i}$$

$C_i$  = concentration of entity i

$S_i$  = stoichiometry of entity i,

We know,

$$\text{RATE-DIMENSION} = \text{TERM-DIMENSION} * \text{CONSTANT-DIMENSION}$$

$$\text{Where the TERM-DIMENSION} = \prod D_i^{S_i}$$

$D_i$  = dimension of  $C_i$

I'll leave it as an exercise to figure out the dimensionality-checking. You can see one solution in `b/biochem/std-rate-functions`.