Reference Manual of the Programming Language School

Roberto Ierusalimschy
Noemi de La R. Rodriguez

February 28, 1996

Contents

1 Modules

The *module* is the compilation unit of School. The main purposes of modules is to allow separate compilation and to control visibility of names.

module	→	**module** *name* **is**
		[*mod-include*]
		{*definition*}
		[*constructor*]
		end [*name*] '.'
definition	→	[**private**] *mod-feature* | *class-operation*

A module defines *features* (types, classes, procedures, variables, and values; see Section 2). Because classes can be quite complex, their operation bodies (*class-operations*) are defined in the outer level of the module. However, they are not a module feature, but a part of a module feature (the class).

All module features are *exported* by the module, unless they are *private*. Variables and values are always private. All other features are non-private unless explicitly so declared. An exported feature may refer to a private type.

A module *imports* features from other modules by listing them in an include clause.

mod-include → **include** {*mod-name* ';'}$^+$

There can be no cycles in the inclusion hierarchy.

Whenever a module is included, all of its exported features are available inside the including module, through qualified names in the form *moduleName'featureName*.

The *whole name* of a feature is the name of the module where it is defined plus its name in this module.

All names directly (that is, without qualification) visible in any point of a module must be distinct.

2 Features

mod-feature	→	*variable-defs*
	|	*value-defs*
	|	*procedure-def*
	|	*type-def*
	|	*class-def*

All features defined in a module are visible from the point of definition until the end of the module. However, types can be referred to in other type definitions before being defined.

2.1 Variable and Value Definitions

variable-defs	→	**var** {*variable-def* ';'}$^+$
variable-def	→	*name* ':' *type* [':=' *exp*]
value-defs	→	**value** {*value-def* ';'}$^+$
value-def	→	*name* ':' *type* [':=' *exp*] | *name* ':=' *exp*

A variable definition creates a new variable with the given type. A value definition gives a name to an object. A Value is similar to a variable, but it refers to a fixed object through its life time. All variables and values defined in a module are created during module initialization (see Section 2.3) and exist until the program ends.

If *exp* is present, it is evaluated and the variable or value is initialized with it, when the module is initialized (see Section 2.3). The type of *exp*, if present, must be a subtype of *type*. If a value is defined without a type, its type is the type of *exp*. A value must be initialized in its definition or in the module constructor.

2.2 Procedures

procedure-def $\rightarrow$ **proc** *name* [*formal-type-params*] *signature*
    where-clause
  is
    block
  end [*name*] ';'

Procedures can be parameterized by types, that is, they can be polymorphic. This facility, together with *formal-type-params* and *where-clause*, is explained in 7.3.

Procedures can have multiple parameters, and can return multiple results.

signature $\rightarrow$ [*formal-params*] [*results*]
formal-params $\rightarrow$ '(' *formal-param* {',' *formal-param*} ')'
formal-param $\rightarrow$ [*name* ':'] *type*
results $\rightarrow$ '$\rightarrow$' *formal-params* | '$\rightarrow$' *type*

The syntax $\rightarrow$ *type* is syntactic sugar for $\rightarrow$ *(*Result *: type)*.

The semantics of procedure calls, including parameter passing, are explained in 4.6.

2.3 Module Constructor

A module constructor can be used to complete the initialization of a module.

constructor $\rightarrow$ **constructor is** *block* **end** ';'

The initialization of a module is done in the following way: first all included modules are initialized, in the order in which they appear in the include list. Then, module variables and values with initial expressions are initialized, in the order in which they are defined. Finally, the module constructor, if present, is executed. All values defined in a module without an initial expression must be initialized in the constructor.

Modules are initialized just once. Therefore, if an already initialized module appears in the include list of a module being initialized, its initialization is skipped.

