For a better understanding of the mechanism of separate compilation, a beforehand definition of the following concepts is necessary: compilation, linking and build.

**Compilation** refers to the processing of source code files and the creation of an 'object' file. This kind of file contains the relocatable code, the static data, and the lists of exports (objects we've defined) and the imports (objects we need). This step doesn't create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled.

**Linking** refers to the creation of a single executable file from multiple object files. In this step, it is common that the linker will complain about undefined elements it can not find. During compilation, if the compiler could not find the definition for a particular function, it would just assume that the function was defined in another file. If this isn't the case, there's no way the compiler would know -- it doesn't look at the contents of more than one file at a time. The linker, on the other hand, looks at multiple files and tries to find references for those elements.

**Build** is the total process of going from source code files to an executable. So, it comprises both the compilation and the linking.

With these concepts now in place, **separate compilation** can be defined as a facility offered by high-level programming languages which permits the programmer to compile different source files at different points in time. It also hints to the separation of phases of building a program mentioned earlier (compilation and linking). A programming language usually offers this functionality through mechanisms of import (say what elements you need from the outside) and export (say what elements are visible to the outside).

The main motivation behind the facility of separate compilation is found in the early problems of computer science. The memory available and the speed of the processors being reduced, more so in comparison with what in on the market today, made the process of compiling the source files for a software system increasingly costly from the point of view of time spent and resources used (a complete build used to be done over the weekend back in the 1980s). Therefore, the programming languages began to be designed with the idea to be able to process the modules/source files of the project in isolation and only when changes are made. The main advantages that separate compilation brings are: one module may be used by many clients; individual modules may be modified and recompiled without recompiling the entire program; clients of a module may be modified and recompiled without recompiling that module; after a module's object file is generated; the source may be removed to hide the details

A useful software product worth mentioning related to the separate compilation is **make** (and **makefiles**). Make is abuild automation tool that automatically builds executable programs and libraries from source code by reading files (makefiles) which specify how to derive the target program. This tool is also used to manage any project where some files must be updated automatically from others whenever the others change. Informally, it can be said that the make tool separately automatically compiles files when they change and then builds the program (the files which are examined in order to detect changes are usually specified in the makefiles).

Another aspect worth being brought up on this subject is the development of modules written in multiple languages. Basically, any language that can be compiled into an object file can be linked together. So, with a little effort, one can link C to Fortran to COBOL to C++ and so on. Problems which arise when creating such a module are related to compatibility. Certain programming languages or libraries may be using different calling conventions (CDECL, STDCALL or FASTCALL), so a common agreement has to be reached when creating the aforementioned module.

Nevertheless, each programming language brings something new to table so the actual motivation for implementing this mechanism can be very specific (by this it is implied that certain functionalities may go hand in hand or need this feature). So, in the following part, analysis in regards to how this mechanism is implemented in several programming languages is presented.

**Java**

To understand how the separate compilation is achieved in Java, we have to see how a Java program is organized and how its compiler works. The structure of a Java program comprises one or more classes, typically organized so that each class is stored in a separate disk file. The name of the file must match the name of the public class within it. The compiler of Java, in contrast to the one in C++ for example, does not generate object files, but bytecode (the files have the extension .class) which is in turn interpreted by the Java Virtual Machine (JVM). Suppose now that we want to invoke the compiler on a class A, which depends on the classes A1, A2, …, An in order to see how separate compilation works (the dependencies may not be explicitly stated by the programmer). For the compilation to work, all the classes on which A depends must be present in bytecode form, i.e. they must have been compiled earlier. If they are present only in the form of the source files, the java compiler forces their compilation. So, the way in which Java resolves unresolved symbols, unlike C++ which uses the linker for this, is to look for them immediately at compile time. The search path for looking up these symbols expands like this: firstly, it looks in the same file containing class A, then in the same directory in which the file A.java was located and finally it looks in the Java library. If no reference was found until the last step, the compiler concludes that it can not be found.

The mechanism by which these external references are made available in Java is by declaring the class and any other method or variable that we want to expose as public.

**Haskell**

The usual way in which a Haskell program is structured is by modules. Each module is placed in a file with the same name and the extension .hs (or .lhs). Each module is made up of symbols, some of which are imported, some of which the module chooses which to expose to other modules and some of which are local. Symbols are exposed by specifying them between parentheses immediately after the module name (ex. “module Algorithm(g)” - the Algorithm module exposes only the symbol g), and they are imported by declaring in which module it imports (ex. “import Sort (quicksort)” - a module specifies that it needs the quicksort symbol from the module Sort; if all the symbols from the module Sort would have been needed, the syntax “import Sort” was enough to achieve that). If the content of a file is not inside any module, it is assumed to be implicitly part of the main module (which in particular is the one which contains the main function).

