Inheritance system for the general purpose programming language.

Alexey Kanatov

[alexey.v.kanatov@gmail.com](mailto:alexey.v.kanatov@gmail.com)

Eugene Zouev

[eugene.zueff@gmail.com](mailto:eugene.zueff@gmail.com)

Innopolis University

**The high-level structure of the paper:**

1. Introduction
2. Types
3. Units
   1. Invariants
   2. Initialization procedures
4. Unit attributes
   1. Routines
   2. Pre- and postconditions
   3. Reference and value forms of objects
5. Compilation units
6. Generics
7. Constant objects
8. Statements
   1. Block
   2. Setters and convertors
9. Overloading
   1. Routine names
   2. Unit names
10. Type checks
    1. NULL safety
    2. Dynamic type checks
    3. “Nullable” types
11. Inheritance
12. Usage - modules
13. Unit extensions
14. Tuples and arrays
15. Concurrency
16. Conclusion

**Abstract**

The paper presents the SLang programming language, its basic concepts, and approaches. Ability to use procedural, object-oriented, and functional programming within type-safe context having multiple inheritance with single and multiple overriding, generics of different kinds as well as extended overloading and unique system of types and constant objects put SLang into the position to be used for a wide range of application development from simple scripts to complicated software systems.

**Keywords**

Object, type, unit, class, module, generics, block, compilation

[CCS](https://dl.acm.org/ccs/ccs.cfm?id=0&lid=0&CFID=1009033506&CFTOKEN=16747759) →  [Software and its engineering](https://dl.acm.org/ccs/ccs.cfm?id=10011007&lid=0.10011007&CFID=1009033506&CFTOKEN=16747759) →  [Software notations and tools](https://dl.acm.org/ccs/ccs.cfm?id=10011006&lid=0.10011007.10011006&CFID=1009033506&CFTOKEN=16747759) →  [General programming languages](https://dl.acm.org/ccs/ccs.cfm?id=10011008&lid=0.10011007.10011006.10011008&CFID=1009033506&CFTOKEN=16747759) →  [Language types](https://dl.acm.org/ccs/ccs.cfm?id=10011009&lid=0.10011007.10011006.10011008.10011009&CFID=1009033506&CFTOKEN=16747759) →  Multiparadigm languages

# INTRODUCTION

There are procedural programming, structured programming, object-oriented programming, functional programming, and concurrent programming paradigms. Can we have all of them work seamlessly together within one programming language – let’s SLang!

In this paper, a highly condensed overview of SLang is given. All examples will use SLang and necessary explanations will be given during the paper flow. Syntax constructions will be presented using simple notation based on the following convention. Where [term] means optional, {term} may be repeated zero or more times, term1 | term2 is the selection of term1 or term2, **bold** font is used to highlight keyword or special symbols.

We will start with the basics to construct a building from the basement to the top of the roof. So, we have a computer which at a very high level has CPU (central processing unit, one or many, their interconnections and architectures are out of interest so far) and memory (RAM – random access memory, set of cells of the same size with a unique address, whether RAM is split into sections for code (set of CPU instructions which can be executed) and data or any other scheme of RAM organization it is also out of scope). The key thing that object is defined as a region or set of regions in RAM plus code of operations which can be applied for the object via execution on CPU. That is all we need. So, we may have an arbitrary number of objects they can interact with each other applying (calling) different operations of each other passing arguments (data) with the call. Next is to define the notion of type as an important characteristic of every object during execution time (runtime). As type fixes the number of operations and their properties (signatures) as well as the size of RAM required to store the object. So, a type is an abstraction used to describe the structure and behavior of objects.

Authors rely on concepts that are well-known by a broad audience of programmers and terms like class or variable will be used without formal definitions. Some definitions will be given right now to simplify the understanding of examples. Unit – named set of members. Where a member can be a routine or an attribute. Routine stands for actions while attribute stands for data. If a routine returns some value as a result of its execution we call it a function otherwise a procedure. If an attribute can change its value during the program execution we call it a variable attribute (or simply variable) otherwise we call it a constant attribute (or simply constant). Unit is very similar to class and the difference is that the unit incorporates proprieties of classes and modules (modules were introduced in Ada (package), Modula-2 (module) – a generally available collection of data and routines with initialization) in one concept and the foundation for types. The usage of a unit allows having classes and modules depending on the nature of the usage. attribute: Type has the meaning that attribute called ‘attribute’ will have static type (type at the compilation time) ‘Type’. attribute **is** Type is equivalent to

attribute: Type **is** Type or

attribute: Type **is new** Type

and is in fact form of type inference (deduction) which allows omitting: Type part. An expression after ‘**is**’ is the initialization of the declared entity. SLang supports anonymous procedures – so, just a sequence of statements that can be treated as a program and can be compiled and executed, thus many examples will use this flow.

attribute **is** Type

attribute.foo ()

That is a correct program that has 2 statements: declaration with initialization (attribute will be created and be associated with an object of type Type) and second is a procedure call having attribute as a call target and foo as the procedure to be called. So, let’s continue with types.

# TYPES

There are 7 (seven) kinds of types – unit type, anchored type, multi type, detachable type, tuple type, range type, and routine type.

Unit type is the most commonly used kind of type. The key thing is that there is a source code of the unit which has an explicit definition of all attributes and routine of this unit – fixing the set of operations over objects of this type and size of objects of this type in memory. Example

**unit** A // Start of unit definition

**var** attribute: Type // Unit attribute

routine **do** … **end** // Unit procedure

**end** // A

a: A // a defined as having type A

Anchored type is the type which is the same **as** another entity has. It works as automatic overriding while inheriting (will explore it later) and allows not to repeat the exact type name. Example

b: **as** a /\* ‘b’ defined as having type the same as ‘a’ has \*/

Multi type states that objects of this type can be one of the types specified in the type definition. So, the set of operations which can be applied to such objects is an intersection of operation from all types included in the multitype definition. So, it allows to produce code which works with objects of already compiled units with no need for inheritance. Example

c: A | B /\* c may be assigned with objects of types A or B \*/

c := **new** A

c := **new** B

c.foo (expression) /\* Both types A and B must have routine foo with the proper signature for the expression to be compatible with both signatures. Exact definition of types compatibility will be given later \*/

Detachable type in the form of ”**?** UnitType” allows us to declare attributes with no value and such attributes can be initialized later with objects of Type or its descendants and dynamic type check can be applied. No unit members can be accessed without dynamic type check. Example

d: **?**A /\* d is just defined as having no value. So, d can not be used unless its type is checked at runtime\*/

**if** d **is** A **do** d.foo **end**

Tuple type defines a group of objects of different types specified in the type declaration. It is very close to C struct and Pascal RECORD. Example

e: (Integer, Real, String) /\* e is a group of values \*/

e := (5, 6.6, “Hello world!”)

