Compilation context is an essential element of the compilation process. It defines search zones where compilation units will be searched to support separate compilation of the unit or units which are being compiled at the moment. If compilation context specify the output target of the compilation then the compilation artefact can be either executable module or a library.

Any change in the compilation context may affect the compilation artefacts consistency and recompilation module has to take care of consistency for such case.

- High-level view of the compilation process:
  - Input: one or more source files
  - Output: executables or libraries or group of object files
- Key characteristics of an executable:
  - Name: short or full with path valid OS file name
  - Entry point: anonymous procedure or any publically available procedure with 0 or 1 parameter of type array of strings or name of the unit and signature of the init procedure of the unit with its signature if necessary (when unit has both init with no parameters and init with 1 parameter typed as array of strings)
  - Format: COFF or ELF or others
- Key characteristics of a library:
  - Name: short or full with path valid OS file name
  - Entry points: all publically available routines or specified list of entry points (routine names potentially with signatures in case of name overloading)
  - Format: COFF or ELF or others
  - Kind: static or dynamic or both
- Key characteristics of the group of object files
  - Naming of every object file is based on the name of the unit transformed into a valid OS file name
  - Format: COFF or ELF or others

Thus the source may contain:

- description of the project context either executable or library

or

- description of the search zones – list of clusters where to look for required compilation units

and optionally

- a sequence of compilation units of the current source

If descriptions are not provided then there is no connect available and only syntax checks can be performed for the compilation units

So, formally the definition of the grammar rules will look like

Compilation: (Context$_{opt}$ CompilationUnit)$_{rep}$

CompilationUnit: UseDirective$_{rep}$
AnonymousRoutine|StandaloneRoutine|UnitDeclaration

Context: **context**

ProjectDsc | Clusters

end

ProjectDsc:

target OS$_{opt}$ (**exe** ContextName$_{opt}$ exe_type$_{opt}$ (**init** exe_entry)$_{opt}$
| (**lib dll**$_{opt}$ ContextName$_{opt}$ (**init** lib_entry lib_entry$_{rep}$)$_{opt}$)

Clusters

(**foreign** {ContextName [“**:**” foreign_option$_{rep}$ **end**]})$_{opt}$

Clusters:

(**use** (ContextName (“**:**” use_option$_{rep}$ **end**)$_{opt}$)$_{rep}$)$_{opt}$

OS: Win32|Win64|Lin32|Lin64|Android|iOS|All

exe_type: console|GUI|All

exe_entry: UnitName|ProcedureName (“(“Signature”)”)$_{opt}$

lib_entry: RoutineName (“(“Signature”)”)$_{opt}$

ContextName: Identifier| StringConstant

Examples:

Source #1: Simplest program, which uses default context

```
StandardIO.put (“Hello world!\n”)
```

Source #2: Simplest program, which uses the context explicitly

```
context target exe use Kernel end
StandardIO.put (“Hello world!\n”)
```

Source #3: Simplest program, which uses the context explicitly

```
context target exe init foo use Kernel end
foo do StandardIO.put (“Hello world!\n”) end
```

Source #3: Simplest library, which uses the context explicitly

```
context target lib use Kernel end
foo do StandardIO.put (“Hello world!\n”) end
```