# 25

**This revision: Eiffel 4.70-1, 23 May 2000 19:22 (SB time)**
*Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2000. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.*

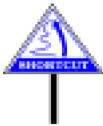
# Agents, iteration and introspection

## 25.1 OVERVIEW

Objects represent information equipped with operations. Operations and objects are clearly defined concepts; no one would mistake an operation for an object.

For some applications — numerical computation, iteration, writing contracts, building development environments, and "introspection" (a system's ability to explore its own properties) — you may find the *operations* so interesting on their own as to treat them as *objects* and pass these objects around to software elements, which can use them to execute the operations whenever they want. Because this separates the place of an operation's *definition* from the place of its *execution*, the definition can be incomplete, since you can provide any missing details at the time of any particular execution.

You can create **agent** objects to describe such partially or completely specified computations. Agents combine the power of higher-level functionals — operations acting on other operations — with the safety of Eiffel's static typing system.

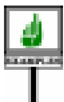



Agents are not for the beginning Eiffel user. If this is your first reading, you should most likely skip this chapter.

## 25.2 A QUICK PREVIEW

Why do we need agents? The rest of this chapter will present a detailed rationale, but it does not hurt to start with a few example uses. This preview contains few explanations, so if this is your first brush with agents some of it may look mysterious; it will, however, give you an idea of the mechanism's power, and by chapter end all the details will be clear.

Assume you want to integrate a function *g (x: REAL): REAL* over the interval [0, 1]. With *your_integrator* of a suitable type *INTEGRATOR* (detailed later) you will simply write the expression




```
your_integrator.integral (~g (?), 0.0, 1.0)
```

Here *~g* (**?**), the first argument to *integral*, is an **agent expression**, distinguished by a tilde character **~** appearing before the function name, *g*. The tilde avoids any confusion with a routine call such as *g* (*3.5*): at the place we call *integral*, we don't want to compute *g* yet! Instead, what we pass to *integral* is a "agent" object enabling *integral* to call *g* when it pleases, as often as it pleases, on whatever values it pleases.

We must tell *integral* where to substitute such values for *x*, at the places where its algorithm will need to evaluate *g* to approximate the integral. This is the role of the question mark **?**, replacing the argument to *g*.

We may use the same scheme in

```
your_integrator.integral (~h (?, u, v), 0.0, 1.0)
```

to compute the integral $\int_0^1 h\,(x, u, v)\,dx$, where *h* is a three-argument function *h* (*x*: *REAL*; *a*: *T1*; *b*: *T2*): *REAL* and *u* and *v* are arbitrary values. As before we use a question mark at the "open" position, corresponding to the integration variable *x*, and fill in the "closed" positions with actual values *u* and *v*.

Note the flexibility of the mechanism: it allows you to use the same routine, *integral*, to integrate a one-argument function such as *f* as well as functions such as *h* involving an arbitrary number of extra values.

You can rely on a similar structure to provide iteration mechanisms on data structures such as lists. Assume a class *CC* with an attribute

```
intlist: LINKED_LIST [INTEGER]
```

and a function

```
integer_property (i: INTEGER): BOOLEAN
```

returning true or false depending on a property involving *i*. You may write

```
intlist.for_all (~integer_property (?))
```

to denote a boolean value, true if and only if every integer in the list *intlist* satisfies *integer_property*. This expression might be very useful, for example, in a class invariant. It is interesting to note that it will work for any kind of *integer_property*, even if this function involves attributes or other features of *CC*, that is to say, arbitrary properties of the current object.

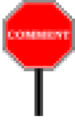
Now assume that in *CC* you also have a list of employees:

```
emplist: LINKED_LIST [EMPLOYEE]
```

and that class *EMPLOYEE* has a function *is_married*: *BOOLEAN*, with no argument, telling us about the current employee's marital status. Then you may also write in *CC* the boolean expression

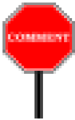
```
emplist.for_all ({EMPLOYEE}~is_married)
```

to find out whether all employees in the list are married. The argument to *for_all* is imitated from a normal feature call *some_employee*. *is_married*, but instead of specifying a particular employee we just give the type {*EMPLOYEE*}, to indicate where *for_all* must evaluate *is_married* for successive targets taken from the the list. Using a tilde instead of a dot signifies that the expression we pass to *for_all* is not the result of a call to *is_married* (a boolean value, invalid here) but the feature *is_married* itself.



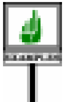

What is remarkable in the last two examples is again the flexibility of the resulting iteration mechanism and its adaptation to the object-oriented form of computation: you can use the same iteration routine, here *for_all* from class *LINKED_LIST*, to iterate actions applying to either:

- The **target** of a feature, as with *is_married*, a feature of class *EMPLOYEE*, with no arguments, to be applied to its *EMPLOYEE* target.
- The **actual argument** of a feature, as with *integer_property* which evaluates a property of its argument *i* — and may or may not, in addition, involve properties of its target, an object of type *CC*.




It seems mysterious that a single iterator mechanism can handle both cases equally well We will see how to write *for_all* and other iterators accordingly. The trick is that they work on their "open" operands, and that when we call them we may choose what we leave open: either the argument as in the *is_positive* and *integral* case, where the open position is represented by a question mark, or the target, as in the *is_married* case.

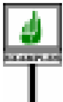
Now assume that you want to pass to some other software component, in the style of STL — the C++ "Standard Template Library" — the mechanisms needed to execute the cursor resetting and advance operations, *start* and *forth*, on a particular list. Here nothing is left open: you fix the list, and the operations have no arguments. You may write



```
other_component. some_feature (your_list~start, your_list~forth)
```

All operands — target and arguments — of the agents passed to *other_component* are "closed", so *other_component* can execute call operations on such objects without providing any further information.

At the other extreme, you might leave an agent expression fully open, as in



```
other_component.other_feature ({LINKED_LIST}~extend (?))
```

so that *other_component*, when it desires to apply a call operation, will have to provide both a linked list and an actual argument to execute *extend*.

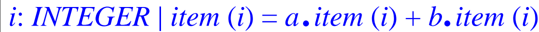
You will indeed be able, whenever you have an agent object, to apply to it a procedure *call*, whose arguments are the open operands of the original agent expression (*call* has no arguments if all operands are closed, as in the next-to-last example). This will have the same effect as an execution of the original feature — *start*, *forth*, *extend* — on a combination of the closed and open arguments.

In the end an expression such as {*LINKED_LIST*}~*extend* (**?**), which can in fact be written just {*LINKED_LIST*}~*extend* without any explicit argument, denotes a "**routine object**": a representation of the routine *extend* from *LINKED_LIST*, such as could be used by browsing tools or other *introspective* facilities.

All these examples used, to define the agents, a routine of a class. This is indeed the most common case. But for more flexibility — especially useful when you use agents to express advanced contracts — you may also write an **inline agent**, built from an arbitrary expression or instruction, with explicit entities representing the open positions. For example we may rewrite the earlier agent expression ~*integer_property* (**?**) as an inline agent

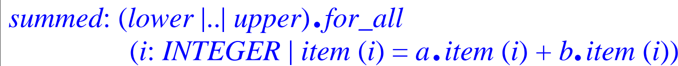
*i*: *INTEGER* | *integer_property* (*i*)

which means exactly the same thing; the convention is simply that you name and declare the open argument *i* — as if it were a routine argument — instead of referring to it implicitly through a question mark. The vertical bar | is the defining mark of inline agents. If *integer_property* is a function of the enclosing class this form is not very useful, and in fact the non-inline variant ~*integer_property* (**?**) is more concise and usually preferable. But with the inline form you can also write an agent such as



*i*: *INTEGER* | *item* (*i*) = *a*.*item* (*i*) + *b*.*item* (*i*)

which could be useful for example in a postcondition



```
summed: (lower |..| upper).for_all
        (i: INTEGER | item (i) = a.item (i) + b.item (i))
```

expressing that for every element *i* of the interval *lower* |..| *upper* the value of the item at position *i* (in a structure such as an array or list) is the sum of the corresponding values in *a* and *b*. Here too the non-inline form is possible in principle, and more concise, since we can write the agent as *is_sum_of* (**?**, *a*, *b*), using a boolean-valued function *is_sum_of* such that *is_sum_of* (*i*, *x*, *y*) is true if and only if *item* (*i*) = *x*.*item* (*i*) + *y*.*item* (*i*). But if our postconditions and invariants need lots of properties of this kind, we will end up introducing numerous routines such as *is_sum_of* with no other use in the class. In such cases, inline agents are useful. For an agent involving a single routine such as *integer_property*, *integral*, *is_married*, *extend* and the other examples above, the original non-inline form using the tilde ~ and question mark **?** is shorter, more abstract and hence preferable.

You may wonder how this can all work in a type-safe fashion. So it is time to stop this preview and cut to the movie.

