520 — Principles of Programming Languages

23: Modula-3

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

---

History

- Defined by a committee from Olivetti and DEC.
- Design goal: language design document should fit in 50 pages.
- Roughly equivalent in power to C++ (except C++ has multiple inheritance and Modula-3 has garbage collection), but infinitely cleaner.
- Not an extension of Modula-2, but a complete re-design.

---

Errors

- A static error must be detected by the implementation before program execution.
- A checked runtime error must be detected and reported at runtime.
- An unchecked runtime error is not guaranteed to be detected. Unchecked runtime errors can occur only in unsafe modules.

---

Ordinal Types

<table>
<thead>
<tr>
<th>Enumeration Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE T = {id1, id2,..., idn}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subrange Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE T = [Lo...Hi]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Predeclared Ordinal Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER, CARDINAL, BOOLEAN, CHAR.</td>
</tr>
</tbody>
</table>
**Ordinal Types...**

```plaintext
TYPE
  T1 = {A, B, C};
  T2 = {A, B, C};
  U1 = [T1.A..T1.C];
  U2 = [T1.A..T2.C];
  V = {A,B};

T1 and T2 are the same type. Modula-3 uses *structural type equivalence*. Two types are the same if they look the same once they have been expanded.

T1 and U1 are different types, one is an enumeration, the other a subrange.
```

**Arrays**

```plaintext
TYPE T = ARRAY [1..3] OF REAL;
TYPE M = ARRAY [1..3],[1..3] OF REAL;
VAR a := ARRAY [1..3] OF REAL {1.0,2.0,3.0};
```

**Records**

```plaintext
TYPE T = RECORD
  seconds: INTEGER;
  millis: [0..999];
END;

TYPE Complex = RECORD
  Re: REAL := 0.0;
  Im: REAL := 0.0;
END;
```

**Packed Types**

```plaintext
(* Syntax: TYPE T = BITS n FOR Base *)

TYPE T = ARRAY [0.255] OF BITS 1 FOR BOOLEAN;

- Each element of T occupies one bit of storage.
```
Sets

TYPE T = SET OF CHAR;
TYPE U = SET OF [0..1023];

Each element of a set is expected to occupy one bit of storage.

References

TYPE
T = REF INTEGER;
U = UNTRACED REF INTEGER;
S = BRANDED "myType" REF INTEGER; (* Explicit *)
V = BRANDED REF INTEGER; (* Compiler-generated *)

An untraced object resides on the non-garbage-collected heap.

REFANY is "void*" for traced objects.

ADDRESS is "void*" for untraced objects.

The NULL type contains only the NIL constant.

Branded types are different from all other types. It is a way of circumventing structural type equivalence.

Procedures

PROCEDURE P (
VALUE x : INTEGER;
VAR y : REAL;

z : TEXT := "HELLO"; (* VALUE by default *)
READEONLY w : REAL)
: BOOLEAN (* Function return type. *)
RAISES {SomeException} =
BEGIN
...
END P;

Procedure Types

TYPE T =
PROCEDURE (
VALUE x : INTEGER;
VAR y : REAL;

z : TEXT := "HELLO";
READEONLY w : REAL)
: BOOLEAN (* Function return type. *)
RAISES {SomeException} =

VAR q : T := P;

q is a variable of procedure type that has the procedure P as its value.
Objects

An object is either NIL or a reference to a data record paired with a method suite, which is a record of procedure that will accept the object as a first argument. [SPwM3]

- $T.m$ is method $m$ in object type $T$.
- ROOT — predeclared object type, the root of all traced objects.
- UNTRACED ROOT — predeclared object type, the root of all untraced (not garbage collected) objects.

Syntax of object declarations:

```
TYPE T = super-type
OBJECT
  Fields
  METHODS
    name sig := proc_name
    Overrides
    Overrides
END
```

Methods are declared by giving their name and signature and a “value”: the `proc_name` is a top-level procedure whose first argument is $T$.

Note that unlike Java, method bodies are not given in the class/object itself.

In Java, the module and the object type are the same. In Modula-3, they are distinct constructs.
Objects...

TYPE
A = OBJECT a:INTEGER; METHODS p() END;
AB = A OBJECT b:INTEGER END;

PROCEDURE Pa(self: A) = ...
PROCEDURE Pab(self: AB) = ...