(x1, x2) := SolveSquareEquation (a, b, c)

// Type of object (x1, x2) is (Real, Real)

Range type explicitly defines a set of possible values objects of this type may have. There are 2 kinds of this type. Example

f: 1..6 /\* f can have Integer values between 1 and 6 \*/

g: 1|3|5|7 /\* g can have odd Integer values between 1 and 7 \*/

f := g // Incorrect!

g := f // Incorrect!

Routine type defines objects which are routines and it means that activation (call or application) of the routine associated with the object can be done later. Example

foo (h: **rtn** (Type1, Type2): Type3) **do**

/\* foo can be called with routine object which has the type function with 2 arguments of types Type1 and Type2 retrurning objects of type Type3\*/

x **is** h (**new** Type1, **new** Type2)

**end**

foo (**rtn** (Type1; Type2): Type3 **do** **return** **new** Type3 **end**)

/\* That is a valid call to foo with inline function \*/

# Units

Any unit is a named collection of attributes or members. Such definition sets away routines but if to consider routines as constant attributes of routine type initialized with the routine signature and body then this definition becomes consistent. Of course, the unit has other characteristics related to inheritance and usage and they will be explored in the paper later.

UnitDeclaration: [**final**] [**ref**|**val**|**concurrent**|**abstract**|**extend**]

**unit** [Identifier](#Identifier) [[AliasName](#AliasName)] [[FormalGenerics](#FormalGenerics)] [[InheritDirective](#InheritDirective)]  
 [[UseDirective](#InheritDirective)]

{

[MemberSelection](#FeatureSelection)|

[InheritedMemberOverriding](#InheritedFeatureOverriding)|

[InitProcedureInheritance](#InitProcedureInheritance) |

[ConstObjectsDeclaration](#ConstObjectsDeclaration)|

[MemberDeclaration](#FeatureDeclaration)

}

[[InvariantBlock](#InvariantBlock)]

**end**

One may notice that unit is a central component and has a lot of elements. So, let’s review them one by one. Let's start with the unit header.

UnitDeclaration: [**final**] [**ref**|**val**|**concurrent**|**abstract**|**extend**]

**unit** [Identifier](#Identifier) [[AliasName](#AliasName)] [[FormalGenerics](#FormalGenerics)] [[InheritDirective](#InheritDirective)]  
 [[UseDirective](#InheritDirective)]

As one unit may inherit members from other units **final** specifier prevents inheritance from this unit. Implying no descendants for this unit.

**ref** | **val** – they specify the default form of objects which will be created using this unit as a type.

**val unit** Integer /\* by default we like all objects of type Integer to be numbers (values) not a reference to the number \*/

**…**

**end** // Integer

i: Integer **is** 5 /\* i is a value object \*/

ir: **ref** Integer **is** 5 /\* ir is a reference object\*/

If no object kind specifier is provided then the default kind of object is a reference one. And using **ref** | **val** while declaring an attribute, it is possible in a flexible manner to control the kind of object being created. This can be applied to any unit not only to Integer as in the example above.

The next is **concurrent**, it allows us to specify that objects of this unit will be processed (executed) by a processing element that is different from the one which is used for all objects which are not marked as concurrent. The processing element is a general term for a physical processor, thread, process, remote server, or whatever computing machine. We do not specify exactly how execution be done we just specify that execution will be done concurrently. Mapping between concurrent unit and actual physical executors is to be doen outside of the programming lanaguge and it is not described here.

**concurrent unit** Philosopher

/\* There 5 of them eating spaghetti \*/

**end**

If we like to ensure that there will be no objects created for the unit it is to be marked as **abstract**. Of course, if there are some abstract routines within the body of the unit it is not possible to create an object of this unit type. So, it is not mandatory to mark such units as abstract as the compiler knows this, but if one likes to prevent objects creation for some units with having all routines as non-abstract then marking the unit abstract will allow to make it. The example below

**abstract unit** AnArray [G]

And the last specifier is **extend,** it allows to extend already compiled unit with new members.

**unit** A

foo **do** … **end**

**end**

**extend unit** A

goo **do** … **end**

**end**

a **is** **new** A

a.foo

a.goo

So, the second call to routine goo is valid if and only if the A unit extension was provided.

It is essential to note that **final** will not work together with **abstract** as it is out of sense to create a unit when it is not possible to create objects of this unit and unit descendants are prohibited as well.

And now let’s explore what can be put after the unit name. Some programmers do not like Integer they prefer int or INTEGER, so for such purpose one may specify another alias name ([AliasName](#AliasName)) which can be used as the unit name

**val unit** Integer **alias** Int

As we follow the style guideline that unit names should start with the capital letter.

Next is the optional parametrization of the unit with some unit type, constant or routine. For such kind of parametrization, term genericity will be used. ([FormalGenerics](#FormalGenerics))

**abstract unit** AnArray [G]

where G is the name of the type which is to be provided to get particular instantiation of the unit. Please note that square brackets [] are no longer associated with array access.

**abstract unit** OneDimentionalArray [G **extend** Any **init** ()]

G can be constrained meaning that any type which will be used for instantiation is to be conformant to the type specified as a constraint. In the case of the example above it should be a descendant of Any. And if it is necessary to create objects of the formal generic type we need to know which initialization procedure to be used – in this example there should be a procedure without arguments.

**unit** Array [G **extend** Any **init** (), N: Integer] **extend** OneDimentionalArray [G]

Here we have two generic parameters and the second one is the constant of the type which is specified.

Next is [InheritDirective](#InheritDirective) which specifies from which units this unit will inherit members. Keyword extend is used as it is used in popular programming languages.

