MATHEMATICAL BACKGROUND

This supplementary chapter (present on the CD-ROM version of the book only) describes some of the mathematical concepts behind the graphical techniques introduced in chapter 32 of *Object-Oriented Software Construction, second edition*. It is extracted from the ISE manual on EiffelBuild [M 1995e].

The conventions, and any cross-reference that you may encounter in this chapter, are those of [M 1995e] rather than the rest of *Object-Oriented Software Construction*.

Pages are numbered 1076.1, 1076.2 and so on to avoid any confusion with the page numbers of the rest of the book, as they appear in the printed version.

This page intentionally blank

This page intentionally blank

32A Mathematical background

32A.1 OVERVIEW

EiffelBuild relies on simple properties of functions. This chapter presents a summary of the necessary notions.

You can use EiffelBuild without having read this discussion, and in fact if you are eager to get your hands on EiffelBuild you may prefer to skip this chapter on first reading and move immediately to the following chapter and the Guided Tour. But an understanding of the elementary mathematical notions discussed below will help you get the most out of EiffelBuild, especially for advanced uses.

"Introduction to the Theory of Programming Languages", Prentice Hall, 1991, ISBN 0-13-498510-9 (0-13-498502-8 pbk).

Many of the topics of this chapter are also useful for the formal study of programming languages, and are covered in more details in the book *Introduction to the Theory of Programming Languages*.

32A.2 FINITE SETS, CARTESIAN PRODUCT

A finite set may be given by the list of its members in braces, for example

PERSON ≜ {*Hélène, Kiyoko, Laura, Roberto, Helmut*}

COUNTRY ≜ {*Japan, France, Italy, UK*}

where ≜ means "is defined as".

A note about naming conventions: the example sets used in this chapter, such as *PERSON* and *COUNTRY*, follow the Eiffel rules for types (classes): they are written in upper-case letters, and use the singular rather than the plural. So *PERSON* denotes a set of persons and *COUNTRY* a set of countries. A mathematical text might call these sets *PEOPLE* and *COUNTRIES*, but for a programmer it is more attractive to think of declarations of the form

Hélène: *PERSON* -- (Eiffel syntax)

meaning "*Hélène* represents an object of type *PERSON*", hence the singular.

If X and Y are sets, then $X \times Y$, called the cartesian product of these sets, is the set of all pairs of the form $<x, y>$ where x is a member of X and y is a member of Y. For example the set *PERSON* × *COUNTRY* contains all the pairs such as <*Hélène, Japan*>, <*Hélène, France*>, ..., <*Kiyoko, Japan*>, <*Kiyoko, France*> and so on.

Cartesian product is also applicable to infinite sets. For example if $\boldsymbol{N}$ is the set of natural (non-negative) integers, then $\boldsymbol{N} \times \boldsymbol{N}$ is the set of all possible pairs of natural integers.

32A.3 RELATIONS

Let X and Y be two sets. A relation with source set X and target set Y is a set of pairs of the form $<x, y>$ such that, in every such pair, x is a member of X and y is a member of Y.

In other words, a relation is one particular subset of the cartesian product $X \times Y$.

For example. the set of pairs

citizenship $\triangleq$ {<*Kiyoko*, *Japan*>, <*Hélène*, *France*>, <*Laura*, *Italy*>,
<*Roberto*, *Italy*>, <*Hélène*, *UK*>}

is a relation with *PERSON* as source set and *COUNTRY* as target set. This relation could represent the intuitive notion "x is a citizen of y". Note that Hélène would then be a person with dual citizenship. As indicated, this relation will be called *citizenship*.

This example is a finite relation. We can also have infinite relations; for example with the set of natural integers $\boldsymbol{N}$ serving both as source set and target set, we can define the infinite set of pairs

neighbors $\triangleq$ $\{<0, 1>,$
$<1, 0>, <1, 2>,$
$<2, 1>, <2, 3>,$
$<3, 2>, <3, 4>,$
$\ldots\}$

denoted more precisely and concisely (with the bar | used to mean "such that") as

$\{<x, y> \in\ \boldsymbol{N} \times \boldsymbol{N} \mid y = x + 1 \ \textbf{\textit{or}}\ y = x - 1\}$

and representing the relation whose pairs all contain elements that differ from each other by either +1 or –1. This relation, as indicated, will be called *neighbors*.