TYPE T1 = AB OBJECT OVERIDES p := Pab END;
TYPE T2 = A OBJECT OVERIDES p := Pa END;
TYPE T3 = AB OBJECT OVERIDES p := Pa END;
TYPE T4 = A OBJECT OVERIDES p := Pab END;

T4 is statically wrong. (Why?)

Subtyping Rules

T <: U means that T is a subtype of U and that U is a supertype of T.
If T <: U then every value of type T is also a value of type U.

[u..v] <: B if u and v have base type B.
[u..v] <: [a..b] if [u..v] ⊆ [a..b].

ARRAY OF)^m ARRAY J_1 OF ... ARRAY J_n OF ARRAY
K_1 OF ... ARRAY K_p OF T <:
ARRAY OF)^m (ARRAY OF)^n ARRAY I_1 OF ... ARRAY
I_p OF T
if NUMBER(I_i) = NUMBER(K_i) for i = 1,...,p.

That is, an array type A is a subtype of an array type A' if they have the same element type, the same number of dimensions, and, for each dimension, either both are open, or A is fixed and A' is open, or they are both fixed and have the same size.

Subtyping Rules

NULL <: REF T <: REFANY.
REFANY contains all traced references.
NIL is a member of every reference type.
NULL <: UNTRACED REF T <: ADDRESS
ADDRESS contains all untraced references.
**Subtyping Rules – Procedures**

- **NULL <: PROCEDURE(A):R RAISES S**
  - NIL is a member of every procedure type.

- **PROCEDURE(A):Q RAISES E <: PROCEDURE(B):R RAISES F**
  - if signature (B):R RAISES F covers signature (A):Q RAISES E.
  - For procedure type, T <: T' if they are the same except for parameter names, defaults, and the raises set, and the raises set for T contains the raises set for T'.

**Subtyping Rules – Objects**

- **ROOT <: REFANY**
  - Every traced object is a reference.

- **UNTRACED ROOT <: ADDRESS**
  - Every traced object is a reference.

- **NULL <: T OBJECT . . . END <: T**
  - NIL is a member of every object type.
  - every subtype is included in its supertype.

**Subtyping Rules – Packed Types**

- **BITS n for T <: T and T <: BITS n FOR T**
  - BITS FOR T has the same values as T.

**Subtyping Rules**

- **T <: T for all T**
  - <: is reflexive.

- **T <: U, U <: V ⇒ T <: V.**
  - <: is transitive.

- **T <: U and U <: T does not imply that T and U are the same, since the subtype relation is unaffected by parameter names, default values, and packing.**

- **T and U are subtypes of each other, but are not the same:**
  ```plaintext
  TYPE T = [0..255];
  U = BITS 8 FOR [0..255];
  ```
Statements — Assignment

A type $T$ is assignable to a type $U$ if
1. $T < : U$
2. $U < : T$ and $T$ is an array.
3. $U < : T$ and $T$ is a reference type.
4. $T$ and $U$ are ordinal types with at least one member in common.

An expression $e$ is assignable to a variable $v$ if
1. $\text{TYPE}(e)$ is assignable to $\text{TYPE}(v)$.
2. The value of $e$ is a member of $\text{TYPE}(v)$.
3. $e$ is not a local procedure.
4. If $e$ is an array then it has the same shape as $v$.

Only point 1 can be checked statically.

Syntax: $v := e$.

The order of evaluation is undefined.

$e$ must be evaluated before $v$ is updated.

If $v$ and $e$ are overlapping sub-arrays then no element is written to before it is read.

VAR

\begin{verbatim}
x : REFANY;
a : REF INTEGER;
b : REF BOOLEAN;
\end{verbatim}

\begin{verbatim}
a := b; (* static error *)
x := a; (* no possible error *)
a := x; (* checked runtime error? *)
\end{verbatim}

All REFS carry runtime types which allows the compiler to insert runtime checks when necessary.

ADDRESS is a raw address without runtime type information. Hence no runtime type checks can be made.
**Statements — Procedure Call**

PROCEDURE P (ch: CHAR; n: INTEGER := 0)  
the following calls are equivalent:
- P('a', 0);
- P('a');
- P(ch := 'a');
- P(n := 0, ch := 'a');
- P('a', n := 0);

the following is illegal:
- P(); (* no value for ch *)
- P(n := 0, 'a'); (* positional bindings must follow keyword bindings *)

**Statements — EVAL**

EVAL Thread.Fork(p)

- EVAL ignores the result of an expression.
- Compare C where programmers habitually ignore function return values/result codes.

