INTERNATIONAL STANDARD

ISO/IEC 10514-2

First edition
1998-12-01

Information technology — Programming languages —

Part 2:
Generics in Modula-2

Technologies de l'information — Langages de programmation —

Partie 2: Éléments génériques en modula 2

Reference number
ISO/IEC 10514-2:1998(E)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Contents

Page

Foreword iii

Introduction iv

1 Scope 1

2 Normative References 1

3 Definitions, Structure and Conventions 2

4 Requirements for Implementations 3

5 The Lexis 6

6 The Language 6

7 System Modules 27

8 Required Library Modules 27

9 Standard Library Modules 27

Annex A (Normative): Changes To the Syntax of the Base Language 28

Annex B (Normative): Collected Concrete Syntax 29

Annex C (Informative): Rationale 30

Annex D (Informative): Translations of Example Refinements to Standard Modula-2 33

Annex E (Informative): File Names 43

Annex F (Informative): Participating Individuals 44

Bibliography 45

© ISO/IEC 1998

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and micro-film, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 10514-2 was prepared by Joint Technical Committee ISO/IEC JTC 1 *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces.*

ISO/IEC 10514 consists of the following parts, under the general title *Information technology — Programming languages*:

— *Part 1: Modula-2, Base Language*

— *Part 2: Generics in Modula-2*

— *Part 3: Object Oriented Modula-2*

Annexes A and B form an integral part of this part of ISO/IEC 10514. Annexes C, D, E, and F are for information only.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Introduction

This part of ISO/IEC 10514 specifies the form and meaning of programs written in ISO Standard Modula-2 with Generic extensions and by reference to that specification lays down requirements for implementations of ISO Standard Modula-2 with Generic extensions.

The reader is referred to International Standard ISO/IEC 10514-1 (herein referred to as "the Base Language") for introductory remarks on the programming language Modula-2.

This part of ISO/IEC 10514 defines ISO Standard Modula-2 with Generic extensions by additions to the Base Language without changing the meaning of any parts of the Base Language.

This part of ISO/IEC 10514 does not provide a formal specification of ISO Standard Modula-2 with Generic extensions, although it is the intention of WG13 to construct the appropriate VDM-SL descriptions for the syntax and semantics described herein when committee resources permit.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Information technology — Programming languages —

Part 2:
Generics in Modula-2

1 Scope

1.1 General

This part of ISO/IEC 10514 specifies extensions to allow generic programming facilities to be added to the base Modula-2 language defined in International Standard ISO/IEC 10514-1 without altering the meaning of valid programs allowed by the Base Language (except for the use of the new keyword introduced by this standard—see clause 5).

1.2 Specifications included in this part of ISO/IEC 10514

In addition to the specifications included in the Base Language this part of ISO/IEC 10514 provides specifications for:

— required symbols for programs written in ISO Standard Modula-2 with Generic extensions;
— the lexical structure and semantics of programs written in ISO Standard Modula-2 with Generic extensions;
— the syntax of programs written in ISO Standard Modula-2 with Generic extensions;
— violations of the rules for the use of the Generic extensions that a conforming implementation is required to detect;
— further compliance requirements for implementations, including documentation requirements.

1.3 Relationship to ISO/IEC 10514-1

This part of ISO/IEC 10514 is part two of the multi-part Standard ISO/IEC 10514. This part of ISO/IEC 10514 extends and modifies the Base Language ISO/IEC 10514-1, but the adoption of this part of ISO/IEC 10514 is optional with respect to the Base Language. This part of ISO/IEC 10514 is also independent of any other parts of ISO/IEC 10514, except for part 1, and can be adopted either together with or independently of such other parts.

1.4 Specifications not within the scope of this part of ISO/IEC 10514

In addition to the categories of specifications excluded by the Base Language this part of ISO/IEC 10514 provides no specifications for:

— the method by which specific refinements are constructed from generic library modules;
— the method by which generic library modules, their associated refining modules, and the refinements produced by these are stored (including any correspondence between the module names and system file names where files are used).

2 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 10514. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO/IEC 10514 are encouraged to investigate the possibility of applying

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 10514-1:1996, *Information technology — Programming languages — Part 1: Modula-2, Base Language.*

3 Definitions, Structure and Conventions

3.1 Definitions

For the purposes of this part of ISO/IEC 10514, the definitions given in ISO/IEC 10514-1 and the following definitions apply.

3.1.1 Generic separate module

A new kind of separate module having formal parameters that can be either type parameters and/or constant value parameters. A generic separate module serves as a template for constructing specific refinements of itself that have been customized using actual types and/or actual constant expressions.

NOTE 1 — A generic separate module consists of a generic definition module and a generic implementation module. The generic definition module is a template from which a definition module can be refined. The generic implementation module is a template from which an implementation module can be refined.

3.1.2 Refiner or Refining module

A new kind of module that is a means of supplying actual parameters for the purpose of creating a refinement of the generic separate module from which the refinement is being made.

NOTE 2 — A refining module can be a separate module, in which case its definition module produces a refinement of the Generic Modula-2 definition module of the generic separate module from which the refinement is being made. The implementation module of such a refining separate module refines the implementation module of the generic separate module from which the refinement is being made.

NOTE 3 — A refining module can be a local module, in which case its refinement has as its qualified export list the items defined in the generic definition module of which it is a refining module. Such a refining local module refines the implementation module of the generic separate module from which the refinement is being made.

3.1.3 Refinement

An abstract entity (either a whole module or an item it contains) constructed from a generic separate module by specifying in a refining module the name of a generic separate module to be refined from and actual types and/or actual constant expressions to evaluate and then substitute for the formal parameters of the generic separate module.

NOTE 4 — The refining module is not the same entity as the refinement produced from it.

3.1.4 Refine

The act of constructing a refinement from a generic separate module by using a refining module.

NOTE 5 — Abstractly, the refining of a generic separate module to a specific refinement is the task of the translator. However, nothing in this part of the multi-part standard precludes an implementation separating this particular translation task from others. For instance, refinement could be performed separately before other translation tasks.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTE 6 — The effect of refinement is the same as if a refinement of the generic separate module had been written directly in the base language with the formal parameters replaced by the results of evaluating the actual parameters. Depending on the implementation strategy, this might not be quite the same as saying that refinement results in a program in the base language (except in the abstract sense), for there is no requirement in this part of ISO/IEC 10514 for a specific intermediate form in which a refinement finds some expression as a file.

TERMINOLOGY NOTE —WG13 is already using the term "instantiate" for use in Object Oriented Modula-2, and has chosen to use "refine" in ISO Standard Modula-2 with Generic extensions.

3.1.5 Generic

A property of both the whole and of any item defined within a generic separate module, regardless of whether the item itself contains or needs module parameters.

3.2 Structure of the Formal Definition

This part of the multi-part International Standard states its requirements in the same form as the Base Language with the exception that it does not include formal expression of semantics in VDM-SL at this time.

3.3 Conventions

The conventions used in this part of ISO/IEC 10514 are to be interpreted in the same way as in the Base Language with the exception that this part of ISO/IEC 10514 does not include VDM-SL at this time.

4 Requirements for Implementations

4.1 General Requirements

A conforming implementation of ISO Standard Modula-2 with Generic extensions meets the requirements for Modula-2 implementations that are laid down in the Base Language. In addition, it meets the requirements of this clause:

4.2 Translation

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall accept compilation modules for translation from source code when they contain the additional lexical form defined in clause 5, and when this is in the syntactic form specified in clause 6. It shall also accept the lexical forms defined in the Base Language when they are used in the (new) syntactic forms specified in ISO Standard Modula-2 with Generic extensions.

NOTE — The effect of accepting for translation the new compilation modules is discussed in the appropriate subclauses of clause 6 of this part of the multi-part standard.

4.3 Source Code Representation

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall provide the additional keyword specified in clause 5 and shall recognize keywords and identifiers as specified in that clause, including those situations where keywords and symbols from the base language are used in new syntactic constructs in this part of ISO/IEC 10514.

4.4 Ordering of Declarations

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall have rules identical in this respect to a conforming Modula-2 implementation.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.5 Predefined Entities

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall have rules identical in this respect to a conforming Modula-2 implementation.

4.6 Library Modules

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall have rules identical in this respect to a conforming Modula-2 implementation. No new library modules are specified by this part of ISO/IEC 10514.

NOTE — This does not preclude the possibility that later editions of or amendments to this part of the multi-part standard might include such modules.

4.7 Errors

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall have rules identical in this respect to a conforming Modula-2 implementation, with the following changes:

— It shall detect any new errors defined in this part of ISO/IEC 10514 in a manner consistent with the detection and reporting of errors required by the Base Language.
— In standard mode a conforming implementation of ISO Standard Modula-2 with Generic extensions shall treat the use of extensions that are not specified by this part of ISO/IEC 10514 or by another standard extension of the Base Language as errors. Conformance to standards parallel to this one (if any) is on an additive basis, so that two or more such standard extensions can be conformed to simultaneously.

NOTE — The intent of this provision is to allow a version of Modula-2 to support, for example, both genericity and object orientation (see ISO 10514-3).

4.8 Exceptions

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall have rules identical in this respect to a conforming Modula-2 implementation. The rules shall be applied after the construction of any refinements.

NOTE 1— This means, for instance, that two refinements of a single generic separate module constitute two different sources for exceptions.

Because the process of refinement logically takes place before the translation of the refinement, the only new errors defined in this part of ISO/IEC 10514 are syntactical and shall be detected by the translator. No new exceptions are therefore defined.

NOTE 2 — This does not preclude the possibility that later editions of or amendments to this part of the multi-part standard might include library modules that define exceptions. Nor does it prohibit implementations from providing library modules with their own exceptions.

4.9 Implementation-dependencies

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall have rules identical in this respect to a conforming Modula-2 implementation.

4.10 Documentation

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall have rules identical in this respect to a conforming Modula-2 implementation.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.11 Statement of Compliance

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall have rules identical in this respect to a conforming Modula-2 implementation and in addition, a separate compliance statement shall be made citing the degree of compliance with this part of ISO/IEC 10514.

4.12 Minimum requirements

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall have rules identical in this respect to a conforming Modula-2 implementation.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5 The Lexis

5.1 Relationship to the Base Language

The lexis of ISO Standard Modula-2 with Generic extensions is based directly on the lexis of the base language as defined in ISO/IEC 10514-1. All syntax elements of the base language keep their representation and all elements of the base language retain their same semantics; a new keyword that is needed to express the new language elements is defined in this clause.