→ *For other interesting applications see "TWO ADVANCED EXAMPLES", 25.11, page 661.*

## 25.3 NORMAL CALLS

First we should remind ourselves of the basic properties of **feature calls**. When programming with Eiffel we rely all the time on this fundamental mechanism of object-oriented computation. We write things like

*← Feature calls were studied in chapter 23 and their type properties in chapter 24.*

| [Q] | *a0.f (a1, a2, a3)* |
|---|---|

to mean: call feature *f* on the object attached to *a0*, with actual arguments *a1*, *a2*, *a3*. In Eiffel this is all governed by type rules, checkable statically: *f* must be a feature of the base class of the type *a0*; and the types of *a1* and the other actuals of the call must all conform to the types specified for the corresponding formals in the declaration of *f*.

In a frequent special case *a0*, the **target** of the call, is just *Current*, denoting the current object. Then we may omit the dot and and the target altogether, writing the call as just

| [U] | *f (a1, a2, a3)* |
|---|---|

which assumes that *f* is a feature of the class in which this call appears. The first form, with the dot, is a *qualified* call; the second form is *unqualified* (hence the names [Q] and [U] given to our two examples).

In either form the call is syntactically an expression if *f* is a function or an attribute, and an instruction if *f* is a procedure. If *f* has been declared with no formals (as in the case of a function without arguments, or an attribute) we omit the list of actuals, *(a1, a2, a3)*.

The effect of executing such a call is to apply feature *f* to the target object, with the actuals given if any. If *f* is a function or an attribute, the value of the call expression is the result returned by this application.

To execute properly, the call needs the value of the target and the actuals, for which this chapter needs a collective name:

**Operands of a call**

The operands of a call include its target (explicit in a qualified call, implicit in an unqualified call), and its arguments if any.

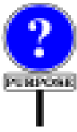
In the examples the operands are *a0* (or *Current* in the unqualified version [U]), *a1*, *a2* and *a3*. Also convenient is the notion of *position* of an operand:

**Operand position**

The target of a call (implicit or explicit) has position 0. The *i*-th actual argument, for any applicable *i*, has position *i*.

Positions, then, range from 0 to the number of arguments declared for the feature. Position 0, the target position, is always applicable.

## 25.4 FROM CALLS TO AGENTS



For a call such as the above, we expect the effect just discussed to occur as a direct result of executing the call instruction or expression: the computation is immediate. In some cases, however, we might want to write an expression that only *describes* the calls intended computation, and to *execute* that description later on, at a time of our own choosing, or someone else's. This is the purpose of agent expressions, which may be described as **delayed calls**.

Why would we delay a call in this way? Here are some typical cases:

A •We might want the call to be applied to all the elements of a certain structure, such as a list. In such a case we will specify the agent expression once, and then execute it many times without having to re-specify it in the software text. The software element that will repeatedly execute the same call on different objects is known as an **iterator**. Function *for_all*, used earlier, was an example of iterator.

B •In an iterator-like scheme for numerical computation, we might use a mechanism that applies a call to various values in a certain interval, for example to approximate the integral of a function over that interval. The first example in this chapter relied on such an *integral* function.

C •We might want the call to be executed by another software element: passing an agent object to that element is a way to give it the right to operate on some of our own data structures, at a time of its own choosing. This was illustrated with the calls passing to *other_component* some agent expressions representing operations applicable to *your_list*.

D •We might specify that any future creation of objects of a certain type apply the call to initialize these objects.

E •We might want to ensure that the call is executed only when and if needed, and then only once for any particular object. This would give us a "once per object" mechanism along the lines of "once functions" (which are executed once per system).

*Once functions see "ROUTINE BODY", 8.6, page 270. The once per object mechanism using agents is described below.*

F •Finally, we may be interested in the agent as a way to gain information about the feature itself, whether or not we ever intend to execute the call. This may be part of the more general goal of providing **introspective** capabilities: ways to enable a software system to explore and manipulate information about its own properties.

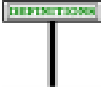
*Introspection is also called* **reflection**, *but the first term appears more appropriate.*

These examples illustrate one of the differences between agent expressions and calls: to execute a call we need the value of all its operands (target and actuals); but for an agent expression we may want to leave some of the operands open for later filling-in. This is clearly necessary for cases A and B, in which the iteration or integration mechanism will need to apply the feature repeatedly, using different operands each time. In an integration

$$\int_{x=a}^{x=b} g(x)\, dx$$

we will need to apply $g$ to successive values of the interval $[a, b]$.

For an agent we need to distinguish between two moments:



**Construction time, call time**

The **construction time** of an agent object is the time of evaluation of the agent expression defining it.

Its **call time** is when a call to its associated operation is executed.

Since the only way to obtain an agent initially is through *agent expressions*, as specified next, it is meaningful to talk about the "agent expression defining it".

→ *A precise definition of "open" and "closed" operands appears on page 666.*

*Readers familiar with lambda calculus may think of open as "free" and closed as "bound".*

For a normal call the two moments are the same. For an agent we will have one construction time (zero if the expression is never evaluated), and zero or more call times. At construction time, we may leave some operands unspecified; they they will be called the *open* operands. At call time, however, the execution needs all operands, so the call will need to specify values for the open operands. These values may be different for different executions (different call times) of the same agent expression (with a single construction time).

There is no requirement that **all** operands be left open at creation time: we may specify some operands, which will be closed, and leave some other open. In the example of computing, for some values $u$ and $v$, the integral

$$\int_{x=a}^{x=b} h(x, u, v)\, dx$$

where $h$ is a three-argument function, we pass to the integration mechanism an agent that is closed on its last two operands ($u$ and $v$) but open on $x$.

Nothing forces you, on the other hand, to leave **any** operand open. An agent with all operands closed corresponds to the kind of application called C above, in which we don't want to execute the call ourselves but let another software element *other_component* carry it when it is ready. We choose the construction time, and package the call completely, including all the information needed to carry it out; *other_component* chooses the call time. This style is used by iterators in the C++ STL library.

At the other extreme, an agent with **all operands open** has no information about the target and actuals, but includes all the relevant information about the feature. This is useful in application F: passing around information about a feature for introspection purposes, enabling a system to deliver information about its own components.

## 25.5 WHAT IS AN AGENT EXPRESSION?

A normal call is a syntactical component — instruction or expression — meant only for one thing: immediate execution. If it is an expression (because the feature is a function), it has a value, computed by the execution, and so denotes an object.

DEFINITION

An agent expression has a different status. Since construction time is separate from call time, the agent expression can only **denote an object**. That object, called an *agent*, contains all the information needed to execute the call later, at various call times. This includes in particular:

- Information about the routine itself and its base type.
- The values of all the closed operands.

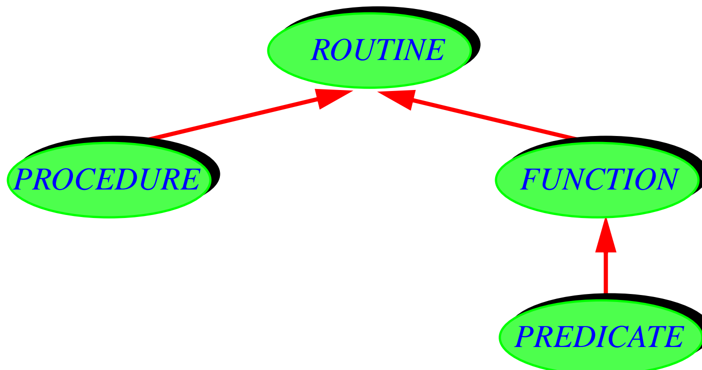
What is the type of an agent expression? Four Kernel Library classes are used to describe such types: *ROUTINE*, *PROCEDURE*, *FUNCTION* and *PREDICATE*. Their class headers start as follows:

```
deferred class ROUTINE [BASE, OPEN –> TUPLE]

class PROCEDURE [BASE, OPEN –> TUPLE] inherit
    ROUTINE [BASE, OPEN]

class FUNCTION [BASE, OPEN –> TUPLE, RES] inherit
    ROUTINE [BASE, OPEN]
class PREDICATE [BASE, OPEN –> TUPLE] inherit
    FUNCTION [BASE, OPEN]
```

In the actual class texts, the formal generic matters have names *BASE_TYPE*, *OPEN_ARGS* and *RESULT_TYPE* to avoid conflicts with programmer-chosen class names. This chapter uses shorter names for simplicity.

→ *A.6.26 to A.6.28 in the ELKS chapter, starting on page 896.*

If the associated feature is a procedure the agent will be an instance of *PROCEDURE*; for a function or attribute, we get an instance of *PREDICATE* when the result is boolean, of *FUNCTION* with any other type. Here for ease of reference is a picture of the inheritance hierarchy:




***Agent classes***

The role of the formal generic parameters is:

- *BASE*: type (class + generics if any) to which the feature belongs.
- *OPEN*: tuple of the types of open operands, if any.
- *RES*: result type for a function.

One of the fundamental features of class *ROUTINE* is

```
call (v: OPEN) is
        -- Call feature with all its operands, using v for the open operands.
```

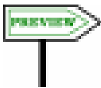
In addition, *FUNCTION* and *PREDICATE* have the feature

```
last_result: RES
        -- Function result returned by last call to call, if any
```

and, for convenience, the following function combining *call* and *last_result*:

```
item (v: like open_operands): RES is
            -- Result of calling feature with all its operands,
            -- using v for the open operands.
            -- (Uses call for the call.)
    ensure
            set_by_call: Result = last_result
```

Note that the formal generic parameters for *ROUTINE*, *PROCEDURE*, *FUNCTION* and *PREDICATE* provide what we need to make the agent mechanism statically type-safe. In particular the *OPEN* parameter, a tuple type, gives the exact list of open operand types; since the argument to *call* and *item* is of type *OPEN*, the compiler can make sure that at call time the actual arguments to *call* will be of the proper types, conforming to the original feature's formal argument types at the open positions. The actuals at closed positions are set at construction time, again with full type checking. So the combination of open and closed actuals will be type-valid for the feature.



