IMPLEMENTING SEPARATE COMPILATIONS IN PASCAL

G. Nani

Department of Mathematics, University of Genova
Via L. B. Alberti,4 - 16132 Genova, Italy

ABSTRACT

PIC (Pascal Interface Controller) is a tool for the development of large, modular software systems in Pascal. The new kind of modularity, implemented by PIC, is analyzed in the paper. Two different versions of PIC have been developed, to interface with Pascal compilers providing different support to program decomposition into compilation units.

1. INTRODUCTION

Standard Pascal[1] programs exhibit a monolithical structure, a feature which can be profitably employed especially in education. This feature, however, is in contrast with modular programming, a vital technique for the development of large software systems. Several proposals for Pascal extensions were discussed and implemented. Most of these proposals rely on specific compiler features, thereby limiting programs transportability. For the many many different implementations of modularity presently available in commercial Pascal systems, a large program can be partitioned into several compilation units. A compilation unit is a file containing a collection of constant, type, variable, procedure or function declarations. Only one compilation unit, the main program, may contain an executable part.

In the Pascal system by Kieburtz et al.[2], variables belonging to the outermost scope of a compilation unit ('global level', in our terminology) define a permanent environment for all procedures and functions declared in the same unit. Procedure, function and variable identifiers can be exported: a universal global environment for all units is made by all exported identifiers of all units. When a unit is compiled, only its internal congruence is checked; inter-unit checks are performed in a pre-linking phase.

The OMSI Pascal-2 system[3] implements a global environment composed by all data structures, which is used by all compilation units for inter-unit communication; global procedures and functions are selectively imported by each unit. However, it is left to the programmer the responsibility of a correct use of the declarations. To this purpose, it is recommended to collect all global declarations in one file, which can be concatenated, at compile time, with all compilation units.
This latter form of modularity is simple but sufficiently powerful for many applications.

An alternative approach to the above has been experienced in the Source Linker system [4,5,6], which extends standard Pascal with module visibility rules.
The functionalities of such proposal make it well suited for the design of

library modules to be "synthesized" into small to medium size programs (bottom-up programming).

The use of precompilation phases to enforce program modularity has been exploited by C preprocessors [7]. Because of the lack of type checking mechanisms in the C compiler, these tools cannot implement a separate compilation mechanism, leaving interface consistency checks to the programmer responsability.

2.THE PIC SYSTEM: GENERALITIES

We propose a tool, called Pascal Interface Controller (PIC), to extend Pascal with the module concept. PIC [8] has been designed to support "programming in the large", that is, it is well suited for the design and development of medium to large software systems. To this purpose, PIC limits the elaboration of non-declarative parts (procedure bodies) to a minimum, in order to be efficient in large systems processing.

PIC implements a separate compilation mechanism upon the basic facilities for independent compilation, which are already present in all commercial Pascal systems. That is, PIC analyzes a collection of modules (each one expressed according to a syntax which shall be detailed in Sect. 3), and modifies their texts, producing a set of output modules (compilation units) ready for being independently compiled by the underlying Pascal compiler. If such compiler does not perform interface consistency checks among modules, PIC ensures that their interfaces are consistent, because of the way each compilation unit is built up.

Presently, two different versions of PIC have been implemented.
The first PIC implementation has been designed for interfacing with compilers (such as OMSI Pascal-2 for DEC computers, VAX/VMS Pascal, Berkeley Unix Pascal) providing

- directives to define global procedures (or functions) as external identifiers (i.e., identifiers that can be referenced by other compilation units);
- mechanisms for compile-time inclusion of a file containing data declarations (i.e. variables being global to the whole program) to allow correct "communication" across compilation units.

Only the first of the two above requirements is truly indispensable for a correct transformation of modules, since, if automatic inclusion is not supported by the compiler, the system can be extended so as to perform inclusion directly (before compilation), or by the use of other system utilities (e.g. editors).