The set of all possible relations between two sets X and Y is written $X \leftrightarrow Y$. For example the relation *citizenship* is a member of the set *PERSON* $\leftrightarrow$ *COUNTRY*; and the relation *neighbors* is a member of $\boldsymbol{N} \leftrightarrow \boldsymbol{N}$. The precise definition of $X \leftrightarrow Y$ *is* that it is the set $\boldsymbol{P}\,(X \times Y)$, using the notation $\boldsymbol{P}\,(A)$, for any set A, to mean the powerset of A, that is to say, the set of all possible subsets of A.

The **domain** of a relation is the set consisting of all elements x such that the relation contains a pair of the form $<x, y>$ for some y — that is to say, a pair with x as its first element. The domain is a subset of the source set. In the case of relation *citizenship*, the source set is {*Hélène*, *Kiyoko*, *Laura*, *Roberto*}; it does not contain *Helmut*, even though this element has been listed as a member of the source set *PERSON*, because no pair in *citizenship* has it as its first element. (The relation does not give any information about the citizenship of *Helmut*.)

The **range** of a relation is the inverse notion: the set consisting of all members of the target set that appear as second element of at least one pair in the relation.

A relation is **total** if its domain covers its entire source set — that is to say, if for every member x of its source set there is at least one pair of the form $<x, y>$ (for some y) in the relation. It is **partial** otherwise. If we say "relation" without further qualification the relation may be partial or total. Relation *neighbors* is total, but relation *citizenship* is not because *Helmut*, a member of its source set, is not in the relation's domain.

Note that the notion of relation used here is limited to binary relations, that is to say relations between two sets (the source and the target). If necessary, we can model a more general notion of relations involving any number of sets by using this notion: for example we can capture relations between three sets X, Y and Z by taking relations in $X \leftrightarrow (Y \leftrightarrow Z)$. This idea will further be applied below to "currying".

32A.4 FUNCTIONS

A function is a relation as just defined, with the extra property that for any x in the source set X, there is **at most one** pair of the form $<x, y>$ for some y — that is to say, at most one pair with x as its first element — in the relation.

The relations used above as examples are not functions. In the case of *citizenship*, there are two pairs with *Hélène* as their first element. In the case of *neighbors*, for each number n except 0, there are two pairs with n as their first element: the pair $<n, n + 1>$ and the pair $<n, n - 1>$.

If we prohibit dual citizenship, that is to say if we remove one of the pairs for *Hélène*, then *citizenship* becomes a function.

The set of pairs $<n, n + 1>$, for all possible natural integers n, is a function. Let us call it *next*. (This is a subset of relation *neighbors*.)

Like a relation, a function may be partial or total. As with relations, the word "function" without further qualification means a function that may be total or partial. Putting together the definitions of "function" and "total relation" we see that with a total function there is, for every member x of its source set, **exactly one** y in the target set such that $<x, y>$ is in the function.

More generally, if f is a function and x is a member of its domain, there is exactly one y such that $<x, y>$ is in the function. This legitimates the usual notation for expressing the value of that y:

$f\,(x)$

For example, *next* (*6*) has the value *7*; and, once we have made *citizenship* a function by removing one of the two pairs for *Hélène*, *citizenship* (*Kiyoko*) has the value *Japan*.

Remember, however, that the notation $f\,(x)$ has a taboo associated with it: unless the function is known to be total, the notation is meaningless without a guarantee that x belongs to the domain of f.

The set of all possible functions with source set X and target set Y is written $X \rightarrow\!\!\!\!\!+\; Y$. Since every function is a relation, $X \rightarrow\!\!\!\!\!+\; Y$ is a subset of $X \leftrightarrow Y$.

