B

This draft: Version 4.82-00-00, 21 September 02 12:15 (Zürich time)
Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2001. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

Specifying systems in Lace *(in progress)*

B.1 OVERVIEW

As you start producing clusters of classes, you will expect the supporting environment to provide language processing tools — compilers, interpreters, documenters, browsers — to process these classes and assemble them into systems.

These tools will need a specification of where to find the classes and what to do with them. Such a specification is called an **Assembly of Classes in Eiffel**, or Ace for short. This appendix presents a notation, the **Language for Assembling Classes in Eiffel**, or Lace, for writing Aces. Although Lace is separate from Eiffel, Eiffel environments must support it.

Two words of encouragement if you feel that after reaching page 843 of the description of Eiffel you should not have to learn yet another language. First, Lace is very much Eiffel-like, so you'll find yourself treading familiar ground in this chapter. But more importantly, for anything other than advanced uses of Eiffel you don't need to study the details of Lace — the way you would study a design or programming language — since you may expect an Eiffel environment to provide an interactive tools that lets you fill out your project's specific needs, provides defaults for everything else, and generates the Ace for you as a result. ISE Eiffel, for example, provides a graphical **Project Wizard** that does all this.

For a good understanding of what's going on behind the scenes it is useful to have a basic understanding of Lace. You can obtain it by reading the basic Ace example of the next section. What comes after that is detailed reference and may be skipped on first reading.

If you are familiar with Lace basics and simply need a reminder on some details, you may find it profitable to use the complete example of a later section. You will find the complete Lace grammar at the end of this chapter.

B.2 A SIMPLE EXAMPLE

Typical elements of an Ace include information about directories and files containing the text of the system's clusters and classes, compilation options (assertion monitoring, debugging etc.) for the classes involved, name of the root class (used to start off execution), location of non-Eiffel elements such as external libraries, target file for the compilation's output.

To help you get quickly a idea of the basic concepts of Lace, here is a simple but typical Ace.

Although simple, this example includes the Lace facilities that suffice for many practical Eiffel systems.

```
system browser root
        EB (browsing)
default
        assertions (ensure); trace (no)
        collect (yes); debug (no)
cluster
        "$INSTALLATION/library/support"
        "$INSTALLATION/library/parsing"

        browsing: "tilda/current/browser"
                default
                        assertion (all)
                option
                        debug (yes): LAYOUT, FUNCTIONS
                end
end
```

This describes a system called browser. The root of this system is a class called EB. The text of EB is to be found in cluster *browsing* (described a few lines below in the Ace); this mention of the root class's cluster, in parentheses, is optional if the entire system has only one class of the name given, here EB.

Default compilation options for the classes of this system are: for assertions, check postconditions (**ensure** clauses), which also implies checking preconditions; do not trace execution; enable garbage collection (*collect*); do not execute **debug** instructions.

These options are system-wide defaults; individual clusters may override them through their own **default** clauses, as does cluster *browsing*. Individual classes may also override the system and cluster defaults through the **option** clause.

The Clusters part, beginning with the **cluster** Lace keyword (reminiscent of the **feature** keyword introducing a Features part in Eiffel), defines the set of clusters; clusters, as you remember, are groups of classes, and the system's classes are collected from the classes of its clusters.

← *Chapter 3 introduced the structure of systems and the notion of cluster.*

The specification of the first two clusters only gives directory names, each written as a Manifest_string. By default, the cluster consists of all classes to be found in the files having names ending with **.e** in this directory. Each one of these files may contain one or more classes. The **.e** name convention is the default; we shall see below how to include files with other names, or to exclude some **.e** files.

A Manifest_string ef

The first two clusters are elements of the Basic Libraries (support and parsing). Their names use Unix-like conventions for environment variables (such as *$INSTALLATION*) to facilitate using the same Ace on different machines. Clearly, such conventions are operating-system-dependent.

The last cluster also has a directory name (this is always required), preceded here by a Cluster_name, *browsing*, and a colon. You will need to include such a Cluster_name whenever other elements of the Ace refer to the cluster: here, for example, the Root clause refers to cluster *browsing* through its Cluster_name, to indicate that this is where the root class EB is.

