Egenhofer, M., and A. U. Frank. "A Precompiler for Modular, Transportable Pascal." *SIG Plan Notices* 23, no. 3 (1988): 22-32.

SIG PLAN NOTICES

A Monthly Publication of the Special Interest Group on Programming Languages

VOLUME 23 NUMBER 3 MARCH 1988

CONTENTS:

EDITORIAL		1
LETTERS		2
ANNOUNCEMENTS		5
TECHNICAL CONTRIBUTIONS:		
D. Hemmendinger	: Unfair Process Scheduling in Modula-2	7
F. W. Calliss	: Problems With Automatic Restructurers	13
M. J. Egenhofer A. U. Frank	: A Precompiler for Modular, Transportable Pascal	22
D. Yeh U. Kastens	: Automatic Construction of Incremental LR(1)—Parsers	33
A. L. Furtado	: Towards Functional Programming in Prolog	43
N. Soundararajan R. L. Costello	: Responsive Sequential Processes	53
M. Klerer H. Liu	: A New Benchmark Test to Estimate Optimization Quality of Compilers	63
J. Bergin S. Greenfield	: What Does Modula-2 Need to Fully Support Object Oriented Programming?	73
N. Holsti	: Using Formal Procedure Parameters to Represent and Transmit Complex Data Structures	83
P. D. Coward	: Determining Path Feasibility for Commercial Programs	93
G. Q. Maguire, Jr. J. M. Smith	: Process Migration: Effects on Scientific Computation	102
P. Ranger	: Some Comments on the Forthcoming "Extended Pascal" Standard	107

NOTE: ACM SIGSOFT '88 CALL FOR PAPERS ON PAGE 52

A Precompiler For Modular, Transportable Pascal

Max J. Egenhofer
Andrew U. Frank
University of Maine
Orono, ME 04469, USA
MAX@MECAN1.bitnet
FRANK@MECAN1.bitnet

Abstract

Highly modular, object-oriented software systems require support from programming languages to produce reusable code. Pascal lacks such features that easily propagate type definitions and routines from one module to another, such as MODULA-2 or ADA. This paper describes a Pascal extension which includes directives to mark data definitions and operations as externally visible. A precompiler which handles import directives translates the source into the compiler-specific Pascal syntax. This precompiler has been successfully used for five years in education and research.

1 Introduction

The lack of suitable support of modular programming techniques in Pascal [Jensen 1978] has been discussed for a long time and lead to several proposals for extending Pascal [Nani 1987]. Recently, commercial compilers such as Lightspeed Pascal [Think 1986] Pascal/VS [IBM 1981], and VAX Pascal [Digital 1987], incorporated means for separate compilation of programming units, propagating type definitions and routines across compilation units; however, the formats vary from compiler to compiler. Usually, manual editing is required, and definitions are not automatically updated when the original definitions are changed. Modularity and separate compilation have been an issue in the forthcoming standard for an extended Pascal [Joslin 1986].

This paper presents an extension to Standard Pascal which has served during the last five years as a suitable tool for 'programming in the large' both in education at universities and in research. Due to the simplicity of the syntax extension, students could quickly learn the concepts of modular programming. On the other hand, the tools were powerful enough to develop large software systems [Frank 1982], [Frank 1984]. The conversion

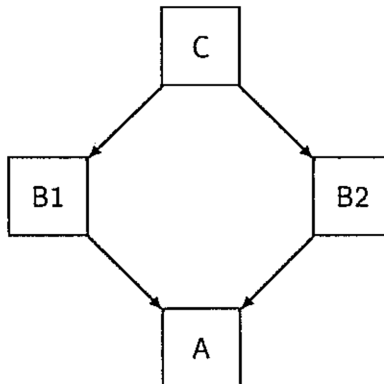
from the source code to the machine specific Pascal is done by a precompiler providing all the required types and routines of externally, separately compiled modules. The precompiler automatically adapts to whatever format the compiler expects for separately compiled units. Transportability of the code is guaranteed by a conditional compilation feature in the precompiler. This Pascal is different from MODULA-2 [Wirth 1982] or the proposed standard, because it combines the definition and implementation of routines in the same file, and thus no manual copy of the interface is needed. Instead, the interface is derived from the implementation by a precompiler.

2 Concepts

The modular programming concept is closely related to abstract data types (ADT) [Guttag 1977], [Goguen 1978]. In order to use such a concept, programming tools are needed to propagate the interface of one abstract data type to another, higher level abstract data type, and to make use of lower level abstract data types when composing higher level types.