The bar across the arrow in the $\rightarrow\!\!\!\!\!+$ symbol reminds us that the functions may be partial. The set of total functions from X to Y, a subset of $X \rightarrow\!\!\!\!\!+\; Y$, is written $X \rightarrow Y$ without a bar.

32A.5 FINITE FUNCTIONS

Like any set, a relation or function may be finite or infinite. Finite functions, that is to say functions made of a finite set of pairs, will be of particular interest for the rest of this discussion.

Function *citizenship* is finite; function *next* is infinite.

If both X and Y are finite, as in the case of *citizenship*, then their cartesian product $X \times Y$ is also finite, so any function in $X \nrightarrow Y$ (and more generally any relation in $X \leftrightarrow Y$), being a subset of $X \times Y$, will be finite. If X or Y or both are infinite, however, a function with source set X and target set Y may be finite or infinite. So in spite of the *next* example you can have a finite function between infinite sets; for example, with $\boldsymbol{N}$ as both source and target, the function {*<0, 1>*, *<237, 118>*} is finite even though $\boldsymbol{N}$ is infinite.

A function is finite if and only if both its domain and its range are finite.

The set of finite functions with source set X and target set Y, a subset of $X \nrightarrow Y$, is written $X \xrightarrow[f]{} Y$, where the *f* stands for “finite”.

32A.6 VISUALIZING A FINITE FUNCTION

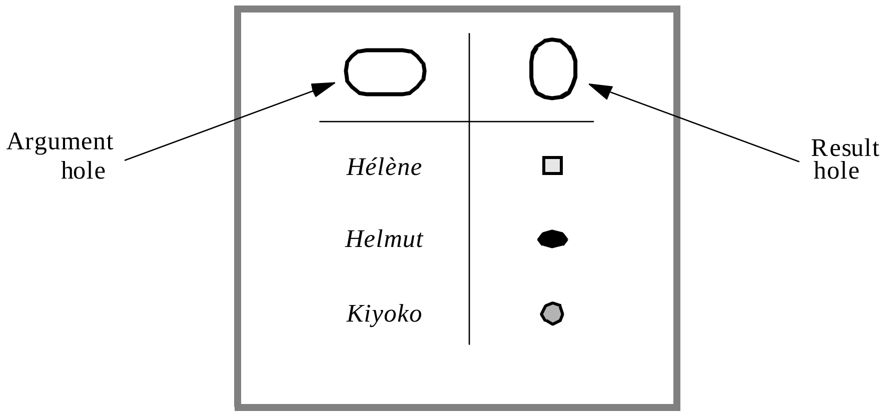
Finite functions and relations are of particular interest for software applications because they can be easily represented in the memory of a computer. But for a graphical application builder such as EiffelBuild finite functions have an even more specifically useful property: they are easy to construct and manipulate visually, using an obvious graphical representation in the form of a table. For example you can graphically display the three-pair function {<*Hélène*, □>, <*Helmut*, ⬬>, <*Kiyoko*, ⯃>} — a function in $PERSON \xrightarrow[f]{} SHAPE$ where *SHAPE* is a set of graphical icons — as

Hélène	□
Helmut	⬬
Kiyoko	⯃

This representation has the immediate visual advantage that for a reasonably small function it is easy to check visually that what is displayed is a function, not just a general relation: just look at the first column (the function’s domain) and check that it has no duplicate. This will be even easier if the source set has a simple order relation so that elements in the first column can be kept sorted, as has been done above using alphabetic order.

Typed pick-and-throw is described in section 4.5 of ''ISE Eiffel: The Environment".

This convention blends particularly well with the visual principles of ISE Eiffel, as described in *ISE Eiffel: The Environment*. In particular, the environment's typed pick-and-throw mechanism immediately suggests a convenient way for users to build a function:



To add a new pair to the function, you will drag-and-drop an element from the source set to the top-left hole. This will only have an effect if that element is not already in the function's domain, that is to say if it is distinct from all the elements in the left column. Dropping the element into the hole will cause the hole to become occupied (according to the conventions of the environment a small dot in the hole should indicate this). When you then drag-and-drop to the top-right hole a member of the target set, the newlywed pair clunks down the table to its proper station in life.