## Invariants

Unit invariant is a set of predicates that state when objects of this unit type and its descendants be consistent. It is a requirement to objects consistency – that is why the keyword ‘**require**’ is being used to highlight that. See example:

**abstract unit** Numeric

one: **as this abstract**

zero: **as this abstract**

// Definitions of \* and + are skipped

**require**

**this** = **this** \* one

zero = **this** \* zero

**this** = **this** + zero

end // Numeric

So, every numeric object of a type which is a descendant of Numeric should implement concepts of one (1) and zero (0) and should be consistent with the invariant stated in Numeric. So, if some operation is applied to an object of some type then after completing the operation the unit invariant is to be checked to ensure that object is still in the consistent state and ready again to perform new operations.

## Initialization procedures

And when an object is being created there should be a way to put it into a consistent stage which fully matches its invariant. That is why we need an initialization procedure (constructor or creation procedure in other programming languages) as the only task it has is to initialize all attributes of the unit which must get their initial values and match the unit invariant. The straightforward choice for the name was “**init**” and as the name of the initialization procedure is known it can be skipped when a new object is being created. So, here is a reduced example of the initialization procedure of unit Boolean

**val unit** Boolean

**init do**

data := 0xb

**end**

{} **var** data:

Bit [Platform.BooleanBitsCount]

**end** // Boolean

Variable attribute data that is not visible to the clients of Boolean is initialized with zero, interpreted as false. So, there are no default values – all units including basic ones explicitly define their default values.

b **is** Boolean

This means that object b will be created with the value false. This is a short cut for the declaration like this

b: Boolean **is** **new** Boolean.**init**()

Of course, a unit may have several init procedures and the programmer is to select the one which is required for the particular case.

# Unit attributes

The text of the section starts here.

## Routines

## Pre and post conditions

## Reference and value forms of objects

# Compilation units

There 3 (three) kinds of compilation units – unit declaration, standalone routine, and anonymous routine. The unit declaration was already explored, standalone routines are just routines that are put outside of any unit and anonymous routine is just a sequence of statements – a kind of script. All these kinds were already used in examples. So, if the target of compilation is to build an executable we have also 3 (three) options for the program entry point – naturally, any anonymous routine is an entry point. Any standalone routine with no arguments or with the single argument array of strings can be an entry point and any publically visible initialization procedure of a non-abstract unit with the same characteristics as standalone routine cab be an entry point. The way how the build process is described is not part of the language and is not explored here. But the build process description guarantees that every unit of the build has a unique name.

# Generics

The text of the section starts here.

# Constant objects

The text of the section starts here.

# Statements

There are 10 statements in SLang.

Statement: [Assignment](#Assignment) | [LocalEntityCreation](#LocalAttributeDeclaration) | [IfCase](#IfCase) | [Loop](#Loop) | **break** [“:”[Label](#Label)] | [MemberCallOrCreation](#FeatureCallOrCreation) | **?** [Identifier](#Identifier) |**return** [[Expression](#Expression)] |[HyperBlock](#HyperBlock) **end** | **raise** [[Expression](#Expression)]

attr := exression // Assignment

a **is** A

c: C **is new** C.init ()

/\* local entity declarations with mandatory initialization short and long forms\*/

b: **?** A /\* local entity explicitly non-initialised \*/

**?** b // b becomes non-initialized

// first form of if

**if** booleanExpression

**do** statements

**else** statements

**end** // if

// second form of if

**if** expression **is**

tag1: statements

tag2: statements

**else** statements

**end** // if

/\* general form of the block with optional assertions and exception handling clauses \*/

**require** predicate

**do** statements

**when** Type **do** statements

**when** entity: Type **do** statements

**ensure** predicate

**end** // do

/\* And here is an example of the loop with break and optional label\*/

:label **while** condition **do**

**break** label

**end** // loop

// Another form decorated with assertions

**require** predicate

/\* This is loop invariant, which si to be true for every iteration \*/

**do** statements

**while** condition

**ensure** predicate

/\* This is the assertion which is to be true after loop is fully executed \*/

**end** // loop

## Block

## Setters and convertors

# Overloading

The text of the section starts here.

## Routine names

## Unit names

# Type checks

The text of the section starts here.

## NULL safety

## Dynamic type checks

## “Nullable” types

# Inheritance

The text of the section starts here.

# Usage – modules

The text of the section starts here.

# Unit extenstions

The text of the section starts here.

# Tuples and arrays

The text of the section starts here.

# Concurrency

The key idea about concurrent execution is that the procedure call is an asynchronous call and the functional one is a synchronous one. |So, if we have a concurrent entity

p **is new concurrent** Type

p.doSomething () // That is asynchronous call

x **is** p.getSomething () /\* is the synchronous one. Current execution flow will wait till the result of getSomething call be available \*/

And here is an example together with comments which shows how to use it.

philosophers **is** (

**new** Philosopher.init("Aristotle"),

**new** Philosopher.init("Kant"),

**new** Philosopher.init("Spinoza"),

**new** Philosopher.init("Marx"),

**new** Philosopher.init("Russell"))

// Array of philosophers

forks **is** (**new** Fork.init (1), **new** Fork.init(2), **new** Fork.init(3), **new** Fork.init(4), **new** Fork.init(5))

// Array of forks

**require**

philosophers.count = forks.count or else philosophers.count = 1 and then forks.count = 2

/\* The task is valid if number of forks is equal to the number of philosophers or if there is only one philosopher then there are 2 forks available \*/

**do end**

**while** true **do** /\* Let them eat forever\*/

**while** seat **in** philosophers.lower **..**

philosophers.upper

**do**

StandardIO.put ("Philosopher '" + philosophers (seat).name + "' is ready for lunch\n")

eat (philosophers (seat), forks (seat), forks (**if** seat = philosophers.upper **then** forks.lower **else** seat + 1)

**end** // loop

**end** // loop

eat(philosopher: Philosopher; left, right:Fork) **do**

/\* This procedure has 3 conccurrent arguments. Call to this procedure creates a crtical section and control flow eneters this section only when all concurrent arguments are exclusively available – philosopher be able to grab both forks at the same time. No deadlocks scheme \*/

StandardIO.put ("Philosopher '" + philosopher.name + "' is eating with forks #" + left.id + " and #" + right.id + "\n")

