Expressions

28.1 OVERVIEW

Through the various forms of Expression, software texts can include denotations of run-time values — objects and references.

Previous discussions have already introduced some of the available variants of the construct: Formal, Local, Call, Old, Manifest_tuple, Agent. The present one gives the full list of permissible expressions and the precise form of all but one of the remaining categories: operator expressions, equality and locals. The last category, constants, has its own separate presentation, just after this one.

28.2 GENERAL FORM OF EXPRESSIONS

An expression will use one of the following variants:

- Basic_expression
- Special_expression
- Parenthesized
- Address
- Once_string
- Boolean_expression

Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Basic_expression</th>
<th>Special_expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic_expression</td>
<td>Read_only</td>
<td>Local</td>
</tr>
<tr>
<td>Special_expression</td>
<td>Manifest_constant</td>
<td>Manifest_tuple</td>
</tr>
<tr>
<td>Parenthesized</td>
<td>&quot;(&quot; Expression &quot;)&quot;</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>&quot;$&quot; Variable</td>
<td></td>
</tr>
<tr>
<td>Once_string</td>
<td>once Manifest_string</td>
<td></td>
</tr>
<tr>
<td>Boolean_expression</td>
<td>Basic_expression</td>
<td>Boolean_constant</td>
</tr>
</tbody>
</table>
“Basic expressions” correspond to the common forms of expression derived from ordinary mathematical notation, such as variables and operator expressions. “Special expressions” include manifest constants — constants given directly by their values — as well as original Eiffel mechanisms such as agents and tuples.

Summarizing the variants of Basic_expression:

- **Read_only**, which includes formal arguments of routines and Current, and Local variables were seen in the discussion of entities.
- A **Call**, denoting a feature call, is an Expression if and only if the feature of the call is a query, that is to say, an attribute or a function. (Otherwise it would be an Instruction.)
- As a special case of Call, you may use an attribute x as an expression in the text of its class. (In the syntax of Call, just use x as Feature_name of an Unqualified_call, without Parenthesized_qualifier or Actuals.)
- **Precursor** enables the redefined version of a routine — in the case of expressions, a function — to refer to the original version. We studied the mechanism in connection with inheritance.
- **Equality** expressions cover both equality and inequality tests, using the symbols =/=, as well as ~ and /~ for comparing objects. Although they are syntactically similar to the next category, Operator_expressions, it is preferable to treat them separately because their semantics, studied in an earlier chapter, is not that of a call.
- An **Operator_expression** is built using unary and binary operators. Operator expressions have the semantics of calls but a special syntax, requiring rules of operator precedence.
- You can use **Parenthesized** subexpressions to override these rules.
- A **Bracket_expression** again has the semantics of a call but uses bracket syntax, as in your_table[your_index], typically an abbreviation for a feature call your_table.item(your_index) where item has been declared with a bracket alias: alias "[1]".
- An **Old_expression**, usable only in a postcondition, denotes the earlier value of an expression. This was discussed with assertions.
- **Creation_expression** yields a newly created object and was studied in the discussion of creation.

Special_expression completes this panoply:

- A **Manifest_constant** has a fixed value, given by the form of the constant, as in the Integer_constant −87. A special case is a Manifest_string: we may view as constant because it denotes a fixed string descriptor object, but it gives access to a character sequence which may in fact change. (This is a potential source of confusion and will be explained in detail.)
§28.2  GENERAL FORM OF EXPRESSIONS

Not all constants are manifest: by declaring a **constant attribute** you may use an Identifier to denote a constant value. Syntactically, as noted above, constant attributes used as expressions are a special case of Call, but it will be convenient to **study them** together with manifest constants.

- **A Manifest_tuple** is a tuple given by the list of its elements.

- **Agent** expressions, representing partially evaluated calls, were studied in the previous chapter.

- **An Object_test**, of the form \{loc: T\} \(\text{exp}\) as studied in the discussion of eradicating void calls, yields true if and only if the expression \(\text{exp}\) is attached on evaluation to an object of type \(T\), and then binds it locally to the entity loc.

- **A Once_string** is a manifest string constant qualified by the keyword `once` to indicate that if it appears in an expression the string will be evaluated just once, rather than denoting a new string object for each evaluation of the expression. Since this notion is closely tied to the semantics of strings it will be studied in the next chapter as part of the general discussion of strings. Non-once strings — the more common case — are examples of **Manifest_constant**, covered by the previous case.

- **Address** expressions, of the form `\$f` where \(f\) is the name of a feature (or `Current` or `Result`) serve to pass the address of an Eiffel object to non-Eiffel software.

Finally, **Boolean_expression** requires its own construct because a few other constructs — assertion clauses, Conditional, Exit conditions of loops — specifically expect a boolean expression. **Object_test** is one of the variants.

This chapter needs only occasionally to refer to the variants studied in depth in their own chapters: Read_only, Local, Call (for which we saw how to determine the value and type of a Call serving as an expression), **Precursor** (whose validity rule stated under what condition you may use a Precursor as an expression), **Equality**, **Agent**, **Old**, **Object_test**, **Address**, **Creation_expression**; for Call, we still need to define explicitly the value of a call expression. After introducing the notion of “subexpression”, the following sections will explore the remaining variants in the order of the syntax. Then we’ll explore general properties of expressions, in particular the notion of Equivalent Dot Form, how to determine the type of an expression, and the syntactical benefit (having to do with making semicolons **always optional**) of the distinction between **Basic_expression** and **Special_expression**.