The order in which you select the pair's elements is immaterial: you can drag-and-drop the result element into its hole first if you prefer.

See "Building a behavior pair", 5.6.6, page 63 and "THE STATE TOOL", 6.11, page 150.

The scheme just described is the basis for building **behaviors** in EiffelBuild, the fundamental mechanism for producing context-event-command associations and hence interactive graphical applications. It also governs the **State Tool** of EiffelBuild.

32A.7 DEALING WITH MULTI-ARGUMENT FUNCTIONS

The example functions discussed so far in this chapter are unary, that is to say take a single argument (a member of the source set).

It is often necessary to use multiary functions — functions with two or more arguments. Mathematically, this does not raise any particular problem: rather than talking about a function with two arguments from sets A and B, we consider that we have a function whose (single) source set is the cartesian product $A \times B$; the function will be a member of the set of functions $A \times B \nrightarrow C$ for some C, and similarly for more than two arguments.

The practical consequences are awkward, however, if we want to keep the visual representation of functions introduced above. We could generalize that representation to give displays of the form

Hélène	*UK*	□
Kiyoko	*Japan*	⬢
Hélène	*France*	⬬
Kiyoko	*Italy*	⬢
Helmut	*Japan*	□

representing here a function in $PERSON \times COUNTRY \xrightarrow[f]{} SHAPE$, the function

{<<*Hélène*, *UK*>, □>,<<*Kiyoko*, *Japan*>, ⬢>, <<*Hélène*, *France*>, ⬬>,

<<*Kiyoko*, *Italy*>, ⬢>, <<*Helmut*, *Japan*>, □>}

But this representation is less convenient than for unary functions. In particular, it becomes much more difficult to check visually that the table indeed represents a function; since the number of members of $A \times B$ is the product of the numbers of members in each, the size of the table will soon become too large.

To solve this problem we can resort to a mathematical device known as currying.

32A.8 CURRYING

Consider a function *f2* corresponding to the usual notion of a function with two arguments. For any applicable pair of values in *A* and *B* respectively, *f2* yields a result in *C*. As we have seen, *f2* will be for some *A*, *B* and *C* a member of the set

$FUNC2 \triangleq A \times B$ ⇸ C

We can associate with *f2*, through a one-to-one correspondence, a function *f1* — the “curried” form of *f2* — having just one argument. Here is how.

Function *f1* will be a member of the set

$FUNC1 \triangleq A$ ⇸ $(B$ ⇸ $C)$

What *f1* represents, like any other member of *FUNC1*, is a function taking one argument (a member of *A*), and yielding for any applicable value of that argument (that is to say, any member of the function’s domain) **another function**, itself a member of B ⇸ C — that is to say a function that for any applicable value in *B* yields a result in *C*.

Given *f2*, the associated *f1* has as its domain the set of all members *a* of *A* for which there exists a member *b* of *B* such that the pair <*a*, *b*> belongs to the domain of *f2*. Now take such an *a*. The value of *f1* (*a*) is itself a function, from *B* to *C*. Let us call that function

$f1_a$. The domain of $f1_a$ is the set of all members b of B such that $<a, b>$ belongs to the domain of $f2$. For any such b, the value of $f1_a\ (b)$ is simply $f2\ (a, b)$. (If this paragraph is not immediately luminous, just read on the explanations and examples that follow.)

The transformation which for any two-argument function $f2$ yields the corresponding one-argument function $f1$ is called currying (not because it was designed over dinner in some oriental restaurant but after the name of an English mathematician). Currying is itself a function — a total one, since every function, whether finite, partial or total will always have a curried version. If we call *curry* the function that curries functions (over some arbitrary basic sets A, B and C as above) we see that it is a member of the set

$$(A \times B \nrightarrow C) \to (A \nrightarrow (B \nrightarrow C))$$

If we are only interested in finite functions — which is the case for visual programming applications studied in this book — we can replace all the $\nrightarrow$ symbols (but not the $\to$, of course) by $\underset{f}{\nrightarrow}$. Function *curry* is not only total but one-to-one: every two-argument function in $A \times B \nrightarrow C$ has a one-argument curried version in $A \nrightarrow (B \nrightarrow C)$, and every such one-argument function is the curried form of some two-argument function.

