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

---

Enumeration Types

\[
\text{TYPE } T = \{id1, id2, \ldots, idn\}
\]

Subrange Types

\[
\text{TYPE } T = [Lo \ldots Hi]
\]

---

Predeclared Ordinal Types

- INTEGER, CARDINAL, BOOLEAN, CHAR.

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.
**Ordinal Types...**

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**

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\};
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.

Procedure Types

TYPE T =
PROCEDURE ( 
    VALUE x : INTEGER;
    VAR y : REAL;
    z : TEXT := "HELLO"; (* VALUE by default *)
    READONLY w : REAL
    : BOOLEAN (* Function return type. *)
    RAISES {SomeException} =
BEGIN
    ...
END P;

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

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.
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
    method_name method_sig := procedure_name
  OVERRIDES
    Overrides
END
```

- method_name must have been declared in the super-type.

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

- **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.

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

• T4 is statically wrong. (Why?)

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

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

- **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:**
  ```
  TYPE T = [0..255];
  U = BITS 8 FOR [0..255];
  ```

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.
**Statements — Assignment**

VAR

x : REFANY;
a : REF INTEGER;
b : REF BOOLEAN;

a := b; (* static error *)
x := a; (* no possible error *)
a := x; (* possible checked runtime error *)

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

**Statements — Assignment**

VAR

x : ADDRESS;
a : UNTRACED REF INTEGER;
b : UNTRACED REF BOOLEAN;

a := b; (* static error *)
x := a; (* no possible error *)
a := x; (* static error *)

- ADDRESS is a raw address without runtime type information. Hence no runtime type checks can be made.

---

**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.

**Statements — Assignment**

- 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.
Statements — Block Statement

Declarations
BEGIN
   Statements
END

   • Introduces a new scope.

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 — Exceptions

   • RAISE e raises an exception.

TRY
   Body
EXCEPT
   exception (argument) ==> handler
   exception (argument) ==> handler
   exception (argument) ==> handler
ELSE
   handler
END

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 — EXIT

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

Statements — TRY-FINALLY

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

\[
\text{TRY} \\
S_1 \\
\text{FINALLY} \\
S_2 \\
\text{END}
\]

Statements — RETURN

- **RETURN** raises the return exception in a procedure.
- Equivalent to (\( B \) is the body of the procedure):
  
  \[
  \text{TRY} \\
  B; \\
  \text{EXCEPT} \\
  \text{return-exception} => \\
  \text{END}
  \]

Statements — LOOP

- **LOOP** \( S \) **END** repeatedly executes \( S \) until it raises an EXIT-exception.
- Equivalent to:
  
  \[
  \text{TRY} \\
  S; S; S; ... \\
  \text{EXCEPT} \\
  \text{exit-exception} => \\
  \text{END}
  \]
Statements — **WHILE**

```plaintext
WHILE B DO
  S
END
```
equivalent to

```plaintext
LOOP
  IF B THEN
    S
  ELSE
    EXIT
  END
END
```

Statements — **RETURN e**

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

```plaintext
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 — **REPEAT**

```plaintext
REPEAT
  S
UNTIL B
```
equivalent to

```plaintext
LOOP
  S; IF B THEN
    EXIT
  END
END
```

Statements — **IF**

```plaintext
IF B_1 THEN S_1
ELSIF B_2 THEN S_2
  ...
ELSIF B_n THEN S_n
ELSE
  S_0
END
```
Statements — WITH

\[\text{WITH id} = e \text{ DO } S \text{ END} \]

\begin{itemize}
  \item id becomes an alias for the expression \(e\) in \(S\).
  \item \(e\) is evaluated once on entry.
  \item Like LET in SCHEME.
\end{itemize}

Statements — FOR

\[\text{FOR id} := \text{first} \text{ TO last \ BY step } \text{ DO } S \text{ END} \]

\begin{itemize}
  \item id is a read-only variable with the same type as \texttt{first} and \texttt{last}.
  \item \texttt{first, last} and \texttt{step} are executed once.
  \item If \texttt{step} is negative, the loop iterates downwards.
\end{itemize}

Statements — CASE

\[\text{CASE } e \text{ OF} \]

\begin{itemize}
  \item \(L_1 \rightarrow S_1\)
  \item \(L_2 \rightarrow S_2\)
  \item \(\ldots\)
  \item \(L_n \rightarrow S_n\)
\end{itemize}

\text{ELSE}

\[S_0\]

\text{END}

\begin{itemize}
  \item The \(L_i\)s are constant expressions that must not overlap.
\end{itemize}

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

\[\text{VAR i} := \text{ORD(first)}; \text{ done} := \text{ORD(last)}; \text{ delta} := \text{step}; \]

\begin{itemize}
  \item \(i\) is a read-only variable with the same type as \texttt{first} and \texttt{last}.
  \item \texttt{first, last} and \texttt{step} are executed once.
  \item If \texttt{step} is negative, the loop iterates downwards.
\end{itemize}
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 v1,...,vn:  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.