**end** // eat

**concurrent unit** Philosopher

name: String

**new** init (aName: **as** name) **do** name := aName **end**

**end**

**concurrent unit** Fork

id: Integer

**new** init (anId: **as** id) **do** id := anId **end**

**end**

# Conclusion

# RELATION BETWEEN UNIT AND CONSTANT

## At first, it is necessary to define what the unit is. So, a named group of members which can be routines or attributes is a unit. So, it is very close to class in terms of definition as it can also inherit members from parent (base) units, can define type and in addition to the class can be used as a module – singleton object which can be shared. So, unit is 3 in 1 – class, module and type. Looking more carefully at attributes one may notice that some of them could be variable – change they values at run-time and some can be only initialized and then they keep their value unchanged during the execution of the program. The latter case is the constant case. So, more formal definition should look like data entity which will keep its value unchanged after initialization. Another definition can be formulated like constant entity can be in the left side of the assignment statement. Of course initialization phase can be performed at runtime and at compiler time too. See examples below

## const a is 5 /\* a can be initialized at compile time as it is being initialized with constant expression \*/

## const b is new B () /\* b will be initialized at runtime as expression after ‘is’ can be evaluated only at execution time \*/

So, unit may contain constant attributes. And as constant entities are data entities they have the type which is defined by the proper unit. Thus every constant entity has name, type and value and the only difference from a variable that it cannot change its value during execution.

# KINDS OF CONSTANTS

## Unit may have constant attributes and that is the first kind of constants. Standalone routine as well as unit member-routine may have local constant, such constant also belong to the same kind. They all may have arbitrary type. So, let’s call them arbitrary constants. Other kind of constants is constants which are explicitly declared within the unit and they all have the type of the unit they are declared in. These constants are to be called as unit constants. And they can be of several kinds as well. The first kind when we specify the name of the constant and optionally initialization procedure which sets the constant into consistent state. The second is the range expression which defines the rules how all constants can be defined. And the third is the regular expression which defines all the constants. Let’s consider the graph which highlights the constant objects hierarchy and then examples in the SLang programming language. Figure x.

## Please see the example with comments.

## unit A

## const attribute is 5 /\* type of attribute is inferred as Integer based on the type of constant object 5 \*/

## const is

## const1, const2

## /\* Both constant objects const1 and const2 have type A\*/

## end

## end // A

# CONSTANT IS AN OBJECT (creation)

# PRACTICAL USAGE

# CAVEATS (inheritance, import, Real)

# DEEP VS. SHALLOW

## As the object which is constant may have a complicated structure, it may refer to other sub-objects the question what is constant and what can be altered is raised. Shallow constantan status implies that only the top level of object stays constant, while deep implies recursive top-down status of constant object. At first it is necessary to give basic definitions. And

# CONCLUSION

# Backbone - two fundamental constants

When we start learning computer science, we start with 2 simple idioms – 0 and 1 (zero and one). Generalizing we may state that we have 2 signs circle and bar and start defining everything in the digital world combining these signs into sequences and giving different interpretation of such chains. Binary digit (bit) was selected as a term to cover this. So, in fact we have defined some unit Bit which has 2 constant objects Bit.0b0 and Bit.0b1. Notation can be different – for example Bit.0 and Bit.1 or Bit.0b and Bit.1b but to stay with the most widely used C-style languages we will proceed with the form Bit.0b0 and Bit.0b1.

* Every unit may define all known constant objects using const is
* Integer.1 is valid constant object of type Integer
* To skip unit name prefix apply use const

val unit Integer extend Integer [Platform.IntegerBitsCount] …

end

val unit Integer [BitsNumber: Integer] extend Numeric, Enumeration is

const minInteger is - (2 ^ (BitsNumber - 1))

const maxInteger is 2 ^ (BitsNumber - 1) - 1

const is /\* That is ordered set defined as range of all Integer constant values (objects) \*/

minInteger .. maxInteger

end

init is

data := Bit [BitsNumber]

end

hidden data: Bit [BitsNumber]

invariant

BitsNumber > 0 /\* Number of bits in Integer must be greater than zero! \*.

end

abstract unit Any use const Integer, Real, Boolean, Character, Bit, String is

end

////

unit WeekDay

const is Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday end

end

use const WeekDay foo (Monday)

foo (day: WeekDay) is

if day is

Monday .. Friday: StandardIO.put (“Work day – go to the office!\n”)

Saturday, Sunday: StandardIO.put (“WeekEnd – do what you like!\n”)

end

end

unit A

const is a1.init, a2.init (T), a3.init (T1, T2)

end

init is end

init (arg: T) is end

init (arg1: T1; arg2: T2) is end

end

const x is A.a1

y is A.a2

Регулярное выражение имеет вид:

{ шаблон повторения } условие повторения

Любое регулярное выражение при обработке выдает последовательность сущностей через запятую – т.е. некоторый список. Т.е. это по сути внутренность кортежа.

Виды “шаблон повторения”:

{ a } - просто переменная

{123} – просто константа

{‘a’} {“string”}

{a | b | c} – т.е. повторятся могут разные сущности

Сущность, стоящая внутри фигурных скобочек будет повторятся.

Как она повторяется задается условием повторения или шагом внутри шаблона повторения

Виды “условие повторения”:

Целая константа – сколько раз повторяется {6} 3 – повторить 6 три раза дает список 6, 6, 6 – по сути это внутренность кортежа.

{a + b} <= x дает последовательность a, a+ b, a+2\*b, …. a+ i\*b <= x т.е. последовательность с шагом ограниченная значением х

{a} 0+ - неограниченная последовательность. Пустая или произвольной длины a или a, a или a, a, a и так далее

Строка это

{Character.minCharacter |.. Character.maxCharacter} 0+

такое регулярное выражение задет любую строку.

val unit Bit

const is

0b0, 0b1

end

end

val unit Bit [N: Integer] is

const is

{Bit.0b0 | Bit.0b1}N

end

end

unit String [N:Integer] extend AString, Array [Character, N] is

/// String with fixed length

const is

"{Character.minCharacter |.. Character.maxCharacter}N"

end

end

unit String extend AString is

/// Variable length String

const is

"{Character.minCharacter |.. Character.maxCharacter}0+"

end

end

Это полный вынос мозга? :-)

Перечислимые типы или константные объекты.