As a general note on the use of expressions, **remember** that if you have an expression \(\text{exp}\) of type \(U\) and want to use it as if it were of a type \(T\) to which \(U\) conforms or converts, you may rely on the notation

\[
\{T\} \[\text{exp}\]
\]
This section defines a technical notion; you may skip it on first reading.

For some of the definitions and rules that follow, it is convenient to talk about the “subexpressions” of an expression: all the components of the expression that are themselves expressions whose value participates in the evaluation of the expression as a whole. This notion is mostly useful for operator expressions, but it’s convenient to define it for all other kinds as well:

---

### Subexpression, operand

The subexpressions of an expression e are e itself and (recursively) all the following expressions:

1. For a Parenthesized (a) or a Parenthesized_target ([|a|]): the subexpressions of a.
2. For an Equality or Binary_expression a § b, where § is an operator: the subexpressions of a and of b.
3. For a Unary_expression ◊ a, where ◊ is an operator: the subexpressions of a.
4. For a Call or Precursor expression: the subexpressions of the Actuals part, if any, of its Unqualified_part
5. For an Agent: the subexpression of its Agent_actuals if any.
6. For a qualified call: the subexpressions of its target.
7. For a Bracket_expression f[a₁, …, aₙ]: the subexpressions of f and those of all of a₁, …, aₙ.
8. For an Old expression old a: a.
9. For a Manifest_tuple [a₁, …, aₙ]: the subexpressions of all of a₁, …, aₙ.

In cases 2 and 3, the operands of e are a and (in case 2) b.

---

Clause 4 uses “the Unqualified_call part of a Call”: both of the available variants, Object_call and Non_object_call, indeed include an Unqualified_call.

For example the subexpressions of \( b + (c * [d, e]) \) are the whole expression, \( b, (c * [d, e]), c, [d, e], d \) and \( e \).

Not every expression physically contained in an expression is a subexpression: according to the rule, \( a + b + c \) has no subexpression other than itself; neither has \( x + y * z \). Parts such as \( a + b, b + c, x + y, y * z \), although valid expressions, are not subexpressions. This is because such examples use more than one binary operator in succession, giving rise to potential ambiguities; and indeed the part \( x + y \) plays no role in determining the value of \( x + y * z \).

The precedence rules which we’ll study shortly let us define a Parenthesized Form for any expression, such as \( x + (y * z) \) in the second part, with subexpressions \( x \) and \( (y * z) \), both of which participate in the value of the expression as a whole.
28.4 PARENTHESIZED EXPRESSIONS

You may enclose an arbitrarily complex expression in parentheses without changing its semantics:

<table>
<thead>
<tr>
<th>Parenthesized Expression Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>If ( e ) is an expression, the value of the Parenthesized ((e)) is the value of ( e ).</td>
</tr>
</tbody>
</table>

Indeed the parentheses have a syntactic role only. You can use a Parenthesized:

- To override default operator precedence in operator expressions as studied in the next section.
- To apply a certain operation to an expression when the syntax wouldn’t permit it in the original form of the expression.

An important example of the second case is feature application to a complex expression. The syntax of a query call \( \text{exp}.f \) (or \( \text{exp}.f(\text{args}) \) with arguments) restricts the target \( \text{exp} \) to just a few possibilities: a single entity, as in \( \text{an_attribute}.f \), or one or more other calls, as in \( g(x).\text{an_attribute}.f \).

You may not directly apply the feature to a manifest constant, as in \( 3.f \) (invalid) or to an operator expression, as in \( a+b.f \) (which, if valid, would apply \( f \) just to \( b \), not to the addition). You can achieve the desired effect by parenthesizing the target expression, as in

\[
(3).f \\
(a+b).f
\]

(valid if \( f \) is applicable to the respective targets).

28.5 OPERATOR EXPRESSIONS

You may build operator expressions by combining simpler expressions through prefix and infix operators, using parentheses to remove ambiguities if necessary.

Operator expression basics

An example, from the postcondition of procedure \( \text{put_child_left} \) in class \( \text{LINKABLE} \) of EiffelBase, is

\[
\text{not (child_position = 2)} \implies \text{child_position = old child_position + 1}
\]

This uses the infix operators \( \implies \) and \(+\) and the prefix operator \( \text{not} \), applied to subexpressions involving \( \text{Old} \) and \( \text{Equality} \).
Semantically, operator expressions bring nothing new: they are simply a different way to write calls, using conventional operator notation rather than dot notation. Since every feature with an operator alias also has a Feature_name (an identifier), you may ignore operators, writing instead calls in dot notation:

\[(\text{child-position} = 2) \cdot \text{negated} \cdot \text{implication} \quad (\text{child-position} = \text{old} (\text{child-position} \cdot \text{plus} (1)))\]

**Operator expression syntax**

Here is the general form of operator expressions:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operator expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Operator_expression</td>
</tr>
<tr>
<td></td>
<td>Unary_expression</td>
</tr>
<tr>
<td></td>
<td>Binary_expression</td>
</tr>
</tbody>
</table>

Both **Unary** and **Binary** operators can be one of the standard operators, or a “free” operator that you make up according to very flexible rules. The list of standard operators already appeared in the discussion of feature names:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unary</td>
</tr>
<tr>
<td></td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>”+”</td>
</tr>
</tbody>
</table>

**Precedence and Parenthesized Form**

The syntax for Operator_expression is ambiguous: it would make it possible to understand an expression such as

\[a + b + c * d\]

in several different ways (expressed with parenthesization):

The correct interpretation, according to the precedence rules given below, is [E31].
You can always remove ambiguities by adding parentheses as in these last forms. In mathematical practice, however, it is customary not to require parentheses in simple cases based on “precedence”. This custom makes \( a + b \times c \) legal and gives it the same meaning as \( a + (b \times c) \), based on the convention that \( \times \) “binds tighter” than \( + \).

