CERN-Data Handling Division
DD/84/19
November 1984

SEPARATE COMPILATION IN A MODULA-2 COMPILER

David G. Foster

(Submitted to the Software Practice and Experience)

DD-df

Separate Compilation in a Modula-2 Compiler

David G. Foster

Data Handling Division

CERN 1211 Geneva 23, Switzerland

Abstract

A new approach to the compilation of Modula-2(1) is described. The compiler uses a simple data base which provides the basis for a portable programming support environment. The environment consists of a number of tools which operate on the data base and provide the programmer with facilities to create, and easily maintain, a multi-module system.

Key words: Modula-2, Compilers

(1)Modula-2 was designed at the Institut fur Informatik, ETH, Zurich.

Introduction

Modula-2 was introduced by Niklaus Wirth and has been widely described as the successor to Pascal. The first implementation was made available by the ETH, Zurich and several compilers have since been produced, most being derivatives of this original compiler. [1]

The compiler to which the following discussions relate is a cross compiler which currently runs on a VAX UNIX(2) system (Berkeley 4.2) and produces code for the Motorola 68000 microprocessor [2] . The compiler has recently been ported to a VAX VMS system.

Separate compilation requirements in Modula-2

A Modula-2 program consists of a number of files containing the text of DEFINITION and IMPLEMENTATION modules to be compiled. The resulting object modules can then be link edited and loaded. All DEFINITION modules must be compiled before their corresponding IMPLEMENTATION parts, and the definitions of all imported modules must have been compiled before the importing module. Thus to create the object modules of a Modula-2 program, a number of separate invocations of the compiler is required. The type checking by the compiler must extend to objects that have been imported from other, separately compiled, modules.

A DEFINITION module contains the declarations of objects which may be imported into other DEFINITION or IMPLEMENTATION modules. It is these declarations which are used when performing type checking of imported

(2)UNIX is a trademark of AT&T, Bell Laboratories.

objects. The problem becomes one of preserving declarative information so that it can be used subsequently by modules importing from this DEFINITION module.

One solution to this problem has been to save symbol table information in a binary format file which can be found and read by the compiler when required. This technique requires the creation of a new file, for each definition module, with a name that is related to the name of the DEFINITION module. So, typically the compilation of file "test.def" which contains the DEFINITION module "test" will produce the file "test.sym". Compiling another module which contains the statement "IMPORT test" implicitly instructs the compiler to open the file named "test" with the extension "sym" to find the symbol table information. The details of this approach have been well described [3] and it is a rather different approach that will be described here.

In the following discussion the nomenclature (a) <- (b) should be read as the DEFINITION or IMPLEMENTATION module (a) imports from, and depends on, the DEFINITION module (b).

Importing by Recompilation

The compilation of a definition module does not create a new file containing symbol table information, but enters a single line of information into an existing file. This file, or module base, is a rather simple data base of information about separately compiled modules. The contents of a

line of this file are shown below.

Entity number
Module name
Checksum
Definition module file name
Date/Time of compilation of definition module
Import list of definition module
Implementation module file name
Date/Time of compilation of implementation module
Import list of implementation module

For some modules, for example a program module which has no DEFINITION part, some of the fields may not be present. Each entry in the module base is given an "Entity number" so that an entry can subsequently be referenced in a convenient way. For example, the "Import list" in the data base contains the "Entity numbers" of the modules that are imported.

The compiler produces a data structure from this file at the start of every compilation so the module base need only be read once. When an IMPORT statement is encountered in the source text currently being compiled, the subsequent module name is looked up in the module base. If it is not there then the imported module has not been compiled and an error is signalled. Assuming no error, the compiler changes input streams to read from the file indicated by the "Definition module file name" entry in the module base. The main procedure of the compiler is called recursively at this point, and the imported module compiled.

Once the imported module has been compiled a symbol table tree of identifier and structure records describing the imported objects is available. This tree is preserved when the input stream is restored after com-

piling the imported module. When all imported modules have been compiled a linked list of symbol table trees has been formed that are then used when checking type compatibility of imported objects. In the case of a module being imported twice the second importation merely refers to the tree already created.

At the end of compilation of a given module an entry is created or updated in the module base. Having compiled a module, all the modules that it depends on are known to the compiler. These modules are those which were referenced via import statements. Given the system of modules: IMPLEMENTATION (a) <- (b) <- (c) and also DEFINITION (b) <- (d) then (a) depends on (b),(c) and (d). If the "Entity numbers" of (a), (b), (c) and (d) are 4, 3, 2 and 1 respectively then compiling (a) would create an "Import list" of 1,2,3,4. Since an IMPLEMENTATION module always depends on its corresponding DEFINITION module, the "Import list" of (a) contains its own "Entity number". Although (a) only depends on (c) indirectly (via (b)) the "Import list" of (a) shows the dependency.

Consistency checking

A number of checks are needed to ensure that a system of modules is kept consistent. In this particular schema it is necessary to ensure that a given DEFINITION module has not changed between the time its entry was created in the module base and the time it is compiled as an imported module. The module base contains a checksum for each DEFINITION module; this is compared with a newly calculated checksum each time the module is imported. If these checksums differ then the file has been changed, but not recompiled, and an error message is generated.