Понятие перечислимых типов позволяет поднять уровень абстракции на чуть более высокий уровень, чем просто пользование константами и следующий шаг – это понимание того, что надо констант – это набор константных объектов некоторого типа. И в общем случае тип может быть любой. Таким образом, любой юнит на базе, которого порождается тип может задать все известные на момент описания юнита константные объекты данного юнит типа. Для целого типа это задается следующим примером

val unit Integer

const

Platform.minInteger .. Platform.maxInteger

end

end

Таким образом, запись Integer.1 есть полное наименование константного объекта 1, для того чтобы не писать префикс имени юнита необходимо включить константы в тот юнит или фрагмент кода, где это необходимо

abstract unit Any use const Integer, Real, Boolean, Character, Bit, String

end

Вот такая конструкция use const позволяет использовать константные объекты из указанных юнитов без префиксов имен юнитов в данном юните и во всех его потомках. Таким образом, константы, к которым мы так привыкли можно использовать без префиксов в любом юните, так как юнит Any is the base unit for all others.

unit WeekDay

const

Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

end

end

Теперь можно работать с константами типа WeekDay

day: WeekDay is WeekDay.Monday

if day is

WeekDay.Monday .. WeekDay.Friday: StandardIO.putString (“Work day – go to the office!\n”)

WeekDay.Saturday, WeekDay.Sunday: StandardIO.putString (“WeekEnd – do what you like!\n”)

end

А для того, чтобы избежать обязательного префикса, достаточно написать use const WeekDay и после этого тот же самый пример станет короче, но не менее выразителен

use const WeekDay

day: WeekDay is Monday

if day is

Monday .. Friday: StandardIO.putString (“Work day – go to the office!\n”)

Saturday, Sunday: StandardIO.putString (“WeekEnd – do what you like!\n”)

end

А так как юнит может иметь процедуру инициализации, то пример более общего случая будет выглядеть так

unit A

const A1.init, A2.init (T), A3.init (T1, T2) end

init is end

init (arg: T) is end

init (arg1: T1; arg2: T2) is end

end

И теперь уже можно использовать тип A и с константные объекты данного типа A1, A2 и A3.

Например, таким образом

use const A

foo (A1)

foo (argument: A) is

if argument

A1: // Некоторое действие(я)

A2 .. A3: // Другое действие(я)

end

end

Понятие константных объектов является более общим, чем перечислимые типы и позволяет поддержать привычный функционал перечислимых типов и распространить возможность сравнение по образцу (pattern matching) для любых типов.

------------------

Ниже представлены примеры единиц компиляции:

foo **is**  // Это процедура  
 ...  
**end**

goo: Integer **is** // Это функция  
 ...  
**end**

**unit** A // Это поименованный контейнер  
 foo **is** //Он содержит процедуру  
 ...  
 **end  
end**

Следует кратко обсудить причины введения в язык понятия контейнера. В самом деле, в современных ЯП имеются

# Integer, logical and character constants

Несколько упрощенный и сокращенный вариант объявления контейнера (типа) Bit приводится ниже.

**val unit** Bit  
 **const is** 0b0, 0b1  
 **end** & (other: **as** **this**) =>  
 **if** **this** = 0b0 **then** 0b0   
 **elsif** other=0b0 **then** 0b0 **else** 0b1  
  
 | (other: **as** **this**) =>  
 **if** **this** = 0b1 **then** 0b1  
 **elsif** other=0b1 **then** 0b1 **else** 0b0  
  
 ^ (other: **as** **this**) =>  
 **if this** = other **then** 0b0 **else** 0b1  
  
 ~ => **if this** = 0b0 **then** 0b1 **else** 0b0  
  
 + (other: **as** **this**): **as** **this is** **if this** = 0b0 **then** **return** other  
 **elsif** other = 0b1 **then** **return** 0b1  
 **else** ... // Bit overflow  
 **end** **end** // +  
  
 - (other: **as** **this**): **as** **this is** **if this** = other **then** **return** 0b0  
 **elsif this** = 0b1 **then** **return** 0b1  
 **else** ... // Bit underflow  
 **end** **end** // -  
**end**

b0 **is ref** Bit // b0– ссылка на объект Bit

b1 **is val** Bit // b1– это значение типа Bit

b2 **is** Bit // Вид b2 будет тем, который  
 // задан в объявлении Bit

Следующее замечание касается секции **const is**. Семантика этой секции состоит в том, что в ней задаются все возможные (константные) значения типа, внутри которого эта секция указана. Иными словами, объекты, перечисленные в этой секции, по определению имеют тип контейнера. В примере объекты 0b0 и 0b1 существуют в пространстве имен контейнера Bit. Следовательно, чтобы обратиться к таким константным объектам, в общем случае следует использовать точечную нотацию с указанием имени контейнера и имени константного объекта через точку:

x **is** Bit.0b1

Понятно, что подобная форма записи загромождает текст программы и на практике не слишком удобна. Чтобы использовать более естественную и привычную нотацию, можно включить имя типа в текущий контекст программы и использовать константы непосредственно:

**use const** Bit  
y **is** 0b0

Задание знаков операций над объектами типа Bit дает возможность использовать традиционную инфиксную форму выражений. Вместе с тем, при необходимости можно задавать альтернативные формы для этих знаков, например:

& **alias** and (other: **as** **this**) => ...

Альтернативные обозначения можно использовать наравне со знаками:

a & b или a and b или a.and(b)

Заключительное замечание касается задания типа параметра и типа возвращаемого значения в виде конструкции **as this**. Такая запись задает тип текущего контейнера ***либо*** любой тип, производный от текущего. Тем самым обеспечивается механизм автоматического переопределения при наследовании.

Прочие детали приведенного выше фрагмента кода представляются очевидными.

Следующий шаг в строительстве системы типов – определение битовой шкалы (последовательности битов).

**val** **unit** Bit[N: Integer] **is  
  
 const is** // 0b(N)0 .. 0b(N)1  
 **end** := (other: Integer) **is** ... **end** ...  
  