Using the first argument is a matter of convention. It is possible to curry a multi-argument function with respect to any argument.

Were it not for Professor Curry, the function could have been called *specialize*, since this is what it does: taking a function with two arguments and specializing it with respect to its first argument. Starting with the two-argument function $f2$, the curried version $f1$ takes only one argument, and is such that $f1\ (a)$ — what was called $f1_a$ above — is like $f2$ but with the first argument set to be a in all cases.

Consider for example the addition function on integers, which we may call *add2*. It takes two arguments and yields their sum. It is a member of $\boldsymbol{N} \times \boldsymbol{N} \to \boldsymbol{N}$. (Addition on integers is a total function, so in this example all the arrows can be written as $\to$ rather than just $\nrightarrow$.) The function, $add1 \triangleq curry\ (add2)$ — the curried version of *add2* — is a member of

$$\boldsymbol{N} \to (\boldsymbol{N} \to \boldsymbol{N})$$

that is to say a function which, for any integer n, yields a function which, for any integer m, yields an integer.

What concretely is *add1*? Well, for any n, $add1\ (n)$ is the function which, for any m, yields $n + m$. For example, $add1\ (1)$ is the "successor" function on integers:

$$add1\ (1) = next$$

where *next*, as introduced earlier, is the function in $\boldsymbol{N} \to \boldsymbol{N}$ defined by the property that $next\ (n)$ is $n + 1$ for every integer n. Similarly, $add1\ (–1)$ is the function *previous* that subtracts one from any integer; $add1\ (0)$ is the identity function on $\boldsymbol{N}$ — the function that, for any n, returns n itself; $add1\ (2)$ is the function that adds 2 to any integer; and so on.

One way to describe currying informally is to say that this mechanism trades argument complexity (cartesian product level) for result complexity (higher-level functions). Define the argument level of a function as the number of $\times$, plus one, in the definition of its source set (that is to say the number of function arguments), and its result level as the number of arrows, plus one, in its target set. For example a function in $((A \times B) \times C) \nrightarrow (D \nrightarrow E)$ has argument complexity 3 and result complexity 2.

If we curry it we get a function in $(A \times B) \nrightarrow (C \nrightarrow (D \nrightarrow E))$, with argument level 2 and result level 3. If we curry this curried version once more, we get a function in $A \nrightarrow (B \nrightarrow (C \nrightarrow (D \nrightarrow E)))$, with respective levels 1 and 4. Each currying operation reduces the argument level by 1 but increases the function level by 1.

The is example illustrates the need for a notational convention to reduce the number of parentheses. As soon as we enter the spice-rich world of currying we start dealing with higher-level function spaces such as $A \nrightarrow (B \nrightarrow (C \nrightarrow (D \nrightarrow E)))$; so to make things lighter it is customary to allow dropping parentheses with the understanding that arrows associate from the right. As a result the notation $A \nrightarrow B \nrightarrow C \nrightarrow D \nrightarrow E$ means the same as the last expression. Similarly, a state was defined in the previous chapter as a member of the set $CONTEXT \xrightarrow[f]{} EVENT \xrightarrow[f]{} COMMAND$, which should be understood as an abbreviation for $CONTEXT \xrightarrow[f]{} (EVENT \xrightarrow[f]{} COMMAND)$.

For a different grouping or to remove risks of ambiguity, you may of course reinstate the parentheses.