When a DEFINITION module is recompiled it is a simple matter for the compiler, using the import lists, to "invalidate" all modules that depend on the recompiled module (in practice invalidating a module entry in the module base involves setting the date/time field to blank). This is necessary since recompiling a DEFINITION module implies a change of interface, so all modules that use this interface need to be re-compiled. This is true even for indirect dependencies as in the system (a) <- (b) <- (c) where a change to (c) requires recompilation of both (b) and (a). Recompiling a DEFINITION module also invalidates its associated IMPLEMENTATION module since this too uses the DEFINITION module interface.

Tools

A series of tools operates on the information present in the module base. The tools consist of a Pascal program and a shell script (or appropriate command language file). The Pascal program performs the computation while the shell script provides the interface to the operating system and the user. These shell scripts are only a few lines long and the Pascal programs contain about 200 lines each. The Pascal programs are portable since they do not rely on operating system functions, instead the shell script performs these where required.

Automatic Recompilation

In early Modula-2 implementations the only tool available was the compiler itself. Recompiling a DEFINITION module involved studying hand drawn graphical representations of the module interdependencies to determine the other modules that required recompilation.

The recompilation tool operates on the module base and finds those entries that have been invalidated by the compiler. Invalidated definition modules are recompiled first, starting with the lowest level modules. Consider (a) <- (b) <- (c) where recompiling (c) has invalidated both (a) and (b). (b) must be recompiled first since this is the lowest level module requiring recompilation. Clearly if (a) were recompiled first, then (b), this would again invalidate (a) requiring it to be compiled a second time. All associated implementation modules which have been invalidated can then be recompiled in any order.

When the program has finished, the module base is in a consistent state. This guarantees that all the interfaces have been verified and that the text files of the DEFINITION modules truly represent the interfaces used by the compiler. This is an improvement over the symbol file approach where the text file is the interface seen by the programmer, but the symbol file is the interface seen by the compiler. This can lead to problems if a DEFINITION module is changed but not recompiled since an out of date symbol file would be used during subsequent compilations.

Module body initialisation order

When a Modula-2 program is started, all the module bodies are executed, ending with the main program. These module bodies may be used, for example, to initialise global variables of a particular module. A module may import a variable which it then uses in its module body initialisation routine. This possibility requires that the imported module be initialised before the importing module. It is possible for IMPLEMENTATION modules to be mutually dependent, IMPLEMENTATION (a) <- (b) and IMPLEMENTATION (b) <-

(a). In this case the order of initialisation is undefined.

A tool has been provided which first searches the module base for the main program module (one with no DEFINITION module entry). From this entry it then deduces all the other modules that need to be included in the link edit phase. In addition the order of initialisation of the modules is deduced according to the above criterion. If mutual dependencies do exist then a warning message is issued and an order assumed.

This tool allows the use of a general link editor by performing the special functions of the link editors normally used in Modula-2 systems.

Graphical representation

It is generally useful to be able to see a complex system of modules and their dependencies in some graphical form. This is particularly appropriate when planning the inclusion of new modules into an existing system. The module base has been used to produce a grid like representation of the dependencies. The rows and columns represent the modules in the system and the intersection states whether the DEFINITION or IMPLEMENTATION of the row module depends on the DEFINITION of the column module.

Compiler structure

The compiler has been structured in such a way as to be able to call itself recursively to deal with imported modules. It was planned to have a one pass syntactic and semantic analyser, however it is not possible to fulfil completely the requirements of the language by a one pass analyser. Generally in Modula-2 an object may be declared and used anywhere within

the scope in which the declaration makes it visible. Thus although inclusion of Pascal like "FORWARD" keywords may remove the problem of forward references for procedures and functions it does not attack the general problem as it exists in Modula-2. The structure of the compiler enables one pass to be completed and if potential forward references are detected, the symbol table trees are saved and the compiler called recursively. Information from the previous pass is extracted from the trees, where required, during the second pass. Thus the compiler will perform the syntactic and semantic analysis in one pass if possible, otherwise it will take two passes.

Conclusion

The technique of recursively recompiling imported modules has kept the size of the compiler very small by avoiding the necessity of special code to create and read symbol files. A small price is paid in efficiency but in practice this has not been a problem. Creating the module base has enabled a number of small tools to be written that are very powerful in helping the programmer to maintain his Modula-2 system. This has been done in a system independent format wherever possible to enable the compiler, and tools, to be ported to different systems.

Acknowledgments

Thanks are due to Duncan Baillie for writing the tools and to Horst von Eicken, Julian Blake, Petrus van der Stok, Alistair McKeeman and Thorsten von Eicken for their contributions to the development of the compiler.

References

1. Terry L. Anderson, "Seven Modula-2 Compilers Reviewed," Journal of Pascal,Ada and Modula-2, pp. 38-43 (March/April 1984).

2. Julian Blake, Horst von Eicken, and David Foster, "Developing Programs for the Motorola 68000 Microprocessor at CERN," Europhysics conference on Software Engineering, Methods and Tools in Computational Physics, (August 1984).

3. Leo Bernhard Geissmann, Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith, Diser Publication (1983).