 **init** **do** **while** pos **in** 0 .. N - 1 **do** this(pos) := 0  
 **end** **end**  
  
**require** N > 0 // Number of Bits is greater than zero  
**end**

Как видно из примера, битовая шкала определяется как обобщенный тип (generic type), параметризованный целочисленным параметром. Настройка этого типа целочисленной константой времени компиляции дает конкретный тип - битовую шкалу заданного размера. Совместное существование непараметризованного и параметризованного типов с одним и тем же именем в языке допускается. Объекты этих типов объявляются следующим очевидным образом:

x1 **is** Bit  
x2 **is** Bit[8]

Наибольшие затруднения связаны с заданием множества констант-значений битовых шкал. Перечислить все константы в общем случае невозможно, и в данном случае компилятор должен понимать (обладать «тайным знанием»), что последовательность определенных значков 0 и 1 с префиксом 0b будет образовывать константные объекты типа Bit[N].

Стоит обратить внимание на процедуру инициализации контейнера. Формально, семантика инициализации состоит в приведении объекта данного типа в состояние, удовлетворяющее его инварианту. В примере всем элементам шкалы присваивается значение 0b0.

Для контейнера Bit[N] задана операция := (присваивание), которая фактически определяет семантику преобразования типа. Тип параметра присваивания Integer. Такой оператор разрешает присваивания вида

b1 **is** Bit[16]  
b1 := 100

Тип Integer построен по принципам, аналогичным типу Bit: фактически, это семейство, состоящее из «простого» типа и обобщенного типа с тем же именем. Обобщенный тип может настраиваться целочисленной константой, образуя тем самым множество конкретных типов вроде Integer[16], Integer[2], Integer[64] и т.д. Обычный тип Integer не конкретизирует размер объектов, который для него установлен по умолчанию.

**use const** Platform **val unit** Integer

**extend** Integer[IntegerBitsCount] **init**(value: **as this**) **do**  
 data := value.data  
 **end** **init do** data := Bit[IntegerBitsCount]  
 **end  
end**

Как видно из примера, тип Integer наследует настроенный на некоторую константу обобщенный тип Integer[N]. Инициализаторы, определенные для Integer, дают возможность задавать нужные начальные значения объектам этого типа, а также инициализировать их по умолчанию:

i0 **is** 6  
i1 **is** Integer

# Real, string and bit constants

При разработке программного обеспечения возникает необходимость у программиста дописать в некоторый контейнер несколько новых подпрограмм или атрибутов, или

нового поведения.

# General scheme for constant objects.

Наследование как вид отношений между контейнерами дает возможность расширять возможности новых контейнеров, используя уже доступные свойства существующих. Основные проблемы, которые возникают при реализации наследования, – это конфликт имен, неоднозначность версий.

# How constant objects replace enumerations.

Проблема, связанная с неконтролируемым использованием нулевых указателей («пустых» или «повисших» ссылок), является одной из наиболее распространенных в практике

В заключение рассмотрим пример более практического характера, который показывает использование описанного механизма совместно с типовой параметризацией.

// Создаем список из одного элемента  
list1 **is** List[Integer](5)  
// Пустой список   
list2 **:** ?List[Integer]

list2 := list1

**if** list2 **is** List[Integer] **then**  
 // Добавляется еще один элемент  
 list2.add(128)  
 **while** list2 **is** List[Integer] **loop**  
 // Цикл работает, пока очередной list2  
 // имеет значение  
 StandardIO.put(" ",l2.item)  
 // Перейти к следующему элементу списка  
 list2 := list2.next   
 **end**  
**end**

**unit** List[G]  
 item**:** G  
 next**:** ?List[G]  
 setItem(other: G)  
 item := other  
 **end**  
 add(other: G)  
 next := List[G](other)  
 **end**  
  
 **init**(element: G)  
 item := element  
 **end**  
**end**

# СПЕЦИАЛЬНЫЕ ТИПЫ: ДИАПАЗОНЫ И ПЕРЕЧИСЛЕНИЯ

Эти типы базируются на понятии константных объектов и хорошей иллюстрацией служит тема дней недели.

**unit** WeekDay

**const is**

Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

**end**

isWorkDay: Boolean => **this in** Monday **..** Friday

isWeekEndDay: Boolean => **this in** Saturday **..** Sunday

**end** // WeekDay

**use const** WeekDay

workDay: Monday **..** Friday **is** Monday

/\* переменная workDay имеет тип диапазон значение типа WeekDay\*/

weekEnd: Saturday **|** Sunday **is** Saturday

/\* переменная weekEnd имеет тип перечисление значений типа WeekDay\*/

weekDay: WeekDay **is** Monday

workDay := weekDay // Ошибка компиляции

weekDay := workDay // OK

weekEnd := workDay // Ошибка компиляции

Таким образом тип диапазон задается двумя константными объектами, которые задают нижнюю и верхнюю границы диапазона, по сути это подтип типа константных объектов. Схожей концепцией является перечисление, когда мы в явном виде задаем все значения перечисления и тип каждого элемента перечисления есть общий тип константных объектов, использованных для формирования такого перечисления. Оператор in позволяет производить перебор элементов таким типов или проверять принадлежность какого-либо значения такому типу.

# ПРИМЕР ПАРАЛЕЛЛЬНОГО ПРОГРАММИРОВАНИЯ

Если контейнеры философ и вилка описать с префиксом **concurrent**, то тогда можно убрать во всем остальном тексте примера **concurrent**, но был специально выбран вариант, чтобы явно подчеркнуть, где требуется параллельность. Поддержка параллелизма базируется на концепции вычислительного элемента, который ассоциирован с каждым объектом типа **concurrent** и имеет свою очередь ожидания запросов к нему. Запросом на выполнение может быть либо доступ к атрибуту или обращение к подпрограмме. При этом обращение к функции – это запрос с ожиданием результата, а обращение к процедуре без ожидания – синхронный и асинхронный вызовы соответственно. Как указано в примере, синхронизация по данным производится за счет механизма вызова подпрограммы – все параметры **concurrent** типа должны быть полностью доступны и тогда происходит вызов – по сути вход в критическую секцию. Это общее описание языкового механизма поддержки параллельного выполнения, но при использования внешних библиотек может быть использован параллелизм вида OpenMP или компилятор может сам производить автоматическую параллелизацию и выполнять части программ или итерации циклов параллельно.