The second version has been designed so as to interface with Pascal systems, such as Kieburtz's and DEC VAX Pascal [9], supporting directives to declare variables, procedures and functions as exportable from the compilation unit in which they have been defined, and to selectively import them in other units.

3. MODULARITY DEFINED BY PIC

The form of modularity introduced by PIC is closely related to that of Modula-2 [10,11]. As Modula-2, PIC has two kinds of modules: a definition module and an implementation module.

A definition module defines the outside interface of its corresponding implementation module and represents the declarative part of this latter one. Any definition module will contain:
(i) a heading: definition module "identifier" ;
(ii) type, constant, label and variable declarations, procedure or function definitions
(iii) import and export lists, which define exchange of information with the environment.

An export list is introduced by a define clause and specifies those identifiers which are visible within the corresponding module and usable from the outside. Each identifier mentioned in an export list must be visible at a global level. A module may contain at most one export list.

The use clause introduces the set of import lists. A module may have several import lists, each of which specifies those identifiers which are used, but not defined, within the module itself. A module can be imported as a whole, thus importing each exported identifier, or single entities may be imported selectively.

Any type or constant identifier can be explicitely renamed, when it is being imported, by means of the as clause. This feature, which can avoid name conflicts, is supported in the second version of PIC only; it may also be used to improve program readability.

The module name to which an object belongs must be specified in the import list through the of clause. An identifier in an export or import list may belong to a constant, type, variable, procedure or function. Predefined identifiers are automatically imported in any definition module and thus cannot be redefined at a global level.

In the following, an example of two definition modules is given.

```
-------------------------------------------------------------
definition module prstack;
  define
    stack,error,sizestack,tipostack;
  use
    initstack,push,pop of 'stacklib';
  const
    sizestack = 6;     { stack size }
  type
    tipostack = packed array [1..3] of char; {stack elements}
    stack = record
              pila : array [1..sizestack] of tipostack;
              top : integer
            end;
  var
    pila0,pila1 : stack;        { stack instances }
```

```
    . . . .
  procedure  exit;
  procedure  error(i : integer);
 endm
-------------------------------------------------------
definition module stacklib;
  define
    initstack, push, pop;
  use
    sizestack, tipostack, stack, error of 'prstack';
  procedure initstack (var pst : stack);
  procedure pop (var pst: stack; var ptipos: tipostack);
  procedure push (var pst: stack; par : tipostack);
end.
-------------------------------------------------------
```

An implementation module automatically imports its definition module and contains the bodies of procedures or functions declared in the former one. The implementation module of the main program is syntactically identified by the heading

program "module identifier" body

and is the only one having an executable part.
Any other implementation module is headed by

"module identifier" body

The implementation modules corresponding to the above definition modules are the following:

```
-------------------------------------------------------
stacklib body
  procedure initstack;
    begin  ... end;
  procedure pop;
    begin  ... end;
  procedure push;
    begin  ... end;
-------------------------------------------------------
program prstack body
   procedure exit;
     begin   ...  end;
   procedure error;
     begin   ...  end;
begin
      ...
end.
-------------------------------------------------------
```

Exchanges of information regulated by import/export mechanisms are subjected to the following restrictions. Importing a procedure or function identifier requires that the types of its formal parameters, and of the function result, are explicitly imported. The type identifier of any imported variable must be imported. In the case of variables with anonymous type declaration, all type or constant identifiers appearing in the same variable declaration must be imported.

By default, in the first version transparent export is assumed, i.e. the internal structure of each exported object is visible. In more detail, exporting an identifier makes any other identifier, which appears in the right side of its declaration, automatically visible. We call public any identifier globally defined in a module and exported. Because of the export rules adopted here, an identifier defined at a global level must not be in contrast with any public identifier (even a non-imported one) or with any global non-public identifier, which appears in the right side of the declaration of a public identifier. The constraints imposed to the programmer are strictly connected to the compiler functional features and are due to the fact that the first PIC version has been developed for a class of Pascal compilers, which have a minimum set of functions to support independent compilations.