*ROUTINE*, *PROCEDURE*, *FUNCTION* and *PREDICATE* have more features than listed above; in particular, they provide introspection facilities, describing properties of the associated routines and discussed below. For a complete interface specification, see the corresponding sections in the presentation of Kernel Library classes.

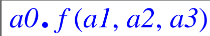
→ *Sections A.6.26 to A.6.28, starting on page 896.*

## 25.6 AGENT EXPRESSIONS

How do we produce agent objects? We use *agent expressions* for which the basic syntactical rule is very simple: start from a normal feature call; in the qualified case, replace its dot with a tilde; in the unqualified case, add a tilde before the feature name. (The syntax is different for *inline* agent expression, discussed in a later section.)

→ *"INLINE AGENTS", 25.17, page 675.*

So if a valid call of the qualified form is

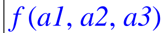


```
a0.f (a1, a2, a3)
```

you get an agent expression by replacing the dot with a tilde:

```
a0~f (a1, a2, a3)
```

In the case of a valid unqualified call



```
f (a1, a2, a3)
```

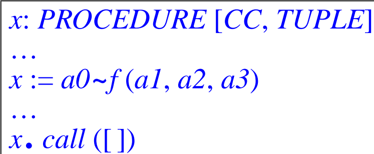
where *f* is a feature of the enclosing class, you obtain the corresponding call expression by adding a tilde:

```
~f (a1, a2, a3)
```

In either case, the new notation is not a call (instruction or expression) any more, but an expression of a new syntactic kind, Feature_agent, which denotes an agent, of a *PROCEDURE* type if *f* is a procedure and a *FUNCTION* type if *f* is a function.

You can do with an agent expression all you are used to do with other expressions. You can assign it to an entity of the appropriate type; assuming *f* is a procedure of a class *CC*, you may write, in class *CC* itself:

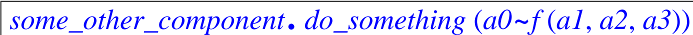
*This example assumes that CC is non-generic, so that it is both a class and a type.*



```
x: PROCEDURE [CC, TUPLE]
…
x := a0~f (a1, a2, a3)
…
x.call ([ ])
```

Note that here all operands are closed since we specified the target *a0* and all the operands *a1*, *a2*, *a3*, so the second formal generic is just *TUPLE*, and the call to *call* takes an empty tuple [ ].

More commonly than assigning a call expression to an entity as here, you will pass it as actual argument to a routine, as in



```
some_other_component.do_something (a0~f (a1, a2, a3))
```

where *do_something*, in the corresponding class, takes a formal *p* declared as

```
p: PROCEDURE [CC, TUPLE]
```

or just

```
p: PROCEDURE [ANY, TUPLE]
```

presumably to call *call* on *p* at some later stage, as we shortly learn to do. This is the scheme that was called C in the presentation of example applications: passing a completely closed agent to another component of the system, to let it execute the call when it chooses to. For example you can pass *your_list~start* or *your_list~extend (some_value)*.

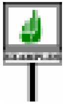
← *Scheme C was on page 652.*

## 25.7 KEEPING OPERANDS OPEN

The examples just seen are still of limited interest because all their operands are closed. What if you want to keep some operands open for latter filling-in at call time, for example by an iteration or integration mechanism?

For arguments (we will see how to handle the target in a short while) the basic technique is very simple: to keep an actual open, just replace it by a question mark. This yields examples such as

→ *About the target see "LEAVING THE TARGET OPEN", page 659.*

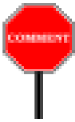


```
w := a0~f (a1, a2, ?)
x := a0~f (a1, ?, a3)
y := a0~f (a1, ?, ?)
z := a0~f (?, ?, ?)
```

The respective types of these call expressions are, assuming that *f* is a procedure with formals declared of types *T1*, *T2* and *T3*:

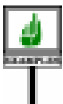
```
w: PROCEDURE [T0, TUPLE [T3]]
x: PROCEDURE [T0, TUPLE [T2]]
y: PROCEDURE [T0, TUPLE [T2, T3]]
z: PROCEDURE [T0, TUPLE [T1, T2, T3]]
```

If *f* were a function, the types would use *FUNCTION* instead of *PROCEDURE*, with one more actual generic representing the result type. For a predicate (boolean-valued function), you may use *PREDICATE*.



You will have noted how the generic parameters of *ROUTINE* and its heirs all provide a full specification of the types involved, enabling static type checking. Consider in particular the role of the second generic parameter *OPEN* in *ROUTINE* [*BASE*, *OPEN* –> *TUPLE*] and its descendants *PROCEDURE*, *FUNCTION* and *PREDICATE*. *OPEN* represents the tuple of types of all the open operands. In the first case above, for *w*, only the last argument, of type *T3*, is left open; in the last case, for *z*, all three arguments are open. (But in all examples so far the target *a0* is closed.) This immediately indicates what argument types are permissible in calls to *call* (or *item* for a function) on the corresponding agents:

→ *The role of the first generic parameter, BASE, will be discussed in "THE BASE CLASS AND TYPE", 25.15, page 668.*



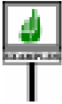
```
w.call ([e3])
x.call ([e2])
y.call ([e2, e3])
z.call ([e1, e2, e3])
```

where the types of expressions *e1*, *e2*, *e3* must conform to *T1*, *T2* and *T3* respectively. The effect of these calls is the same as what we would obtain through the following normal calls (call time same as construction time):

```
a0.f (a1, a2, e3)
a0.f (a1, e2, a3)
a0.f (a1, e2, e3)
a0.f (e1, e2, e3)
```

## 25.8 EXPLICIT TYPES FOR OPEN OPERANDS

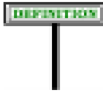
As a variant of the "question mark" specification for open operands, you might want to specify explicitly a certain type for some of the arguments. Instead of the first example



```
w := a0~f (a1, a2, ?)
```

you may write

```
wu := a0~f (a1, a2, {U3})
```




to specify that you are leaving open the last argument, and that the corresponding actual must be of type conforming to *U3*. This will be called the **braces convention**, complementing the "question mark convention" seen so far. In this case the type of *wu* is

```
wu: PROCEDURE [T0, TUPLE [U3]]
```

Clearly, this is only permitted for a type *U3* that conforms to *T3*, the type of the last formal argument. This is a general rule: where a normal call requires an operand of type $T_i$, a corresponding agent expression may use $\{U_i\}$ , for any type $U_i$ conforming to $T_i$. This means that the argument is left open, and that any corresponding actual at call time must be of a type conforming to $U_i$.

We may now reinterpret the question mark convention in terms of the braces convention: a question mark at an argument position is simply an abbreviation for $\{T_i\}$, where $T_i$ is the type of the corresponding formal. For example *y*, defined earlier as *a0~f (a1, ?, ?)* could instead have been written

```
a0~f (a1, {T2}, {T3})
```

## 25.9 LEAVING THE TARGET OPEN

The examples of open operands seen so far were open only for some or all of the arguments; the target was closed. Your may also want to leave the target open. Looking again at our staple example, we see that if we start from a normal call of the qualified form

```
a0.f (a1, a2, a3)
```

we cannot use the question mark convention to replace the target *a0*, since we need to identify the class of which *f* is a feature. But the braces convention will work. You may write

```
s := {T0}~f (a1, a2, a3)
```

to denote an agent open on its target (of type *T0*) and closed on all arguments. Of course you can open any or all of the arguments too, as in

```
t := {T0}~f (a1, a2, ?)
u := {T0}~f (?, ?, ?)
```

→ *The expression for u can be abbreviated to just {T0}~f. See* *"COMPLETELY OPEN AGENTS", page 666*.

where we get with *u*, for the first time, an agent open on all its operands, target and arguments.

The types for the last call expressions are:

```
s: PROCEDURE [T0, TUPLE [T0]]
t: PROCEDURE [T0, TUPLE [T0, T3]]
u: PROCEDURE [T0, TUPLE [T0, T1, T2, T3]]
```

Note once again how the second generic parameter of the *ROUTINE* classes corresponds to the open operands — target and arguments included.

The next section discusses the role of the first generic parameter, always representing the target type.

The order of the open operands is the one in which these operands would appear in a normal call: target, then first argument and so on. Calls to *call* on the corresponding routine objects will be instructions of the form

```
s.call ([e0])
t.call ([e0, e3])
u.call ([e0, e1, e2, e3])
```

with expressions *e0*, *e1*, *e2*, *e3* of types conforming to *T0*, *T1*, *T2*, *T3*. Note how, when it comes to applying *call* (or *item* for a function) to an agent, the target, if left open in the call expression, must be passed as part of the actual argument tuple, in the same way as an open argument.