# Conclusion

В статье дан обзор основных понятий и концепций языка SLang, приведены примеры, дающие представление о программировании на данном языке и о его возможностях.

В настоящий момент полностью проработан синтаксис языка, семантика всех конструкций, идет работа над строгим описанием правил валидности конструкций, а также ведется разработка компилятора переднего плана (front-end) и генераторов кода для нескольких программных и аппаратных платформ, среди которых .NET, LLVM, Эльбрус и др. Проводятся подготовительные работы к раскрутке компилятора (bootstrapping): к переносу реализации на собственный входной язык.

Ожидается что можно будет приступить к разработке приложений для операционных систем Andrоid, Linux, Windows и ОС Эльбрус по готовности всего компилятора и прикладных библиотек, работа над которыми также ведется.

.

# REFERENCES

1. Clemens A. Szyperski: Import is Not Inheritance. Why We Need Both: Modules and Classes, ECOOP 1992.
2. The Python Language Reference, https://docs.python.org/3.3/reference/.
3. Martin Odersky, Lex Spoon, and Bill Venners: Programming in Scala, Second Edition, Artima Press, 2010.
4. International Standard: ISO/IEC 8652:2012 Information technology – Programming Languages – Ada.
5. Bertrand Meyer: Object-Oriented Software Construction, Second Edition. Prentice Hall. ISBN 0-13-629155-4.
6. N.Wirth: The Programming Language Oberon, http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf
7. The Swift Programming Language Reference: <https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AboutTheLanguageReference.html>.

**abstract unit** Day

**end**

**unit** WorkDay **extend** Day

**const**

Mon, Tue, Wed, Thur, Fri

**end**

**end**

**unit** DayOff **extend** Day

**const**

Sat, Sun

**end**

**end**

**unit** WeekDay **extend** WorkDay, DayOff

**end**

**use const** WorkDay, DayOff

d0: Day **is** Mon

d0 := Sat

d1: WorkDay | DayOff is Mon

d1 := Sat

N – число битов, выделенных для хранения Real.

m – число битов на мантиссу

o – число битов на порядок

m + n + 1 = N 1 бит на знак. 0 => +, 1 => -

maxReal = (2 \*\*m \* 10) \*\* (2\*\*o)

minReal =

Вот смотри, что ты предлагаешь

**val unti** Real

**init** (even, fractional: Integer)

**init** (mantissa, order: Integer; fuckTheSameInitNameDummyParameter: Bit)

**end**

a **is** Real.init (5, 5) // instead of a is 5.5

b **is** Real.init (4, 6, Bit.0b0) // instead of b is 4E6

**Regular expressions:**

**Регулярное выражение имеет вид:**

{ шаблон повторения } условие повторения

Любое регулярное выражение при обработке выдает последовательность сущностей через запятую – т.е. некоторый список. Т.е. это по сути внутренность кортежа.

**Виды “шаблон повторения”:**

{ a } - просто переменная

{123} – просто константа

{‘a’} {“string”}

{a | b | c} – т.е. повторятся могут разные сущности

Сущность, стоящая внутри фигурных скобочек будет повторятся.

Как она повторяется задается условием повторения или шагом внутри шаблона повторения

**Виды “условие повторения”:**

Целая константа – сколько раз повторяется {6} 3 – повторить 6 три раза дает список 6, 6, 6 – по сути это внутренность кортежа.

{a + b} <= x дает последовательность a, a+ b, a+2\*b, …. a+ i\*b <= x т.е. последовательность с шагом ограниченная значением х

{a} 0+ - неограниченная последовательность. Пустая или произвольной длины a или a, a или a, a, a и так далее

Строка это

{Character.minCharacter |.. Character.maxCharacter} 0+

такое регулярное выражение задет любую строку.

**val unit** Bit

**const is**

0b0, 0b1

**end**

**end**

**val unit** Bit [N: Integer] **is**

**const is**

{Bit.0b0 | Bit.0b1}N

**end**

**end**

**unit** String [N:Integer] **extend** AString, Array [Character, N] **is**

/// String with fixed length

**const is**

"{Character.minCharacter |.. Character.maxCharacter}N"

**end**

**end**

**unit String extend AString is**

/// Variable length String

**const is**

"{Character.minCharacter |.. Character.maxCharacter}0+"

**end**

**end**

Это полный вынос мозга? :-)

1. Введение
2. Две основные константы – основа мироздания.
3. Целые, логические и символьные константы.
4. Вещественные, строковые и битовые константы.
5. Общее понятие константных объектов.
6. Как константные объекты заменяют перечислимые типы.
7. Заключение.

Перечислимые типы или константные объекты.

Понятие перечислимых типов позволяет поднять уровень абстракции на чуть более высокий уровень, чем просто пользование константами и следующий шаг – это понимание того что надо констант – это набор константных объектов некоторого типа. И в общем случае тип может быть любой. Таким образом, любой юнит на базе, которого порождается тип может задать все известные на момент описания юнита константные объекты данного юнит типа. Для целого типа это задается следующим примером

**val unit** Integer

**const**

Platform.minInteger **..** Platform.maxInteger

**end**

**end**

Таким образом, запись Integer.1 есть полное наименование константного объекта 1, для того чтобы не писать префикс имени юнита необходимо включить константы в тот юнит или фрагмент кода, где это необходимо

**abstract unit** Any **use const** Integer, Real, Boolean, Character, Bit, String

**end**

Вот такая конструкция use const позволяет использовать константные объекты из указанных юнитов без префиксов имен юнитов в данном юните и во всех его потомках. Таким образом, константы, к которым мы так привыкли можно использовать без префиксов в любом юните.