Statements — TYPECASE

\[
\text{TYPECASE } e \text{ OF} \\
T_1(v_1) \Rightarrow S_1 \\
| T_2(v_2) \Rightarrow S_2 \\
| \ldots \\
| T_n(v_n) \Rightarrow S_n \\
\text{ELSE} \\
S_0 \\
\text{END}
\]

- e is of reference type.
- The Ti’s are a subtype of the type of e.
- For the minimum i such that the type of e is a subset of Ti, Si is executed.

Statements — INC/DEC

- INC(v,n) is equivalent to
  \[
  \text{WITH } x = v \text{ DO} \\
  x := \text{VAL(ORD(X)+n, T)} \\
  \text{END}
  \]
- DEC(v,n) is equivalent to
  \[
  \text{WITH } x = v \text{ DO} x := \text{VAL(ORD(X)-n, T) END}
  \]
- 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.
Opaque Types – REVEAL

- A partial revelation: \texttt{REVEAL T <: U}. This reveals that \texttt{V} is a supertype of \texttt{T}. \texttt{T} has previously been declared an opaque type.
- A complete revelation: \texttt{REVEAL T = U}. This gives the final, concrete, type of \texttt{T}. It’s a (link-time) error if any type revealed as a supertype of \texttt{T} is not a supertype of \texttt{V}.

Opaque Types — Example

```plaintext
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;
```

- An importer of \texttt{I} can only allocate \texttt{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.

Recursive Declarations

- Modula-3 allows free order of declarations. The consequence is that declarations can be recursive. Some recursive declarations are illegal.
- A declaration \texttt{N = E} or \texttt{N : E} is recursive if \texttt{N} occurs in any partial expansion of \texttt{E}.
- A recursive declaration is legal if every occurrence of \texttt{N} in any partial expansion of \texttt{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.
**Interfaces**

- An interface has the form:
  ```
  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.

**Legal Recursive Declarations**

- \(TYPE\) \(List = \) 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**

- \(TYPE\) \(T =\) 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).

**Modules**

- An interface has the form:
  ```
  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 \(I\).

**Import Statement**

- \(\texttt{IMPORT}\ I\ \texttt{AS}\ J\) imports the interface whose global name is \(I\) and gives it the local name \(J\).

- \(\texttt{IMPORT}\ I\ \texttt{IS\ SHORT\ FOR}\ \texttt{IMPORT}\ I\ \texttt{AS}\ I.\)

- \(\texttt{FROM}\ I\ \texttt{IMPORT}\ N\) introduces \(N\) as the local name for the entity declared as \(N\) in the interface \(I\).
Example

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

Example...

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;
BEGIN
END Stack.

Modules...

This is illegal, since two names in exported interfaces coincide:

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:

INTERFACE I;
  PROCEDURE X(); ... 

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

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

```
INTERFACE StackRep; IMPORT Stack;
REVEAL Stack.T =
    BRANDED OBJECT item: REAL; link: Stack.T END
END StackRep.
```

```
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

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

- A generic module has the form

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

- An instance of G has the form

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

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

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

- An instance of G has the form

  ```
  INTERFACE I = G(A_1, \ldots, 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:

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

- 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.

**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.

**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.
```

```
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.
```
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...

- If r is a reference to a record, then \( r.f \) is shorthand for \( r^\cdot f \).
- 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.
- **SUBARRAY(a: Array; from, for: CARDINAL): ARRAY OF 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: \( R\{Bindings\} \) where R is a record type and Bindings is a list of keyword or positional bindings, exactly as in a procedure call.

Designators

- \( r^\cdot \) 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 \( a^\cdot[i] \).
- If r denotes a record, \( r.f \) denotes its f field.
A allocation operation has the form: \( \text{NEW}(T, \ldots) \) where \( T \) is a reference type.

- If \( T \) is declared as an opaque type, \( \text{NEW}(T) \) 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 \( \text{NEW} \) operation has the form: \( \text{NEW}(T, n_1, \ldots, n_k) \) 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 \( \text{NEW} \) operation has the form: \( \text{NEW}(T, \text{Bindings}) \)

### Type operations

- \( \text{ISTYPE} \ (x: \text{Reference}; T: \text{RefType}) : \text{BOOLEAN} \) is \( \text{TRUE} \) if and only if \( x \) is a member of \( T \). \( T \) must be an object type or traced reference type.
- \( \text{NARROW} \ (x: \text{Reference}; T: \text{RefType}) : T \) 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.
- \( \text{TYPECODE} \ (T: \text{RefType}) : \text{CARDINAL} \) 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.

**Readings and References...**


Most of the text and the examples in this lecture are taken from the language specification.