These agent expressions will have the same effect as the normal calls

```
e0.f (a1, a2, a3)
e0.f (a1, a2, e3)
e0.f (e1, e2, e3)
```

In the unqualified case, a normal call of the form

```
f (a1, a2, a3)
```

may be viewed as an shorthand for the qualified form *Current.f (a1, a2, a3)*. Correspondingly, you may write the open-target agent expression as

```
{CC}~f (a1, a2, a3)
```

*CC is the enclosing class, assumed to be non-generic.*

but you may also use a question mark for the target:

```
?~f (a1, a2, a3)
```

Here too you may of course leave some of the arguments open, or all of them as in

```
v := ?~f (?, ?, ?)
```

→ *The expression for v can be abbreviated to just* **?~** *f.* *"COMPLETELY OPEN AGENTS", 25.13, page 666.*

## 25.10 A SUMMARY OF THE POSSIBILITIES

Although you may have the impression that the agent mechanism has many variants, it is in fact very simple. So to avoid any confusion, or impression of confusion, here is an informal description listing all the possibilities:

→ *This only covers non-inline agents. See also* *"INLINE AGENTS", 25.17, page 675.*

### How agent expressions are made

To obtain an agent expression, you *must*:

1 • Start from a valid routine call — of any form, qualified or not.

2 • **Replace the dot by a tilde** if the call is qualified; otherwise, **add a tilde** before the routine name.

In addition, to make some operands open, you *may*:

3 • **Replace any operand** (the target, or any argument) by a question mark, or a type in braces, as in *{YOUR_TYPE}*

4 • If **all the arguments** are question marks, omit the argument list (and the parentheses) altogether.

That's all there is to it!

In the next sections we continue exploring the details, and study the precise syntax, validity and semantics of agent expressions.

## 25.11 TWO ADVANCED EXAMPLES

We have now seen cases of all the variants of agent expressions, but before proceeding to a detailed analysis of all their properties let's gain further appreciation for the power and versatility of the mechanism by looking at two interesting applications: error processing and "once per object".

The first example addresses a frequent situation in which we perform a sequence of actions, each of which might encounter an anomaly that prevents continuing as hoped. The problem here is that it's difficult to avoid a complex, deeply nested control structure, since we may have to get out at any step. The straightforward implementation will look like this:

```
action1
if ok1 then
        action2
        if ok2 then
                action3
                ... More processing, more nesting ...
        end
end
```

For example we may want to do something with a file of name *path_name*. We first test that *path_name* is not void. Then that the string is not empty. Then that the directory exists. Then that the file exists. Then that it is readable. Then that it contains what we need. And so on. A negative answer at any step along the way must lead to reporting an error situation and aborting the whole process.

The problem is not so much the nesting itself; after all, some algorithms are by nature complex. But often the normal processing is not complicated at all; it's the error processing that messes everything up, hiding the "useful" processing in a few islands lost in an ocean of error handling. If the error processing is different in each case (**not** *ok1*, **not** *ok2* and so on) we can't do much about it. But if it is always of the form: "Record the error source and terminate the whole thing", then the above structure may seem unpleasantly over-complicated. Although we might use exceptions to address the problem, they are often overkill.

An agent-based technique is useful in some cases. It assumes that you write the various actions — *action1* ... *action3* above — as procedures, each with a body of the form

```
...Try to do what's needed...
controlled_check (execution_ok, "...Appropriate message...")
```

with *execution_ok* representing the condition that must be satisfied for the processing to continue. Then you can rewrite the processing above as just:

```
controlled_execute ([
        ~ action1,
        ~ action2 (...),
        ~ action3 (...)
        ])
if controlled_glitch then
        warning (controlled_glitch_message)
                -- Procedure warning is an error reporting mechanism
end
```

This linear structure is much simpler than the original.

The features whose names start with *controlled_* come from the EiffelBase class *CONTROLLED_EXECUTION*, of which the class containing the above scheme should be a descendant. These procedures are not difficult to write; for example *controlled_check* sets *controlled_glitch* and *controlled_glitch_message*, and *controlled_execute* looks like this:

*The routine as it appears in the library has a few extra instructions to record the glitch step and, on option, raise an exception.*

```
controlled_execute
        (actions: ARRAY [PROCEDURE [ANY, TUPLE]]) is
                -- Execute actions, stopping if encountering a glitch.
        local
                i: INTEGER
        do
                from
                        controlled_glitch := False; i := actions.lower
                until i > actions.upper or else controlled_glitch loop
                        actions.item.call ([ ])
                        i := i + 1
                end
        end
```

The second example, also supported by an EiffelBase class, provides an elegant "once per object" mechanism. You know, of course, Eiffel's "once routines", executed only once per system execution. They define a "once per class" mechanism: all instances of a class share the result of a once function. (All these concepts are applicable to procedures, but for this discussion we restrict ourselves to functions.) Now assume you need functions that compute a result specifric to each instance of the class, and computed just once for that instance, the first time it's requested — if at all. A typical application would be large pieces of information associated with objects of a certain type, but stored in a database; for example each instance

← *For an introduction to once routines see "ROUTINE BODY", 8.6, page 270.*

of a class *COMPANY* may have *stock_history* information, of type *HISTORY*, which may be huge. We only want to retrieve the information on demand; given the size of the information and the number of instances of the class, it is not acceptable to load everything ahead of time. Even if an instance of *COMPANY* is in memory, we want to retrieve the associated *HISTORY* from the database only when and if we need access to the company's *stock_history*.

Agents provide us with a general solution to all problems of this kind. In class *COMPANY* you will simply declare

```
stock_history: ONCE_PER_OBJECT [HISTORY]
```

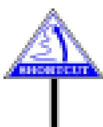
and obtain the value, when and if needed, as

```
stock_history.item (~ retrieved_history)
```

Here *retrieved_history* is the function that computes the needed result — the one that you want to call once for each object. That's all you have to do! Note that this scheme allows you to have as many "once per object" functions as you like in any given class. It relies on a general-purpose EiffelBase class *ONCE_PER_OBJECT* of the following form:

```
expanded class
      ONCE_PER_OBJECT [G]
feature -- Access
      item (f: FUNCTION [ANY, TUPLE, G]): G is
                  -- Value of f, computed once for each object;
                  -- subsequent calls return same value for same object.
            do
                  if not computed then
                        internal_result := f.item ([ ])
                        computed := True
                  end
                  Result := internal_result
            end
feature {NONE} -- Implementation
      computed: BOOLEAN
            -- Has  item already been requested?
      internal_result: G
            -- Result, if already computed
end -- class ONCE_PER_OBJECT
```

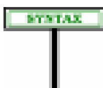
## 25.12 AGENT EXPRESSION SYNTAX




We have now seen examples of all the variants of agent expression, so it is time to give the syntax. (This section introduces no new concept, so the hurried reader may skip to the next one.)

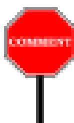
*If hurried skip to "COMPLETELY OPEN AGENTS", 25.13, page 666.*

The new construct is Agent, a variant of Expression:




| | | |
|---|---|---|
| Agent | ≜ | Feature_agent \| Inline_agent |
| Feature_agent | ≜ | [Agent_target] Agent_unqualified |
| Agent_unqualified | ≜ | "~" Feature_name [Agent_actuals] |
| Agent_target | ≜ | Entity \| Parenthesized \| Type_descriptor |
| Type_descriptor | ≜ | Explicit_type_descriptor \| Placeholder |
| Explicit_type_descriptor | ≜ | "{" Type "}" |
| Placeholder | ≜ | "?" |

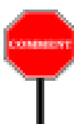
Agent_actuals *represents the actual arguments, if any, to the agent, and is specified next.*




The construct of interest for the moment is Feature_agent, corresponding to the tilde form. The inline form, Inline_agent is studied in a separate section.

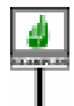
*"INLINE AGENTS", 25.17, page 675.* *Routine agents use a vertical bar |.*

The major difference between the syntax of a Feature_agent and of a normal Call is the tilde of an Agent_unqualified, which has no equivalent in the corresponding construct, Unqualified_call. This guarantees that an agent expression can never be confused for a normal call.