To formalize this practice, we complement the syntax by **precedence rules**. Every possible operator has a precedence, a numerical value between 1 and 13 determined by the table below. The values themselves are not important; what matters is the comparison of the precedence values of any two operators appearing consecutively in an expression. For example, \( \times \) has precedence 8 and \( + \) has precedence 3. In the absence of intervening parentheses, the one with the higher precedence binds tighter.

### Operator precedence levels

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>. (Dot notation, in qualified and non-object calls)</td>
</tr>
<tr>
<td>12</td>
<td>old (In postconditions)</td>
</tr>
<tr>
<td></td>
<td>not + – Used as unary</td>
</tr>
<tr>
<td></td>
<td>All free unary operators</td>
</tr>
<tr>
<td>11</td>
<td>All free binary operators.</td>
</tr>
<tr>
<td>10</td>
<td>(^) (Used as binary: power)</td>
</tr>
<tr>
<td>9</td>
<td>(* / /\ ) (As binary: multiplicative arithmetic operators)</td>
</tr>
<tr>
<td>8</td>
<td>(+ – ) Used as binary</td>
</tr>
<tr>
<td>7</td>
<td>(\ldots) (To define an interval)</td>
</tr>
<tr>
<td>6</td>
<td>(= /= \sim \neg &lt; &gt; &lt;= &gt;=) (As binary: relational operators)</td>
</tr>
<tr>
<td>5</td>
<td>and and then</td>
</tr>
<tr>
<td></td>
<td>(Conjunctive boolean operators)</td>
</tr>
<tr>
<td>4</td>
<td>or or else xor</td>
</tr>
<tr>
<td></td>
<td>(Disjunctive boolean operators)</td>
</tr>
<tr>
<td>3</td>
<td>implies (Implicative boolean operator)</td>
</tr>
<tr>
<td>2</td>
<td>[ ] (Manifest tuple delimiter)</td>
</tr>
<tr>
<td>1</td>
<td>; (Optional semicolon between an Assertion_clause and the next)</td>
</tr>
</tbody>
</table>
This precedence table includes the operators that may appear in an Operator_expression, the equality and inequality symbols used in Equality expressions, as well as other symbols and keywords which also occur in expressions and hence require disambiguating: the semicolon in its role as separator for Assertion_clause; the old operator which may appear in an Old expression as part of a Postcondition; the dot . of dot notation, which binds tighter than any other operator.

The operators listed include both standard operators and predefined operators (|= /= ~ |~). For a free operator, you cannot set the precedence: all free unaries appear at one level, and all free binaries at another level.

This precedence table is the basis for the rule removing potential syntactic ambiguities in operator expressions. We’ll just work from a form that adds parentheses wherever needed:

**Parenthesized Form of an expression**

The parenthesized form of an expression is the result of rewriting every subexpression of one of the forms below, where § and ‡ are different binary operators, ◊ and ♣ different unary operators, and a, b, c arbitrary operands, as follows:

1. For a § b § c where § is not the power operator ^: (a § b) § c (left associativity).
2. For a ^ b ^ c : a ^ (b ^ c) (right associativity).
3. For a § b ‡ c: (a § b) ‡ c if the precedence of ‡ is lower than the precedence of § or the same, and a § (b ‡ c) otherwise.
4. For ◊ ♣ a: ◊ (♣ a)
5. For ◊ a § b: (◊ a) § b
6. For a § ◊ b: a § (◊ b)
7. For a subexpression e to which none of the previous patterns applies: e unchanged.

Since the notion of subexpression was defined recursively, the rewriting must be applied recursively too. Both notions are interesting for the case of an Operator_expression but are defined for general expressions, allowing the recursion to work properly.

The Parenthesized Form of

\[ a + b \ast c \uparrow \text{old} \ d \]

is

\[ a + (b \ast (c \uparrow (\text{old} \ d))) \]
The Parenthesized Form is not always fully parenthesized; it only adds the parentheses necessary to remove ambiguities. Here it doesn’t put any around the full expression, or around entities \( a, b, c, d \).

Operator \( ^ \) gets a special treatment in clauses 1 and 2 of the definition because basic arithmetic types (\( \text{INTEGER}, \text{REAL} \) and their sized variants) use it as \textbf{power} operator: the mathematical notation \( a^{bc} \) is traditionally understood as meaning \( a^{(bc)} \) — the only interesting interpretation since \( (a^b)^c \) is just \( a^{bc} \).

Special cases in rules are unpleasant, but it is dangerous to go against long-standing mathematical conventions. Here a left-associative rule could cause errors for people trained in mathematics or physics. To avoid worrying about such issues, just use parentheses wherever there might be any doubt.

Clause 4 reflects that, in the above precedence table, all unary operators have the same precedence; and the last two clauses, that unary operators bind tighter than all binary operators.

\( \odot \) and \( \div \) can be the same operator, used as unary in one case and binary in the other. So clause 6 tells us that \( a--b \) — where the two signs must be separated by a break, lest we take them to start a comment — means \( a-(b) \).