For this cluster, the default assertion monitoring option, overriding the default specified at the system level, is **all** (monitor everything). Furthermore, the *debug* option is enabled for two classes of the cluster, *LAYOUT* and *FUNCTIONS*.

This example is typical of Aces used to assemble and compile systems without any advanced options.

B.3 ON THE ROLE OF LACE

Before showing the remaining details of Lace, it is important to ponder briefly over the connection of this description to the rest of this book.

Lace support, it was mentioned above, is not a required element of an Eiffel implementation. Why then talk about Lace at all as part of a specification of the Eiffel language? There are two reasons, one pedagogical and one practical.

The pedagogical reason is that since some Lace-like mechanisms, at least elementary ones, will be necessary anyway to execute your software, you would not get a full picture of Eiffel software development without some understanding of possible assembly mechanisms.

the preceding two sections are probably sufficient to get a general idea of the purpose of Lace, but the rest of this appendix will give more details for those readers who are seriously interested. As mentioned already, these details are not essential on first reading, hence the SHORTCUT sign which signals the rest of this appendix as non-crucial material.

The practical reason for paying attention to Lace involves **portability**, and should be of particular concern to authors of Eiffel implementations.

True, because of the variety of possible implementation platforms (hardware, operating systems, user interfaces) and of possible implementation techniques such as interpretation, compilation to machine code, compilation to an intermediate assembly-like code such as C etc., one may not guarantee total portability or enforce a fully general Lace standard. For one thing, an implementation could altogether bypass text-based descriptions such as those of Lace, in favor of interactive input of compilation and assembly options (with a modern graphical or "point and click" user interface); then it would have no need for a description *language* in the textual sense of this word, even though it will still provide the Lace semantics — specification of compilation options, class text location etc. — in some other way.

Even if system descriptions use a textual form, an individual implementation may have non-portable characteristics, stemming for example from the peculiarities of the file and directory system of the underlying platform, or from specific optimization options provided by the implementor.

Along with such non-portable aspects, however, certain facilities will be needed in every implementation. For example, it is necessary to let developers specify whether or not to monitor the assertions of any given class at run time. Then everyone will benefit if all implementations using text-based system descriptions rely on a common set of notations and conventions. This does not guarantee full portability, but avoids unjustified sources of non-portability.

The design of Lace is a result of these considerations. It suggests a default notation for the standard components of system descriptions, while leaving individual implementors the freedom to add platform-dependent or implementation-specific facilities.

B.4 A COMPLETE EXAMPLE

For ease of reference (especially meant for those readers who already know the basics of Lace but are coming back to this presentation for a quick and informal reminder on the form of some clauses), here is a complete Ace using most of the available possibilities. The various components are explained in subsequent sections.

Because it illustrates all the major Lace facilities, this Ace is more complex than most usual ones, which tend to use the basic facilities illustrated by the simple example on page

```
system browser root
        EB (browsing)
default
        assertions (ensure)
        trace (no)
        collect (yes)
        debug (no)
cluster
        "$INSTALLATION/library/support"
        basics: "$INSTALLATION/library/structures"
        parsing: "$INSTALLATION/library/parsing"
        browsing: "not/current/browser"
                use
                        ".lace"
                include
                        "commands"
                exclude
                        "g.t.e"
                default
                        debug ("level2"); debug ("io_check")
                option
                        assertion (all): CONSTANTS, FULL_TEXT
                        trace: FUNCTIONS, QUIT, RENAMED
                        debug (yes): LAYOUT, FUNCTIONS
                        debug ("format"): FULL_TEXT, OUTPUT
                        debug ("numerical_accuracy"): OUTPUT
                visible
                        CONSTANTS as BROWSING_CONSTANTS
                        LAYOUT
                        EB
                                creation
                                        initialize
                                export
                                        execute, set_target, initialize
                                end
                end
```

```
external
        Object: "object_name.o", "../basics.o",
        "-ltermcap", "otherlib.a"
        C: "previous.h", "/usr/$MACHINE/src/scree-.c"
        Make: "../Clib/makefile"
generate
        executable: "$INSTALLATION/bin"
        C (yes): "$INSTALLATION/src/browser/package/eb.c"
        Object (no): "$INSTALLATION/bin/browser"
end
```

B.5 BASIC CONVENTIONS

Let us now proceed to the details of Lace.