2.4 Type Definitions

Types describe abstract interfaces of objects. The description of an object interface is composed of the signatures of the procedures these objects support. An include clause may be used to include all operations of other types in the new one. *Warning: this clause is NOT implemented.*

type-def $\rightarrow$ **type** *name* [*formal-type-params*] **is**
    [**include** {*type* ';'}$^+$]
    {*header* ';'}
  end [*name*] ';'
header $\rightarrow$ **proc** *name signature*

2.5 Class Definitions

A class describes an implementation for objects. This implementation includes the variables and values which comprise the state of the object, operations over this state, and constructors to create new objects with this implementation. The definition of a class states its instance variables, values, and the signatures of its operations and constructors. The bodies of operations and constructors are written in the outer level of the same module (see Section 3).

class-def	→	[**partial**] **class** *name* [*formal-type-params*] [':' *type*]
		where-clause **is**
		[*class-include*]
		{*class-feature*}
		end [*name*] **;**
class-include	→	**include** *class* [*with-clause*] ';'
class-feature	→	*value-defs*
	|	*variable-defs*
	|	*header* [**is deferred**] ';'
	|	*constructor-header* ';'
constructor-header	→	**constructor** *name* [*formal-params*]

A class may include another class. In this case, the class inherits all variables, values, and operations of the included class. Constructors are not inherited, although a constructor of the new class must call a constructor of the old one (see Section 3). Unless changed by a with-clause, all inherited variables and values are invisible inside the new class.

with-clause	→	**with** *rename* {',' *rename*}
rename	→	*name* | *name* **as** *name* | *name* **as** '_'

A with-clause may be used to rename an inherited feature (*name* **as** *name*), to make visible an otherwise invisible feature (*name*), or to make invisible an otherwise visible feature (*name* **as** _).

All feature names in a class must be distinct. In order to redefine an inherited feature, it first must be renamed or made invisible.

The collection of the headers of all procedures of a class comprise the *internal type* of a class. If a type is given in a class definition, this type is the *external type* of the class, and it must be a super-type of its internal type. If a type is not given, the external type of the class is its internal type. If a class includes another one, its internal type must be a subtype of the internal type of the included class. The external type of a class is always considered an opaque type (see Section 7).

Like global procedures, classes can be generic, that is, they can be parameterized by types.

Class constructors are used to create new objects of the class. Each new object has its own set of variables and values.

Classes can be declared *partial*. A partial class can have deferred procedures, that is, procedures without a body. Partial classes cannot be instantiated, being only used as superclasses for other classes. Partial classes can have constructors. Deferred operations are not inherited. Therefore, a subclass must redeclare all deferred methods of its superclass, redeferring them if needed.

3 Class Operations

Class operations define the bodies of constructors and procedures of a class.

class-operation → *class-constructor* | *class-procedure*

class-constructor → **constructor** *op-name* [*formal-params*] **is**
  [**include** *name* [*actual-params*] ';']
  block
end [*op-name*] ';'
class-procedure → **proc** *op-name* *signature* **is**
  block
end [*op-name*] ';'
op-name → *name* '.' *name*

All class operations are executed inside the context of an object, the *receiver* of the operation. Therefore, their statements can refer to instance values and variables, other operations of the same object, and *self* (see Section 5).

If a class has a superclass, its constructors must call a constructor from the superclass. Such "super" constructor is executed before the body of the local constructor, and is denoted by preceding its name and arguments with the keyword **include**.

A class constructor must initialize all class values defined without an initial expression.

Because class operations are defined outside the class definition, they must refer to the class they belong. This is done by preceding the operation name with the class name, separated by a period.

4 Statements

block → *statements*
statements → *statement* {';' *statement*}
statement → *local-dec*
| *assign*
| *loop*
| *exit*
| *if*
| *proc-call*
| *return*
| *empty-statement*
empty-statement →

4.1 Local Declarations

A local declaration declares a local variable or value.

local-dec → *local-var-dec* | *local-value-dec*
local-var-dec → **var** *name* ':' *type* [':=' *exp*]
local-value-dec → **value** *name* [':' *type*] ':=' *exp*

Local variables and values are visible from the point of declaration until the end of the inner block where they are declared.