**Statements — Block Statement**

Declarations
BEGIN
    Statements
END

- Introduces a new scope.

**Statements — Exceptions**

- RAISE e raises an exception.

TRY
    Body
EXCEPT
    exception (argument) ==> handler
    exception (argument) ==> handler
    exception (argument) ==> handler
ELSE
    handler
END
Statements — TRY-FINALLY

- $S_1$ terminates normally ⇒ the statement is equivalent to $S_1; S_2$.
- $S_1$ raises an exception, $S_2$ terminates normally ⇒ the exception is re-raised after $S_2$ is executed.
- $S_1$ and $S_2$ both raise an exception ⇒ the outcome of the TRY is the exception from $S_2$.

```
TRY
  $S_1$
FINALLY
  $S_2$
END
```

Statements — LOOP

- LOOP $S$ END repeatedly executes $S$ until it raises an EXIT-exception.
- Equivalent to:

```
TRY
  $S$; $S$; $S$; ...
EXCEPT
  exit-exception =>
END
```

Statements — EXIT

- EXIT raises the exit exception to prematurely exit a loop.
- EXIT must be nested within a LOOP, WHILE, REPEAT, FOR.

Statements — RETURN

- RETURN raises the return exception in a procedure.
- Equivalent to ($B$ is the body of the procedure):

```
TRY
  $B$
EXCEPT
  return-exception =>
END
```
Statements — RETURN $e$

- RETURN $e$ raises the return exception in a function.
- Equivalent to ($B$ is the body of the procedure):
  TRY
    $B$; (* error: no returned value *)
  EXCEPT
    return-exception($v$) =>
  (* the result becomes $v$ *)
END
- It's a checked runtime error to not return a value from a function.

Statements — IF

IF $B_1$ THEN $S_1$
ELSIF $B_2$ THEN $S_2$
... 
ELSIF $B_n$ THEN $S_n$
ELSE
  $S_0$
END

Statements — WHILE

WHILE $B$ DO
  $S$
END

equivalent to

LOOP
  IF $B$ THEN
    $S$
  ELSE
    EXIT
  END
END

Statements — REPEAT

REPEAT
  $S$
UNTIL $B$

equivalent to

LOOP
  $S$; IF $B$ THEN
    EXIT
  END
END

END
**Statements — WITH**

**WITH id = e DO**
\[ S \]
**END**

- `id` becomes an alias for the expression `e` in `S`.
- `e` is evaluated once on entry.
- Like `LET` in SCHEME.

**Statements — FOR**

**FOR id := first TO last BY step DO**
\[ S \]
**END**

- `id` is a read-only variable with the same type as `first` and `last`.
- `first`, `last` and `step` are executed once.
- If `step` is negative, the loop iterates downwards.

**Statements — FOR...**

**FOR id := first TO last BY step DO S END**

↓ ↓ ↓

VAR i := ORD(first); done := ORD(last); delta := step; BEGIN
  IF delta >= 0 THEN
    WHILE i <= done DO
      WITH id=VAL(i,T) DO S END; INC(i,delta);
    END
  ELSE
    WHILE (i >= done DO
      WITH id=VAL(i,T) DO S END; INC(i,delta);
    END
  END END END

**Statements — CASE**

**CASE e OF**
\[ L_1 \Rightarrow S_1 \]
\[ L_2 \Rightarrow S_2 \]
\[ \ldots \]
\[ L_n \Rightarrow S_n \]
**ELSE**
\[ S_0 \]
**END**

- The `L_i`s are constant expressions that must not overlap.
**Statements — TYPECASE**

```
TYPECASE e OF
  T_1(v_1) => S_1
  | T_2(v_2) => S_2
  | ...
  | T_n(v_n) => S_n
ELSE
  S_0
END
```

- `e` is of reference type.
- The `T_i`s are a subtype of the type of `e`.
- For the minimum `i` such that the type of `e` is a subset of `T_i`, `S_i` is executed.