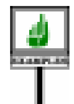

Note that the mechanism is applicable not only to identifier features but also to operator features (Infix and Prefix). The technique is very simple: just designate the feature by its Feature_name, and add a tilde to it as you would do with an identifier feature. You may remember that the Feature Name Consistency principle allows us, if § is the operator of an infix feature, to treat **infix** "§" as a normal feature name and use it in any place where a feature identifier would be legal; same thing for **prefix** "‡" if ‡ is a prefix operator. So you can use agent expressions such as

*← Feature Name Consistency principle: page 203.*



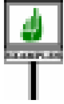

```
a ~infix "+" (b)        -- All closed
~infix "+" (?)          -- Open on argument, closed on target
? ~ prefix "+"          -- All open (open on target, no argument)
```

Coming back to the general case, the Agent_target may be absent, in which case the call will be considered closed on its target: the current object. In this case the associated feature must be a feature of the current class, and the Feature_agent will start with a tilde and a feature name, as in



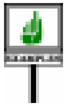

```
~f (a1, a2, a3)
~f
```

In all other cases the agent expression has an explicit Agent_target, for which the specification shows three possibilities: Entity, Parenthesized and Type_descriptor. Examples of a Agent_target of each kind are



```
e0                   -- An Entity
(a.k (x).l (y).m)    -- A Parenthesized containing a complex expression
{U0}                 -- A Type_descriptor (explicit)
```

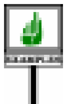
This syntax requires you, if you want to use as target an expression other than a simple entity, to enclose it in parentheses, as in the second example. There is no loss of expressiveness, since the expression you put in parentheses can still be as complicated as you like. The reason for forcing parentheses is a concern for readability. With suitable precedence rules, it would not be hard for a compiler to parse *a*.*k* (*x*). *l* (*y*).*m*~*f* (*a1*, *a2*, *a3*). Instead, you must write



```
(a.k (x).l (y).m)~f (a1, a2, a3)
```

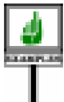
where the parentheses around the target remove any confusion arising from the presence of both dots (part of the multi-level qualified Call serving as target of the agent expression) and tildes.

The third possibility for a target includes a Type_descriptor. This may be an Explicit_type_descriptor listing the target type in braces:



```
{T0}~f (a1, a2, a3)
```

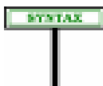
or simply a question mark, called a Placeholder, indicating an open target of the current type:



```
?~f (a1, a2, a3)
```

This expression assumes that *f* is a feature of the enclosed class; it represents an agent that is open on its target.

The part after the tilde is what the syntax productions call Agent_unqualified, which resembles the Unqualified_call component of normal calls, but with two more possibilities for an Agent_actual:



Agent_actuals $\triangleq$ "(" Agent_actual_list ")"

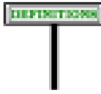
Agent_actual_list $\triangleq$ {Agent_actual "," …}

Agent_actual $\triangleq$ Actual | Type_descriptor

One of the variants of Agent_actual is Actual, meaning a normal actual argument for a call (Expression or, for an external routine, Address). The new variant is Type_descriptor, covering two forms: Placeholder (a question mark) and Explicit_type_descriptor, a type in braces.

We can define precisely what "open" and "closed" mean for the operands of an agent expression:

← *The operands of a call were defined on page 651 as including its target, and its arguments if any.*



**Open and closed operands**

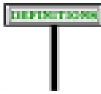
The **open operands** of a Feature_agent include:

- Its target if there is a Agent_target and it is a Type_descriptor (Explicit_type_descriptor or Placeholder).
- Any Agent_actual that is a Type_descriptor.

The **closed operands** include all non-open operands.

An earlier definition also introduced the notion of *operand position*, which we can now extend to a definition of open and closed positions:

← *"Operand position" was defined on page 651: the target position is 0, and the argument positions start at 1.*



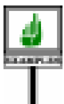
**Open and closed operand positions**

The **open operand positions** of a Feature_agent are the operand positions of its open operands, and the **closed operand positions** those of its closed operands.

## 25.13 COMPLETELY OPEN AGENTS

In some cases it will be useful to write an agent expression that denotes an an agent open on all of its arguments, and possibly on its target too.

We have seen how to obtain this effect by using question marks (Placeholder) at every argument position:



```
a0 ~ f (?, ?, ?)      -- Qualified, arguments open, target closed
~ f (?, ?, ?)         -- Unqualified, arguments open, target closed
```

← *The first expression was called z on page 657; the second one was called v on page 660.*

An abbreviation is permitted for this case: omit the parenthesized argument list (the Agent_actuals part) altogether, yielding respectively

```
a0 ~ f                -- Qualified, arguments open, target closed
~ f                   -- Unqualified, arguments open, target closed
```

These examples all have closed targets; in the unqualified case the target is the current object. The fully open variants, with open targets, are:

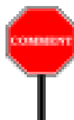
```
{T0} ~ f    -- Qualified, all operands open (target and arguments)
? ~ f       -- Equivalent to previous one if T0 is current type
```

← *The first expression has the same value as u as defined on page 659.*

The syntax just given explicitly allows all these abbreviated forms. It is the same as the full form if *f* has no arguments; but if *f* has arguments, a normal call without actuals, such as *a0*. *f*, or just *f* in the non-qualified case, would violate the Argument rule. In contrast, the Agent Expression rule, introduced later in this chapter, explicitly allows you to omit the Agent_actuals, even for a feature with arguments, as an abbreviation for a list of completely open actuals.

← *The Argument rule was on page 640.*

→ *The Agent Expression rule will appear on page 681.*



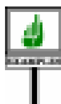
This abbreviated form has the advantage of conveying the idea that the denoted agent is a true "feature object", carrying properties of the feature in its virginal state, not tainted by any particular choice of actual argument. The last two variants shown do not even name a target. This is the kind of object that we need for such *introspective* applications as writing a system that enables its users to browse through its own classes.

## 25.14 ACCESSING FEATURE PROPERTIES

As part of introspection support, class *ROUTINE* and its descendants provide features to access the precondition and postcondition of a routine:

```
precondition (args: OPEN) BOOLEAN
        -- Do args satisfy routine's precondition in present state?

postcondition (args: OPEN) BOOLEAN
        -- Does current state satisfy routine's postcondition
        -- for operands args?
```

This enables you to check the precondition before you apply an agent, as in

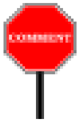


```
if your_agent.pre (your_operands) then
        your_agent.call (your_operands)
end
```

where *your_agent* is an agent expression and *your_operands* is a valid tuple of operands for that agent.

There is, as will be seen next, a similar facility for class invariants.

## 25.15 THE BASE CLASS AND TYPE




Introspection support is also one of the concerns behind the first generic parameter of *ROUTINE*, *PROCEDURE*, *FUNCTION* and *PREDICATE*. The specification

```
ROUTINE [BASE, OPEN –> TUPLE]
```

includes, as first generic parameter, the type *BASE* representing the type (class with generic parameters) to which an agent's associated feature belongs. This is the type of the target expected by the feature.

*→ Although intuitively clear, the notion of "associated feature" of an agent has not yet been defined precisely. The definition is part of the Agent Expression rule, page 681.*

The examples seen so far do not use *BASE* at all, because procedure *call* does not need it. If the agent is closed on its target, as in

```
y := a0~f (a1, ?, ?)
```

then it includes, here through *a0*, the target information that a later call to *call* may require. In the other case — open target — as in

```
t := {T0}~f (a1, a2, ?)
```

then the target type is specified, here *T0*, and provides the information needed to determine the right version of *f*. In this case the *BASE* generic parameter is in fact redundant, since it is identical to the first component of the tuple type corresponding to *OPEN*; the type of *t*, for example, is

```
ROUTINE [T0, TUPLE [T0, T3]]
```

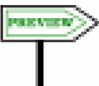
where the two tuple components correspond to the two open operands: the target, and the last argument.

In both the closed target and open target cases, then, we don't need the *BASE* generic parameter if all we do with agents is execute *call* on them.

*BASE* is useful for other purposes. Without *BASE* a call closed on its target, as with *y* above, could not contain any information about the class (and associated type) where the call's associated feature is defined. To open the gate to full *introspection* services — enabling a system to explore its own properties — class *ROUTINE* uses a feature

```
base_type: TYPE [BASE]
```

that yields the type to which the agent's feature belongs. Class *TYPE* [*G*] from the Kernel Library provides information about a type *G* and its base class.




Class *TYPE* is, even more fundamentally than *ROUTINE* and its heirs, the starting place for introspection. Example features include:

- *name*: *STRING*, the upper name of the type's base class.
- *generics*: *ARRAY* [*TYPE* [*ANY*]], the actual generic parameters, if any, used in the type's derivation.
- *routines*: *ARRAY* [*ROUTINE* [*ANY, TUPLE*]], the routines of a class, each an instance of *PROCEDURE*, *FUNCTION* or *PREDICATE*.

- *attributes*: *ARRAY* [*FUNCTION* [*ANY*]], the attributes.
- *invariant* (*obj*: *G*): *BOOLEAN*, telling us whether an instance *obj* satisfies the invariant.

Class *ANY* has a feature

```
generator: TYPE [like Current]
```

→ On ANY and universal features see chapter 30.

which yields an object describing the type of the current object.

So within a class of which *f* is a feature, *generator* has the same value as (~*f*)**.** *base_type*; if *a* is of type *T* and *f* is a feature of *T*, then *a***.***generator* has the same value as ({*T*}~*f*)**.** *base_type*.

A more complete interface specification of *TYPE* appears in the description of the Kernel Library classes.

→ On class TYPE see A.6.30, page 900.

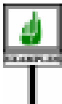
Thanks to the presence of *BASE* among the generic parameters of *ROUTINE* and its descendants, we can give a proper type to *base_type*, and as a result gain access to a whole library of introspection mechanisms.

## 25.16 USING AGENTS

All the details of the agent mechanism have now been introduced, although we haven't yet taken the trouble to look at the validity rules and precise semantics. We should now revisit and extend the examples sketched at the very beginning of this chapter and see how to make them work in practice: not just the client side (integrating a function, iterating an operation) but the suppliers too (the integrator, the iterators).