A local name must be distinct from any other name visible in the point of declaration.

4.2 Assignment

assign → *name* ':=' *exp*

This command sets the current value of the variable or value *name* to be the object resulting from *exp*. The left side must be the name of a visible variable or value. The type of *exp* must be a subtype of the type of the variable or value.

Values can be assigned to only inside a constructor.

4.3 Loops

loop	→	**loop** *block* **end**
	|	**while** *exp* **loop** *block* **end**
exit	→	**exit**

A loop executes its block repeatedly.

An exit statement can only appears inside a loop. It terminates the inner loop enclosing it, and the control goes to the statement following the loop.

The form **while** *exp* **loop** *block* **end** is syntactic sugar for **loop if** *exp* **then** *block* **else exit end***;* **end**.

4.4 If

if	→	**if** *exp* **then** *block* {*elsif*} [**else** *block*] **end**
elsif	→	**elsif** *exp* **then** *block*

An if command first evaluates its expression. If the expression results in a value different from *nil*, the **then** block is executed. Otherwise the **else** block is executed. The construction **if** E **then** S **end** is syntactic sugar for **if** E **then** S **else end**. The construction **if** E **then** S **elsif** . . . **end** is syntactic sugar for **if** E **then** S **else if** . . . **end end**.

4.5 Return

return → **return**

A return statement can be used only in a procedure or constructor body. It interrupts the sequential control flow and returns to the caller.

4.6 Procedure Calls

proc-call	→	*identifier* [*actual-type-params*] [*actual-params*] [*actual-results*]
	|	*dynamic-call* [*actual-params*] [*actual-results*]
actual-params	→	'(' [*exp* {',' *exp*}] ')'
actual-results	→	'→' '(' *var* {',' *var*} ')'

The first syntax is used to call module procedures. The second form is used to call methods of an object, using dynamic-binding; see Section 6.

Actual type parameters are explained in Section 7.3.

Actual parameters are expressions. The types of the actual parameters must be subtypes of the formal ones. Actual results are variables. The types of the actual results must be supertypes of the formal ones.

Inside the body of a procedure, all formal parameters and results are local variables. Parameters are initialized with the value of the respective actual parameter. When the procedure returns, its results are assigned to the respective actual results.

5 Expressions

$$
\begin{array}{rcl}
\textit{exp} & \rightarrow & \textit{simple-exp} \\
 & | & \textit{if-exp} \\
 & | & \textit{constructor-exp} \\
 & | & \textit{closure} \\
\textit{simple-exp} & \rightarrow & \text{'(' } \textit{exp} \text{ ')'} \\
 & | & \textit{identifier} \\
 & | & \textbf{self} \\
 & | & \textit{exp-call} \\
 & | & \textit{constant}
\end{array}
$$

Expressions evaluate to objects. Any expression has a static type, and the type of any object resulting from an expression is always a subtype of the expression type. Parenthesis may be used to group expressions.

A single identifier, when used as an expression, must denote a visible variable or value. This variable or value may be a module feature, an instance feature, a local value/variable, a formal parameter, or a formal result. The result of this expression is the current value of the variable/value. The type of this expression is the declared type of the variable/value.

The expression **self** results in the receiver of a method. It can only be used in the body of a class operation. The type of **self** is the internal type of the class.

5.1 Calls

Procedures can be called inside an expression.

$$
\begin{array}{rcl}
\textit{exp-call} & \rightarrow & \textit{identifier } [\textit{actual-type-params}] \ \textit{actual-params} \\
 & | & \textit{dynamic-call } [\textit{actual-params}]
\end{array}
$$

The first syntax is used to call module procedures.[1] The second form is used to call methods of an object, using dynamic-binding; see Section 6.

Actual type parameters are explained in Section 7.3.

A procedure called as an expression must return a single result. This result is the value of the expression.

The type of a call is the (single) declared return type of the procedure.