Transparent export has a slightly different implementation in the second version. Given the identifier 'id', its qualified notation is the identifier obtained by prefixing 'id' with its module identifier, followed by a special alphanumeric symbol (e.g. underscore, '_'). By exporting an identifier, all identifiers occurring on the right side of its definition are made visible in their respective qualified notation. However, an identifier which is visible in qualified notation can always be explicitely imported (and renamed), so as to be referenced as wished.

4. COMMENTS AND CONCLUSIONS

The paper has presented PIC, a tool for the development of large programs in Pascal. The supported form of modularity is simple, yet sufficiently powerful. It allows only explicit information exchange (thus being a statical form of modularity) among modules, which is functionally similar to that of Modula-2. The main difference with the latter lies in PIC simplicity, which also gives a better homogeneity of modules treatment (especially for the lack of local modules).

We concentrated on the possibility of supporting separate compilations of program portions, rather than in the visibility (information hiding) facilities of modular programming. This choice was due to the need of designing a minimal set of functionalities to introduce modularity in the Pascal language. These extensions are sufficient to implement a separate compilation facility by means of the independent compilation facility (which is supported by all commercially available Pascal implementations), without extending Pascal too much.

Two PIC versions were developed, designed so as to interface with compilers providing different features for indipendent compilation.
With respct to the second version of PIC, the first version is limited by the assumption of the presence of a global environment, which is included by all compilation units, and by the requirement to limit, as much as possible, the elaboration on implementation modules.
Because of the functional characteristics of PIC, a modular implementation of the system is required which enhances portability and the possibility of adapting the system to any programming environment.Infact, the system can be adapted to output suitable directives, according with the different syntax for compilation units in the underlying compiler, by suitably modifying the output procedures for the declarative regions into which

definition modules have been transformed.

The whole system is written by use of its own form of modularity, and consists of 10 definition modules (plus other 10 implementation modules) comprising about 6000 code lines.
Up to now, PIC has been used in a few software projects; the largest of them being the development for a modular ,easily transportable version, of one already existing compiler.
By this way, a unique source code version need to be kept, from which the Pascal-2 and/or Pascal-VAX versions can be automatically derived. We experienced that this greatly simplifies the maintenance process for the compiler.

From the above observations, and experiences, some concluding remarks can be thus summarized:

- PIC is simple to be used
- PIC outputs readable modules
- PIC limits to a minimum elaboration over implementation modules, which makes it efficient and of practical use in large software implementations.

REFERENCES

[1] D. Cooper, "Standard Pascal User Reference Manual" ,W.W. Norton Company , 1983.
[2] R.B. Kieburtz, W. Barabash and C.R. Hill, "A Type-checking Program Linkage System for Pascal", Proc. 3rd Int. Con. on Software Engeneering, Atlanta 1978.
[3] "Pascal-2 User Manual", Oregon Software, 1983.
[4] M.Ancona, L.De Floriani, G.Dodero, S.Mancosu, "Integrating library modules into Pascal programs", Proc. 6th International Conference on Software Engineering, Poster Session, IEEE, Tokyo, 13-16 September 1982.
[5] M. Ancona, L. De Floriani, G. Dodero, P. Thea "Program Development by using a Source Linker" , Proceedings of the 4th Jerusalem Conference Information Techonology , 1984.
[6] G. Nani , "Source Linker User Manual " , Tech. Rep. n. 160 Istituto per la Matematica Applicata, Genova, 1984 (in Italian).
[7] S. Boyd, "Modular C", SIGPLAN Notice, 18(4) , 1983.
[8] G. Nani, "Implementing separate compilation by means of independent compilation", Tech. Rep. n. 206, Istituto per la Matematica Applicata, Genova, 1986 (in Italian).
[9] "VAX Pascal User Manual", Digital Equipment Corporation , 1982.
[10] N. Wirth , "Programming in Modula-2", Springer Verlag, 1982.
[11] N. Wirth, "The Module : a system structuring facility in high level programming languages" , Proceedings Symposium on Language Design and Programming Methodology , Sydney, 1979.