→ Validity and semantics are in the next section, 25.19, page 680.

The first set of examples was about integration. We assumed functions



```
g (x: REAL): REAL
h (x: REAL; a: T1; b: T2): REAL
```

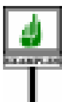
and wanted to integrate them over a real interval such as [0, 1], that is to say, approximate the two integrals

$$\int_{x=0}^{x=1} g(x)\,dx \qquad \int_{x=0}^{x=1} h(x, u, v)\,dx$$

We declare

```
your_integrator: INTEGRATOR
```

and, with the proper definition of function *integral* in class *INTEGRATOR*, we will obtain the integrals through the expressions



```
your_integrator.integral (~g (?), 0.0, 1.0)
your_integrator.integral (~h (?, u, v), 0.0, 1.0)
```

The question mark indicates, in each case, the open argument: the place where *integral* will substitute various real values for *x* when evaluating *g* or *h*.

Note that if we wanted in class *D* to integrate a real-valued function from class *REAL*, such as *abs* which is declared in *REAL* as

```
abs: REAL is
        -- Absolute value
    do … end
```

we would obtain it simply through the expression

```
your_integrator.integral ({REAL}~abs, 0.0, 1.0)
```

Let us now see how to write function *integral* to make all these uses possible. We use a primitive algorithm — this is not a treatise on numerical methods — but what matters is that any integration technique will have the same overall form, requiring it to evaluate *f* for various values in the given interval. Here class *INTEGRATOR* will have a real attribute *step* representing the integration step, with an invariant clause stating that *step* is positive. Then we may write *integral* as:

```
integral
    (f: FUNCTION [ANY, TUPLE [REAL], REAL];
    low, high: REAL): REAL is
            -- Integral of f over the interval [low, high]
    require
        meaningful_interval: low <= high
    local
        x: REAL
    do
        from
            x := low
        invariant
            x >= low ; x <= high + step
            -- Result approximates the integral over
            -- the interval [low, low.max (x – step)]
        until x > high loop
            Result := Result + step * f.item ([x])
            x := x + step
        end
    end
```

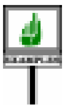
The boxed expression is where the algorithm needs to evaluate the function *f* passed to *integral*. Remember that *item*, as defined in class *FUNCTION*, calls the associated function, substituting any operands (here *x*) at the open positions, and returning the function's result.The argument of *item* is a tuple (of type *OPEN*, the second generic parameter of *FUNCTION*); this is why we need to enclose *x* in brackets, giving a one-argument tuple: [*x*].

In the first two example uses, ~*g* (**?**) and ~*h* (**?**, *u*, *v*), this argument corresponds to the question mark operands to *g* and *h*. In the last example the call expression passed to *integral* was {*REAL*}~ *abs*, where the open operand is the target, represented by {*REAL*}, and successive calls to *item* in *integral* will substitute successive values of *x* as targets for evaluating *abs*.

In the case of *h* the closed operands *u* and *v* are evaluated at the time of the evaluation of the agent expression ~*h* (**?**, *u*, *v*), and so they remain the same for every successive call to *item* within a given execution of *integral*.

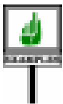
Note the type *FUNCTION* [*ANY*, *TUPLE* [*REAL*], *REAL*] declared in *integral* for the argument *f*. It means that the corresponding actual must be a call expression describing a function from any class (hence the first actual generic parameter, *ANY*) that has one open operand of type *REAL* (hence *TUPLE* [*REAL*]) and returns a real result (hence *REAL*). Each of the three example functions *g*, *h* and *abs* can be made to fit this bill through a judicious choice of open operand position.

Now the iteration examples. In a class *CC* we want to manipulate both a list of integers and a list of employees



```
intlist: LINKED_LIST [INTEGER]
emplist: LINKED_LIST [EMPLOYEE]
```

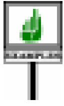
and apply the same function *for_all* to both cases:



```
if intlist.for_all (~is_positive (?)) then … end
if intlist.for_all (~over_threshold (?)) then … end

if emplist.for_all ({EMPLOYEE}~ is_married) then … end
```

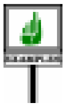
The function *for_all* is one of the iterators defined in class *TRAVERSABLE* of EiffelBase, and available as a result in all descendant classes describing traversable structures, such as *TREE* and *LINKED_LIST*. This boolean-valued function determines whether a certain property holds for every element of a sequential structure. The property is passed as argument to *for_all* in the form of a call expression with one open argument.

Our examples use three such properties of a very different nature. The first two are functions of the client class *CC*, assessing properties of their integer argument. The result of the first depends only on that argument:

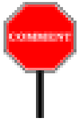


```
is_positive (i: INTEGER): BOOLEAN is
        -- Is i positive?
    do Result := (i > 0) end
```

Alternatively the property may, as in the second example, involve other aspects of *CC*, such as an integer attribute *threshold*:

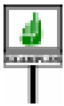


```
over_threshold (i: INTEGER): BOOLEAfsN is
        -- Is i greater than threshold?
    do Result := (i > threshold) end
```




Here *over_threshold* compares the value of *i* to a field of the current object. Surprising as it may seem at first, function *for_all* will work just as well in this case; the key is that the call expression **~***over_threshold* (**?**), open on its argument, is closed on its target, the current object; so the agent object it produces has the information it needs to access the *threshold* field.

In the third case, the argument to *for_all* is {*EMPLOYEE*} **~** *is_married*; this time we are not using a function of *CC* but a function *is_married* from another class *EMPLOYEE*, declared there as



```
is_married: BOOLEAN is do ... end
```

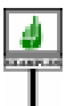
Unlike the previous two, this function takes no argument since it assesses a property of its target; We can still, however, pass it to *for_all*: it suffices to make the target open.

The types of the call expressions are the following:

```
PREDICATE [CC, TUPLE [INTEGER]]
    -- In first two examples (is_positive and over_threshold)
PREDICATE [EMPLOYEE, TUPLE [EMPLOYEE]]
    -- In the is_married example
```

*This assumes again that CC is non-generic, so that it is both a class and a type. Remember that a PREDICATE is a FUNCTION with a BOOLEAN result type.*

You may also apply *for_all* to functions with an arbitrary number of arguments, as long as you leave only one operand (target or argument) open, and it is of the appropriate type. You may for example write the expressions



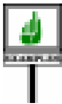
```
intlist.for_all (~some_criterion (e1, ?, e2, e3))
emplist.for_all ({EMPLOYEE}~some_function (e4, e5))
```

assuming in *CC* and *EMPLOYEE*, respectively, the functions

```
some_criterion (a1: T1; i: INTEGER; a2: T2; a3: T3)      -- In CC
some_function (a4: T4; a5: T5)                          -- In EMPLOYEE
```

for arbitrary types *T1*, ..., *T5*. Since operands *e1*, ..., *e5* are closed in the calls, these types do not in any way affect the types of the call expressions, which remain as above: *PREDICATE* [*CC*, *TUPLE* [*INTEGER*]] and *PREDICATE* [*EMPLOYEE*, *TUPLE* [*EMPLOYEE*]].

Let us now see how to write the iterator mechanisms themselves, such as *for_all*. They should be available in all classes representing traversable structures, so they must be introduced in a high-level class of EiffelBase, *TRAVERSABLE* [*G*]. Some of the iterators are unconditional, such as



```
do_all (action: ROUTINE [ANY, TUPLE [G]]) is
        -- Apply action to every item of the structure in turn.
    require
        … Appropriate preconditions …
    do
        from start until off loop
            action.call ([item])
            forth
        end
    end
```

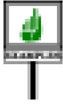
This uses the four fundamental iteration facilities, all declared in the most general form possible as deferred features in *TRAVERSABLE*: *start* to position the iteration cursor at the beginning of the structure; *forth* to advance the cursor to the next item in the structure; *off* to tell us if we have exhausted all items (**not** *off* is a precondition of *forth*); and *item* to return the item at cursor position.

*Descendants of TRAVERSABLE effect these features in various ways to provide iteration mechanisms on lists, hash tables, trees and many other structures.*

The argument *action* is declared as *ROUTINE* [*ANY*, *TUPLE* [*G*]], meaning that we expect a routine with an arbitrary base type, with an open operand of type *G*, the formal generic parameter of *TRAVERSABLE*, representing the type of the elements of the traversable structure. Feature *item* indeed returns a result of type *G* (representing the element at cursor position), so that it is valid to pass as argument the one-argument tuple [*item*] in the call *action*. *call* ([*item*]) that the loop repeatedly executes.

We normally expect *action* to denote a procedure, so its type could be more accurately declared as *PROCEDURE* [*ANY*, *TUPLE* [*G*]]. Using *ROUTINE* leaves open the possibility of passing a function, even though the idea of treating a function as an action does not conform to the Command-Query Separation principle of the Eiffel method.