5.2 Conditional Expressions

$$\textit{if-exp} \ \rightarrow \ \textbf{if } \textit{exp} \textbf{ then } \textit{exp} \textbf{ else } \textit{exp} \textbf{ end}$$

Conditional expressions are used to write non-strict expressions. A conditional expression evaluates its first expression. If the result is non nil, it evaluates its second expression, and the result of this expression is the result of the if. If the result is nil, it evaluates its third expression, and the result of this expression is the result of the if.

[1] Notice that in this case the parenthesis after the procedure name are mandatory, to distinguish calls from variable/value access.

The type of a conditional expression is the type of its second or third expression, if this type is a supertype of the type of the other expression. Otherwise the type of a conditional expression is Bool.

5.3 Infix and Prefix Operators

All prefix and infix operators are syntactic sugar for other language constructs. Below is a list of such operators, with their meaning. The other list shows the relative priorities of all operators, from higher to lower. All infix operators are left associative.

'−' *exp*	*exp.neg*
exp1 '*' *exp2*	*exp1.mul(exp2)*
exp1 '/' *exp2*	*exp1.div(exp2)*
exp1 '+' *exp2*	*exp1.add(exp2)*
exp1 '−' *exp2*	*exp1.sub(exp2)*
exp1 '=' *exp2*	*exp1.eq(exp2)*
exp1 '≠' *exp2*	**not** *(exp1=exp2)*
exp1 '<' *exp2*	*exp1.lt(exp2)*
exp1 '>' *exp2*	*exp1.gt(exp2)*
exp1 '≤' *exp2*	*exp1.le(exp2)*
exp1 '≥' *exp2*	*exp1.ge(exp2)*
exp1 '[' *exp2* ']'	*exp1.at(exp2)*
not *exp*	**if** *exp* **then** *false* **else** *true* **end**
exp1 **and** *exp2*	**if** *exp1* **then** *exp2* **else** *false* **end**
exp1 **or** *exp2*	**if** *exp1* **then** *true* **else** *exp2* **end**

. (method call)
− **not** (prefix)
* /
+ −
< > ≤ ≥ = ≠
and
or

5.4 Constants

constant → *int-const* \| *real-const* \| **nil** \| *string-const* \| *char-const*

5.5 Constructor Expressions

constructor-exp → *class* '!' [*name*] [*actual-params*]
class → *identifier* [*actual-type-params*]

Constructor expressions are used to create and initialize new objects. A constructor expression first creates a new object of the specified class, and then executes the given constructor on the new object. If the named class is generic, actual type parameters must be supplied. If the constructor name is omited, a constructor named *new* is called.

Inside the body of a constructor, all formal parameters are local variables. Parameters are initialized with the value of the respective actual parameter.

The type of a constructor expression is the external type of its class.

5.6 Closure Expressions

Warning: this feature is NOT implemented.

A closure is an abstraction of an operation, together with the environment to execute this operation. In School, this environment can be the receiver of the operation, or nil, if the operation is a module operation.

Closure expressions are used to create closures.

closure	→	**closure** '(' *closure-param* ')'
closure-param	→	*dynamic-call* | *identifier* [*actual-type-params*]

The first form makes a closure of a method call over a receiver. The second form makes a closure of a global operation.

The result of a closure expression is an object with only one operation, named *apply*. The signature of this operation is the same as of the closed operation. When executed, the operation *apply* calls the closed operation, passing its own parameters.

6 Dynamic-Binding

To call a class operation over an object, School always uses dynamic-binding (also called late-binding).

dynamic-call → *simple-exp* '.' *name*

The object resulting from the expression is the *receiver*. The operation to be called is the operation with the given name defined in the class of the receiver object.

The type of *simple-exp* must have an operation with the given name, and the signature of this operation is used for all type checking purposes.

7 The Type System

An important property of the type system of School is that types are used solely for program verification. It is impossible to change the semantics of a program by changing its type definitions, although that can easily change the validity of the program.