Each abstract data type must be implemented as a module, a separate compilation unit without a main program, and thus the terms *module* and *ADT* will be used equivalently. On the other hand, programs are compilation units with a single executable main program. Programs cannot define parts which are externally usable.

In an object-oriented, modular architecture, type definitions are inherited from other type definitions. The structure of such objects is not strictly hierarchical. The same holds for routines: in general, a software system is neither linear nor purely hierarchical; on the contrary, modular structures aim to have reusable code with interfaces which control the structure and provide independence from different implementation details. A useful separate compilation mechanism must include the treatment of structures like the ones in the following example: A fundamental module A provides a type definition for the A_type and some operations. Two other modules B1 and B2 define the B1_type and the B2_type, both having the A_type as part of them. Finally, a module C defines the C_type which is a synthesis of B1_type and B2_type. So, the modules B1 and B2 both need the A_type, and in consequence, C needs B1_type, B2_type, and A_type to compile successfully. Simply collecting all types along the paths between the modules would lead to having the A_type included twice: once from B1, and once more from B2. Most compilers complain about such definitions since the A_type would be interpreted using the same type name for two different definitions.



Most commercial Pascal compilers have some functionality for independent compilation, but they are founded on the 'include'-principle which does not work for non-hierarchical, nested imports.

- A more sophisticated mechanism is needed to sort out multiple definitions of the same type.

Large software systems suffer from the inherent problems of modifications in the routine heads of exportable operations. It is very cumbersome and error-prone to reproduce the definitions manually. Errors resulting from such omitted updates are often fatal and hard to locate.

- A suitable modular Pascal must provide an automatism for updating changes in exported types and routines.

3 Separate Compilation

The crucial parts for modular systems with separate compilation are the externally visible definitions of constants, types, and routines. The following two ways for declaring external usage are used in this extended Pascal:

- Exported constants and types are included in an

  ```
  EXPORT TYPE;
  ...
  PRIVATE;
  ```

  bracket. Each module can have several of these brackets.

- Exported functions and procedures are flagged with the keyword

  EXPORT OP;

For example, the abstract data type iv1 for a one-dimensional interval of integers exports the type definition of the interval type and some routines. The syntax for this module looks as follows:

```
MODULE iv1;
INTENT1 the ADT for a one-dimensional interval;
AUTHOR2 af, may 1985;

EXPORT TYPE;
TYPE boundType = integer;
     iv1Type = RECORD
                 low, high: boundType;
               END;
PRIVATE;

EXPORT OP;
FUNCTION iv1make (l, h:  boundType):  iv1Type;
INTENT1 to make an iv1;

VAR i: iv1Type;
BEGIN
  i.low := l;
  i.high := h;
  iv1make := i;
END;
```

Externally defined constants, types, and operations are made available with the two statements

IMPORT <module_name > FROM <group_name>;

for importing constants and types only, and

[1] INTENT is an optional keyword in both modules/programs and routines to systematically state their purpose. In the implementation, INTENTs are converted to comments.

[2] AUTHOR is an additional, optional keyword to state the author and modifications.

IMPORT OP <module_name > FROM <group_name>;

for importing constants, types, and operations. <group_name> specifies the name of the corresponding group. Groups are structures for collecting several modules to a larger unit.

Type imports are implicitly propagated, that is, importing a type which is based on the definition of other, externally defined types implies that all required types are automatically provided. IMPORT OP accesses all the exported routines and their external types for the specific module. For example, the abstract data type for a two-dimensional interval relies on the one-dimensional intervals: the type of the 2-d interval is composed of two 1-d intervals, and the operations for the 2-d intervals are built by combining the 1-d operations. The code for the ADT iv2 looks as follows:

```
MODULE iv2;
INTENT the ADT for a 2-dimensional interval;
AUTHOR af, may 1985;

EXPORT TYPE;
IMPORT OP iv1 from intervals;
TYPE iv2Type = RECORD
                 low, high: iv1Type;
               END;
PRIVATE;

EXPORT OP;
FUNCTION iv2make (ll, lh, hl, hh: boundType): iv2Type;
INTENT to make an iv2;
VAR i: iv2Type;
BEGIN
  i.low := iv1Make (ll, lh);
  i.high := iv1Make (hl, hh);
  iv2make := i;
END;
```

Like Standard Pascal, first a type must be defined before it can be used in another definition. The same holds for the imports: the import of iv1 must precede the definition of the iv2type. Types and operations are exported together with the newly defined iv2Type, since this type depends on the type definition in iv1. Other imports which are

not related to the interfacing types would be written outside of the EXPORT TYPE; ...PRIVATE; bracket.

