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.

Duck typing in SLang. It is based on the same mechanism – dynamic type checking – conformance test. Let's consider the following example

**unit** Duck

fly **do**

print("Duck flying")

**end**

**end**

**unit** Sparrow

fly **do**

print("Sparrow flying")

**end**

**end**

**unit** Whale

swim **do**

print("Whale swimming")

**end**

**end**

**while** animal **in** (Duck, Sparrow, Whale) **do**

**if** animal **is** (fly ()) **do**

animal.fly

**end**

**end**

И сейчас еще подумав пришел к выводу, что не надо вводить has как ключевое слово внутри if. У нас есть проверка типа is и понятие кортежа и тогда пример модифицируется как

while animal in (Duck(), Sparrow(), Whale()) do

if animal is (fly()) do

animal .fly()

end

end

Т.е. после is в if может стоять кортеж, который описывает по сути набор свойств некоторого несуществующего юнита которому в нашем случае animal должен быть конформен, т.е. иметь все свойства указанные в кортеже. Более навороченный пример, который показывает всю мощь аппарата

if a is (f(T1), f(T2), f(T3, T4), attr: T5, goo(T1, T5):T5) do

/\* если а имеет 4 рутины с именами f и goo, и один атрибут

/т.е. по сути то что стоит в скобках - это кортеж который описывает некоторый тип, а не объект \*/

a.f(T1())

a.f(T2())

a.f(T3(), T4())

var t5 is a.attr

t5 := goo (T1(), T5())

end

Все это статически проверяется, а в рантайме идет работа cхожая с работой классического is Type

2. СЛанг базируется на понятии согласованности типов - type A is compatible with type B if and only if type A conforms to type B or it has conversion function from type A to type B

Пример, а то я опять слов понаписал 😊

unit A

end

unit B extend A

:= (other: A) do end

end

Здесь юнит B конформен юниту А, а юнит И имеет процедуру конвертации объектов типа А в тип В, следовательно следующий присваивания валидны

var a is new A

var b is new B

a := b // Это полиморфное присваивание - конформность работает

b := a // Это присваивание вида type cast - работает процедура конвертации

Просто чтобы тебя не пугать я всегда говорю о конформности типов, хотя надо говорить об их совместимости или согласованности (какой термин лучше?), имея в виду что присваивать можно если один объект конформен другому или есть рутина преобразования (конверторы) ...

Я надеюсь что ты помнишь про конверторы

unit A

:= (other: Type) do end // Это чтобы можно было объектам типа А присваивать объекты типа Type

:= (): Type do end // А это в обратную сторону присваивать объекта типа А объектам типа Type

end

И отсюда твой текст про то что две сущности a и b имеют разные типы, а главное что нельзя присваивать неверно

var a is (1, '2', "3456")

var b is (1, '2', "3456")

a := b // Можно типы а и b конформны друг другу