Types are names plus actual type parameters. A type has a signature, which is a collection of operation signatures. Moreover, a type can be classified as opaque. An *opaque* type has no subtypes other than NIL and itself.

The *signature* of an operation is composed by the list of parameter types and the list of return types.

7.1 Types

The type of an entity can be a formal type parameter, a defined type, or the external type of a defined class. In this case the class reference is preceded by the symbol '#', and the resulting type is opaque. If the type or class is generic, type parameters must be supplied.

type	→	*name*
	|	*identifier* [*actual-type-params*]
	|	'#' *class*

An exported feature in a module may refer to a private type. When a type is not visible, it is treated as an opaque type, with an empty signature.

7.2 Subtyping

Two types are considered equal if they have the same whole name and their actual type parameters are equal.

Given two types $T1$ and $T2$, the following algorithm defines if $T1$ is a subtype of $T2$, denoted as $T1 \prec T2$.

1. if $T1$ and $T2$ are equal, or if $T1$ is NIL, return true.

2. otherwise, if $T2$ is opaque, return false.

3. otherwise, check their signatures. For each operation O in $T2$, with signature

   $$(P_1, \cdots, P_n) \rightarrow (R_1, \cdots, R_m),$$

   there must be an operation with the same name O in $T1$, and with signature

   $$(Q_1, \cdots, Q_n) \rightarrow (S_1, \cdots, S_m),$$

   where for all $i \leq m$, $S_i \prec R_i$, and for all $j \leq n$, $P_j \prec Q_j$. During these checks, the algorithm assumes that $T1 \prec T2$.

7.3 Generic Features

Most features in a module can be *generic*, that is, parameterized by types.

formal-type-params → '{' *name* {',' *name*} '}'

A generic feature can only be used when instantiated. To *instantiate* a feature is to supply actual types for its parameter types.

actual-type-params → '{' *type* {',' *type*} '}'

Because types do not take part in the semantics of a program, the instantiation does not need to create any code, and is done only for type checking. For this purpose, instantiation of a signature is done by textual replacement of formal parameters by actual parameters.

Where-clauses specify valid actual types in an instantiation. They are used to check the correctness of an instantiation and the correctness of the generic feature. If a generic feature is correct, it will be correct with any valid instantiation.

where-clause	→	[**where** {*type-restriction*}$^+$]
type-restriction	→	*name* **has**
		{*header* ';'}
		end [*name*] ';'

An absent restriction is equivalent to a restriction with an empty signature.

If the generic definition has a where-clause, the actual types must satisfy it. That means that the signature of each actual type must be a subsignature of the correspondent "where" signature, instantiated with the actual parameters.

Inside a generic feature, a formal type parameter is assumed to have the signature given by its where-clause. Moreover, a formal type parameter is always assumed to be opaque.

8 Pre-defined Features

The following definitions are visible in all modules:

```
type Bool is
end Bool;

value false : Bool := nil;
value true : Bool := 1;

type Int is
   proc add (Int) → Int;
   proc sub (Int) → Int;
   proc mul (Int) → Int;
   proc div (Int) → Int;
   proc lt (Int) → Bool;
   proc gt (Int) → Bool;
   proc le (Int) → Bool;
   proc ge (Int) → Bool;
end Int;

type Char is
   proc lt (Char) → Bool;
   proc gt (Char) → Bool;
   proc le (Char) → Bool;
   proc ge (Char) → Bool;
   proc Ord → Int;
end Char;

proc Ascii (Int) → Char;

type Real is
   proc add (Real) → Real;
   proc sub (Real) → Real;
   proc mul (Real) → Real;
   proc div (Real) → Real;
   proc lt (Real) → Bool;
   proc gt (Real) → Bool;
   proc le (Real) → Bool;
   proc ge (Real) → Bool;
   proc Trunc → Int;
   proc Round → Int;
end Real;

proc Float (Int) → Real;

...
```