NOTE - A conforming program written in the base language (or in the base language augmented by some other standard extension parallel to this part of the multi-part standard) is translated correctly by a translator conforming to ISO Standard Modula-2 with Generic extensions except in the case that the keyword defined in this part of the multi-part standard is used as an identifier in the program.

5.2 Keyword

Only one new keyword is defined; it is used to denote generic separate modules.

Concrete Syntax

keyword = base language keyword | "GENERIC" ;

NOTE — ISO Standard Modula-2 with Generic extensions does not define any new pervasive identifiers or any other new lexis elements beyond those already present in the base language.

6 The Language

6.1 The Model

ISO Standard Modula-2 with Generic extensions is an extension of the base language to provide facilities for generic programming. The provided model is characterized as follows:
— Genericity is defined at the separate module level only. (All the contents of a generic separate module are regarded as generic, but no item within a module can be declared as generic independently of the module containing it).
— Generic separate modules have formal parameters that are either type parameters or constant value parameters.
— Refining separate modules have actual parameters that are compatible with the formal parameters of the generic separate module from which the refinement is being made.
— Refining local modules have actual parameters that are compatible with the formal parameters of the generic separate module from which the refinement is being made.
— Formal/actual module parameter correspondence semantics are identical to those employed in the base language for constant value parameters of procedures. (There are no variable parameters allowed in generic separate module parameter lists; neither can variables be used to pass values to constant value parameters in generic separate module parameter lists).
— Refinement takes place not later than translation time.

ISO Standard Modula-2 with Generic extensions incorporates all the syntax and semantics of the base language without alteration. This clause indicates only the additions (new syntax and semantics) required to support generic programming.

6.2 Programs, Program Modules, and Separate Modules

6.2.1 The New Modules

Generic separate modules are extensions of the separate modules defined in the Base Language. Each has a generic definition module that shall exist in order to check the well-formedness of any other module that depends on it. A generic separate module also has a generic implementation module. Another module does not import items from a generic separate module; rather, it imports them from a refinement of a generic separate module.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Refining separate modules are variations of the separate modules defined in the Base Language. Each has a refining definition module that shall exist in order to check the well-formedness of any other module that depends on it. A refining separate module also has a refining implementation module. Each refining separate module depends upon some generic separate module, and this imposes the constraint that both the refining separate module and the generic separate module from which it refines shall exist before the well-formedness of the refinement produced by a refining separate module can be checked.

The fact that the translation of a program depends upon the prior translation of the modules it imports means that the refinement shall exist in order to be translated in the correct order. Thus, the process of refinement logically precedes that of the establishment of program compilation dependencies, which in turn logically precedes that of the translation of the program. However, the physical means by which this is actually achieved is implementation defined. Thus, implementations are free to separate the step of refinement from that of translation, or to combine the two as they see fit. This also means that, although a refinement of a generic separate module can be thought of in a logical sense as itself being a module (separate or local), there is no need to define its syntax or semantics as such, for these derive from the generic separate module and the refiner creating the refinement from it and in turn are correct (or not) according to the rules of the Base Language.

NOTE — A consequence of this is that in systems that employ files, a refinement might find physical expression as a file independent of the file(s) containing the refining module. However, this is not required, and if the implementation strategy employed is to have the translator construct the refinement as part of the translation process, no such independent files need be created.

6.2.2 Programs and Compilation Modules

Four new kinds of compilation modules are added to the list provided in the Base Language. This produces the following change:

Concrete Syntax

compilation module = program module | definition module | implementation module | generic definition module | generic implementation module | refining definition module | refining implementation module ;

6.2.3 Generic Definition Module

A generic definition module is an extension of the definition module in the Base Language, and the rules for definition modules given in the Base Language apply to generic definition modules, with the following additions.

Concrete Syntax

generic definition module =
"GENERIC", "DEFINITION", "MODULE", module identifier, [formal module parameters], semicolon,
import lists, definitions,
"END", module identifier, period ;

Semantics

The effect of processing a generic definition module by a translator is implementation defined, but
— Any identifiers it imports shall be distinct from each another and from any identifiers it defines.
— Any identifiers it defines shall be distinct from each other and from any identifiers it imports.
— Its own name is added to the environment. This allows other modules to import it.
— The identifiers it defines are NOT added to the environment. (When the module is later refined, the refinements of those identifiers will be added to the environment instead of the identifiers defined by the generic separate module). This means that the translator shall report as an error any attempt to perform unqualified import of an identifier defined in a generic definition module or to employ such an identifier when qualified by the name of a generic definition module.
— Apart from its own name being added to the environment, no translation of a generic definition module takes place until a refining definition module is translated, and at that time the effect of the translation depends on the actual parameters supplied by the refining module.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTE 1— The effect of this part of the multi-part standard is to require that refinements be checked in the same way as any base language module. These rules do not either require or preclude an implementation having the translator perform such consistency checks as might be possible on the definition module of a generic separate module when presenting it to the translator.

NOTE 2— The entities defined in a generic separate module are not available for use by other modules until they have been refined by the translation of a refining module that refines from the generic separate module in question. (They can of course be used within the generic separate module itself). It is, therefore, an error that the translator shall report, to attempt to employ in another body items from a generic separate module rather than from a refinement of one. This does not of course preclude any module (including a generic one) from importing a generic separate module for the purpose of refining it.

Examples

Conventional data structures such as lists, queues, and stacks can be constructed generically, and parameterized to provide them with sufficient information about the data expected to be entered into the structure.

NOTE 3 — Several of these examples are dealt with further in the clauses on generic implementations and refining definitions and implementations.

Example 1: The first example is a generic definition of a data structure. In such applications, it is expected that the structure will be manipulated independent of the kind of element it contains. All the work of programming is in the structure manipulation; the provision of the items to enter into it is a refining detail.

```
GENERIC DEFINITION MODULE Stacks (Element: TYPE);

CONST
  StackSize = 100;

PROCEDURE Push (item : Element);

PROCEDURE Pop (VAR item : Element);

PROCEDURE Empty () : BOOLEAN;

END Stacks.
```

NOTE 4— Specification of a constant such as *StackSize* in the manner of this example constrains all the refinements of this module. If that were not the intention, such an item could instead be made a module parameter and then specified by the refinements.

NOTE 5 — Such a constant can be used in the corresponding generic implementation module. However, depending on the implementation strategy chosen, the correctness of such use might not be checked until the refinement of that implementation is performed.

Example 2: Sometimes the data structure itself needs parameterization.

```
GENERIC DEFINITION MODULE Matrix (Rows, Cols : CARDINAL; MatrixElement : TYPE);

TYPE
  TMatrix = ARRAY [0 .. Rows-1]
       OF ARRAY [0 .. Cols-1] OF MatrixElement;

PROCEDURE Invert (VAR m : TMatrix);

END Matrix.
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Example 3: This illustration shows a generic technique, as opposed to a generic Abstract Data Type (ADT).

```
GENERIC DEFINITION MODULE Validate (PType : TYPE; PValidProc : ValidProcType);

TYPE
  ValidProcType = PROCEDURE (item : PType) : BOOLEAN;
   (* Note the forward reference in the module parameter list *)

PROCEDURE Valid (item : PType) : BOOLEAN;

END Validate.
```

Example 4: (An outline of a generic sort) This example illustrates the abstraction of a generic technique applied to an existing kind of structure (an array), rather than to a user-defined structure.

```
GENERIC DEFINITION MODULE Sorts (Item : TYPE; GenCompare : CompareProc);

FROM Comparisons IMPORT
  CompareResults;

TYPE
  CompareProc = PROCEDURE (Item, Item) : CompareResults;

PROCEDURE Quick (VAR data : ARRAY OF Item);

  (* Other procedures and functions could be included as well. *)
END Sorts.
```

NOTE 6 — The type identifiers of formal parameters can forward reference types defined in the definition module of the generic separate module itself.

Example 5: (Parameters are optional) This example illustrates that although in the majority of cases generic separate modules will be parameterized, this is not required, and non-parameterized instances might also be useful. This module defines a counter, any number of instances of which can be refined under different names.

```
GENERIC DEFINITION MODULE Counter;

PROCEDURE Inc;

PROCEDURE Reset;

PROCEDURE Count () : CARDINAL;

END Counter.
```

6.2.4 Generic Implementation Module

A generic implementation module is an extension of the implementation module in the sense of the Base Language, and the rules for implementation modules (and their relationships with their definition modules) given in the Base Language apply, with the following additions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Concrete Syntax

generic implementation module =
"GENERIC", "IMPLEMENTATION", "MODULE", module identifier, [interrupt protection], [formal module parameters], semicolon,
import lists, module block,
module identifier, period ;

Semantics

The effect of translating the implementation module of a generic separate module by a translator is implementation defined, but:
— The definition module of the generic separate module shall already exist and have both the same name and the same parameters.
— Any identifiers that the implementation module of a generic separate module imports shall be distinct from each other, from any identifiers it defines, and from any identifiers defined in its corresponding generic definition module.
— Any identifiers that the implementation module of a generic separate module defines shall be distinct from each other, from any identifiers it imports, and from any identifiers defined in its corresponding generic definition module.
— Other than possibly checking such consistencies, the translation of a generic implementation module cannot be performed until it is refined by a refining module, and at that time the effect of the translation depends on the actual parameters supplied by the refining module.
— When a refinement of a generic implementation module is translated, the effect is the same as translating a copy of the generic implementation module with the results of evaluating the actual parameters supplied in the refinement substituted for the formal parameters in the generic separate module, and at that time, only the rules of the base language need be applied to perform the translation.

NOTE 1 — The effect of this part of the multi-part standard is to require that refinements be checked in the same way as any base language module. These rules neither require nor preclude an implementation having the translator perform such consistency checks as might be possible on the implementation module of a generic separate module by presenting it to the translator separately.

The parameters of the generic implementation module shall be identical to the parameters of the corresponding generic definition module.

NOTE 2 — The interrupt protection on the generic implementation module (when provided) applies to all refinements of this module, whether separate or local, and such refinements can not have a protection expression of their own.

If a programmer employs code in the generic implementation that assumes something about a type that is passed to it via one of the formal parameters, and then a refinement is done passing an actual type inconsistent with this assumption, the error shall be detected upon attempting to translate the refinement of the implementation (i.e. after the refinement has been constructed by the refiner). For instance, suppose a generic implementation used the operator "<" rather than having a compare procedure passed to it as a parameter (as done in example 4 in 6.2.3 and 6.2.4. The genericity is then restricted to data types that could use "<" and if some other data type were employed, the translator is required by the base language standard to report the error when attempting to translate the refinement of the implementation module. Without imposing complicated syntax on the form of generic separate modules to specifically delimit or control the use of built-in operations and procedures, there is no solution to this difficulty other than ensuring that the code for generic separate implementation modules is itself written in as "generic" a fashion as will permit the envisioned refinements of it to be correct.

NOTE 3 — A consequence of such application of the rules of the Base Language to the translation of refinements (i.e. after the refinement is done) is that a generic separate module that is syntactically correct and has been refined by a syntactically correct refining module can still result in a refined module that is not correct.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Examples

Example 1: What follows is a possible generic implementation of the first example in 6.2.3:

```
GENERIC IMPLEMENTATION MODULE Stacks (Element : TYPE);

VAR
  stack    : ARRAY [0..StackSize] OF Element;
  stackPtr : CARDINAL;
(* One could also arrange for StackSize to be a parameter. *)

PROCEDURE Push (item : Element);
BEGIN
 stack[stackPtr] := item;
  INC (stackPtr);
END Push;

PROCEDURE Pop (VAR item : Element);
BEGIN
  DEC (stackPtr);
  item := stack[stackPtr];
END Pop;

PROCEDURE Empty () : BOOLEAN;
BEGIN
  RETURN stackPtr = 0
END Empty;

BEGIN (* module body initialization *)
  stackPtr := 0
END Stacks.
```

NOTE 4 — The constant *StackSize* is from the corresponding definition module. It can be used in this generic implementation module. However, depending on the implementation strategy chosen, the correctness of such use might not be checked until the refinement of that implementation is performed.

Example 2: The next module is a corresponding implementation for the generic definition of example 2 in 6.2.3.

```
GENERIC IMPLEMENTATION MODULE Matrix (Rows, Cols : CARDINAL; MatrixElement : TYPE);

(* Note that, as in any implementation module, the generic implementation module has access to the <generic> types defined
in the corresponding definition. *)

PROCEDURE Invert (VAR m : TMatrix);
BEGIN
  (* your favourite technique *)
END Invert;

END Matrix.
```

Example 3: The next module is a corresponding implementation for the generic definition of example 3 in 6.2.3.

```
GENERIC IMPLEMENTATION MODULE Validate (PType : TYPE; PValidProc : ValidProcType);

PROCEDURE Valid (item : PType) : BOOLEAN;
BEGIN
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
  RETURN PValidProc (item)
END Valid;

END Validate.
```

Example 4: Here is a sketch of a generic implementation of the *Sorts* module defined in example 4 of 6.2.3.

```
GENERIC IMPLEMENTATION MODULE Sorts (Item : TYPE; GenCompare : CompareProc);

FROM Comparisons IMPORT
  CompareResults;

PROCEDURE Swap (VAR a, b : Item);
VAR
  temp : Item;

BEGIN
  temp := a;
  a := b;
  b := temp;
END Swap;

PROCEDURE Quick (VAR data : ARRAY OF Item);
BEGIN
(* typical quicksort algorithm, except that compares are done by a procedure Compare <actual one supplied as parameter>
which returns values of type Comparisons.CompareResults. Swaps are done using the refinement of the generic swap above.
*)

END Quick;

END Sorts.
```

Example 5: (Parameters are optional) This example implements the non-parameterized counter found in 6.2.3.

```
GENERIC IMPLEMENTATION MODULE Counter;
VAR
  CurrentCount : CARDINAL;

PROCEDURE Inc;
BEGIN
  INC (CurrentCount);
END Inc;

PROCEDURE Reset;
BEGIN
  CurrentCount := 0;
END Reset;

PROCEDURE Count () : CARDINAL;
BEGIN
  RETURN CurrentCount
END Count;

BEGIN (* main *)
  Reset;
END Counter.
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.2.5 Refining Definition Module

A refining definition module is similar to a definition module in the sense of the Base Language, and the rules for definition modules given in the Base Language apply, with the following additions or changes.

The imports of a refining definition module are the imports of the definition module of the generic separate module of which it is a refiner together with the results of evaluating the actual module parameters (i.e. these values can be treated as imports).

The exports of a refining definition module are the refinements of the items defined in the definition module of the generic separate module of which it is a refiner.

Concrete Syntax

refining definition module =
"DEFINITION", "MODULE", module identifier, equals, generic separate module identifier, [actual module parameters], semicolon,
"END", module identifier, period ;

generic separate module identifier = identifier

A refining definition module cannot have additional imports or declarations of its own in addition to the ones in the generic separate module from which it refines.

Semantics

The effect of translating a refining definition module shall be the same as constructing and then translating a definition module (in the sense of the Base Language) obtained from the definition module of the generic separate module that it refines from but with the results of evaluating the actual parameters of the refining definition module substituted for the formal parameters of the definition module of the generic separate module that is being refined. The resulting refinement that is translated is a definition module in the sense of the base language, and the translation that is done following refinement employs only the rules of the base language.

NOTE 1 — One consequence of this is that if a generic separate module exports a type, and this module is then refined more than one time so as to produce several types, then any one of these is incompatible with all the others, even though they all were refined from the same generic separate module.

The parameters of the refining definition module shall match the parameters of the generic definition module of which it is a refiner. If the definition module of the generic separate module has no parameters, the refining definition module shall also have none, not even an empty parameter list.

NOTE 2 — It is not the intention of this part of the multi-part standard to specify the exact means by which the substitutions are carried out. That is, implementations are free to separate this step from the other functions of the translator and carry it out prior to other translation tasks (perhaps creating a new physical file expressing the refinement as an ordinary definition module). Implementations are also free to have the translator carry out the refinement in some other manner, so long as the effect is as described. That is, the actual refinement might or might not have a separate physical existence from the refiner (perhaps as a file in its own right), and whether it does or not is an implementation defined issue.

This part of the multi-part standard permits a client program module to be translated once any refining definition modules (and therefore also all the definition modules needed to do the refinement) have been translated. Although the definition modules of the generic separate modules from which the refinements are being made shall also exist, the corresponding implementation modules of the generic separate modules and refining separate modules need not exist at this point.

Examples

For each example, the outcome of one possible preprocessor refinement strategy is shown in informative Annex D.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Example 1: What follows is a possible refiner of the first example in 6.2.3:

DEFINITION MODULE CardStack = Stacks (**CARDINAL**);
END CardStack.

Example 2: In order to refine the module Sorts as a separate module, one must first supply the necessary Compare procedure in a separate module. For instance to refine an integer sort, first supply:

DEFINITION MODULE IntegerInfo;

FROM Comparisons **IMPORT**
CompareResults;

PROCEDURE Compare (a, b : INTEGER) : CompareResults;

END IntegerInfo.

When this is done, refining is by a module such as:

DEFINITION MODULE IntSorts = Sorts (**INTEGER**, IntegerInfo.Compare);
END IntSorts.

NOTE 3 — Once the separate definition module IntegerInfo has been processed, accessibility to such items as IntegerInfo.Compare in the parameter list of a refining definition module follows the rules for accessibility to imported identifiers in any definition module.

The generic implementation module must also be supplied and refined.

NOTE 4 — The refined type
IntSorts.CompareProc = PROCEDURE (INTEGER, INTEGER) : CompareResults
is now exported by the refined module. This might not be necessary, but is consistent with normal language rules. There seems to be no compelling reason to make this export "disappear" in the refining process.

Example 3: Any number of specific instances of the generic counter can be refined, though this might not be a common use of the facility:

One refiner that illustrates the syntax is:

DEFINITION MODULE ACount = Counter;
END ACount.

NOTE 5 — See also example 4 in 6.3 for multiple local refinements.

Example 4: The generic matrix is refined with constants, as in the following:

DEFINITION MODULE RealMatrix45 = Matrix (4, 5, REAL);
END RealMatrix45.

6.2.6 Refining Implementation Module

A refining implementation module is similar to an implementation module in the sense of the Base Language, and the rules for implementation modules (and their relationships with the corresponding definition modules) given in the Base Language apply, with the following additions and changes.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The imports of a refining implementation module are the imports of the implementation module of the generic separate module of which it is a refiner together with the actual parameters of the refining implementation module (i.e. the values can be treated as imports).

Concrete Syntax

refining implementation module =
"IMPLEMENTATION", "MODULE", module identifier, equals, generic separate module identifier, [actual module parameters], semicolon,
"END", module identifier, period ;

generic separate module identifier = identifier ;

Semantics

The effect of translating a refining implementation module shall be the same as constructing and then translating an implementation module (in the sense of the Base Language) obtained from the implementation module of the generic separate module that it refines from, but with the results of evaluating the actual parameters of the refining module substituted for the formal parameters of the generic separate module that is being refined. The resulting refinement that is translated is an implementation module in the sense of the base language, and the translation that is done following refinement employs only the rules of the base language.

The parameters of the refining implementation module shall be the same as the parameters of the corresponding refining definition module.

The parameters of the refining implementation module shall match the parameters of the implementation module of the generic separate module of which it is a refinement.

NOTE 1 — It is not the intention of this part of the multi-part standard to specify the exact means by which the substitutions are carried out. That is, implementations are free to separate this step from the other functions of the translator and carry it out prior to other translation tasks (perhaps creating a new physical file expressing the refinement as an ordinary implementation module). They are also free to have the translator carry out the refinement in some other manner, so long as the effect is as described. That is, the actual refinement might or might not have a separate physical existence from the refiner (perhaps as a file in its own right), and whether it does or not is an implementation defined issue.

A refining implementation module cannot have a protection expression, additional imports, or declarations of its own in addition to the ones it refines from its generic counterpart. It can not contain a refining local module. Neither can it have a body of its own (that is, in addition to any body it might refine from the implementation module of the generic separate module of which it is a refiner).

NOTE 2 — Since the refinement of a generic implementation module depends on the contents of the implementation module of the generic separate module at the time the refinement is performed, any change to the implementation module of the generic separate module will only be reflected in refinements if they are re-refined. That is, changes to the generic implementation must be intentionally propagated to the refinements in the same manner as changes to definition modules must be propagated to all modules dependent on them.

Examples

Example 1: What follows is a possible refining module of the first example in 6.2.4

IMPLEMENTATION MODULE CardStack = Stacks (**CARDINAL**);
END CardStack.