Finally, the two-dimensional ADT can be used in a main program. The code for this program looks as follows:

```
PROGRAM ivDrive;
INTENT to show the use of the adt iv2;
AUTHOR max, august 1987;

IMPORT OP iv2 FROM intervals;

VAR i: iv2Type;
    b1low, b1high, b2low, b2hogh: boundType;
BEGIN
   ..
  i := iv2Make (b1low, b1high, b2low, b2high)
   ..
END;
```

Note that only iv2 is imported, while not only the iv2Type, but also boundType—defined in iv1—is accessible; however, the operations of iv1 cannot be used since they are not explicitly imported.

4 Conditional Compilation

Conditional compilation is a method to maintain several versions of the same code in a single source file. It can be used to produce code which is transportable between diverse compilers in a similar manner to the one in [Sorens 1986]. The code which is specific for one compiler is only included for compilation if needed. Moreover, this feature can be used to include code which is exclusively used for testing as well.

The construct of the conditional compilation is the IFC directive. IFC closes with an ENDC and has the alternatives ELSEC and ELSEIFC.

```
IFC <condition>;
[ELSEC; | ELSEIFC <condition> ;]
ENDC;
```

For example, the command to open a sequential file varies among different compilers. With the conditional compilation statement the following code can be written such that it is transportable between IBM and VAX Pascal compilers:

```
MODULE sfile;
INTENT operations on sequential files;

PROCEDURE sfWOpen (fn:string; VAR fil:text);
INTENT to open a sequential file for writing;
VAR f:string;
BEGIN
IFC ibm;
   f := 'name = ' || fn;
   rewrite (fil, f);
ELSEIFC vax;
   open (fil, fn);
   rewrite (fil);
ELSEC;
   expand on other machines;[3]
ENDC;
END;
```

Conditions have the boolean typed values TRUEC or FALSEC, and they are by default FALSEC. Conditions can be set with the commands

TRUEC <condition>;

FALSEC <condition>;

Conditions can be only set in the head of a module or program. This restriction guarantees that each condition is consistently the same within a compilation unit. The specific value of the current compiler in the example above were set with the command

TRUEC ibm;

or

TRUEC vax;

[3]This 'comment' will produce a compilation error if neither IBM nor VAX are TRUEC and remind the programmer to adapt this specific part for the current compiler.

The precompiler always sets the value for the currently used complier to TRUEC, such that the programmer need not care about the initialization; however, other conditions depend on the programmer's initializations.

When incorporating dialects in large modular software systems, incompatible statements can be treated in two different ways:

- Statements which are used once or only a few times, such as the command for opening a file, are explicitly included into the code by the programmer as conditional compilation.
- Syntax features that occur all over, such as the alternative OTHER, OTHERWISE, OTHERS, etc. in the CASE statement, or the diverse treatment of the last END in a module or program, can be built into the precompiler once at its installation, avoiding the repetition of varying statements throughout the code.

This combination reduces the troublesome maintainance [Braunschober 1987] of portable systems to a minimum.

5 Implementation of the Precompiler

The precompiler was written in its own specific language and modular form. It consists of 12 modules (including a string handler), all together about 3000 lines of code. It was developed at the Swiss Federal Institute of Technology, Zuerich [Frank 1983] on a DECsystem-10 under TOPS-10, and transported to IBM 307 system under VM/CMS with PascalVS, and to VAX/MicroVAX under VMS with VAX Pascal at the University of Maine, Orono. The precompilation is split into two major parts:

- The I/O intensive completion of the imports. The precompiler checks uniqueness for multiply imported type names via different import paths.
- Preparing the exports for more efficient further imports. Brousing through the whole source code of each imported module for each compilation should be avoided; instead, a definition file is generated which contains all exported constants and type definitions, and the routine heads of the exported operations. This method speeds up precompilation significantly. The definition file is a text file and can be used for documentation, too.

6 Further Improvements

The introduced precompiler was a beneficial implementation of modular software techniques because it overcame some shortcomings of Pascal in a simple matter. Nevertheless, it still suffers from some restrictions. The following deficiencies and disadvantages were observed:

- Only bottom-up development is possible. At least the interfaces must be defined before higher-level modules can be implemented using definitions of lower parts.