You should not have been too surprised by the syntax, which is Eiffel-like. The syntax descriptions below use the conventions applied to Eiffel throughout the rest of this book.

← *Chapter 2 introduced the conventions for syntax description.*

Comments, as in Eiffel, begin with two consecutive dashes -- and extend to the end of the line.

The grammar of Lace also uses some of the same basic components as Eiffel:

- Identifier, such as *A_CLASS_NAME*
- Manifest_string, such as *"A STRING$"*
- Integer_constant, such as *–4562*

← *"IDENTIFIERS", 29.10, page 695; "MANIFEST STRINGS", 27.7, page 617; "INTEGERS", 29.13, page 699.*

As in Eiffel, letter case is not significant for identifiers. The recommended standard is to use upper case for class names and lower case for everything else. Letter case is also not significant for strings except when they refer to outside elements such as file names, directory names or linker options; such strings will be passed verbatim to outside tools (such as the operating system or linker), which may or may not treat letter case as significant.

Lace has the following keywords, which you may not use as identifiers:

adapt	**all**	**as**	**check**	**cluster**	**creation**	**default**	**end**	**ensure**
exclude	**export**	**external**	**generate**	**ignore**	**include**	**invariant**	**keep**	**loop**
no	**option**	**require**	**rename**	**root**	**system**	**use**	**visible**	**yes**

An important convention applied throughout the Lace syntax is that an Identifier is syntactically legal wherever a Manifest_string is, and conversely. For this purpose, the grammar productions given below do not refer directly to these two constructs, but use the construct Name, defined as

Name ≜ Identifier | Manifest_string

As a consequence, if your system contains a class called CLUSTER, which is not a valid Lace identifier since it conflicts with one of the keywords in the above list, you may still refer to it in the Ace by using the Manifest_string "*CLUSTER*". Similarly, although you may give a simple file name such as *my_file* as an identifier, one which does not conform to Lace identifier conventions, such as "tilda*/directory/my_file*", will have to be expressed as a string.

For clarity, all the examples of this presentation use strings for file and directory names.

A consistency condition applies to names used in an Ace: the Cluster_name must be different for each cluster. It is valid, however, to use the same identifier in two or more of the roles of Cluster_name, System_name, Class_name.

ACE STRUCTURE

The structure of an Ace is given by the following grammar.

All clauses were present in the long example above; the earlier, shorter example had all clauses except Externals and Generation.

The Defaults clause gives general options which apply to all classes in the system, except where overridden by cluster defaults or options specified for individual classes. It may also indicate options that apply to the system as a whole; for example, the option

collect (**yes**)

requests garbage collection to be turned on; this only makes sense for the whole system. The precise form of options is explained below.

→ *"SPECIFYING OPTIONS", B.9, page 854*.

The Clusters part lists individual clusters and the associated options.

The Externals clause gives information about any non-Eiffel software element needed to assemble the system.

The Generation clause indicates where to store the output of system assembly and compilation (executable module, object code, code in another target language). By default the output will be produced in the directory where the compilation command is executed.

The order of these clauses should be easy to remember: first you give the system a name (System) and express where it starts its execution (Root); then you specify the options that apply across the board, except where specifically overridden (Defaults); you list the Clusters that make up the system's universe; you indicate what else is needed, beyond Eiffel clusters, to assemble the system (Externals); finally, you indicate where the outcome of the assembly and compilation process must be generated (Generation).

The next sections study the various clauses of an Ace.

B.6 BASICS OF CLUSTER CLAUSES

In an Ace containing a Clusters part, the keyword **cluster** will be followed by zero or more Cluster_clause, each specifying the location in the file system of one of the clusters of the universe, and the properties applying to the classes of that cluster.