Where *do_all* applies *action* to all elements of a structure, other iterators provide conditional iteration, selecting applicable items through another call expression argument, *test*. Here is the "while" iterator:

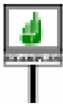


```
while_do
	(action: ROUTINE [ANY, TUPLE [G]]
	 test: PREDICATE [ANY, TUPLE [G]]) is
		-- Apply action to every item of structure up to,
		-- but not including, first one not satisfying test.
		-- If all satisfy test, apply to all items and move off.
	require
		… Appropriate preconditions …
	do
		from start until
			off or else not test.item ([item])
		loop
			action. call ([item])
			forth
		end
	end
```

Note how the algorithm applies *call* to *action*, representing a routine (normally a procedure), and *item* to *test*, representing a boolean-valued function. In both cases the argument is the one-element tuple [*item*].

The iterators of *TRAVERSABLE* cover common control structures: *while_do*; *do_while* (same as *while_do* but with "test at the end of the loop", that is to say, apply *action* to all items up to *and including* first one satisfying *test*); *until_do*; *do_until*; *do_if*.

Yet another of the iterators of *TRAVERSABLE* is *for_all*, which we used in the examples. It is easy to write a *for_all* loop algorithm similar to the preceding ones, but easier yet to define *for_all* in terms of *while_do*:



```
for_all (test: PREDICATE [G, TUPLE [G]]): BOOLEAN is
		-- Do all items satisfy test?
	require
		… Appropriate preconditions …
	do
		while_do (~do_nothing, test)
		Result := off
	end
```

Procedure *do_nothing*, from class *ANY*, has no effect; here we simply apply it as long as *test* is true of successive items. If we find ourselves *off* then *for_all* should return true; otherwise we have found an element not satisfying the *test*.

→ *do_nothing is cited in 30.6, page 796.*

Assuming a proper definition of *do_until*, the declaration of *exists*, providing the second basic quantifier of predicate calculus, is nicely symmetric with *for_all*:

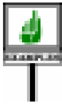
```
exists (test: PREDICATE [G, TUPLE [G]]): BOOLEAN is
        -- Does at least one item satisfy test?
    require
        … Appropriate preconditions …
    do
        do_until (~do_nothing, test)
        Result := not off
    end
```

## 25.17 INLINE AGENTS

Agents as seen so far do not name their open operands, representing them instead by question marks in *~is_positive* (**?**), by a type in braces in {*EMPLOYEE*}~ *is_married*, or just leaving them implicit in *~is_positive*.

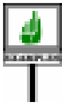
*~ is_positive means the same as ~ is_positive* (**?**).

As previewed at the beginning of this chapter there is also an inline variant, where you name the open operands. Its distinctive mark is the vertical bar |. Two examples defining function agents are



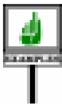
```
(i: INTEGER | is_positive (i))
(e: EMPLOYEE | e.is_married)
```

equivalent respectively to ~ *is_positive* (**?**) and {*EMPLOYEE*}~ *is_married*. The outermost parentheses are not part of the syntax for Inline_agent, but will be included in the examples for clarity. The common case of using an inline agent as routine argument requires parentheses anyway, as in



```
emplist.for_all (e: EMPLOYEE | e.is_married)
```

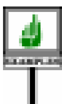
You may include two or more open operands of the same type, as in



```
(e, f: EMPLOYEE | e.salary > f.salary)
```

which represents a boolean-valued operation that, given two objects of type *EMPLOYEE*, returns true if and only it the query *salary* yields a higher result for the first than for the second.

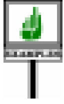
For operands of different types, use successive vertical bars:



```
(e, f: EMPLOYEE | p: POSITION | (e.job = p) and (f.job = p))
```

These were all examples defining *function* agents; accordingly, the part after the vertical bar was an expression. It is also possible to define inline *procedure* agents; in that case the definition uses one or more instructions

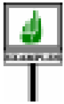
enclosed in the keywords **do** … **end**, as in the following example, using an inline agent passed as argument to an iterator, which will raise by 10 percent the salary of every employee of first name "Bertrand":



```
emplist.do_all (e: EMPLOYEE |
    do
        if equal (e.first_name, once "Bertrand") then
            e.set_salary (1.1 * e.salary)
        end
    end)
```

→ *Defining the string as* **once** *is not strictly necessary but improves performance by avoiding repeated evaluations; see "Once strings and the semantics of manifest strings", page 704*

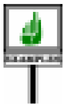
Inline agents do not give us anything fundamentally new, since we can always rewrite them as non-inline agents — of the form discussed in preceding sections, using tildes — after defining appropriate functions. For example we can rewrite the next-to-last one as *~same_job*, or *~same_job* (**?**, **?**, **?**), with the function definition



```
same_job (a, b: EMPLOYEE; pos: POSITION): BOOLEAN is
        -- Do a and b both have position pos?
    do
        Result := ((a.job = pos) and (b.job = pos))
    end
```

Although this tells us that in principle we could do without inline agents, they are useful if you want to avoid writing functions such as *same_job* when their only purpose is to define agents.

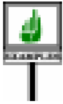
This case arises in particular for agents that express advanced contract specifications. Here is a typical example. Assume that in a class describing sequentially extendible structures (such as *LIST* [*G*]) you write a procedure that appends an element. It might include a postcondition as follows:



```
extend (x: G) is
        -- Add x at end; keep other items
    require
        ...
    do
        ...
    ensure
        one_more: count = old count + 1
        added_at_end: item (count) = x
        others_unchanged: (1 |..| old count).for_all
            (i: INTEGER | item (i) = (old twin). item (i))
    end
```

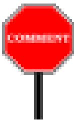
In the last postcondition clause — the one of interest for this discussion — *1* |..| **old** *count* is the interval from 1 to **old** *count*, to whose items *for_all* applies the agent property on the following line. The property expresses that the item at position *i*, for arbitrary *i*, is equal to the original item at that position (more precisely, to the item at position *i* in **old** *twin*, a copy of the list taken on entry to the procedure). This is typical of how agents enable us to express non-trivial postcondition or invariant properties, stating that a whole set of items have not changed, or have a certain association with the corresponding set of items in another structure.

We could restate the inline agent (the argument to *for_all*) in non-inline form as ~*equal_item* (**old** *twin*, **?**), but this assumes a function

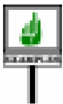


```
equal_item (l: like Current; i: INTEGER): BOOLEAN is
        -- Is item at position i equal to corresponding one in l?
    do
        Result := (item (i) = l.item (i))
    end
```

If you want to specify your software completely — expressing not only straightforward properties such as *item* (*count*) = *x*, but also those involving entire substructures — you may end up writing many such functions. Although they add interesting information, one may also feel that, being only used for assertions, they needlessly complicate the class. They may destabilize the software since any effort at better specification may cause the addition of a whole set of new features, used only in the assertions and of no other interest to clients of the class. Inline agents solve this problem.




Here is another example application of inline agents. The agents described in this chapter represent delayed *calls*; you may have wondered whether we also need an expression construct to denote delayed *object creation*, perhaps something like ~ **create** {*SOME_TYPE*}.*make* (*a1*,**?**). The answer is no, since we can achieve the intended effect (assuming we need it) by using a creation expression as part of an inline agent in



```
b1: B | create {SOME_TYPE}.make (a1, b1)
```

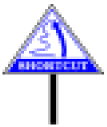
where *B* is the type of *make*'s second argument.

Inline agents, such as *l*: **like** *Current* | *i*: *INTEGER* | *item* (*i*) = *l*.*item* (*i*), are like little routine declaration; in fact we could call them **anonymous routines**, similar to anonymous *classes* (tuple types) and anonymous *objects* (tuples). It's as we had written *equal_item* inline and without a routine name.

Inline and non-inline agents, however, are not completely interchangeable. More precisely, every inline agent has a non-inline equivalent (illustrated in the last example by the form using *equal_item*); but the converse is not always true, because inline agents only provide a restricted form of anonymous routine.

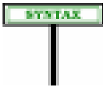
In particular, an explicit (non-anonymous) routine such as *equal_item* may have, apart from argument declarations and a Routine_body (**do** clause), other clauses such as Precondition, Postcondition and Rescue. An inline agent *args*: *SOME_TYPE* | *expr* can only specify its operands *args* and a result *expression*, equivalent to a **do** clause with a single instruction of the form *Result* := *expr*. If you want anything else — assertions, or exception handling — you must write a non-anonymous routine such as *equal_item* and use it to define a non-inline agent.

## 25.18 SYNTAX, VALIDITY AND SEMANTICS OF INLINE AGENTS



We have now seen all the agent-related mechanisms; there remains to study the precise validity and semantics. This will introduce no new concept, so on first reading you may skip to the next chapter.

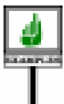
It is convenient to start with inline agents by showing how to define their validity and semantics in terms of the more general case, non-inline agents, detailed in the next sections. Here is the syntax of inline agents, distinguished by the vertical bar |:



Inline_agent ≜ [Agent_formals] "|" Inline_body
Agent_formals ≜ {Entity_declaration_group "|" ...}$^+$
Inline_body ≜ {Expression | Inline_procedure}$^+$
Inline_procedure ≜ **do** Compound **end**}$^+$

Entity_declaration_group, a construct seen in the discussion of routines, represents sequences of entities followed by a colon and a type name, as in

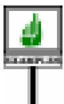
← *Page 266*.