**unit** WeekDay

**const**

Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

**end**

**end**

Теперь можно работать с константами типа WeekDay

day: WeekDay **is** WeekDay.Monday

**if** day **is**

WeekDay.Monday **..** WeekDay.Friday: StandardIO.putString (“Work day – go to the office!\n”)

WeekDay.Saturday, WeekDay.Sunday: StandardIO.putString (“WeekEnd – do what you like!\n”)

**end**

А для того, чтобы избежать обязательного префикса, достаточно написать **use const** WeekDay и после этого тот же самый пример станет короче, но не менее выразителен

**use const** WeekDay

day: WeekDay **is** Monday

**if** day **is**

Monday **..** Friday: StandardIO.putString (“Work day – go to the office!\n”)

Saturday, Sunday: StandardIO.putString (“WeekEnd – do what you like!\n”)

**end**

А так как юнит может иметь процедуру инициализации, то пример более общего случая будет выглядеть так

**unit** A

**const** A1.init, A2.init (T), A3.init (T1, T2) **end**

**init is end**

**init** (arg: T) **is end**

**init** (arg1: T1; arg2: T2) **is end**

**end**

И теперь уже можно использовать тип A и с константные объекты данного типа A1, A2 и A3.

Например, таким образом

**use const** A

foo (A1)

foo (argument: A) **is**

**if** argument

A1: // Некоторое действие(я)

A2 .. A3: // Другое действие(я)

**end**

**end**

Понятие константных объектов является более общим, чем перечислимые типы и позволяет поддержать привычный функционал перечислимых типов и распространить возможность сравнение по образцу (pattern matching) для любых типов.

Так как теперь a: Type есть описание константного атрибута, то все аргументы, которые описываются именно таким образом - константы. Далее я начал думать над понятием, а что есть константа. И пришёл к четкому пониманию, что константа - это константный объект(опять Америку открываю😊), и это значит что он не только не может быть в левой части оператора присваивания, но и над ним не могут вызывается его методы-рутины, которые меняют его внутреннее состояние. Rigid константный объект, это когда не только первый уровень неизмененен, но и все последующие. Т.е. константа вне зависимости от её вида **ref** or **val**, нельзя сделать ++. Нам не важен вид объекта ref он или val, семантика константности от этого не меняется. И есть всего два вида объектов изменяемые (**var**) и неизменяемые (const ,но это по умолчанию все, и ключевое слово для этого не нужно и **rigid**, полностью все дерево).

В С++ void f() const {} - это полный аналог safe foo do end , а **var** относится к каждому агрументу, и его отсутствие не даёт вызывать те рутины, что изменяют внутреннее состояние target of the call. Пример.

**unit** A

**var** attr: Type := (other: Type) **do**

/\*As this attribute has assignemtn procedure(setter), it can be assigned with := form \*/

attr := other

**end**

foo (arg: Type) **do**

arg.attr := Type /\*Compile time error as arg is a constant object!\*/

**end**

goo (**var** arg: Type) **do**

arg.attr := Type // OK! As we explicitly stated that arg is variable

**end**

**end**

i **is** 6

ir: **ref** Integer **is** 6

**var** j **is** 5

**var** jr: **ref** Integer **is** 5

i++ // Compile time error

j++ // OK!

ir++ // Compile time error

jr++ // OK!

Т.е. все рутины любого юнита делятся на два подкласса - те что изменяют данные объекта такого юнита и те что не изменяют (другими словами в теле руины есть присваивания в атрибуты данного юнита или нет- статически отлавливается) и следовательно если хочешь использовать рутины, что изменяют состояние объектов, то объекты не должны быть константными - т.е. надо явно написать **var**

А главное что семантика val & ref не влияет на константность объекта! val & ref влияют только на семантику of the default := operation

Ну очевидно, что код вида 5++ не имеет смысла, а если 5 заменить на имя константы 5, то как раз станет понятно о чем речь

five is 5

five++ // Compile time error

Аналогично в общем случае

entity0 **is** Type

entity0.someRotuineCallWhich ChangesEntityObject // Compile time error

**var** entity1 **is** Type

entity1.someRotuineCallWhich ChangesEntityObject // OK

Т.е. семантика var - не совпадает с семантикой var в языках Pascal and Modula-2 - совсем!

ну вроде разжевал почти полностью - осталось только табличку семантики присваивания приложить.

Очень приятно что циклы сразу стали чище

**while** i **in** 1..100 **do**

**end**

Внутри цикла (для каждой интерации) i константа - ей нельзя присваивать!

**while** **var** j **in** 1..100 **do**

**end**

А вот так нельзя :-) Запрещено ☺

**Attributes, routines, immutability**

As any computer has memory and memory logically consists of cells (bytes, words, etc) then the high-level image of a memory cell or cells is a data attribute or simply attribute. Attributes may be a part of the routine (procedure or function) and then we call them local attributes or locals, or they can belong to a unit, and then we call them unit attributes or just attributes or properties. Some parts of memory (regardless of the HW architecture) we interpret as a code and it can be executed, so, at a high-level, there are some “attributes” which should be executed as well. So, some attributes should be routines or better say constant attributes of the routine type. So, after all these definitions let’s consider examples that illustrate this. Let’s start with local attributes

foo **do**

l1: Type1 **is** expression1 /\* Local l1 will be treated as having Type1 and will be initialized with the value which is produced when expression1 evaluated with no errors. The static type of expression1 should conform to Type1 \*/

l2 **is** expression2 /\* Local l2 will get the type identical to the static type of expression2 – in other words, the compiler will automatically deduce the type of l2, typical example of type inference \*/

l3 **is** 5 // l3 is Integer initialized with value 5

l4 **is new** Integer.init (5)

**end**

How to deal with the variable number of arguments? The answer is rather straightforward – anything which is put in parenthesis is a tuple, so, we should declare a parameter of type tuple.

foo (args: ()) **do**

**while** arg **in** args **do**

/\* 1st option, arg will be of type Any deduced by the compiler \*/

StandardIO.put (arg)

**end**

**while** index **in** 1 .. args.count() **do**

// 2nd option access arguemnts using position

arg **is** args(index) // arg will be of type Any deduced by the compiler

StandardIO.put (arg)

**end**

**end**

**инициализацию полей кортежа - ты предлагал использовать : а я is так как у нас : всегда используется в описании. а не в выражении**

**т.е. вместо**

**a is (f1: 5; f2: "A string")**

**должно быть**

**a is (f1 is 5; f2 is " A string")**

**По сути это создание с инициализацией**

**И как всегда есть разница между сигнатурой - т.е. кортежем с описаниями и кортежным выражением**

**foo (f1: Integer; f2: String) do ... end**

**foo (f1: Integer; f2 is "Default value") do ... end**

**Это еще раз подчеркивает семантическую разницу между : и is**