Example 2: To complete the refiner of the generic sorting module in 6.2.3, one still needs:

IMPLEMENTATION MODULE IntSorts = Sorts (**INTEGER**, IntegerInfo.Compare);
END IntSorts.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A client has:

```
FROM IntSorts IMPORT
  Quick;
```

or, if more than one refinement to separate modules is done:

```
IMPORT IntSorts, RealSorts, RecSorts;
```

Example 3: Here is a refining implementation part of the counter example in 6.2.4:

```
IMPLEMENTATION MODULE ACount = Counter;
END ACount.
```

Example 4: To illustrate the necessity of the parameters for both parts of the module being the same, here is the refining implementation for example 2 in clause 6.2.4:

```
IMPLEMENTATION MODULE RealMatrix45 = Matrix (4, 5, REAL);
END RealMatrix45.
```

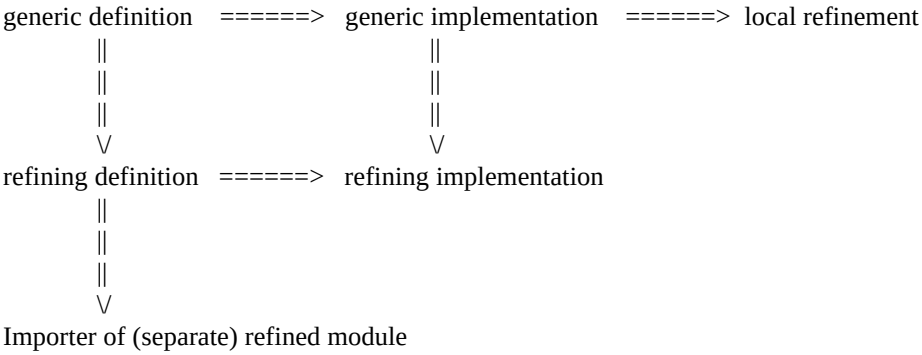
6.2.7 Module Compilation Order

The module compilation order in ISO Standard Modula-2 with Generic extensions is determined by applying the rules of the Base Language after the refinements of any generic separate modules have been completed.

6.2.8 Module Dependencies

The module dependencies of ISO Standard Modula-2 with Generic extensions are the same as those in the Base Language, with the following additions:
— The implementation module of a generic separate module shall be translated after the corresponding definition module of that same generic separate module.
— The definition module of a refining separate module shall be translated after the definition module of the generic separate module of which it is a refiner.
— The implementation module of a refining separate module shall be translated after the corresponding definition module of that same refining separate module.
— The implementation module of a refining separate module shall be translated after the implementation module of the generic separate module of which it is a refiner.
— A module containing one or more refining local modules (see 6.4) shall be translated after all the implementation modules of the generic separate modules for which it contains refining local modules.
— A module importing one or more refining separate modules shall be translated after all the definition modules of the refining separate modules that it imports.

Chart of dependencies:



Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.2.9 Nested Module Refinement Order

If an implementation module of a generic separate module contains one or more refining local modules, then the order of refinement is the same as the order of initialization would be once the modules were fully refined. This means that the outer module is refined first, then the local modules in the textual order in which they appear, and then any local modules contained in the generic separate modules from which they are being refined, and so on.

This process is not recursive, whether directly or indirectly. If the implementation part of a generic separate module contains a refining local module, that refining local module cannot refine from the same generic separate module in which it is contained. Neither can two generic separate modules each contain a refiner to the other, whether directly or in some longer circular chain.

6.2.10 Module Initialization Order

The module initialization order in ISO Standard Modula-2 with Generic extensions is determined by applying the rules of the Base Language after the refinements of any generic separate modules have been completed.

NOTE — This means that if a program uses two different refinements of the same generic separate module, these are regarded as different modules in the sense of the Base Language. Each has its own initialization, and the two initializations are done in an order determined by the import lists of the program.

6.2.11 Module Termination

The module termination order in ISO Standard Modula-2 with Generic extensions is determined by applying the rules of the Base Language after the refinements of any generic separate modules have been completed.

NOTE — This means that if a program uses two different refinements of the same generic separate module, the two modules are terminated in an order determined by their initialization.

6.2.12 Import Lists

The rules for import lists in a module in ISO Standard Modula-2 with Generic extensions are identical to those of the Base Language and are applied after the refinement of any generic separate modules has taken place. In addition, the rules for imports into generic separate modules are the same as the rules for any other separate modules.

NOTE — Because the names of formal parameters are in the scope of a generic separate module, and they are treated as imports, they can not be the same as any identifiers named in an import list of that module.

6.2.13 Export Lists

A refined separate module has as its qualified export list the items defined in the generic definition module of which its refining separate module is a refiner.

A refined local module (see 6.4) has as its qualified export list the items defined in the generic definition module of which its refining local module is a refiner.

6.3 Definitions and Declarations

6.3.1 Relationship to the Base Language

The rules for definitions and declarations in ISO Standard Modula-2 with Generic extensions are identical to those in the Base Language with the additions noted below.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.3.2 Formal Module Parameters

A generic separate module declaration shall optionally include formal module parameters, each with its formal type.

Concrete Syntax

formal module parameters = left parenthesis, formal module parameter list, right parenthesis ;

formal module parameter list = formal module parameter, {semicolon, formal module parameter} ;

Semantics

The identifiers declared as formal module parameters in the formal module parameter list of a generic separate module shall be distinct from each other, and distinct from the formal type identifiers.

6.3.3 Formal Module Parameter Consistency

The formal module parameter list of a generic implementation module shall be identical to the formal parameter list of the corresponding generic definition module. The identifiers used to denote the parameters shall be the same, and the identifiers used to denote the formal types of the parameters shall be the same. The number of formal parameters shall also be the same.

NOTE — This rule is to ensure that any changes to the dependencies employed by the implementation module of the generic separate module necessitate first making the same changes in the generic definition module, retaining the notion of the definition module of the generic separate module as a reliable contract with both dependent modules and human readers with respect to the resources that will be employed by any later refinements of the generic separate module.

6.3.4 Formal Module Parameters

6.3.4.1 The Model for Formal Module Parameters

As an optional part of the definition and implementation modules of a generic separate module, a formal module parameter provides an explicit interface between the body of the generic separate module and the body of any module refined from it. If the generic separate module has formal parameters, any refiner of it shall provide corresponding actual parameters. These are evaluated, and the resulting arguments are accessed in the refined module through the identifiers of the formal parameters.

There are two kinds of formal module parameters: constant value parameters and type parameters. The types of the former are specified by formal types; the latter are types, and this is specified by the keyword TYPE.

Concrete Syntax

formal module parameter = constant value parameter specification| type parameter specification ;

6.3.4.2 Constant Value Parameters

Constant value parameters provide a means of passing a constant to the refinement of a generic separate module. Corresponding actual parameters shall be constant expressions of a type compatible to the formal parameter type.

Constant value parameters are value parameters in the same sense as the value parameters used with procedures (as defined in the Base Language ISO/IEC 10514-1) except that the corresponding actual parameters can only be constant expressions compatible with the type of the formal parameter.

Concrete Syntax

constant value parameter specification = identifier list, colon, formal type;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTE 1 — The base language rules for translation are not applied until after the refinement has been performed. The effect of translating a refiner is to translate a separate module (in the sense of the base language) constructed from the generic separate module that is being refined with the results of evaluating the actual parameters supplied by the refiner substituted for the formal parameters of the generic separate module. Thus, when the refinement is being done, the types of the formal parameters of the generic separate module and any types defined therein are deemed to be available to the translator.

NOTE 2 — Implementations are free to allow the translator to perform consistency checks between generic separate module parameters and any forward referenced type definitions those generic separate modules can contain, but this does not constitute a constraint on the scope of those formal parameters per se. Since the presentation of a generic separate module to a translator need have no other result than adding its own name to the environment, all consistency checks can be postponed until the refinement is translated.

Thus, the type of a formal constant value parameter can be:
— any pervasive type,
— any type previously named as a formal module type parameter in the same module parameter list,
— any type imported into the generic separate module in its import list, or
— any type defined in the generic separate module itself.

In the latter two cases, the type shall be imported into or defined in (respectively) the generic definition module because the parameter lists of the definition and implementation modules are required to be the same.

NOTE 3 — A consequence of allowing types previously named in the same parameter list is that such parameter lists must be evaluated from left to right, a restriction that is not present in the base language for the parameter lists of procedures.

NOTE 4 — The internal types Z-type, R-type, C-type, and S-type cannot be used as they have no formal names in the language. This is a restriction, and could have been dispensed with by removing type names altogether from the parameter lists, but that would prevent any type checking prior to translation, and would be more likely to generate inconsistencies in the refined modules.

This rule applies only to generic definition and implementation modules. The translation of a refiner is to have the effect of translating a copy of the generic separate module (without any parameters) with the values of the actual parameters in the place of the formal parameter names. From a logical point of view, all that exists at the conclusion of this process is the translation of a module in the sense of the Base Language; therefore the new rule need only apply to generic separate modules, and is not retroactive to other situations.

NOTE 5 — Applications of the rule allowing types defined in generic separate modules to be used in the parameter list of the same module are in effect forward references similar to the forward referencing for pointers to records.

NOTE 6 — One consequence of this rule is that a constant value parameter can be of a type mentioned previously in the parameter list as a type parameter.

6.3.4.3 Type Parameters

Type parameters provide a means of passing a type identifier to the refinement of a generic separate module. Corresponding actual parameters shall be types.

Concrete Syntax

type parameter specification = identifier list, colon, "TYPE" ;

NOTE 1 — This is a new explicit syntactical use of a keyword already existing in the Base Language. Some pervasive procedures already have such a use implicit in their definitions. For instance, VAL is implicitly

PROCEDURE VAL (aType : TYPE, value : anyType);

It is not, however, the intention of this part of ISO/IEC 10514 to change or extend the use of "TYPE" to allow it in other contexts such as the declaration of formal procedure headings. The latter remain as they are in the base language.

NOTE 2 — "TYPE" is to be thought of as a keyword even in this situation where it is being used as an identifier of a class of entities, each of which is a type.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTE 3 — If the actual parameter passed to a formal type parameter is an opaque type, the refinement might not work unless it is written to take this possibility into account.

NOTE 4 — If the refinement assumes numeric operations, then the actual type shall be numeric or the translator will report this as an error when the refinement is being translated.

6.4 Refining Local Module Declarations

A refining local module is similar to a local module in the sense of the Base Language, and the rules for local modules given in the Base Language apply, with the following additions or changes.

Concrete Syntax

refining local module declaration =
"MODULE", module identifier, equals, generic separate module identifier, [actual module parameters], semicolon,
[export list],
"END", module identifier ;

generic separate module identifier = identifier

NOTE 1 — The identifier of the generic separate module named in the refining local module declaration needs to be visible in the scope of the refining local module.