32A.9 A DIGRESSION: CORRECTNESS OF COMPILERS AND INTERPRETERS

In a totally different application domain, currying helps understand notions related to software, programming languages and programming tools. In particular, the notions of compiler and interpreter are not always well understood, so it is useful to provide precise definitions. This example is not directly relevant to EiffelBuild, but it should help you improve your appreciation of the concepts. As on the other topics of this chapter, the book *Introduction to the Theory of Programming Languages* provides more extensive discussions and examples.

Consider a programming language, say Eiffel (although any other example would do). It defines the set *EIFFEL_SYSTEM* of valid systems (programs). Let *EXECUTABLE* be the set of machine-language programs for a certain architecture, say Intel X86. We may abstractly consider that set as a function:

$$EXECUTABLE \triangleq INPUT \nrightarrow OUTPUT$$

where *INPUT* is the set of all possible program inputs, and *OUTPUT* the set of all possible program outputs. What this definition expresses is that a machine program defines a function that, for any possible input, should produce the corresponding output. Such functions are partial (hence the $\nrightarrow$ and the word "should" in the preceding sentence) because a program may for some inputs enter into an infinite loop or recursion, or crash, and hence fail to yield an output.

Now consider a compiler for Eiffel. It is a mechanism to transform Eiffel systems into machine programs; mathematically this means that the compiler is the implementation of a function *compiler* — a member of the set

$$EIFFEL\_SYSTEM \nrightarrow EXECUTABLE$$

Next consider an interpreter. Unlike a compiler, it is able to execute a system directly without first translating it to another form. What you feed into such execution is not just the input but also the software itself; to do its job, the interpreter has an equal need for both of these two elements.

Looking at the interpreter properties from a mathematical perspective, we may understand the interpreter as implementing a two-argument function *interpreter*, a member of the set

$$EIFFEL\_SYSTEM \times INPUT \nrightarrow OUTPUT$$

Some software development environments provide both a compiler and an interpreter for the same language. The goal is to let environment users use the interpreter when they need fast turnaround, and the compiler when they need the highest possible performance.

ISE Eiffel applies a more sophisticated form of this technique: its Melting Ice Technology integrates both compiled and interpretative techniques, but the choice between the two kinds is done automatically by the environment according to the needs of the moment, rather than by users selecting one or the other.

The presence of both a compiler and an interpreter raises a tricky issue: how to guarantee that the semantics is the same — in other words, that you will not get some results when you are using the interpreter during development, and different ones when you move to production and start relying on the compiler?

In the above framework the semantics compatibility requirement may be expressed simply: what we want is

$$\boxed{compiler = curry\ (interpreter)}$$

32A.10 CURRYING FOR VISUAL DISPLAY

Let us return now to graphical application building. Currying provides us with a simple solution to the problem of displaying multi-argument functions. Displaying such a function directly in a tabular form is, as we saw, neither convenient nor convincing.

The tabular format is only satisfactory with two columns, one for the input and one for the output — that is to say for single-argument functions, so that it is easy to check visually for the function property. The solution, then, is to curry multi-argument functions as many times as needed until we only have single-argument equivalents.

Our earlier example was the display of the following function, with argument level 2:

{<<*Hélène*, *UK*>, □>, <<*Kiyoko*, *Japan*>, ⯃>, <<*Hélène*, *France*>, ⬬>,

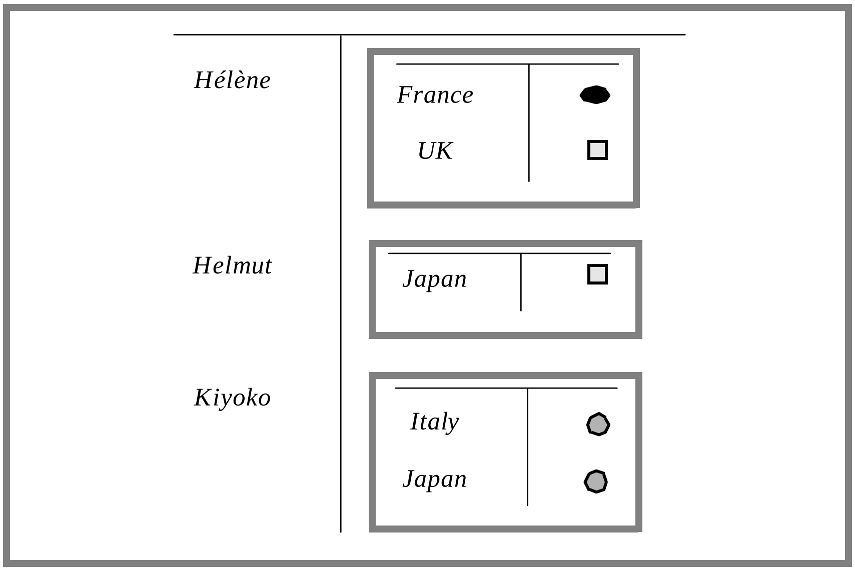
<<*Kiyoko*, *Italy*>, ⯃>, <<*Helmut*, *Japan*>, □>}

Currying it means specializing on the first argument; or, to put it differently, considering separately the country-to-shape correspondences induced by each of the persons: the correspondence for Julie, the correspondence for Kiyoko and so on. Here is the curried version, one line per person, source sets alphabetically ordered:

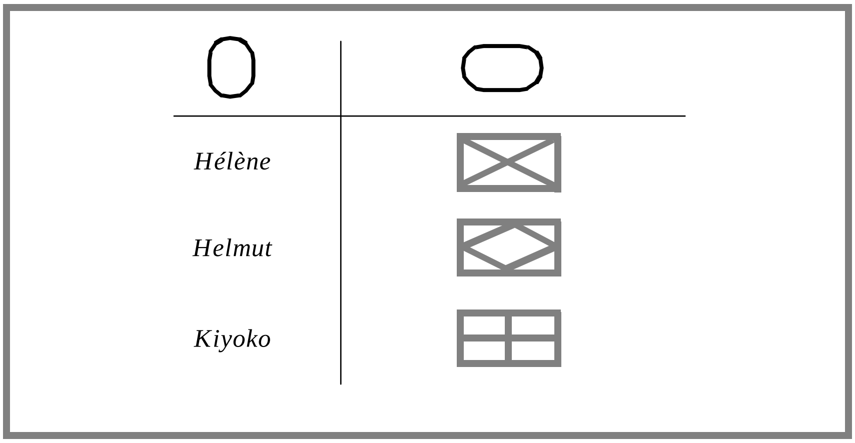
{<*Hélène*,	{<*France*, ⬬>, <*UK*, □>}>,
<*Helmut*,	{<*Japan*, □>}>
<*Kiyoko*,	{< *Italy*, ⯃>, < *Japan*, ⯃>}>}

The first element on each line is a person; the second is a finite function from countries to shapes.

The visual representation readily adapts to this curried form: since we are back to dealing with one-argument functions only, it suffices to nest the tabular display devised for such functions:

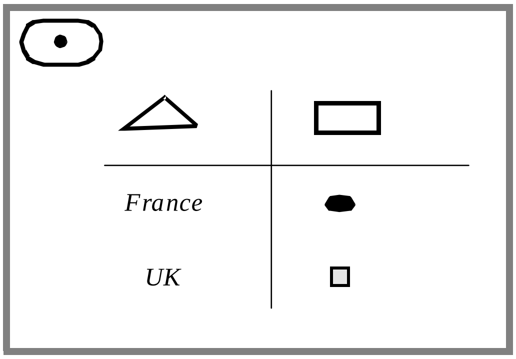


For displaying an existing finite function on paper this nested technique may be convenient if the function is not too large. With an interactive computer system, too much nesting may lead to an impractical user interface, especially if you are building or expanding a function rather than just exploring an existing one. We may in his case replace nesting by zooming: a function of the above form may appear initially as something like



where the elements on the right are placeholders for the right-hand side functions. Note that this figure includes again the argument and result holes introduced earlier as the basic device enabling interactive users to build and modify the function.