To override the meaning implied by this rule, you may always use parentheses. For any \textbf{Binary} operator, the first operand of \( \& \) in

\[
(\text{exp}) \& \text{other_exp}
\]

is always \( \text{exp} \), regardless of the precedence of \( \& \) and of the operators appearing in \( \text{exp} \); the last operand of \( \& \) in

\[
\text{exp} \& (\text{other_exp})
\]

is always \( \text{other_exp} \); and for any \textbf{Unary} operator \( \odot \), the expression

\[
\odot (\text{exp})
\]

always denotes the application of \( \odot \) to the value of \( \text{exp} \).

The precedence rules are easy to remember but competent Eiffel programmers mostly use them to understand the code of their macho colleagues. Don’t hesitate to put parentheses around subexpressions to clarify intent and avoid errors. In particular, you should always use parentheses when a boolean expression uses different conjunctive and disjunctive operators in succession, as in \( (a \text{ or } (b \text{ and } c)) \).

We will build the Equivalent Dot Form of an expression, on which its validity and semantics are based, from its Parenthesized Form. In other words, thanks to this notion we can for all the rest of the discussion forget about any matters of ambiguity and operator precedence.
Accounting for target conversion

We need one more definition to handle all cases of operator expressions. It covers the mechanism that we studied in the chapter of conversions, allowing you to follow traditional mathematical practice by writing mixed-mode expressions such as your_integer + your_real when you really mean to use the “+” operator from class REAL, converting the first operand to REAL. To make this possible, you must specify convert in the declaration of the operator, in class INTEGER:

```
plus alias "+" convert (other: INTEGER): INTEGER...
```

In this case the standard unfolding of your_integer + your_real into your_integer,plus (your_real) doesn’t apply, since REAL neither conforms nor converts to INTEGER. We want to understand the expression as (your_integer,converted_to_real) + your_real. Because the first unfolding would be type-wise invalid, there is no danger of confusion.

A simple definition takes care of this case:

**Target-converted form of a binary expression**

The target-converted form of a Binary_expression `x § y`, where the one-argument feature of alias § in the base class of x has the Feature_name f, is:

1. If the declaration of f includes a convert mark and the type TY of y is not compatible with the type of the formal argument of f: `({TY} [x]) § y`
2. Otherwise: the original expression, `x § y`.

`({TY} [x])` denotes x converted to type TY. This definition allows us, if the feature from x’s type TX cannot accept a TY argument but has explicitly been specified, through the convert mark, to allow for target conversion, and TY does include the appropriate feature accepting a TX argument, to use that feature instead.

The archetypal example is your_integer + your_real which, with the appropriate convert mark in the “+” feature in INTEGER, we can interpret as ([REAL] [your_integer]) + your_real, where “+” represents the plus feature from REAL. (where `{II} [x] denotes x converted to type II). In fact that all we need: the validity and semantics, in this case, will simply rely — through the Equivalent Dot Form — not on the original expression but on its target-converted form. There is no need for any special rule, either for validity or for semantics.
Operator expression validity and semantics

Once no syntactical ambiguity remains, the validity and semantic properties of an Operator_expression are essentially those of a corresponding Call.

For every Operator_expression there will be an Equivalent Dot Form, syntactically a Call, illustrated above for a postcondition clause of class LINKABLE. As another example, here is the Equivalent Dot Form of our earlier expression \( a + b + c \ast d \):

\[
(a, \text{plus}(b)), \text{plus}(c, \text{multiplied}(d))
\]

This assumes that if \( x \)'s type has a base class \( C \) with operator features plus alias "+" and multiplied alias "\( \ast \)".

The next section gives a precise definition of the Equivalent Dot Form, although the above examples suffice to make the idea clear. Then the validity constraint on operator expressions is straightforward:

![Operator Expression rule](image)

The Feature Declaration rule tells us that a given operator may serve as alias for a unary feature (a feature without argument), or a binary feature (with one argument), or both, as in the case of \(+\) in INTEGER and REAL. In this last case, two features will match the requirement of clause 1; but that’s OK because the form of the expression, unary or binary, will remove any ambiguity thanks to the definition of the Equivalent Dot Form.

This rule ensures that every operator is used with the proper number of arguments. For example INTEGER and other basic arithmetic classes have a one-argument function product alias "\( \ast \)"; but not zero-argument version, as would be required for a Unary. Then of the expressions

\[
\begin{align*}
2 \ast 2 \\
\ast 2
\end{align*}
\]

the first is valid but not the second.
The rule also explains why some binary operators can be used as "multiary" — meaning with three or more operands, of types all compatible with the type of the first — others are limited to two arguments. An example of multiary operator is + on integers; relational operators such as <, on the other hand, are binary but not multiary. This is clear from the Equivalent Dot Forms. With integer operands, the Operator_expression

\[
1 + 2 + 3 + 4
\]

has the Parenthesized Form

\[
((1 + 2) + 3) + 4
\]

yielding the valid Equivalent Dot Form

\[
((1 + 2), plus (3), plus (4))
\]

By the same rules, the Operator_expression

\[
1 < 2 < 3
\]

would yield the Equivalent Dot Form

\[
(1, is\_less (2)) < 3
\]

is not valid since the highlighted operand is of type BOOLEAN, but BOOLEAN does not have a function aliased to <, violating clause 1 of the Operator Expression rule.

If BOOLEAN had a function is\_less alias "<", perhaps with false considered less than true, this would still not make the expression valid: such a function would expect an argument of type BOOLEAN, not INTEGER. In this case it’s clause 2 that would fail. A true multiary operator, such as "+" on integers, must accept successive operands of the same or compatible type.

In summary, there is no need to define binary and multiary operators as separate syntactical categories. The grammar lists both kinds as Binary; whether a given operator may be used in multiary form depends on the signature of the corresponding function and on the precedence rules.
There remains to define the semantics of an Operator_expression. You are probably guessing from the preceding discussion that — as with validity — it is simply the semantics of its Equivalent Dot Form. You are guessing almost right; “almost” because (life not always being as simple as we would like) we must account for a special case, semistrict operators:

### Expression Semantics (strict case)

The value of an Expression, other than a Binary_expression whose Binary is semistrict, is the value of its Equivalent Dot Form.

This semantic rule and the preceding validity constraint make it possible to forego any specific semantics for operator expressions (except in one special case) and define the value of any expression through other semantic rules of the language, in particular the rules for calls and entities.

This applies in particular to arithmetic and relational operators (for which the feature declarations are in basic classes such as INTEGER and REAL) and to boolean operators (class BOOLEAN): in principle, although not necessary as implemented by compilers, \( a + b \) is just a feature call like any other.

The excluded case — covered by a separate rule — is that of a binary expression using one of the three semistrict operators: **and then**, **or else**, **implies**. This is because the value of an expression such as \( a \text{ and then } b \) is not entirely defined by its Equivalent Dot Form \( a \text{ conjuncted_semistrict } (b) \), which needs to evaluate \( b \), whereas the **and then** form explicitly ignores \( b \) when \( a \) has value **False**, as the value of the whole expression is **False** even if \( b \) does not have a defined value, a case which should not be treated as an error.

### 28.6 SEMistrict BOOLEAN OPERATORS

The semantic rule for operator expressions set out the special case of three boolean operators, known as “semistrict”. We’ll now take a look at these operators to understand why they are needed, and obtain the semantic rule for this case.

The ordinary (“strict”) boolean operators **not**, **and**, **or**, and **xor**, defined in the Kernel Library class BOOLEAN, define operations on boolean values. The value of **not** \( a \) is true if and only if \( a \) has value false. The others are binary operators; the value they yield when applied to a first operand of value \( v1 \) and a second operand of value \( v2 \) is defined as follows:

- For **and**: true if and only if both \( v1 \) and \( v2 \) are false.
- For **or**: false if and only if either \( v1 \) or \( v2 \) is false.
• For xor: true if and only if \( v_1 \) and \( v_2 \) have different values. In other words, \( a \ xor \ b \) has the same value as \( (a \ or \ b) \ and not (a \ and \ b) \).

Three operators, also defined in BOOLEAN, complement and and or, from which they differ by a special semantic property known as semistrictness.

### Semistrict operators

A semistrict operator is any one of the three operators and then, or else and implies, applied to operands of type BOOLEAN.

For operands of values \( v_1 \) and \( v_2 \) they yield the following results:

- **and then** (semistrict conjunction): false if \( v_1 \) is false, otherwise the value of \( v_2 \).

- **or else** (semistrict disjunction): true if \( v_1 \) is true, otherwise the value of \( v_2 \).

- **implies** (semistrict implication): true if \( v_1 \) is false, otherwise the value of \( v_2 \). (In other words, \( a \ implies b \) has the same value as \( not a \ or else b \).)

At first sight, and then seems equivalent to and, or else to or, and implies to or with the first argument negated. The difference is that any one of these operators may in some cases yield a result on the sole basis of its first argument \( v_1 \), if the value of \( v_1 \) suffices to determine the outcome – even if the second argument does not have a value. They are “strict” (demand a value) for the first argument only, hence the term “semistrict”.

The difference arises for and then when \( v_1 \) is false (result: false), for or else when \( v_1 \) is true (result: true), and for implies when \( v_1 \) is false (result: true). In these three cases the implementation must not evaluate the second argument \( v_2 \). No such rule applies to and and or, which are not required to produce any result for an undefined second argument, and so may use a strict implementation as well as a semistrict one.

As a consequence, the semistrict operators, in contrast with their counterparts in standard mathematical logic, are not commutative: they do not treat their operands symmetrically. For example, \( a \ and then b \) does not necessarily have the same effect as \( b \ and then a \). To be more accurate, any values these expressions yield will be the same, but it is possible for the second to yield a value when the first does not.
Because they enable you to write two-operand boolean expressions whose second operand need not have a value if the first operand’s value leaves only one possible result, semistrict operators are particularly useful for a certain kind of loop used to traverse a data structure. Here is an example from a search routine in class `LINKED_LIST` in EiffelBase:

```eiffel
search_same (v: like first)
  -- Move cursor to first position (at or after current one)
  -- where v appears; move "off" if no such position.
  do
    from
    ... (Initialization omitted)...
    variant
      count = position + 1
    until
      off or else (item = v)
    loop
      forth
    end
  end
ensure
  (not off) implies (item = v)
end
```

The loop will terminate whenever the cursor moves after the last element (off), or hits an element whose value, as given by `item`, is equal to the argument v. The Exit expression tests for either of these conditions to occur. When the first condition (off) is true, however, we do not want to evaluate the second (item = v): not only would its contribution to the result be useless (since a disjunction with one true operand may have no value other than true); evaluating it would in fact be improper since function `item` is only defined when the cursor is on an actual element, which is not the case when it is off: (This is reflected in the precondition for `item`, which includes the condition not off.)

To guarantee the desired result, the Exit condition uses or else rather than or. In the same way, the postcondition only makes sense because of the semistrictness of implies. In other words, the semistrict semantics of or else and implies guarantees that `search_same` will work properly even if v does not appear in the list.

This common loop scheme is captured by iterator routines of EiffelBase, — `do_all, do_while, for_all` and others — declared in high-level classes such as LINEAR and hence available for most practical data structures. To use these routines, it suffices to pass them the appropriate agents as arguments, as in `your_list, for_all (agent your_condition)` which returns true if and only if every element of `your_list` satisfies `your_condition`.
This discussion leads us to the general semantic definition for nonstrict boolean operators:

**Operator Expression Semantics (semistrict cases)**

For $a$ and $b$ of type `BOOLEAN`:

- The value of $a$ and then $b$ is: if $a$ has value false, then false; otherwise the value of $b$.
- The value of $a$ or else $b$ is: if $a$ has value true, then true; otherwise the value of $b$.
- The value of $a$ implies $b$ is: if $a$ has value false, then true; otherwise the value of $b$.

For each of the three forms, if the first condition listed holds, the computation of the expression’s value must not cause evaluation of $b$.

The semantics of other kinds of expression, and Eiffel constructs in general, is **compositional**: the value of an expression with subexpressions $a$ and $b$, for example $a + b$ (where $a$ and $b$ may themselves be complex expressions), is defined in terms of the values of $a$ and $b$, obtained from the same set of semantic rules, and of the connecting operators, here $+$. Among expressions, those involving semistrict operators are the only exception to this general style. The above rule is not strictly compositional since it tells us that in certain cases of evaluating an expression involving $b$ we should not consider the value of $b$. It’s not just that we may ignore the value of $b$ in some cases — which would also be true of $a$ and $b$ (strict) when $a$ is false — but that we must ignore it lest it prevents us from evaluating the expression as a whole.

It’s this lack of full compositionality that makes the above rule more **operational** than the semantic specification of other kinds of expression. Their usual form is “the value of an expression of the form $X$ is $Y$”, where $Y$ only refers to values of subexpressions of $X$. Such rules normally don’t mention order of execution. They respect compositionality and leave compilers free to choose any operand evaluation order, in particular for performance. Here, however, order matters: the final requirement of the rule requires that the computation first evaluate $a$. We need this operational style to reflect the special nature of nonstrict operators, letting us sometimes get a value for an expression whose second operand does not have any.
What makes a bracket expression possible is a feature declared with a bracket alias clause, as in

\[
\text{item alias } \{[\text{key: } H]\}: G \ldots
\]

which — if this declaration appears in HASH_TABLE, and \textit{your_table} is of type HASH_TABLE [T, U] — allows writing \textit{your_table} [your_key] as an abbreviation for \textit{your_table.item} (your_key).

The Kernel Library class ARRAY [G] relies on this technique to allow accessing array elements through the notation \textit{your_array} [n] as a synonym for \textit{your_array.item} (n) for an integer \( n \). You are not limited to one argument: a class MATRIX3 [G] describing three-dimensional matrices may have

\[
\text{item alias } \{[\text{i, j, k: } G]\}: G \ldots
\]

allowing element access under the form \textit{your_matrix} [\( n_1, n_2, n_3 \)].

This mechanism is also useful in connection with assigner procedures: adding \textit{assign put} (after \( G \)) to any of these examples, with a procedure \textit{put} having the appropriate signature, allows you to use assignment syntax, as

\[
\textit{your_matrix} [\( n_1, n_2, n_3 \)] := v
\]

in the last example, an abbreviation for \textit{your_matrix.put} \((v, n_1, n_2, n_3)\). The left side is, again, a Bracket_expression.

The syntax is simple:

**Bracket expressions**

\[
\begin{align*}
\text{Bracket_expression} & \triangleq \text{Bracket_target } \{[\text{"Actuals" } \text{"\}]} \\
\text{Bracket_target} & \triangleq \text{Target } \mid \text{Once_string } \mid \\
& \quad \text{Manifest_constant } \mid \text{Manifest_tuple}
\end{align*}
\]

\textit{Target} covers every kind of expression that can be used as target of a call, including simple variants like \textit{Local} variables and formal arguments, as well as \textit{Call}, representing the application of a query to a target that may itself be the result of applying calls.
Examples of Bracket_expression are

<table>
<thead>
<tr>
<th>Bracket_expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>your_table [your_key]</td>
</tr>
<tr>
<td>your_matrix [n1, n2, n3]</td>
</tr>
<tr>
<td>table_list, i_th(i) [your_key]</td>
</tr>
</tbody>
</table>

In the first two cases, the Call_chain is just a single query, your_table or your_matrix; such a Bracket_expression could appear respectively in class HASH_TABLE or MATRIX3. The last example, with a longer Call_chain, assumes that in the base class for table_list there’s a function i_th returning a table.

The Bracket_target used to the left of the bracket part allows a number of expression variants; Call_chain is the most common, permitting bracket expressions such as f[x] but also a, b, f[x] (to be understood again as an abbreviation: for a, b, f, item(x) for the appropriate item function). One of the other possibilities is Manifest_tuple, as in [a, b, c][i], taking advantage of a bracket alias for item in TUPLE. If you want a more complex expression as target, use a Parenthesized_target, as in

```
([a + b])[i]
```

which will be valid if the type of a + b has a bracket feature.

The reason for the restriction of Bracket_target to specific kinds of expressions is — as you might not have guessed! — the need to make the semicolon optional in all cases without causing any syntactical ambiguity. If you are interested in understanding this fully, you’ll find the details in the final section of this chapter.

The Equivalent Dot Form of a Bracket_expression simply involves replacing the expression by a call in dot notation, using the associated feature. For the above three examples it is:

<table>
<thead>
<tr>
<th>Dot Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>your_table.item(your_key)</td>
</tr>
<tr>
<td>your_matrix.item(n1, n2, n3)</td>
</tr>
<tr>
<td>table_list.i_th(i).item(your_key)</td>
</tr>
</tbody>
</table>

These examples all assume item as the Feature_name for the bracket feature; this is indeed the most common choice, but of course you may choose any name you like.
Here is the validity rule:

**Bracket Expression rule**

A Bracket_expression $x \{y\}$ is valid if and only if it satisfies the following conditions:
1. A feature of the base class of $x$ is declared as alias "[\]."
2. The expression’s Equivalent Dot Form is a valid Call.

The Feature Declaration rule ensures that at most one feature satisfies clause 1. The Equivalent Dot Form, as defined below, relies on that feature.

### 28.8 THE EQUIVALENT DOT FORM

This section defines precisely the notion of Equivalent Dot Form, already introduced informally through examples, and used extensively in the previous sections. It may be skipped on first reading.

For a full specification of the validity and semantics of an Operator_expression or Bracket_expression, we need a precise description of how to obtain its Equivalent Dot Form. Because such expressions may involve components which are expressions of other kinds (such as calls or constants), the definition must in fact be applicable to any kind of expression. In the following definition the most important cases are the first three, giving dot equivalents for the non-dot forms (operators, bracket):

**Equivalent Dot Form of an expression**

Any Expression $e$ has an Equivalent Dot Form, not involving (in any of its subexpressions) any Bracket_expression or Operator_expression, and defined as follows, where $C$ denotes the base class of $x$, $pe$ denotes the Parenthesized Form of $e$, and $x'$, $y'$, $e'$ denote the Equivalent Dot Forms (obtained recursively) of $x$, $y$, $e$:
1. If $pe$ is a Unary_expression § $x$: $x'.f$, where $f$ is the Feature_name of the no-argument feature of alias § in $C$.
2. If $pe$ is a Binary_expression of target-converted form $x \& y$: $x'.f(y')$ where $f$ is the Feature_name of the one-argument feature of alias § in $C$.
3. If $pe$ is a Bracket_expression $x \{y\}$: $x'.f(y')$ where $f$ is the Feature_name of the feature declared as alias "[\]" in $C$.
4. If $pe$ has no subexpression other than itself: $pe$.
5. In all other cases: (recursively) the result of replacing every subexpression of $e$ by its Equivalent Dot Form.
In the first three cases, the Operator Expression and Bracket Expression rules seen earlier in this chapter guarantee that there is a feature \( f \) of the given alias. The Feature Declaration rule then ensures that in all of the first three cases exactly one feature \( f \) satisfies the requirements.

The Operator Expression and Bracket Expression rules both rely on the definition of Equivalent Dot Form, raising the appearance of circular reasoning. But we are only interested in Equivalent Dot Forms of expressions that satisfy clause 1 of their respective rules; this is enough to make the definition of Equivalent Dot Form applicable, and then to use it in the rule’s second clause. So this mutual dependency does not cause circularity.

In case 2 we draw the feature \( f \) not from the original expression but from its target-converted form as presented in the preceding section. It will usually identical, but allows us for example to accept \( \text{your_integer} + \text{your_real} \), treating it as \((\text{your_integer}.\text{converted_to_real}) + \text{your_real}\).

Case 4 is the terminal case of the recursion, covering Formal, Local, Manifest constant, and any Call consisting of a single query with no arguments. Case 5 makes sure that we apply the rule recursively to all constituents of a complex expression.

Case applies, among others, to a parenthesized expression \((f)\), for which it gives us \((f')\) where \(f'\) is, recursively, the Equivalent Dot Form of \( f \).

### 28.9 BOOLEAN EXPRESSIONS

For \text{Boolean_expression}, the grammar at the beginning of this chapter gave three kinds: \text{Boolean_constant}, \text{Object_test} and \text{Basic_expression}. The two boolean constant are \text{True} and \text{False}. \text{Object_test} has its own validity rule. The third case must satisfy an obvious constraint:

\[
\begin{align*}
\text{Boolean Expression rule} & \quad \text{VWBE} \\
\text{A Basic_expression is valid as a Boolean_expression if and only if it is of type BOOLEAN.}
\end{align*}
\]

Here the “type” of a \text{Basic_expression} is the result of applying the Expression Type definition appearing below.

### 28.10 ENTITIES

Entities do not appear as a separate case in the syntax for Expression because they form a special case of Call (more precisely Unqualified_call). But their role as expressions or components of expressions deserves a few comments.

First, as a reminder, the syntactic definition:
The associated constraint, called the Entity rule, required any entity to be one of: attribute; local variable of the enclosing routine if any (including Result if it is a function); formal argument of the enclosing routine or inline agent; feature of a call; Current.