Semantics

The effect of translating a module containing a refining local module shall be the same as constructing and then translating a module in which the refining local module is replaced by a local module (in the sense of the Base Language) constructed from the implementation module of the generic separate module that it refines from, but with the results of evaluating the actual parameters of the refining local module substituted for the formal parameters of the generic separate module that is being refined. The resulting refinement that is translated is a module containing a local module (in the sense of the base language) and the translation that is done following refinement employs only the rules of the base language.

A refining local module can be employed in any context in which the refined local module is permitted by the Base Language, including a program module, an implementation module of a separate library module, or a local module.

The parameters of the refining local module shall match the parameters of the generic separate module of which it is a refiner.

A refining local module is also subject to the appropriate rules for refining implementation modules:

A refining local module cannot have a protection expression, additional imports, or declarations of its own in addition to the ones it acquires from its generic counterpart. It can not contain a refining local module. Neither can it have a body of its own (that is, in addition to any body it might refine from the implementation module of the generic separate module of which it is a refiner). It can have an export list.

The imports of a refined local module are the merger of the imports of the definition and implementation parts of the generic separate module of which it is a refinement together with the actual module parameters (i.e. these values can be treated as imports).

The export list of a refined local module is the export list specified in the refining local module by which it is refined. The syntax and semantics are those of the Base Language.

Semantic notes:
The interpretation of the refinement of a generic separate module as a local module is that:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

a) The refinement is the merger of the refinements of the library definition and implementation module pair of the generic separate module (the name of which needs to be visible at the place of the refinement).
b) Such a merger only makes sense if the parameter lists of the definition and implementation parts of the generic separate module are the same, and this is one of the reasons why this rule was adopted.
c) The list of exports of the refined local module into its surrounding scope is specified in the refining local module. Because these exports can be qualified, two refinements (under different names) of the same generic separate module could both be made in one scope without causing a name clash.
d) The only permitted refinement of a generic separate module within another module is as a local module.
e) If the implementation part of a generic separate module contains a refining local module, that refining local module cannot refine from the same generic separate module in which it is contained—whether directly or indirectly. That is, circular refinement is not permitted. The translator is required to detect and report this error.
f) The translator shall be able to detect a new syntax error, for it is not possible to use (apart from refinement) items from a generic separate module directly in another module. For instance, if one were to import for the purpose of local refinement the module "Sorts", a use of Sorts.Quick is invalid.
g) The name "Sorts" does have to be imported into the scope of the refinement, however. This import provides only the name of the module, not the names of any items in it. The latter must be refined to be used.

Examples

Example 1: What follows is a sketch of a program client containing two local refinements of the first example in 6.2.3:

```
MODULE StackClient;
IMPORT Stacks; (* the generic name has to be imported *)

TYPE
  RecDef =
    RECORD
     c : CHAR;
     i : INTEGER;
    END (* record *);

MODULE CardStack = Stacks (CARDINAL);
EXPORT QUALIFIED StackSize, Push, Pop, Empty;
END CardStack;

MODULE RecStack = Stacks (RecDef);
EXPORT StackSize, Push, Pop, Empty;
END RecStack;

VAR
  c : CARDINAL;
  r : RecDef;

BEGIN (* main *)
  CardStack.Push (c);
  Push (r);
END StackClient.
```

The refinement of the local modules of this module shall export according to the export list in the refiner and shall consist of the merger of the corresponding definition and implementation parts of the generic separate modules.

One possible way of achieving the refinement is shown in clause D.3 as an illustration.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Example 2: Generic modules to implement structures and those to implement data manipulation can be defined as in the examples in 6.2.3 and then combined in a new generic definition module using local modules. First create the generic definition module.

```
GENERIC DEFINITION MODULE ValidStacks (PType : TYPE; PValidProc : ValidProcType);

TYPE
  ValidProcType = PROCEDURE (item : PType) : BOOLEAN;

PROCEDURE PushValid (item : PType);

END ValidStacks.
```

Then, combine the two by refining both within the implementation via local modules and then employing services from both so that, in this case, only valid items are pushed on the stack. In addition, note that the data items are to be defined in another module and are assumed to have ADT components:
(i) *PType* that can be mapped both to the *Element* required by *Stacks*
and the *Item* required by *Validate* and
(ii) *PValidProc* compatible with the *Validation* required by *Validate.*

NOTE 2 — This example could also have been solved in an object oriented manner, where those facilities are available. However, the generic approach allows the programmer to have a client program that imports only *ValidStacks*, without also having *Stacks* and *Validate* present.

```
GENERIC IMPLEMENTATION MODULE ValidStacks (PType : TYPE; PValidProc : ValidProcType);
IMPORT Stacks, Validate;

MODULE MyStacks = Stacks (PType);
EXPORT QUALIFIED StackSize, Push, Pop, Empty;
END MyStacks;

MODULE MyValidate = Validate (PType, PValidProc);
EXPORT QUALIFIED Valid;
END MyValidate;

PROCEDURE PushValid (item : PType);
BEGIN
  IF MyValidate.Valid(item) THEN
    MyStacks.Push (item);
  END (* if *)
END PushValid;

END ValidStacks.
```

These can then be refined for a specific data type and validation procedure.

Example 3: The generic matrices defined in example 2 of 6.2.3 can be refined locally using literal or constant data established in the main module.

```
MODULE Client;

IMPORT Matrix;

  MODULE Mat10x20 = Matrix (10, 20, CARDINAL);
  EXPORT QUALIFIED TMatrix, Invert;
  END Mat10x20;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
CONST
  row = 4;
  col = 6;

  MODULE MatRowXCol = Matrix (row, col, CARDINAL);
  EXPORT QUALIFIED TMatrix, Invert;
  END MatRowXCol;

VAR
  m : Mat10x20.TMatrix;
  n : MatRowXCol.TMatrix;

BEGIN
 (*.
   .
  .*)
  Mat10x20.Invert (m);
  MatRowXCol.Invert (n);
END Client.
```

Example 4: Two independent local refinements of the module counter can be performed, in effect creating two ADT counters both of which are hidden from the program and modifiable only through the refined procedures.

```
MODULE NeedsACounter;

IMPORT Counter;

VAR
  countingCondition1, countingCondition2 : BOOLEAN;

MODULE Duke = Counter;
EXPORT QUALIFIED Inc, Reset, Count;
END Duke;

MODULE Baron = Counter;
EXPORT QUALIFIED Inc, Reset, Count;
END Baron;

BEGIN (* main program module *)

IF countingCondition1
  THEN
    Duke.Inc;
  ELSIF countingCondition2 THEN
    Baron.Inc;
  END;

END NeedsACounter.
```

Example 5: The refinement of two instances of the same generic module is possible, and it has some uses (as shown in example 4) but it is more likely that ISO Standard Modula-2 with Generic extensions will be used to refine Abstract Data Types (ADTs) as separate library modules that are then used as many times as desired in an application. Along the way, new facilities could be added to a generic separate module to form a new generic separate module that could in turn be refined with a specific data type.

-`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Suppose one begins with a classical data structure such as a list, parameterized for the type to list. A framework could look like:

```
GENERIC DEFINITION MODULE Lists (DataType : TYPE);
TYPE
  List;
PROCEDURE Create (VAR l : List);
PROCEDURE Destroy (VAR l : List);
PROCEDURE Add (l : List; item : DataType);
PROCEDURE Delete (l : List; item : DataType);
PROCEDURE Find (l : List; item : DataType);
END Lists.

GENERIC IMPLEMENTATION MODULE Lists (DataType : TYPE);
TYPE
  NodePointer = POINTER TO Node;
  Node =
    RECORD
      theItem : DataType;
      next : NodePointer;
    END;
  List = POINTER TO ListRecord;
  ListRecord =
    RECORD
      Head : NodePointer;
      numberActive : CARDINAL;
    END;
(* all stubs, testing concept only *)
PROCEDURE Create (VAR l : List);
END Create;
PROCEDURE Destroy (VAR l : List);
END Destroy;
PROCEDURE Add (l : List; item : DataType);
END Add;
PROCEDURE Delete (l : List; item : DataType);
END Delete;
PROCEDURE Find (l : List; item : DataType);
END Find;
END Lists.
```

This generic separate module can be refined as it is to produce new separate or local modules implementing lists for a specific data type, and as many of these lists as desired can then be employed. Alternately, one could refine in a generic module to produce, say, generic sorted lists. One first creates a new generic definition module and then refines the generic separate module above in the implementation, with appropriate exports per the definition:

```
GENERIC DEFINITION MODULE ListsSorted (DataType : TYPE; Compare : CompareProc );

FROM Comparisons IMPORT
  CompareResults;

TYPE
  List;
  CompareProc = PROCEDURE (DataType, DataType) : CompareResults;

PROCEDURE Create (VAR l : List);
PROCEDURE Destroy (VAR l : List);
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
PROCEDURE Delete (l : List; item : DataType);
PROCEDURE Find (l : List; item : DataType);
PROCEDURE Insert (theList : List; theItem : DataType);
END ListsSorted.

GENERIC IMPLEMENTATION MODULE ListsSorted (DataType : TYPE; Compare : CompareProc);

MODULE SLists = Lists (DataType);
EXPORT List, Create, Destroy, Delete, Find;
END SLists;

PROCEDURE Insert (theList : List; theItem : DataType);
END Insert;

END ListsSorted.
```

This new generic separate module can now be refined to produce a separate module for lists of a particular data type; for instance:

```
DEFINITION MODULE IntListsSorted = ListsSorted (INTEGER, IntegerInfo.Compare);
END IntListsSorted.
```

A program module can be written that imports and uses this separate module, creating as many of the sorted lists of integers as desired.

6.5 Module Parameter Compatibility

6.5.1 Actual and Formal Module Parameter Correspondence

Every refiner of a generic separate module that has a formal parameter list shall supply a corresponding actual module parameter list. The actual parameters and the formal parameters shall match. The translation of the refiner of the generic separate module involves the evaluation of any actual parameters, the binding of the results of this evaluation to the corresponding formal parameters, and the translation of the refined module that results from this action.

The correspondence between actual module parameters and formal module parameters is established by the positions of the parameters in the lists of actual and formal parameters respectively.

NOTE — To avoid name clashes, the semantics of module parameters is the same as that of procedure parameters. That is, the formal parameters are semantically local aliases for the actual parameters. This does not preclude the creation of a preprocessing tool that actually works by textual substitution; it merely imposes on such a tool the necessity of ensuring that if some local name conflicts with that of an actual parameter, the conflict shall be resolved, perhaps by renaming.