**Inc/Dec**

```
INC(v, n) is equivalent to
  WITH x = v DO
  x := VAL(ORD(X)+n, T)
END
DEC(v, n) is equivalent to
  WITH x = v DO x := VAL(ORD(X)-n, T) END
```

- `T` is the type of `v`.
- `ORD(elm)` : `INTEGER` converts an element of an enumeration to an integer that represents its position.
- `VAL(INTEGER, Type)` : `Type` converts an integer into an element of an enumeration.

**Declarations**

```
TYPE T = U.
CONST id: T = C.
CONST id = C. The type of `id` is the type of `C`.
VAR id: T := E.
VAR v_1, ..., v_n: T := E.
EXCEPTION id(T). T is the type of the argument.
```

**Opaque Types**

- Modula-3 has more flexible opaque types than Modula-2. An opaque type can be partially revealed, so that some parts of the program can have more access to its internals than others.
- Java’s solution is the `private` and `protected` declarations.
- A C++ class has `friend` classes which are allowed more access than other classes.
Opaque Types – REVEAL

A partial revelation: \texttt{REVEAL T <: U}. This reveals that \( V \) is a supertype of \( T \). \( T \) has previously been declared an opaque type.

A complete revelation: \texttt{REVEAL T = U}. This gives the final, concrete, type of \( T \). It's a (link-time) error if any type revealed as a supertype of \( T \) is not a supertype of \( V \).

Opaque Types — Example

\begin{verbatim}
INTERFACE I;
  TYPE T <: ROOT;
  PROCEDURE P(x:T):T;
END I;

INTERFACE IClass;
  IMPORT I; REVEAL I.T <: MUTEX;
END IClass;

INTERFACE IRep;
  IMPORT I;
  REVEAL I.T = MUTEX BRANDED OBJECT count:INTEGER END;
END IRep;
\end{verbatim}

An importer of \texttt{I} can only allocate \( T \) objects, pass them to \texttt{I.P}, or create subtypes of \texttt{I.T}.

An importer of \texttt{IClass} knows that every \texttt{IClass.T} object is a \texttt{MUTEX} and can therefore lock on it.

An importer of \texttt{IRep} can access the \texttt{count} field.

Opaque Types — Example...

\begin{verbatim}
INTERFACE I; TYPE T <: ROOT; PROCEDURE P(x:T):T; END I;

INTERFACE IClass; IMPORT I; REVEAL I.T <: MUTEX; END IClass;

INTERFACE IRep; IMPORT I;
  REVEAL I.T = MUTEX BRANDED OBJECT count:INTEGER END;
END IRep;
\end{verbatim}

Recursive Declarations

- Modula-3 allows free order of declarations. The consequence is that declarations can be recursive. Some recursive declarations are illegal.
- A declaration \( N = E \) or \( N : E \) is recursive if \( N \) occurs in any partial expansion of \( E \).
- A recursive declaration is legal if every occurrence of \( N \) in any partial expansion of \( E \) is 1. within some type constructor \texttt{REF} or \texttt{PROCEDURE}, 2. within a field or method type of the \texttt{OBJECT} type constructor.
Recursive Declarations

Legal Recursive Declarations

```plaintext
TYPE List = REF RECORD x:REAL; link:List END;
TYPE T = PROCEDURE(p:T);
TYPE XList = X OBJECT link:XList END;
CONST N = BYTESIZE(REF ARRAY [0..N] OF REAL);
VAR v : REF ARRAY[0..BYTESIZE(v)] OF INTEGER;
```

Illegal Recursive Declarations

```plaintext
TYPE T = RECORD x:T END;
TYPE U = OBJECT METHODS m() := U.m END;
CONST N = N+1;
```

Intuitively, recursive declarations are only legal if they go through a level of indirection (a pointer).

Import Statement

- **IMPORT I AS J** imports the interface whose global name is I and gives it the local name J.
- **IMPORT I** is short for **IMPORT I AS I**.
- **FROM I IMPORT N** introduces N as the local name for the entity declared as N in the interface I.

Interfaces

- An interface has the form:
  ```plaintext
  INTERFACE id;
  Imports;
  Decls
  END id.
  ```
  id is an identifier that names the interface, Imports is a sequence of import statements, Decls is a sequence of declarations that contains no procedure bodies or non-constant variable initializations.
  It is a static error for two or more interfaces to form an import cycle.