Similar to C++, the compiler of Haskell generates as a result an object file (usually with the extension .o). However, besides this, it also generates an interface file (usually with the extension .hi). The interface file contains the information that the compiler needs in order to compile further modules that depend on this module and it contains things like the types of exported functions, definitions of data types, and so on.

Analogous with the Java compiler, if the current module depends on symbols from other modules, it would force their compilation too, if their .hi files are not already present. However, this is achieved only if the compilation command is called with the option “--make” .In the case this option is not specified, the compiler assumes that the .hi files for the module exist and concludes that the symbols can not be found if it can not find them . The search path for the compiler is the following: a list of directories, which by default contains only the current directory but can be modified with certain commands, in which the compiler looks for files with the extension .hi (if the --make option is not passed), and for files with the extension .hs or .lhs (if the option is passed).

Because of the way certain modules are imported inside others, circular dependencies can appear (for example the module A import functions from module B and module B imports functions from module A). Haskell offers a mechanism for dealing with this kind of problem: .hs-boot file. Every cycle in the module import graph must be broken by a hs-boot file which must be compiled first, before the source are compiled. Each .hs-boot file belongs to a source file (and module by extension) and it must live in the same directory as its parent source file. This kind of file acts as a messager for the compiler in the sense that it informs him with the bare minimum of information needed to get the bootstrapping process started.

Haskell also offers the possibility to export and import functions from other programming languages through the Foreign Function Interface (FFI) as well. The basic way to import a function is illustrated in the following example:

foreign **import** **ccall** "exp" c\_exp **::** Double **->** Double

The keyword foreign is used to denote the fact a communication with other programming language is established. After the import keyword (which signales the fact that the function comes from the outside), the function call standard is specified. The name of the outside function and the name of the function inside the Haskell module are then specified. The last thing to be needed is the signature of the function . A function can be exported from Haskell as shown below, with all the keywords mentioned above maintaining their purpose (export signales that the function is available outside):

foreign **export** **ccall** triple **::** Int **->** Int

**Fortran**

The way in which Fortran implements the mechanism of separate compilation is very similar to C++. The compiler generates object code files which are then given to the link-editor. However, in the case of a software system written only in Fortran, you need first to compile all files other than the main program into object files and then compile the main code together with these object files.

Fortran structures its program in modules. A module consists of two parts: a specification part for statements declaration and a contains part for subroutine and function definitions. A module can be incorporated in a program or subroutine by the use statement (eg “use maths\_functions” - makes the module maths\_functions available inside). By default, all the symbols from a module are exported through the use statement. The keyword which changes the accessibility of a symbol is “private”.

Fortran 2003 introduced language features which can guarantee interoperability between C and Fortran (and to more languages by using C as an intermediary). These features are mostly accessed through the intrinsic module “iso\_c\_binding”. To call a C function, one would need to do something like this:

! Interface to C routine

interface

integer(c\_int) function how\_many\_geese(flock\_num) bind(C,'howManyGeese')

! Interface blocks don't know about their context,

! so we need to use iso\_c\_binding to get c\_int definition

use, intrinsic :: iso\_c\_binding, only : c\_int

integer(c\_int) :: flock\_num

end function how\_many\_geese

end interface

First an interface must be declared. This is essentially equivalent to the C function prototype, and lets the compiler know about the number and type of the arguments, etc. The bind attribute is used to tell the compiler the name of the function in C, which may be different to the Fortran name. Finally, the Fortran program needs to be linked against the C library that includes the implementation of “howManyGeese()”, and then “how\_many\_geese()” can be called from Fortran.

**Modula-2**

Modula-2 was designed to support separate compilation and data abstraction in a straightforward way. The way in which it structures its program is through modules. Each Modula-2 module is made up of two parts: a definition module, the interface portion, which contains only those parts of the subsystem that are exported, and an implementation module, which contains the working code that is internal to the module. Except for standard identifiers no object from the outer world is visible inside a module unless explicitly imported and no internal module object is visible from the outside unless explicitly exported. In the example below, all that module M1 exposes are the objects a, b, c, P and all that module M2 imports are the objects exposed by M1. To access the objects, M2 would have to use the M2 qualifier like this: “M2.a”.

**DEFINITION** **MODULE** M1;

**EXPORT** **QUALIFIED** a, b, c, P;

**MODULE** M2;

**IMPORT** M1;