6.5.2 Actual Module Parameters

Actual module parameters are either a constant expression of a type compatible to the formal parameter type (if the corresponding formal parameter is a constant value parameter) or a type identifier (if the corresponding formal parameter is a type parameter).

At the point of refinement actual parameters have to be provided according to the following rules:

a) For a formal TYPE parameter the identifier of a visible type shall be provided. See 6.9.1.3 in the Base Language.
b) For the other parameters a constant expression of a type compatible with the requested type shall be provided. (NOTE — a procedure is a constant value). See 6.9.1.2 in the Base Language.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTE 1 — In either case, any type that is visible at the point of the refining module is potentially usable as an actual parameter, or as the type of an actual parameter. This includes pervasive types, user defined types, types qualified by a visible module name, or any other types provided by other parts of this multi-part standard. (e.g. classes)

NOTE 2 — This check of the parameters does not in itself guarantee that the refinement can be compiled without errors.

NOTE 3 — There are no variable module parameters as there are for procedures.

NOTE 4 — A variable designator cannot be used to pass a value to a constant value parameter.

NOTE 5 — Since a module parameter list can include items of types named previously in the list, such lists shall be evaluated left to right—a restriction not present in the base language for evaluating parameters of procedures.

Other than the restrictions on refining module actual parameters (to constant expression parameters and type parameters), and the requirement of left-to-right evaluation, the rules for these parameters (matching, compatibility) are the same as those given in the Base Language for actual procedure parameters of these kinds.

Concrete Syntax

actual module parameters = left parenthesis, actual module parameter list, right parenthesis ;

actual module parameter list = actual parameter, {comma, actual parameter} ;

actual parameter = constant expression | type parameter ;

6.5.3 Module Parameter Matching

A refinement of a generic separate module shall supply actual parameters that match the formal parameters given in the definition of the generic separate module.

Semantics

The formal module parameters of a generic separate module shall match the actual module parameters specified in any refiner of that generic separate module. The formal module parameters match the actual module parameters if and only if the number of formal module parameters is equal to the number of actual module parameters and each formal module parameter is parameter-compatible with the corresponding actual module parameter — see 6.5.4. This correspondence shall be established by the positions of the parameters in the lists of formal and actual module parameters respectively.

6.5.4 Module Parameter Compatibility

Semantics

A formal module parameter shall be parameter-compatible with an actual module parameter if one of the following two statements is true:

a) The formal parameter is a constant value parameter and the actual parameter is a constant expression that is assignment compatible with the type of the formal module parameter.

b) The formal parameter is a type parameter and the actual parameter is a type identifier.

NOTE — These rules are a subset of the rules in the main standard 6.9.3 for procedure parameters, restricted only by the fact that there are no variable module parameters and that actual constant value module parameters can only be constant expressions compatible with the type of the formal constant value parameter.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.6 Exceptions

A conforming implementation of ISO Standard Modula-2 with Generic extensions shall have rules for detection and reporting of exceptions identical in every respect to a conforming Modula-2 implementation.

If exceptions are defined in a generic separate module they are also generic (templates) until a refinement has been done. If such an exception is raised, its source is the refinement. Thus, two different refinements of the same generic separate module raise different exceptions from different sources.

7 System Modules

The system modules of ISO Standard Modula-2 with Generic extensions are based directly on those of the base library as defined in ISO/IEC 10514-1. All system modules of the base library keep their representation and meaning; no new system modules are defined for ISO Standard Modula-2 with Generic extensions.

NOTE — This does not preclude the possibility that later editions of or amendments to this part of the multi-part standard might include such modules.

8 Required Library Modules

The required library modules of ISO Standard Modula-2 with Generic extensions are based directly on those of the base library as defined in ISO/IEC 10514-1. All required library modules of the base library keep their representation and meaning; no new required library modules are defined for ISO Standard Modula-2 with Generic extensions.

NOTE — This does not preclude the possibility that later editions of or amendments to this part of the multi-part standard might include such modules.

9 Standard Library Modules

The standard library modules of ISO Standard Modula-2 with Generic extensions are based directly on those of the base library as defined in ISO/IEC 10514-1. All standard library modules of the base library keep their representation and meaning; no new standard library modules are defined for ISO Standard Modula-2 with Generic extensions.

NOTE — This does not preclude the possibility that later editions of or amendments to this part of the multi-part standard might include such modules.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex A
(normative)
Changes To the Syntax of the Base Language

keyword = base language keyword | "GENERIC" ;

compilation module = program module | definition module | implementation module | generic definition module | generic implementation module | refining definition module | refining implementation module ;

local module declaration = Base Language local module declaration | refining local module declaration ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex B
(normative)
Collected Concrete Syntax

generic definition module =
"GENERIC", "DEFINITION", "MODULE", module identifier, [formal module parameters], semicolon,
import lists, definitions,
"END", module identifier, period ;

generic implementation module =
"GENERIC", "IMPLEMENTATION", "MODULE", module identifier, [interrupt protection], [formal module parameters], semicolon,
import lists, module block,
module identifier, period ;

refining definition module =
"DEFINITION", "MODULE", module identifier, equals, generic separate module identifier, [actual module parameters], semicolon,
"END", module identifier, period ;

refining implementation module =
"IMPLEMENTATION", "MODULE", module identifier, equals, generic separate module identifier, [actual module parameters], semicolon,
"END", module identifier, period ;

generic separate module identifier = identifier ;

formal module parameters = left parenthesis, formal module parameter list, right parenthesis ;

formal module parameter list = formal module parameter, {semicolon, formal module parameter} ;

formal module parameter = constant value parameter specification| type parameter specification ;

constant value parameter specification = identifier list, colon, formal type;

type parameter specification = identifier list, colon, "TYPE" ;

refining local module declaration =
"MODULE", module identifier, equals, generic separate module identifier, [actual module parameters], semicolon,
[export list],
"END", module identifier ;

actual module parameters = left parenthesis, actual module parameter list, right parenthesis ;

actual module parameter list = actual parameter, {comma, actual parameter} ;

actual parameter = constant expression | type parameter ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex C
(informative)
Rationale

C.1 General

Programmers often face the situation where data must be handled within a structure in a manner that is dependent on the structure, but independent of the data stored in those structures. Since it might well be the case that more than one (almost identical) instance of such structures (e.g. linked lists) is required to contain different types of data, it is desirable to write the code for the manipulation of such structures only once and in a manner independent of any possible contents—that is, in a generic fashion. In like manner, it might be necessary to code procedures and functions (such as sorts) that operate on data stored in structures already recognized by the base language (such as an array). Such manipulations (apart from comparison) are also independent of the data type of the structure elements.

In both cases, there is a need to devise a method to produce a general solution that can be refined for any number of specific data types. This is not possible in the base language of Modula-2 without sacrificing type safety by employing parameters such as ARRAY OF LOC. Moreover, the type compatibility rules prevent the building of generic procedure types even with this mechanism unless their parameters are the even more unsatisfactory ADDRESS. See (3) and (5) for a full discussion of the difficulties where it is concluded: "Thus, in the critical matter of producing re-usable code, Modula-2 fails to deliver on safe type-checked techniques, and so is a clumsy tool at best for generic software."

C.2 Rationale For Some Decisions

C.2.1 Genericity at compile time

At various stages of the discussion, proposals have been advanced to make refinement dynamic, in the sense that it would take place at run time rather than requiring prior compilation. However, moving refinement (and therefore memory allocation) to run time would create inefficient code, and be difficult to implement.

C.2.2 Constant/Type parameters instead of Module names

The initial proposal had only module names as parameters in much the manner of Modula-3. This constant/Type parameter style was added as an option, but it subsequently became clear that this addition made the ability to use module names as parameters redundant.

— The use of local refining modules is less baroque, as there is no need to create other local modules for passing information as parameters. The information can be passed from the scope of the program module instead.
— The contract for using a generic separate module becomes much more readable as all used entities must be listed as parameters in the module heading.
— If module names are employed, it is not possible to tell from the definition module which entities from those modules will ultimately be employed in the implementation. The contract between user and provider of a generic separate module could in that case be invalidated just by using another entity from the parameter module inside the generic implementation without changing its corresponding definition module. In the style chosen for this part of the multi-part standard, any new entity one wishes to employ in the implementation must first be explicitly listed as a parameter in the definition.

C.2.3 Allowing refining local modules

The question of whether (and how) to permit refinement of a generic separate module as a local module is more difficult to answer than the corresponding question for library modules as Modula-2 does not permit the equivalent of DEFINITION/IMPLEMENTATION pairs in a program module or in an implementation module. One option might have been to forbid this possibility, but there is important functionality that would be impossible without local refinement, namely the ability to make a new generic separate module from an old one (nested refinement).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

C.2.4 Export statements in refining local modules

The preprocessor renaming approach is not possible unless the items renamed in the definition module are exported unqualified into the scope of the implementation module by the refining local module. Since at other times (when making two refinements of the same generic separate module in the same scope) qualified export is more desirable, control of the nature and extent of exports has been moved into the refiner. This also adds flexibility for the programmer and reduces the amount of "magic" in the concept of a merger of definition and implementation modules.

C.2.5 Not making redundant items disappear

The name of a generic separate module is required to be imported into the scope of the refinement. Following the actual refinement, this name seems to be of no use. However, it seems inappropriate either to change the normal import rules or to specify that the unneeded import shall "disappear" upon refinement , and the inclusion of this line leaves implementation strategies more open, and makes the code more human readable. This part of the multi-part standard avoids the "magic" of simply making the name disappear when it was no longer needed. The present approach, while somewhat redundant, is more consistent with what a user familiar with the Base Language would expect from the use of the module framework for generics.

A generic separate module is a new kind of module entity, so for the sake of simplicity and consistency, the use of the word IMPORT in such situations is genuine import, in the sense that it makes an identifier visible. The only legal use of such an identifier is for local refinement. The omission of the import of the generic name would mean that there is no flag to the reader or to the compiler of a program containing a refining module. The entire module would have to be examined to determine if it contained any such references.

A question was raised as to whether this slightly different sense of importation (the name is visible only for the purposes of refinement, and the program module can make no use of it because it is a generic separate module name) ought to be flagged by syntax such as GENERIC IMPORT, but this appeared to be unnecessary.