Modules

- An interface has the form:
  ```plaintext
  MODULE id EXPORTS Interfaces;
  imports;
  Block id.
  ```
  Interfaces is a list of distinct names of interfaces exported by the module.
  If module M exports interface I, then all declared names in I are visible without qualification in M.
  Any procedure declared in I can be redeclared in M, with a body.
  The signature in M must be covered by the signature in I.
This is illegal, since two names in exported interfaces coincide:

```plaintext
INTERFACE I;
PROCEDURE X(); ...

INTERFACE J;
PROCEDURE X(); ...

MODULE M EXPORTS I, J;
PROCEDURE X() = ...;
```

This is illegal, since the visible imported name X coincides with the top-level name X:

```plaintext
INTERFACE I;
PROCEDURE X(); ...

MODULE M EXPORTS I;
FROM I IMPORT X;
PROCEDURE X() = ...;
```

Example

```plaintext
INTERFACE Stack;
    TYPE T <: REFANY;
    PROCEDURE Create(): T;
    PROCEDURE Push(VAR s: T; x: REAL);
    PROCEDURE Pop(VAR s: T): REAL;
END Stack.
```

Example

```plaintext
MODULE Stack;
    REVEAL T = BRANDED OBJECT item: REAL; link: T END;
    PROCEDURE Create(): T = BEGIN RETURN NIL END Create;
    PROCEDURE Push(VAR s: T; x: REAL) =
        BEGIN s := NEW(T, item := x, link := s) END Push;
    PROCEDURE Pop(VAR s: T): REAL =
        VAR res: REAL;
        BEGIN res := s.item; s := s.link; RETURN res END Pop;
END Stack.
```
Example...

If the representation of stacks is required in more than one module, it should be moved to a private interface, so that it can be imported wherever it is required:

```pascal
INTERFACE Stack (* ... as before ... *) END Stack.

INTERFACE StackRep; IMPORT Stack;
    REVEAL Stack.T =

MODULE Stack; IMPORT StackRep;
    (* Push, Pop, and Create as before *)
    BEGIN
    END Stack.
```

Generics

In a generic interface or module, some of the imported interface names are treated as formal parameters, to be bound to actual interfaces when the generic is instantiated.

A generic interface has the form

```pascal
GENERIC INTERFACE G(F_1, ..., F_n);
    Body
    END G.
```

$F_1, \ldots, F_n$ is a list of identifiers, called the formal imports of $G$, and $\text{Body}$ is a sequence of imports followed by a sequence of declarations.

Generics

An instance of $G$ has the form

```pascal
INTERFACE I = G(A_1, ..., A_n) END I.
```

$A_1, \ldots, A_n$ is a list of actual interfaces.

The instance $I$ is equivalent to an ordinary interface defined as follows:

```pascal
INTERFACE I;
    IMPORT A_1 AS F_1, \ldots, A_n AS F_n;
    Body
    END I.
```

Generics

A generic module has the form

```pascal
GENERIC MODULE G(F_1, ..., F_n);
    Body
    END G.
```

An instance of $G$ has the form

```pascal
MODULE I EXPORTS E = G(A_1, ..., A_n) END I.
```

The instance $I$ is equivalent to an ordinary module defined as follows:

```pascal
MODULE I EXPORTS E;
    IMPORT A_1 AS F_1, \ldots, A_n AS F_n;
    Body
    END I.
```
Example

GENERIC INTERFACE Stack(Elem);
(* where Elem.T is not an open array type. *)
TYPE T <: REFANY;
PROCEDURE Create(): T;
PROCEDURE Push(VAR s: T; x: Elem.T);
PROCEDURE Pop(VAR s: T): Elem.T;
END Stack.

Example...

GENERIC MODULE Stack(Elem);
REVEAL T = BRANDED OBJECT
n: INTEGER; a: REF ARRAY OF Elem.T END;
PROCEDURE Create(): T =
BEGIN RETURN NEW(T, n := 0, a := NIL) END Create;
PROCEDURE Push(VAR s: T; x: Elem.T) =
BEGIN ... END Push;
PROCEDURE Pop(VAR s: T): Elem.T =
BEGIN DEC(s.n); RETURN s.a[s.n] END Pop;
BEGIN END Stack.

Example...