*a*, *b*, *c*: *T*

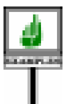
This serves to declare the listed entities as being of type *T*.

You may not use an Entity_declaration_list, which could involve more than one Entity_declaration_group and hence more than one type, as in *a*, *b*: *T1*; *c*: *T2*, since this would be ambiguous (at least to the human reader); but you can obtain the same effect by using successive groups separated by vertical bars, as in the inline agent



*a*, *b*: *INTEGER* | *c*: *REAL* | *a* + *b* > *c*

Note that the Agents_formals is optional: you may define agents without formal arguments, such as



```
| p + q > r          -- where p, q, r are queries of the class
| create {SOME_TYPE}.make (some_entity)
| f                  -- where f is a query of the class; same meaning as ~f
```

Some terminology will be useful

DEFINITION

**Formal arguments, defining expression of an inline agent**

The **formal arguments** (or **operands**) of an inline agent are the entities listed in any Entity_declaration_group of its Agent_formals part. Its **body** is its Inline_body part.

*"Formal argument" recalls the connection with routines; "operand", the open operands of an agent. So both names are useful.*

This enables us to consider that an inline agent is derived from an anonymous routine:

DEFINITION

**Associated routine of an inline agent**

The **associated routine** of an inline agent *ia* is a fictitious routine *r*, declared in the enclosing class *C* as follows:

- The name of *r* is chosen not to conflict with any other feature name in *C* and its descendants.
- The formal arguments of *r* are the same as those of *ia*, if any.
- If the body of *ia* is an Expression *exp*, then *r* is a function whose result type is the type of *exp* and whose body is of the **do** form, containing the single instruction *Result* := *exp*, and has none of the optional clauses (Precondition, Postcondition, Rescue).
- Otherwise the body of *ia* is an Inline_procedure and *r* is a procedure whose Routine_body is the body of *ia*.

This definition allows us to treat inline agents like non-inline ones

SEMANTICS

**Non-inline form of an inline agent**

The validity and semantics of an inline agent *ia* are those of its **non-inline form**: the agent ~ *af*, where *af* is *ia*'s associated routine.

This will spare us the need to define the semantics of inline agents; we can just rely on the next section's specification of the non-inline case.

This approach is applicable to the validity rule as well; but here, to enable compilers to provide directly usable messages in case of an erroneous inline agent, it is useful to provide a direct rule, derived through the above definitions from the Formal Argument rule of routines:

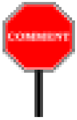
← *Formal Argument rule: page 266.*

**Inline Agent rule** *CPIA*

An Inline_agent *ia* appearing in a class *C* is valid if and only if the list of identifiers *formals* obtained by concatenating every Identifier_list of every Entity_declaration_group in the Agent_formals part of *ia*, if any, satisfies the following two conditions:

1 • No Identifier appears twice in *formals*.

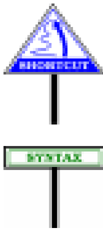
2 • No Identifier *e* appearing in *formals* is the final name of a feature of *C*, or of a formal argument or local entity of the enclosing routine if any.




The validity of the Inline_body part doesn't require any particular rule; it is covered by the Entity rule, which states that any entity appearing in the Inline_body must be a formal argument of the inline agent itself (such as *l* and *i* in *l*: **like** *Current* | *i*: *INTEGER* | *item* (*i*) = *l*.*item* (*i*)) or otherwise legal from the context (feature of the enclosing class, formal argument or local entity of the enclosing routine).

← *“Entity rule”, page 517.*

## 25.19 VALIDITY AND SEMANTICS OF FEATURE AGENTS



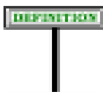

It remains to provide the validity and semantic rules of non-inline agent expressions, complementing the syntax already given. This material may, like the previous section, be skipped on first reading.

For ease of reference here is a repetition of the syntax productions:

← *These productions first appeared on pages 664 and 665.*

Feature_agent ≜ [Agent_target] Agent_unqualified
Agent_target ≜ Entity | Parenthesized | Type_descriptor
Type_descriptor ≜ Explicit_type_descriptor | Placeholder
Explicit_type_descriptor ≜ "{" Type "}"
Placeholder ≜ "?"
Agent_unqualified ≜ "~" Feature_name [Agent_actuals]
Agent_actuals ≜ "(" Agent_actual_list ")"
Agent_actual_list ≜ {Agent_actual "," …}
Agent_actual ≜ Actual | Type_descriptor

To define the validity of an agent expression we need to be able to consider its “target type”, explicit or implicit:




**Target type of an agent expression**

The target type of a Feature_agent is:

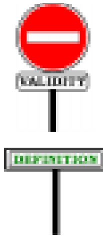
1 • If there is no Agent_target, or an Agent_target which is a Type_descriptor of the Placeholder kind, the current type.

2 • If there is a Agent_target and it is an Entity or Parenthesized, its type.

3 • If there is a Agent_target and it is a Type_descriptor of the Explicit_type_descriptor kind, the type that it lists (in braces).

← *The “current type” is the enclosing class, with generic parameters added if necessary to make up a type. See* *“CURRENT TYPE, FEATURES OF A TYPE”, 12.8, page 384.*

This is enough to introduce the validity rule, which also defines the notion of "associated feature" of an agent expression:




**Agent Expression rule** *CPAE*

A Feature_agent appearing in a class *C*, with a feature identifier *fi* and target type *T0*, is valid if and only if it satisfies the following six conditions:

1 • *fi* is the name of a feature of *T0*, called the **associated feature** of the agent.

2 • If there is a Agent_target, that feature is export-valid for *T0* in *C*.

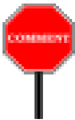
3 • If the Agent_actuals part is present, the number of elements in its Agent_actual_list is equal to the number of formals of *f*.

4 • Any Agent_actual of the Actual kind is of a type conforming to the type of the corresponding formal in *f*.

5 • Any Agent_actual which is a Type_descriptor of the Explicit_type_descriptor kind lists, between the braces, a type conforming to the type of the corresponding formal in *f*.

6 • If *T0* is separate, any non-expanded formal of *f* is separate.

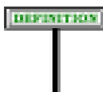
Clause 6 is a consistency condition for concurrent computation, and parallels a similar clause discussed in the chapter on normal calls.




The rule's phrasing makes certain forms of the construct automatically valid:

- If any Agent_actual is of the Placeholder kind, represented simply by a question mark, neither clause 4 nor clause 5 applies, so the argument raises no type validity problem. This is as expected, since such an argument is left open for future filling-in.
- If there is no Agent_actuals part, clauses 3 to 5 do not apply. If *f* has no formals, we are calling an argumentless feature with no actuals, as we should. If *f* has one or more formal arguments, we view the absence of explicit actuals of an abbreviation for actuals that are all of the Placeholder kind (question marks): assuming *f* takes three arguments, *a0~f* is simply an abbreviation for *a0~f* (**?**, **?**, **?**). In this case the implicit arguments are all open, and hence automatically valid.

We may formalize the last observation through a definition which will also be useful for the semantics:

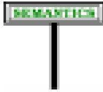



**Unfolded form of an agent expression**

The unfolded form of a Feature_agent *dc* is:

- *dc* itself if it includes an Agent_actuals part, or if the associated feature has no formals.
- Otherwise, *dc* extended with a Agent_actuals part made up of the appropriate number of Agent_actual components, all of the Placeholder (question mark) kind.

We now have enough to define the semantics of an agent expression:



← "Open operand position" was defined on page 666.

## Type and value of an agent expression

Consider a Feature_agent *d*, whose associated feature *f* has a generating type *T0*. Let *i1*, …, *im* ($m \geq 0$) be its open operand positions, if any, and let $T_{i1}$, .., $T_{im}$ be the types of *f*'s formals at positions *i1*, …, *im* (taking $T_{i1}$ to be *T0* if *i1* = 0).

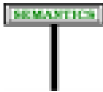
The type of *d* is:

- *PROCEDURE* [*T0*, *TUPLE* [$T_{i1}$, .., $T_{im}$]] if *f* is a procedure;
- *FUNCTION* [*T0*, *TUPLE* [$T_{i1}$, .., $T_{im}$], *R*] if is a function of result type *R*.

Evaluating *d* at a certain *construction time* yields a reference to an instance D0 of the type of *d*, containing information identifying:

- *f*.
- The open operand positions.
- The values of the closed operands at the time of evaluation of *d*.

Although this will be an implicit consequence of the preceding description, it doesn't hurt to state explicitly what some of the information in D0 is good for: enabling calls on agent objects.



## Effect of executing *call* on an agent

Let D0 be an agent object with associated feature *f* and open positions *i1*, …, *im* ($m \geq 0$). The information in D0 enables a call to procedure *call*, executed at any **call time** posterior to D0's construction time, with target D0 and (if required) actual arguments $a_{i1}$, .., $a_{im}$, to perform the following:

- Produce the same effect as a call to *f*, using the closed operands at the closed positions and $a_{i1}$, .., $a_{im}$, evaluated at call time, at the open positions.
- In addition, if *f* is a function, setting the value of query *last_result* for D0 to the result returned by such a call.

← *last_result* from class *FUNCTION*, giving the result of the last evaluation, was introduced on page 655.