C.2.6 Including Refining Implementation Modules

Although the illustrations in the sample code in Annex D showing a preprocessor approach to refinement might lead the reader to suspect that these modules are unnecessary, refining implementation modules were included so as to maintain the Base Language method of requiring both definition and implementation parts of a separate library module. A generic separate definition module can be refined and client programs translated in the light of this refinement prior to the refining of the generic separate implementation module, and even (in the usual case of refining to a separate module) before the generic separate implementation module has been written. This is the behaviour that users of the Base Language would expect.

In addition, if the generic separate definition module contains an opaque type, the refinement of the associated generic separate implementation module is of its declaration of the item. The type remains abstract with respect to clients, but this behaviour allows the type to be transparent to local refinements in case additional procedures need to be written in the surrounding environment.

C.2.7 Generic and dynamic modules

Consideration of the relationship between generic local modules and dynamic modules led to the conclusion that the two features are orthogonal.

C.2.8 Generic program modules have not been provided

No functionality has been identified for them.

C.2.9 No variable module parameters

Variable parameters for modules and the use of variables to pass values to value module parameters have both been forbidden, because either would force refinement to be done dynamically (at run time) and open up many new issues. (E.g. If a variable changes its value, how, and at what point is the module that depends on that value get re-refined to reflect the new value?)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

C.2.10 Leaving the END in refining modules

The END in a separate refining module is in some sense redundant. However, leaving it out would tend to imply that a refiner is *not* a module, but something else.

The later decision to add an export clause to the refining module to correct a problem in refining as a local module in an implementation module requires that the END be retained.

C.2.11 Requiring a Keyword tag only for Generic Modules, not refiners

This is not inconsistent because a program module cannot contain anything generic whatsoever. It can only contain refiners, and refining modules need have no keyword tag to mark them as such. The translator must determine what action to take when it encounters refining code; there is no need to tag the overall file, though an implementor *might* wish to do this through file names.

C.3 Possible Future Work

There is nothing in this part of ISO/IEC 10514 that would preclude a later addition of other generic types and procedures that are not encapsulated in a generic module, and in which the keyword GENERIC is more broadly available.

C.4 Relationship to Other Languages

C.4.1 ISO Standard Modula-2 with Generic extensions and Object Oriented Modula-2

Because of the decision of WG13 not to use the module as the basis for object classes, and because the rules for refinement of generic separate modules are applied before translating a Modula-2 program with unaltered Base Language syntax and semantics, the provisions of this part of ISO/IEC 10514 are orthogonal to the provisions of par three of this multi-part standard Modula-2—(ISO/IEC 10514-3:1998(E)—on object oriented Modula-2, with the single exception that "GENERIC" becomes a keyword and is not available as an identifier to programs.

It is not the intention that ISO Standard Modula-2 with Generic extensions facilities compete with those of the OO part of ISO/IEC 10514. Their primary use will be in the construction of frameworks or templates of reusable code for classical Abstract Data Types (ADTs) such as lists, and for the coding of generic techniques such as sorting, where, in both cases, the target data type will now no longer have to be specified ahead of time.

Generic modules are frameworks or templates for structures and techniques that need only to be fleshed out by specifying one or more target types or values. If actual values were allowed to be variables (a dynamic model) generic modules could in that case indeed be used as a limited form of value-based object class. WG13 considered and rejected several dynamic models for generic programming on the specific grounds that they would trespass on OO. Since it is not generally practical to employ Generic modules as object classes since variables cannot be used in a refinement, this part of ISO/IEC 10514 does not substantially overlap the OO part of ISO/IEC 10514.

This part of ISO/IEC 10514 does not obstruct any evolution of the OO part of ISO/IEC 10514, or any further evolution of this part of ISO/IEC 10514.

Generic modules are not value based objects; they are templates (not unlike those of C++) for the development of abstract data types (e.g. lists) and techniques for manipulating abstract data types (e.g. sorts) where both are free of the constraint of having to represent the specific types at the outset.

C.4.2 Generics in Other Notations

The semantics of ISO Standard Modula-2 with Generic extensions have some similarities to those used for similar facilities in Modula-3, Ada, and C++. The reader is referred to the respective language definitions for authoritative information.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex D
(informative)
Translations of Example Refinements to Standard Modula-2

D.1 Introduction

This Annex contains a set of transformations, one for each example, to show one *possible* way to interpret refinement in terms of the Base Language with *approximate* semantics using a preprocessor tool. There are other ways of implementing the semantics more accurately using facilities in a compiler, for example. The reason that these illustrations provide only approximate semantics is that the module parameters are translated into imports and/or definitions that rename them as the formal parameters. Because there might also be items in the scope with names identical to an actual parameter used in this way, this technique then fails to produce the correct results. (Refinement with name semantics is not the same as refinement with value semantics).

D.2 Examples from 6.2.3, 6.2.4

Example 1:

A possible transformation of *Stacks* by a preprocessor using the refiner *CardStack*:

```
DEFINITION MODULE CardStack;
(* Renamed generic items by refining parameter *)
TYPE
  Element = CARDINAL;

CONST
  StackSize = 100;

PROCEDURE Push (item : Element);

PROCEDURE Pop (VAR item : Element);

PROCEDURE Empty () : BOOLEAN;

END CardStack.
```

A possible transformation of the implementation of *Stacks* by a preprocessor using the refiner *CardStacks*:

```
IMPLEMENTATION MODULE CardStack;
VAR
   stack    : ARRAY [0..StackSize] OF Element;
   stackPtr : CARDINAL;
(* One could also arrange for StackSize to be a parameter. *)

PROCEDURE Push (item : Element);
BEGIN
 stack[stackPtr] := item;
  INC (stackPtr);
END Push;

PROCEDURE Pop (VAR item : Element);
BEGIN
  DEC (stackPtr);
  item := stack[stackPtr];
END Pop;
```

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
PROCEDURE Empty () : BOOLEAN;
BEGIN
  RETURN stackPtr = 0
END Empty;

BEGIN (* module body initialization *)
  stackPtr := 0
END CardStack.
```

In this simple approach, the renaming required to handle the parameters is all done in the refinement of the definition part and need not be repeated here. The refined module itself needs the new name of course, and some parameter correspondence can be checked in such a preprocessor approach. Alternately, some of this could be left until such time as the refined module is compiled.

Example 2:

A possible transformation of *Sorts* by a preprocessor using the refiner *IntSort*:

```
DEFINITION MODULE IntSorts;
FROM IntegerInfo IMPORT
  Compare;
FROM Comparisons IMPORT
  CompareResults;
(* Renamed generic items by refining parameter *)
TYPE
  Item = INTEGER ;
CONST
  GenCompare = Compare ;

TYPE
  CompareProc = PROCEDURE (Item, Item) : CompareResults;

PROCEDURE Quick (VAR data : ARRAY OF Item);

  (* Other procedures and functions could be included as well. *)
END IntSorts.
```

An obvious limitation of this simple approach is that the generic parameter cannot be originally named *Compare* or the illegal line "Compare = Compare" would be produced, unless the preprocessor is "smart enough" to bypass such obvious conflicts. Preventing all possible naming conflicts of this kind is not very practical in a simple preprocessor, however. Using the correct semantics of parameter substitution in an actual compiler avoids the potential problems generated by this kind of crude transformation into Standard Modula-2.

A possible transformation of the implementation of *Sorts* by a preprocessor using the refiner *IntSort*:

```
IMPLEMENTATION MODULE IntSorts;
FROM Comparisons IMPORT
  CompareResults;

PROCEDURE Swap (VAR a, b : Item;)
VAR
  temp : Item;

BEGIN
  temp := a;
  a := b;
  b := temp;
END Swap;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
PROCEDURE Quick (VAR data : ARRAY OF Item);
BEGIN
(* typical quicksort algorithm, except that compares are done by a procedure Compare <actual one supplied as parameter>
which returns values of type Comparisons.CompareResults. Swaps are done using the refinement of the generic swap above.
*)

END Quick;

END IntSorts.
```

Example 3:

A possible transformation of *Counter* by a preprocessor using the refiner *ACount.*

```
DEFINITION MODULE ACount;
PROCEDURE Inc;

PROCEDURE Reset;

PROCEDURE Count () : CARDINAL;

END ACount.
```

A possible transformation of the implementation of *Counter* by a preprocessor using the refiner *ACount.*

```
IMPLEMENTATION MODULE ACount;
VAR
  CurrentCount : CARDINAL;

PROCEDURE Inc;
BEGIN
  INC (CurrentCount);
END Inc;

PROCEDURE Reset;
BEGIN
  CurrentCount := 0;
END Reset;

PROCEDURE Count () : CARDINAL;
BEGIN
  RETURN CurrentCount
END Count;

BEGIN (* main *)
  Reset;
END ACount.
```

Example 4:

A possible transformation of *Matrix* by a preprocessor using the refiner *RealMatrix45*:

```
DEFINITION MODULE RealMatrix45;
(* Renamed generic items by refining parameter *)
CONST
  Rows = 4 ;
```

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
CONST
  Cols = 5 ;
TYPE
  MatrixElement = REAL ;

TYPE
  TMatrix = ARRAY [0 .. Rows-1]
       OF ARRAY [0 .. Cols-1] OF MatrixElement;

PROCEDURE Invert (VAR m : TMatrix);

END RealMatrix45.
```

NOTE — This particular tool produces one CONST or TYPE statement for each parameter.