To instantiate these generics to produce stacks of integers:
INTERFACE Integer; TYPE T = INTEGER; END Integer
INTERFACE IntStack = Stack(Integer) END IntStack.
MODULE IntStack = Stack(Integer) END IntStack.

Initialization

- The order of execution of the modules in a program is constrained by the following rule:
  - If module M depends on module N and N does not depend on M, then N's body will be executed before M's body, where:
  - A module M depends on a module N if M uses an interface that N exports or if M depends on a module that depends on N.
  - A module M uses an interface I if M imports or exports I or if M uses an interface that (directly or indirectly) imports I.

- Implementations are not expected to share code between different instances of a generic module.
- Implementations are not required to typecheck uninstantiated generics, but they must typecheck their instances.
Safety

- An unsafed module is declared: **UNSAFE MODULE** or **UNSAFE INTERFACE**.
- An interface is safe if there is no way to produce an **unchecked runtime error** by using the interface in a safe module.
- If all modules that export a safe interface are safe, the compiler guarantees the intrinsic safety of the interface.
- If any of the modules that export a safe interface are unsafe, it is the programmer who makes the guarantee.
- It is a static error for a safe interface to import an unsafe one or for a safe module to import or export an unsafe interface.

Designators

- \( r \) dereferences a pointer.
- \( a[i] \) denotes the \((i + 1 - \text{FIRST}(a))\)-th element of the array \( a \).
- \( a[i_1, \ldots, i_n] \) is shorthand for \( a[i_1] \ldots [i_n] \).
- If \( a \) is a reference to an array, then \( a[i] \) is shorthand for \( a^\langle i \rangle \).
- If \( r \) denotes a record, \( r.f \) denotes its \( f \) field.

Designators...

- If \( r \) is a reference to a record, then \( r.f \) is shorthand for \( r^\langle f \rangle \).
- If \( o \) denotes an object and \( f \) names a data field specified in the type of \( o \), then \( o.f \) denotes that data field of \( o \).
- If \( I \) denotes an imported interface, then \( I.x \) denotes the entity named \( x \) in the interface \( I \).
- If \( T \) is an object type and \( m \) is the name of one of \( T \)'s methods, then \( T.m \) denotes the \( m \) method of type \( T \).
- If \( E \) is an enumerated type, then \( E.id \) denotes its value named \( id \).
- \( \text{SUBARRAY}(a: \text{Array}; \text{from}, \text{for}: \text{CARDINAL}): \text{ARRAY OF} \text{ElemType}(a) \) produces a subarray of \( a \).

Set, array, and record constructors

- A set constructor has the form: \( S\{e_1, \ldots, e_n\} \) where \( S \) is a set type and the \( e \)'s are expressions or ranges of the form \( lo..hi \).
- An array constructor has the form: \( A\{e_1, \ldots, e_n\} \) where \( A \) is an array type and the \( e \)'s are expressions.
- A record constructor has the form: \( R\{\text{Bindings}\} \) where \( R \) is a record type and \( \text{Bindings} \) is a list of keyword or positional bindings, exactly as in a procedure call.
NEW

- An allocation operation has the form: NEW(T, ...) where T is a reference type.
- If T is declared as an opaque type, NEW(T) is legal only in scopes where T's concrete type is known completely, or is known to be an object type.
- If T is a reference to an array with k open dimensions, the NEW operation has the form: NEW(T, n_1, ..., n_k) where the n's are integer-valued expressions that specify the lengths of the new array in its first k dimensions.
- If T is an object type or a reference to a record, the NEW operation has the form: NEW(T, Bindings)

Type operations

- ISTYPE (x: Reference; T: RefType) : BOOLEAN is TRUE if and only if x is a member of T. T must be an object type or traced reference type.
- NARROW (x: Reference; T: RefType): T returns x after checking that x is a member of T. If the check fails, a runtime error occurs. T must be an object type or traced reference type.
- TYPECODE (T: RefType) : CARDINAL Every object type or traced reference type (including NULL) has an associated integer code. Different types have different codes. The code for a type is constant for any single execution of a program.

Readings and References

- Systems Programming with Modula-3 (SPwM3), edited by Greg Nelson.
- The Modula-3 report (html): http://research.compaq.com/SRC/m3defn/html/complete.html Most of the text and the examples in this lecture are taken from the language specification.