Let us examine the possibilities by writing a Cluster_clause through successive additions showing most of the available possibilities.

In its simplest form, a Cluster_clause is simply a Directory_name, expressed as a Manifest_string, as in:

On some operating systems, directories may be called differently (for example "folders") or replaced by some other mechanism.

```
"$INSTALLATION/library/browsing"
```

If you must refer to the cluster in other clauses of the Ace, you will need to give it a Cluster_name. (This will be the name for Lace, and is distinct from the cluster's name for the operating system, which appears as the Directory_name.) The Cluster_name will precede the Directory_name, separated by a colon. If you want to call the above cluster *browsing*, you will declare it as

```
browsing: "$INSTALLATION/library/browsing"
```

An optional Cluster_properties part may then appear, specifying further properties of the cluster. It may contain the following paragraphs, all optional, in the order given: Use, Include, Exclude, Name_adaptation, Defaults, Options and Visible. If present, the Cluster_properties part is terminated by an **end** (and, as with an Eiffel routine, a suggested comment repeating the cluster name).

Here is the syntax of the Clusters and Cluster_properties parts:

```
Clusters ≜ cluster {Cluster_clause ";" …}
Cluster_clause ≜ [Cluster_tag]
                 Directory_name
                 [Cluster_properties]
Cluster_tag ≜ Cluster_name ":"
Directory_name ≜ Name
                 Cluster_properties
                 [Use]
                 [Include]
                 [Exclude]
                 [Name_adaptation]
                 [Defaults]
                 [Options]
                 [Visible]
                 end
```

The following sections explore the various Cluster_properties paragraphs. If, in the meantime, you fear that you might forget the order of paragraphs in a Cluster_properties part, remember the following simple principle: the order is the natural one from the point of view of a language processing **tool** that must process the cluster. For example, a compiler which uses a Cluster_properties specification to compile the classes a cluster, and has already obtained any default specifications associated with the cluster (through the Use paragraph), will take the following actions:

As mentioned earlier, developers should produce an Ace by completing a pre-filled template, rather than from scratch. The template will have the paragraphs in the right order.

Find any files to take into account besides the default (Include).

Discard any unneeded files (Exclude).

To prepare for compiling the class texts, find out if any class name appearing there actually refers to a class having another name (Name_adaptation).

Find out the cluster-level compilation options (Defaults). and start compilation of the cluster's various classes.

When compiling a given class, find out if a specific option applies to it (Options).

Having compiled classes, decide which ones of their properties, if any, must be made available to other systems (Visible).

B.7 STORING PROPERTIES WITH A CLUSTER

The Cluster_properties part may begin with a Use paragraph, as in

```
browsing: "~/current/browser"
        use
                "Ace.mswin"
        end
```

to indicate that the cluster's directory contains a "Use file" (here of name *.lace*) containing the specification of some of the cluster's properties. The content of a Use file must itself be a Cluster_properties conforming to the Lace syntax. This makes it possible to specify cluster properties (for example compilation options) in a file that remains stored with the cluster itself.

In the above example, the Cluster_properties part for cluster *browsing* in the Ace has no further paragraphs beside Use, so all the cluster properties for *browsing* will be taken from the Use file. In the examples that follow, however, the Ace will contain other paragraphs for *browsing*, such as Include, Exclude or Options. In such a case the properties specified in the Ace are added to those of the Use file, and they take precedence in case of conflict.

It is a general Lace principle that whenever two comparable properties may apply (here a property specified in a Use file, and a property specified in the Ace after the Use paragraph) the one appearing last is added to the first or, in case of conflict, overrides it.

Here is the syntax of the optional Use paragraph of a Cluster_properties part:

```
Use ≜ use File
File ≜ Name
```

The Cluster_properties part contained in a Use file may itself contain a Use paragraph.

B.8 EXCLUDING AND INCLUDING SOURCE FILES

The next two optional Cluster_properties paragraphs, Include and Exclude, serve to request the explicit inclusion or exclusion of specific source files. Two important applications are overriding the default naming convention for files containing class texts, and selecting non-standard versions of a library class.