Together with the Call rule, the Entity rule governs the use of identifiers in expressions. A simple consequence of these two constraints is:

```
Entities and variables

Entity ≝ Variable | Read_only
Variable ≝ Attribute | Local
Attribute ≝ Identifier
Local ≝ Identifier | Result
Read_only ≝ Formal | Current
Formal ≝ Identifier
```

The restriction “other than as the feature of an Object_call or qualified Call” excludes an identifier appearing immediately after a dot to denote a feature being called on a target object: in \( a + b, c(d) \), the rule applies to \( a, b \) (target of a Call) and \( d \) (actual argument), but not to \( c \) (feature of a qualified Call). For \( c \) the relevant constraint is the Call rule, which among other conditions requires \( c \) to be a feature of the base class of \( b \)'s type.

The Identifier rule is not a full "if and only if" rule; in fact it is conceptually superfluous since it follows from earlier, more complete constraints. Language processing tools may find it convenient as a simple criterion for detecting the most common case of invalid Identifier in expression.

--- REFERENCE TO ENTITY EVALUATION SEMANTICS

28.11 THE TYPE OF AN EXPRESSION

Every expression has a type; this notion is central to the validity rules governing (among others) assignment, argument passing and the construction of larger expressions from smaller ones.
This static type of the expression, entirely deduced from declarations in the software text, shouldn’t be confused with the dynamic type of its value at some instant of execution.

We are now in a position to define precisely the notion of static type for each kind of expression.

A full definition must remove the effect of genericity: if \( a \) is of type \( D \ [U] \) and \( x \) is an attribute or function declared of type \( G \) in class \( D \ [G] \), the type we want for \( a.x \) is not \( G \) — meaningless outside of class \( C \) — but \( U \). This has been taken care of by the Generic Type Adaptation rule, which tells us to apply the actual-to-formal parameter substitutions whenever our types involve generic derivations. By referring to this rule, the following Expression Type definition can ignore genericity for its own specific cases:

**Type of an expression**

The type of an Expression \( e \) is:

1. For the predefined Read_only Current: the current type.
2. For a routine’s Formal argument: the type declared for \( e \).
3. For an Object-Test local: its declared type.
4. For Result, appearing in the text of a query \( f \): the result type of \( f \).
5. For a Local variable other than Result: the type declared for \( e \).
6. For a Call: the type of \( e \) as determined by the Expression Call Type definition with respect to the current type.
7. For a Precursor: (recursively) the type of its unfolded form.
8. For an Equality: BOOLEAN.
9. For a Parenthesized (\( f \)): (recursively) the type of \( f \).
10. For old \( f \): (recursively) the type of \( f \).
11. For an Operator_expression or Bracket_expression: (recursively) the type of the Equivalent Dot Form of \( e \).
12. For a Manifest_constant: as given by the definition of the type of a manifest constant.
13. For a Manifest_tuple \([a_1, \ldots, a_n] \) (\( n \geq 0 \)): TUPLE \([T_1, \ldots, T_n]\) where each \( T_j \) is (recursively) the type of \( a_j \).
14. For an Agent: as given by the definition of the type of an agent expression.
15. For an Object_test: BOOLEAN.
16. For a Once_string: STRING.
17. For an Address \$v: TYPED_POINTER [T] where \( T \) is (recursively) the type of \( v \).

Case 6, which refers to a definition given in the discussion of calls, also determines case 11, operator and bracket expressions.
28.12 EXPRESSIONS AND THE SEMICOLON

We end this review of expressions with a syntactical note (which you may skip on first reading). The distinction between Basic_expression and Special_expression has, among others, a syntactic purpose. Eiffel’s Semicolon rule specifies that the semicolon as separator is always optional. It must be applicable to any AssertionClause, which can be an Unlabeled_assertion_clause and hence directly follow another clause, which could end with

\[
\ldots f \{x\}
\]

using a Bracket_expression, the application of \( f \) to \( x \). To a naive parser, however, this could look like two successive clauses:

\[
\ldots f; [x]
\]

without the semicolon. The second line, a Manifest_tuple, is also an expression, and hence a possible assertion clause if it were valid. It is not valid, since a tuple cannot be boolean as required for an assertion clause; but that’s validity information, whereas it should be possible to parse software texts on the basis of syntactical information only.

Fortunately, the syntax avoids any such problem thanks to the division between Basic_expression and Special_expression. Unlabeled_assertion_clause, and every context where similar ambiguities could arise, only accept a Basic_expression; all the constructs such as Manifest_tuple that could cause such ambiguities are part of Special_expression.

This technique no loss of generality because if you do want to start a component (for example an Unlabeled_assertion_clause) with a legitimate expression that, syntactically, is a Special_expression, you can just put it in parentheses: as Parenthesized is part of Basic_expression this does the trick.

In some cases, you may also use a Parenthesized_target. Note for example the following assertion, valid if \( f \) is of type BOOLEAN:

\[
\text{require } f (([x, y, z]), \text{count} > 0)
\]

This assertion includes two clauses; the first is true if and only if \( f \) is true, and the second is trivially true since it states that a 3-item tuple has a positive number of items.

Such cases are extreme, and in fact the conscientious programmer always labels assertion clauses:

\[
\text{require property_1: f property_2: ([x, y, z]), count > 0}
\]

Valid, assuming the proper declarations (but not the recommended style).

The recommend style.
But this is only a recommendation. The syntax rule guarantee the basic Eiffel right of omitting semicolons between elements on different lines — greatly enjoyed by all users of the language.