- Changes in the interface of an abstract data type are not automatically propagated into the compiled code. It is up to the user to recompile all directly adjacent modules after modifications of the interface. In order to avoid dangerous errors when changing externally used routine interfaces, the following rule guarantees a secure propagation of such changes:

  Parameter names and types of exported routines must not be changed unless the routine is given a new name. The usage of a different name will prevent the programmer from using unadapted code, because at link time the old names will appear as undefined sysmbols.

- Uniqueness of names of the exported routines and types is not completely checked before linking. At an earlier state, uniqueness of names with respect to the imported operations and types is checked at compilation time; however, this is not always sufficient.

- Highly modular systems seduce the user to uncontrolled usage of external calls. This may lead toward undesired loops such as routines in module A calling routines in B and vice versa, which are hard to maintain and may fail to link.

- In order to guarantee portability, first the precompiler must be installed (and adapted to the compiler-specific features) before any software can be used.

- The current version is implemented without nested IFCs. This implementation is an obstacle when writing several versions with machine-specific differences.

- More types than eventually required are passed to the higher levels. The same holds for the imports of routines; however, their overhead is less

than the one of the types since only the routines of the addressed modules are made available. Nevertheless, the compilation units can grow tremendously by imports.

Some of these shortcomings cannot be solved without the support of a sophisticated code management system using database management techniques. Such a tool for computer aided software engineering provides on-line type checking and consistent propagation of changes. We have been designing such a system which is more complex, yet more powerful than this simple Pascal extension.

7 Conclusion

The non-redundant definition and management of exporting types and routines is a secure and reliable feature in a programming language for programming in the large. Only a few syntax extensions marking external usage of types and routines and stating the usage of externally defined types and/or routines were required to turn Pascal to a modular language. Different from other language extensions, these extensions did not introduce some redundancy by separating definitions from implementations; instead, a precompiler is used to derive the essential parts from a single source. By using the precompiler, all the advantages of Pascal are kept and, in addition, the ability of modular programming is gained. Due to the feature of conditional compilation, transportability among any Pascal compiler is guaranteed.

A more efficient and better controlled environment requires a code management system on top of a code database.

References

[Braunschober 1987] W. Braunschober. COMPAS—Compatible Pascal. SIGPLAN Notices, 22(3), March 1987.

[Digital 1987] VAX Pascal User Manual. Digital Equipment Corporation, Maynard (MA), 1987.

[Frank 1982] A. Frank. PANDA—A Pascal Network Database System. In: G.W. Gorsline, editor, Proceedings of the Fifth Symposium on Small System, Colorado Springs (CO), 1982.

[Frank 1983] A. Frank. A Precompiler for Transportable Modular Pascal. 1983. internal documentation, University of Maine at Orono, Department of Civil Engineering, Surveying Engineering, Orono (ME).

[Frank 1984] A. Frank. Extending a Database with Prolog. In: L. Kerschberg, editor, Proceedings of the First International Workshop on Expert Database Systems, Kiawah Island (SC), October 1984.

[Goguen 1978] J. A. Goguen et al. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In: R. Yeh, editor, Current Trends in Programming Methodology, Prentice-Hall, Englewood Clifts (NJ), 1978.

[Guttag 1977] J. Guttag. Abstract Data Types And The Development Of Data Structures. Communications of the ACM, June 1977.

[IBM 1981] Pascal/VS, Language Reference Manual. IBM, second edition, April 1987.

[Joslin 1986] D. A. Joslin. Extended Pascal—Illustrative Examples. SIGPLAN Notices, 21(12), December 1986.

[Jensen 1978] K. Jensen and N. Wirth. Pascal User Manual and Report. Springer-Verlag, New York (NY), 1978.

[Nani 1987] G. Nani. Implementing Separate Compilations in Pascal. SIGPLAN Notices, 22(8), August 1987.

[Sorens 1986] M. Sorens. A Technique for Automatically Porting Dialects of Pascal to Each Other. SIGPLAN Notices, 21(1), January 1986.

[Think 1986] Lightspeed Pascal, User's Guide and Reference Manual. Think Technologies, Inc., Lexington (MA), first edition, August 1986.

[Wirth 1982] N. Wirth. Programming in Modula-2. Springer-Verlag, New York (NY), 1982.