By default, when you list a cluster as part of a system, this includes all the class texts contained in files having names of a certain standard form in the cluster's directory; normally this standard form is *xxx.e* for any string xxx, although certain platforms may have different conventions (for example if periods are not legal characters in file names). The rest of this presentation assumes the *xxx.e* convention.
.lY
A *xxx.e* file may contain one or more classes, written consecutively. It is often a good idea to have just one class per file, with the *xxx* part of the file name being the lower-case version of the Class_name; for example file *cursor.e* would contain the source text for class CURSOR. In some cases, however, you may wish to group the texts of a few small and closely related classes in a single file.

The *xxx.e* convention or its equivalent is only the default. You may wish to remove from consideration a file with a name of this form (because you do not want to include the corresponding classes in your system, or simply because the file contains non-Eiffel text); conversely, you may wish to add to the cluster some classes residing in files having non-conforming names. The **exclude** and **include** clauses achieve this.

Here is a typical use, which excludes file *g.t.e* and includes two files with non-standard names:

```
browsing: "~/current/browser"
        use
                "Ace.mswin"
        include
                "commands"
                "states"
        exclude
        "g.t.e"
        end
```

You may also apply the Exclude facility when you wish a class from a certain cluster to override a class from another cluster. If you exclude a file containing a class of name C, and another cluster contains a class with the same name, this class will override the original C. This is useful in particular if you wish to replace a library class by your own version. Assume for example you want to use your own version of *ANY*, the universal class serving as ancestor to all developer-defined classes. You may achieve this by storing the new version in one of your clusters and excluding the default one (assumed to be in file *any.e* in cluster *default*):

←*See chapter 32 about class ANY.*

```
default: "/usr/local/Eiffel/library/kernel"
        exclude
                "any.e"
        end
```

Here is the syntax of the Include and Exclude optional paragraphs of a Cluster_properties part:

```
Include  ≜  include File_list
Exclude  ≜  exclude File_list
File_list  ≜  {File ";" …}
```

B.9 SPECIFYING OPTIONS

Option values govern actions of the tools that will process the Ace; for example they may affect compilation, interpretation or linking.

An option specification may appear in any of the following three Ace components, all optional:

The Ace-level Defaults clause.

The Defaults paragraph of a Cluster_properties part.

The Options paragraph of a Cluster_properties part.

In the last two cases, the Cluster_properties may be in the Ace itself or in the Use file for one of its clusters.

If two or more conflicting values are given for an option, the last overrides any preceding ones. This means that values in the Options paragraph override cluster-level Defaults values, which override Ace-level Defaults values, and that a value in any of these components overrides any preceding value in the same component.

Here is a specimen of an Ace-level Defaults clause already shown above:

```
default
        assertions (ensure); trace (no)
        collect (yes); debug (no);
```

This example enables options as indicated. It is also acceptable as a cluster-level Defaults, except for the presence of **collect** (enabling garbage collection), which may only be given at the Ace-level since garbage collection applies to an entire system.

To get an example of cluster-level Defaults and Options, let us extend our *browsing* Cluster_clause example:

```
browsing: "~/current/browser"
        use
                "Ace.mswin"
        include
                "commands"
                "states"
        exclude
        "g.t.e"
        default
                debug ("level2");
                debug ("io_check")
        option
                assertion (all): CONSTANTS, FULL_TEXT;
                trace: FUNCTIONS, QUIT, RENAMED;
                debug (yes): LAYOUT, FUNCTIONS;
                debug ("format"): FULL_TEXT, OUTPUT;
                debug ("numerical_accuracy"): OUTPUT
        end
```

The Defaults paragraph overrides any Ace-level default for the **debug** option by enabling execution of Debug instructions in routines of classes of the cluster, for the Debug_key *level2* and the Debug_key *io_check*.

← *"THE DEBUG INSTRUCTION", 16.8, page 371*.

The Options paragraph in turn overrides all preceding Defaults. The syntactic structure of is the same as for a Defaults paragraph, except that here every Option_tag (and optional Option_value in parentheses) may be followed by a Target_list, beginning with a colon, which lists one or more Name; these must be the names of classes in the cluster. In that case the option given overrides the default only for the classes given.

If there is no Target_list, the option applies to all classes in the cluster.

Here is the syntax of Options and Defaults paragraphs:

Defaults ≜ **default** {Option_clause ";" …}
Options ≜ **option** {Option_clause ";" …}
Option_clause ≜ Option_tag [Option_mark] [Target_list]
Target_list ≜ ":" {Class_name "," …}"" sup +
Option_tag ≜ Class_tag System_tag
System_tag ≜ **collect** Free_tag
Class_tag ≜ **assertion** | **debug** | **optimize** | **trace** | Free_tag
Free_tag ≜ Name
Option_mark ≜ "(" Option_value ")"
Option_value ≜ Standard_value | Class_value
Standard_value ≜ **yes** | **no** | **all** | Free_value
Class_value ≜ **require** | **ensure** | **invariant** | **loop** | **check** | Free_value
Free_value ≜ File_name | Directory_name | Name

A Target_list may only appear in an Options paragraph, not in a Defaults paragraph. A System_tag may only appear in an Ace-level Defaults clause.

The syntax permits only one Option_value, not a list of values, after an Option_tag. You may obtain the effect of multiple values by repeating the same Option_tag with different values, as was done in the example with the lines

debug ("*format*"): *FULL_TEXT, OUTPUT*
debug ("*numerical_accuracy*"): *OUTPUT*

which imply enabling the debug option for class *OUTPUT* both for the Debug_key *format* and for the Debug_key *numerical_accuracy*. In case of conflict, as usual, the last value given overrides any preceding ones.

This syntax shows that for an Option_tag as well as an Option_value you may use not just predefined forms (such as *assertion* for an Option_tag and **no** for an Option_value) but also Free forms, each of which is defined just as a Name (Identifier or Manifest_string). This means that along with general-purpose options which are presumably of interest to all implementations of Eiffel (level of assertion monitoring, garbage collection etc.), individual implementors may add their own specific options.

The predefined possibilities for Option_tag (*collect, assertion* etc.) are not Lace keywords, and so may be used as identifiers in an Ace. The predefined possibilities for Standard_value, however, are keywords; they appear in bold italics (***yes, require*** etc.). Remember that you can always use a Manifest_string (such as "*YES*" or "*DEFAULTS*") to write a Lace name, for example the name of a class in the system, which conflicts with a keyword.

When the predefined forms are supported, they should satisfy the constraints and produce the effects summarized in the following table.

Option	Governs	Possible values	Default	Scope
assertion	Level of assertion monitoring and execution of Check instructions	**no**, **require**, **ensure**, **invariant**, **loop**, **check**, **all**. Monitoring at each level in this list also applies to the subsequent levels (**ensure** implies precondition checking etc.). Value **invariant** means class invariant; **loop** means monitoring of loop invariants and of loop variant decrease; **check** adds execution of **check** instructions; **all** means same as **check**.	**require**	
collect	Garbage collection	**no**, **yes**.	**yes**	Entire system
debug	Execution of Debug instructions	**no**, **yes**, **all** or a Name representing a Debug_key. **yes** means same as **all**.	**no**	

optimize	Optimize generated code.	**no**, **yes**, **all**, or a Name representing specific optimization level offered by compiler. In Defaults or Options clause for a given cluster, **yes** governs class-level optimization and **all** means same as **yes**. In Ace-level Defaults clause, **yes** governs system-wide optimization, and **all** means same as **yes** plus class-level optimization.	**no**	
trace	Generate run-time tracing information.	**no**, **yes** or **all**. **yes** means the same as **all**		

B.10 SPECIFYING EXTERNAL ELEMENTS

To assemble a system you may need "external" elements, written in another language or available in object form from earlier compilations. The Externals clause serves to list these elements.

Here is an example Externals clause:

```
external
        Object:
                "object_name.o"; "../basics.o"
                "-ltermcap"; "otherlib.a"
        C: "previous.h"; "/usr/$MACHINE/src/scree-.c"
        Make: "../Clib/makefile"
```

Such a clause contains one or more Language_contribution, each being relative to a certain Language. Every Language is given by an Identifier, such as:

Object: object code, produced by a compiler for some language, to be linked with the result of system compilation or included for interpretation.

Remember that letter case is not significant, so that "FORTRAN" and "make" would also be permitted. Make is a Unix tool, with equivalents on many other operating systems, which works from a dependency list, or Makefile, to recompile or reconstruct software. Make and makefiles are normally not needed for Eiffel classes, but may be needed for external non-Eiffel software.

Ada: Ada language elements.

Pascal: Pascal language elements.

Fortran: Fortran language elements.

C: C language elements.

Make: Descriptions of dependencies needed to recompile non-Eiffel software elements.

The exact list of supported Language possibilities depends on the implementation.

In each Language_contribution, the Language is followed by a semicolon and a list of File names containing the corresponding elements.

The syntax of the Externals clause is the following.

```
Externals ≜ external Language_specifics
Language_specifics ≜ {Language_contribution ";" …}
Language_contribution ≜ Language ":" File_list
Language ≜ Eiffel | Ada | Pascal |
           Fortran | C | Object | Make |
           Name
```

The predefined language names (**Eiffel**, **Ada** etc.) are not Lace keywords, and so may be used as identifiers in an Ace.

B.11 ONCE CONTROL

[To be filled in. Remember to update the “complete example” to include this possibility.]

B.12 GENERATION

The Generation clause indicates what output, if any, should be generated by the assembly process, and where that output should be stored.

A specimen of the clause is:

```
generate
        Executable: "$INSTALLATION/bin"
        C (yes): "$INSTALLATION/src/browser/package"
        Object (no): "$INSTALLATION/bin/M_68040/eb"
```

This Generation clause requests generation of both an executable module and a C package containing the translation of the original Eiffel. Clearly, although any Eiffel environment which is not solely meant for analysis or design will support *executable* generation, the availability of any other target language is implementation-dependent.

ISE's Eiffel compiler generates executable code as well as C packages (including a copy of the run-time system, a Make file and all other elements needed to compile and run the result). The C package generation mechanism provides support for cross-development.

The generate Target, coming after the colon, is either a Directory, as in the first example, or a File, as in the second. If it is a directory, the output will be stored in a file of that directory; the name of that file will normally be the System_name, here *browser*. The tools may also use both the System_name and the name of the root's chosen creation procedure to make up the name of the executable output file.

The Language name (*Executable, C* or Object in the example) may be followed by a Generate_option_value, **yes** or **no**, in parentheses. The absence of this component, as in the first two cases of the example, is equivalent to (**yes**). The last line requests that no *Object* package be generated. The Ace's author may re-enable *Object* generation simply by replacing **no** by **yes**.

Here is the syntax of the Generation clause:

Generation	≜	**generate** Generation_clauses
Generation	≜	{Language_generation **";"** …}
Language_generation	≜	Language [Generate_option] **":"** Target
Generate_option	≜	**"("** Generate_option_value **")"**
Generate_option_value	≜	**yes** | **no**
Target	≜	Directory | File

B.13 VISIBLE FEATURES

As you generate output from a system, you may want to make some of the system's classes available to external software elements that will create instances of these classes (through creation procedures) and apply features to those instances (through exported features).
.lP
Using the Visible paragraph of a Cluster_properties part, you may indicate which classes of the cluster must be externally visible; this will apply by default to all the creation procedures and exported features of these classes, but you may also request external visibility for some of them only. Furthermore, you may make some of them externally available under names which are different from their original names in the class text, for example if they are to be called from a language whose identifier conventions differ from those of Eiffel.

Some external software may also need to refer to the class name itself; this is the case with *eif_proc* and similar functions from the Cecil library, which obtain a routine pointer. If the Eiffel name of the class is not appropriate for this purpose (in particular when it would cause ambiguity), you may define a different external class name.

← *"THE CECIL LIBRARY", 28.16, page 675*.

Here is a Visible added to our example, browsing cluster extended with a Visible paragraph, requesting external visibility for three classes of the cluster, *CONSTANTS*, *LAYOUT* and *EB*:

```
browsing: "~/current/browser"
        … use, include, exclude, adapt, default, option as before…
        visible
                CONSTANTS as BROWSING_CONSTANTS;
                LAYOUT
                        rename
                                choice_menu as "choice.menu",
                                set_reverse as "set.reverse"
                        end
                EB
                        create
                                initialize
                        export
                                execute, set_target, initialize
                        rename
                                set_target as "set.target"
                        end
        end
```

For *CONSTANTS*, you have defined a different external class name, *BROWSING_CONSTANTS*, for use by external software such as Cecil functions.

← *"THE CECIL LIBRARY", 28.16, page 675*.

For *CONSTANTS* and *LAYOUT*, external software can create objects using all the creation procedures of these classes (if any), and call all exported features on these objects. Two features of *LAYOUT* are available to external software (for creation or call) under names different from their Eiffel names, making them callable from a language which prohibits underscores _ in identifiers.

For EB, feature *set_target* is also externally renamed. In addition, you have only requested external availability for specific features of EB: among creation procedures, you only need initialize to be externally available for object creation; and among exported features, you only need *execute*, *set_target* (under its external name *set.target*) and *initialize* to be externally available for calls.

.lA
External software may never use a feature for creating objects unless the class text declares it as a creation procedure, and may never use a feature for calls unless the class text declares it as exported. The Export_restriction subclause (beginning with **export**) and the Creation_restriction subclause (beginning with **creation**) are not permitted to extend external availability beyond what is implied by the Eiffel class text. (For one thing, a secret feature is not required to preserve the invariant, so calling it from external software elements could put an object into an inconsistent state, which is the first step towards Armageddon.)

An "exported" feature is one that is generally available to all clients (exported without restriction). A "secret" feature, which is a special case of non-exported feature, is one which is available to no client except *NONE*.

← *"INFORMATION HIDING", 7.7, page 126.*

.lC
It is not incorrect for an implementation to make all exported features of all classes externally available. With such an implementation, you will usually not need any Visible paragraph. You may still, however, use an Ace (perhaps written for another implementation) that has a Visible paragraph: the semantics of such a paragraph is to specify that certain features should be externally visible; it does not preclude an implementation from providing more externally visible features — the implementation just does more than it has to.

Even an implementation which by default makes all compiled features externally visible may in fact need to support the Visible paragraph. The reason is that a compiler may include a global system optimizer, which will detect routines that are not reachable from the creation procedure of the system's root class, and eliminate such routines from the generated code. The optimizer might also decide to inline all calls to certain routines, and then remove the object code for these routines. In such cases you will need to use a Visible paragraph to guarantee that the routines remain available for use by external software.

The syntax of the optional Visible paragraph is the following:

Visible ≜ **visible** {Class_visibility **";"** …}

B.14 COMPLETE LACE GRAMMAR

For ease of reference, you will find below the complete grammar of Lace, repeating the individual descriptions given earlier in this appendix.

B.15 LACE VALIDITY RULES

The following is a list of Lace validity constraings, presented as a single rule with multiple clauses. As you know, Eiffel validity constraints are presented as "if and only if" rules, letting you know not only what you *must* do to strive for validity, but also how much is is enough that you do to be *assured* of validity. The Lace constraints do not follow this style because some conditions depend on the underlying operating system and its handling of files, folders and other non-language elements that condition the workings of Eiffel tool. So the rules simply state what you must do; I did try to make the rules as complete as possible by including all known platform-independent conditions, but some conditions may have been missed.

Ace validity

An Ace must satisfy the following conditions.

1 • All files listed exist.

2 • All files listed are accessible to Eiffel tools for reading.

3 • All directories listed