If you want to work on one of the right-hand side function you will start the corresponding tool (in the ISE Eiffel sense of this term). For example if the tool is targeted to the result — itself a function — of the curried function applied to *Hélène*, it will show:



Note the (filled) tool hole at the top left. It applies to the tool's contents (its target) as a hole; you can use it to drag-and-drop the entire function into the result hole of the primary function — the top-right hole of the immediately preceding figure.

32A.11 FUNCTIONS AND CURRYING FOR EIFFELBUILD

See "States", 1.4.6, page 7.

The concepts introduced in this chapter are the mathematical basis for EiffelBuild's approach to building GUI applications. In the Context-Event-Command-State model of EiffelBuild, we define each state as a function in the set of states

$$STATE2 \triangleq CONTEXT \times EVENT \xrightarrow[f]{} COMMAND$$

What this means is that a state is defined by how user actions will be interpreted when the session is in that state. The domain of the function is the set of <*context, event*> pairs that are recognized:

- What contexts are known to the system in that state, for example a certain button and a certain window.
- What events will be meaningful for each of these contexts — for example a mouse click on the button, and a mouse movement that brings the cursor into the window or out of it.

For each <*context, event*> pair in the state function's domain, the state function defines a result: the command that must be executed whenever the given event occurs in the given context. For example if the button reads **SAVE**, a mouse click occurring in that button should cause the *Save* command to be executed. Contexts, events, commands and states are all Eiffel objects, either extracted from a catalog of predefined elements or instances of classes defined by the software developer.

An entire interactive application consists of one or more states. Many simple applications will have just one state, but as we saw in the preceding chapter it may be desirable to have several states, interconnected through a transition diagram.

An essential part of building an application with EiffelBuild, then, will be to construct its states, that is to say to specify visually, through the environment's drag-and-drop mechanism, what events should trigger what commands in what states.

Because states are two-argument functions, we should apply currying and its visual counterpart. To define the set of states, then, we will not use *STATE2* as defined above but its curried version:

$STATE1 \triangleq CONTEXT \xrightarrow[f]{} EVENT \xrightarrow[f]{} COMMAND$

(Remember that there are implicit parentheses around the rightmost two sets.)

We could of course have used currying on events rather than contexts, defining states as members of the set

$EVENT \xrightarrow[f]{} CONTEXT \xrightarrow[f]{} COMMAND$

But this would be less convenient because of how interactive applications and the underlying toolkits are typically organized. For each kind of context, only a few events make sense — such as “Click”, “Release” and “Activate” for a certain kind of button; so the normal way to proceed is:

0 • Select a state *st*.

1 • Determine the list of contexts that are relevant for *st*; mathematically, this is the domain of the function *st*.

2 • For each context *ct* in that list, determine the list of events that are applicable to *ct*; mathematically, this is the domain of the function *st* (*ct*) — that is to say, the function from events to commands that results from applying *st* to its domain member *ct*.

3 • For each event *ev* in that list, determine the command that must be executed when *ev* occurs within *ct* for state *st*.

Although it would be possible to change this sequence of steps to specialize on events before contexts, this would mean that once we have chosen a state and an event we may have to build a big function, since an event is often applicable to many contexts. In contrast, once we have chosen a context, only a small number of events will typically be applicable to a given context. So currying on contexts first yields a more modular approach to the interactive construction of a state, with only one potentially large function (the state), all the others (the context-command mappings) being typically small even if there are many of them.

As a result a new development abstraction emerges, important to the practical understanding of EiffelBuild: the **behavior**. A behavior is a member of the set of functions

$BEHAVIOR \triangleq EVENT \xrightarrow[f]{} COMMAND$

So the set *STATE1* introduced above is $CONTEXT \xrightarrow[f]{} BEHAVIOR$; a state is a function that associates a behavior (a finite function from events to commands) with each one of a certain set of contexts.

To build a state is to build a curried two-argument function. In EiffelBuild, the State Tool and the Behavior format of the Context Tool use the principles developed in the earlier sections by enabling you to associate a behavior with t every context applicable to a state.