Once again, a renaming preprocessor tool need do very little once the definition part has been processed.

```
IMPLEMENTATION MODULE RealMatrix45;
(* Note that, as in any implementation module, the generic implementation module has access to the <generic> types defined
in the corresponding definition. *)

PROCEDURE Invert (VAR m : TMatrix);
BEGIN
(* your favourite technique *)
END Invert;

END RealMatrix45.
```

D.3 Examples from 6.4 (local refinements)

Example 1:

The preprocessor renaming approach shown here simply makes two copies of the appropriate code inside the main module. The import of CARDINAL from the surrounding scope is correct, as the local surrounding scope could very well have redefined that name.

```
MODULE StackClient;
IMPORT Stacks; (* the generic name has to be imported *)

TYPE
  RecDef =
    RECORD
     c : CHAR;
     i : INTEGER;
    END (* record *);

MODULE CardStack;
IMPORT CARDINAL;
EXPORT QUALIFIED StackSize, Push, Pop, Empty;

(* Renamed generic items by refining parameter *)
TYPE
  Element = CARDINAL;

CONST
  StackSize = 100;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
VAR
  stack    : ARRAY [0..StackSize] OF Element;
  stackPtr : CARDINAL;
(* One could also arrange for StackSize to be a parameter. *)

PROCEDURE Push (item : Element);
BEGIN
 stack[stackPtr] := item;
  INC (stackPtr);
END Push;

PROCEDURE Pop (VAR item : Element);
BEGIN
  DEC (stackPtr);
  item := stack[stackPtr];
END Pop;

PROCEDURE Empty () : BOOLEAN;
BEGIN
  RETURN stackPtr = 0
END Empty;

BEGIN (* module body initialization *)
  stackPtr := 0
END CardStack;

MODULE RecStack;
IMPORT RecDef;
EXPORT StackSize, Push, Pop, Empty;

(* Renamed generic items by refining parameter *)
TYPE
  Element = RecDef ;

CONST
  StackSize = 100;

VAR
  stack    : ARRAY [0..StackSize] OF Element;
  stackPtr : CARDINAL;
(* One could also arrange for StackSize to be a parameter. *)

PROCEDURE Push (item : Element);
BEGIN
 stack[stackPtr] := item;
  INC (stackPtr);
END Push;

PROCEDURE Pop (VAR item : Element);
BEGIN
  DEC (stackPtr);
  item := stack[stackPtr];
END Pop;

PROCEDURE Empty () : BOOLEAN;
BEGIN
  RETURN stackPtr = 0
END Empty;
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
BEGIN (* module body initialization *)
  stackPtr := 0
END RecStack;

VAR
  c : CARDINAL;
  r : RecDef;

BEGIN (* main *)
  CardStack.Push (c);
  Push (r);
END StackClient.
```

Example 2:

The translation of *ValidStacks* using a renaming preprocessor has been omitted, as this example is only a sketch.

Example 3:

A possible transformation of *Client* by a renaming preprocessor:

```
MODULE Client;

IMPORT Matrix;

MODULE Mat10x20;
IMPORT CARDINAL;
  EXPORT QUALIFIED TMatrix, Invert;

CONST
  Rows = 10 ;
CONST
  Cols = 20 ;
TYPE
  MatrixElement = CARDINAL;

TYPE
   TMatrix = ARRAY [0 .. Rows-1] OF ARRAY [0 .. Cols-1] OF MatrixElement;

PROCEDURE Invert (VAR m : TMatrix);
BEGIN
(* your favourite technique *)
END Invert;

END Mat10x20;

CONST
  row = 4;
  col = 6;

MODULE MatRowXCol;
IMPORT row, col, CARDINAL;
  EXPORT QUALIFIED TMatrix, Invert;

CONST
  Rows = row ;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
CONST
  Cols = col ;
TYPE
  MatrixElement = CARDINAL;

TYPE
  TMatrix = ARRAY [0 .. Rows-1] OF ARRAY [0 .. Cols-1] OF MatrixElement;

PROCEDURE Invert (VAR m : TMatrix);
BEGIN
(* your favourite technique *)
END Invert;

END MatRowXCol;

VAR
  m : Mat10x20.TMatrix;
  n : MatRowXCol.TMatrix;

BEGIN
  (*.
    .
   .*)
  Mat10x20.Invert (m);
  MatRowXCol.Invert (n);
END Client.
```

Example 4:

Processing *NeedsACounter* with a renaming preprocessor could produce:

```
MODULE NeedsACounter;

IMPORT Counter;

VAR
  countingCondition1, countingCondition2 : BOOLEAN;

MODULE Duke;
EXPORT QUALIFIED Inc, Reset, Count;

VAR
  CurrentCount : CARDINAL;

PROCEDURE Inc;
BEGIN
  INC (CurrentCount);
END Inc;

PROCEDURE Reset;
BEGIN
  CurrentCount := 0;
END Reset;

PROCEDURE Count () : CARDINAL;
BEGIN
  RETURN CurrentCount
END Count;
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
BEGIN (* main *)
  Reset;
END Duke;

MODULE Baron;
EXPORT QUALIFIED Inc, Reset, Count;

VAR
  CurrentCount : CARDINAL;

PROCEDURE Inc;
BEGIN
  INC (CurrentCount);
END Inc;

PROCEDURE Reset;
BEGIN
  CurrentCount := 0;
END Reset;

PROCEDURE Count () : CARDINAL;
BEGIN
  RETURN CurrentCount
END Count;

BEGIN (* main *)
  Reset;
END Baron;

BEGIN (* main program module *)

IF countingCondition1
  THEN
    Duke.Inc;
  ELSIF countingCondition2 THEN
    Baron.Inc;
  END;

END NeedsACounter.
```

Example 5:

A possible transformation of *Lists* by a preprocessor using the refiner *ListsSorted*:

```
GENERIC IMPLEMENTATION MODULE ListsSorted (DataType : TYPE ; Compare : CompareProc ) ;

MODULE SLists;
IMPORT DataType;
EXPORT List, Create, Destroy, Delete, Find;

TYPE
  NodePointer = POINTER TO Node;
  Node =
    RECORD
      theItem : DataType;
      next : NodePointer;
    END;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
  List = POINTER TO ListRecord;
  ListRecord =
    RECORD
      Head : NodePointer;
      numberActive : CARDINAL;
    END;
(* all stubs, testing concept only *)
PROCEDURE Create (VAR l : List);
END Create;
PROCEDURE Destroy (VAR l : List);
END Destroy;
PROCEDURE Add (l : List; item : DataType);
END Add;
PROCEDURE Delete (l : List; item : DataType);
END Delete;
PROCEDURE Find (l : List; item : DataType);
END Find;
END SLists;

PROCEDURE Insert (theList : List; theItem : DataType);
END Insert;

END ListsSorted.
```

A possible subsequent transformation of *ListsSorted* by a preprocessor using the refiner *IntListsSorted*:

```
DEFINITION MODULE IntListsSorted;
FROM IntegerInfo IMPORT
  Compare;
FROM Comparisons IMPORT
  CompareResults;
(* Renamed generic items by refining parameter *)
TYPE
  DataType = INTEGER ;

TYPE
  List;
  CompareProc = PROCEDURE (DataType, DataType) : CompareResults;

PROCEDURE Create (VAR l : List);
PROCEDURE Destroy (VAR l : List);
PROCEDURE Delete (l : List; item : DataType);
PROCEDURE Find (l : List; item : DataType);
PROCEDURE Insert (theList : List; theItem : DataType);
END IntListsSorted.
```

and, the refinement of the implementation by

```
IMPLEMENTATION MODULE IntListsSorted = ListsSorted (INTEGER, IntegerInfo.Compare);
END IntListsSorted.
```

can be thought of as resulting in:

```
IMPLEMENTATION MODULE IntListsSorted;
MODULE SLists;
IMPORT DataType;
EXPORT List, Create, Destroy, Delete, Find;
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
TYPE
  NodePointer = POINTER TO Node;
  Node =
    RECORD
      theItem : DataType;
      next : NodePointer;
    END;
  List = POINTER TO ListRecord;
  ListRecord =
    RECORD
      Head : NodePointer;
      numberActive : CARDINAL;
    END;
(* all stubs, testing concept only *)
PROCEDURE Create (VAR l : List);
END Create;
PROCEDURE Destroy (VAR l : List);
END Destroy;
PROCEDURE Add (l : List; item : DataType);
END Add;
PROCEDURE Delete (l : List; item : DataType);
END Delete;
PROCEDURE Find (l : List; item : DataType);
END Find;
END SLists;

PROCEDURE Insert (theList : List; theItem : DataType);
END Insert;

END IntListsSorted.
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex E
(informative)
File Names

In systems where files are employed, it is a common convention (but by no means a standardized practice) to employ the file name suffix ".DEF" to indicate that the contents form a definition module and the suffix ".MOD" to indicate that the contents form an implementation module or a program module. Since an implementation would presumably continue to employ these suffixes for refinements if they had a separate physical existence as files, they might wish to consider using the suffixes:

.GDE for a file expressing a generic definition module
.GMO for a file expressing a generic implementation module
.RDE for a file expressing a refining definition module
.RMO for a file expressing a refining implementation module or containing a refining local module.

Multiple refiners and refinements of the same generic separate module would undoubtedly be required each to be stored under unique file names before these suffixes.

If, on the other hand, implementations choose to incorporate all the functionality of refinement into a single translator and also not use separate files for the final refinements, these suggestions are not necessary.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex F
(informative)
Participating Individuals

This Draft International Standard was prepared by Richard J. Sutcliffe, as project editor on behalf of ISO/IEC JTC1/SC22/WG13. The following individuals participated in the process as the nominated experts of ISO/IEC JTC1/SC22/WG13:

Frank Bender	Germany
Eberhard Enger	Germany
Elmar Henne	Germany
Keith Hopper	New Zealand
Herbert Klaeren	Germany
John Lancaster	UK
David Lightfoot	UK
Cornelis Pronk	Netherlands
Wolfgang Redtenbacher	Germany
Martin Schleusser	Germany
Martin Schönhacker	Austria
Richard J. Sutcliffe	Canada
Albert Wiedemann	Germany

The assistance of the following individuals is also gratefully acknowledged:

Dana Aldom	Canada
Omar Jaber	Canada
Darren Johnson	Canada
Josh Merwin	Canada
Peter Moylan	Australia
Robin Nicholl	Canada
Matt Suderman	Canada

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Bibliography

1. Barnes, John. *Programming in Ada 95.* Addison-Wesley: Wokingham, England, 1996
2. Harbison, Samuel P. *Modula-3.* Prentice Hall: Englewood Cliffs, NJ, 1992
3. Hopper, Keith & Sutcliffe, Richard J. *Proposed Generic/Abstract Type Extension To Modula-2* (Canada #C89 WG13 #D211) 1994
4. Schildt, Herbert. *C++ From the Ground Up.* Osborne McGraw-Hill: Berkeley, 1994
5. Sutcliffe, Richard J. *Modula-2 Extension to Support Generic Programming* (WG13 Wien-8 #D213) 1994
6. Sutcliffe, Richard J. *A Discussion of Generic Data Types in Modula-2* (C3/D11) 1987
7. Sutcliffe, Richard J. *Generic Modula-2* (Canada #C93, C94, C95/D220, C96/D221, C101/D235, C103/D242) [CD versions of this DIS]
8. Sutcliffe, Richard J. *Responses to Comments on Generic Modula-2* (Canada #C98, C100) 1995
9. Sutcliffe, Richard J. *Disposition of Comments on CD 10514-2 (Generics)* (Canada #C103/D241, C106/D247) 1997
10. Sutcliffe, Richard J. *Disposition of Comments on DIS 10514-2 (Information technology -- Programming languages -- Part 2: Modula-2, Generics in Modula-2)* (Canada #C109/D253) 1997

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC

ICS 35.060

Price based on 45 pages

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale