# Diseño

## Palabras reservadas

- $\bullet \quad Comentarios --$
- $\bullet \quad instrucciones$ 
  - for
  - in
  - while
  - do
  - if
  - else
  - print
  - read
- union
- :
- .
- . -
- \*
- .
- \_
- . .
- >
- <=
- >=
- ==
- /=
- ( )
- []
- or
- and
- not
- ::

## Decisiones

## Tipos

- Int
- Float

- Char
- String -- [Char]
   Array -- [Tipo] or Tipo[n] or [Tipo|n]

## Lenguaje de programación SUPERCOOL

SUPERCOOL es un lenguaje de programación de uso general desarrollado para la cadena de Lenguajes de Programación de la USB. El lenguaje es imperativo, con funciones, recursividad, comentarios, y de más.

## Estructura de un programa

Un programa en SUPERCOOL tiene la siguiente estructura:

```
[<instrucción>]
```

Es una lista de instrucciones a ejecutar una tras otra. Un ejemplo de un programa básico en SUPERCOOL es:

```
def plus(a, b) :: Int -> Int {
    return a + b
}

for i in [1..10] do {
    print("numero: " ++ string(plus(1,i)) ++ "\n");
}
```

#### **Identificadores**

Un identificador en SUPERCOOL consiste de una cadena de caracteres de cualquier longitud que comienza por una letra minúscula (a-z), y es seguido por letras minúsculas (a-z), letras mayúscula (A-Z), dígitos (0-9) o el caracter guión bajo (\_).

## Tipos de datos

Se dispone de los siguientes tipos de datos:

- Void para funciones que no devuelven valores (aka. procedimientos).
- Int números enteros con signo de N(32/64) bits.
- Bool representa un valor booleano o lógico, es decir true o false.
- Float números flotantes de N bits, precisión y tal...

- Char caracteres, UTF-8.
- String cadenas de caracteres, escencialmente [Char]
- [Array] arreglos de los otros tipos, no se permiten [Void]
- def(id, id,...) :: Firma funciones, debe especificarse los tipos de entrada y salida.

#### Instrucciones

#### Instrucción vacia

Instrucción que no hace nada, *noop*. No tiene sintaxis. Un ejemplo para ilustrar su uso:

;;

En el ejemplo hay dos operaciones noop.

### Asignación

```
<variable> = <expresión>
```

Ejecutar esta instrucción tiene el efecto de evaluar la expresión del lado derecho y almacenarla en la variable del lado izquierdo. La variable tiene que haber sido declarada previamente y su tipo debe coincidir con el tipo de la expresión, en caso contrario debe arrojarse un error.

## Bloque

Permite colocar una secuencia de instrucciones donde se requiera una instrucción. Su sintaxis es:

```
<instrucciónN-1>;
<instrucciónN>
}
```

Una secuencia de instrucciones separadas por ;. Nótese que se utiliza el caracter ; como separador, no como finalizador, por lo que la última instrucción de un bloque **no debe** terminar con ;, pero puede gracias a la *instrucción vacía*.

Colocar  $\}$  trae implicitamente un ; al final, para poder hacer cosas del estilo: x = 2;  $\{y = x;; \} x = 3$  sin tener que colocar ; despues de cerrar el bloque.

#### Declaración

Declara una variable para el alcance actual.

```
<Tipo> <identificador>;
```

Se escribe primero el tipo de la variable a declarar y luego el identificador de esta.

#### Declaración de funciones

Declara una función, especificando parametros de entrada y de salida.

Nótese que la definir una función no obliga la implementación inmediata, pero debe ser implementada luego. La <firma> especifica la entrada y salida de la función, para cada entrada debe haber una especificación en la firma y una extra señalando la salida. Un ejemplo es:

```
def iguales(a, b) :: Int -> Int -> Bool {
    return a == b
}
```

Podemos ver que la entrada consta de dos Int y tiene una salida de Bool

#### Entrada

```
read/scan ???
```

#### Salida

```
write/print ???
```

#### Condicional

**PENSAR BIEN**: Palabra then es **innecesaria**, trae discrepancia con el resto del lenguaje.

Condicional típicio if then else. La condición debe ser del tipo Bool y en caso de ser cierta, se ejecuta la instrucción siguiente al then, sino se ejecuta la instrucción siguiente al else (en caso de haber).

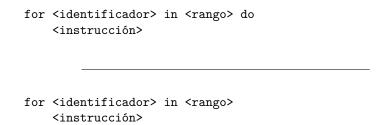
## Condicional por casos

Condicinal por casos, case.

```
case <expresión> of
  <expresión1> -> <instrucción1>
  <expresión2> -> <instrucción2>
   ...
  <expresiónN> -> <instrucciónN>
end
```

**PENSAR BIEN**: Palabra **end** es **necesaria**, trae discrepancia con el resto del lenguaje.

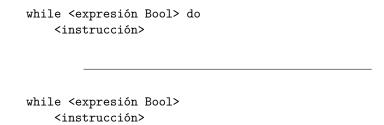
#### Iteración determinada



**PENSAR BIEN**: Palabra do es **innecesaria**, trae discrepancia con el resto del lenguaje.

El campo para <rango> debe ser del estilo [Int..Int], puede ser con identificadores. El <identificador> puede ser modificado dentro del for.

#### Iteración indeterminada



**PENSAR BIEN**: Palabra do es **innecesaria**, trae discrepancia con el resto del lenguaje.

## Reglas de alcance de variables

Para utilizar una variable primero debe ser declarada o ser parte de la variable de iteración de una instrucción for. Es posible anidar bloques e instrucciones for y también es posible declarar variables con el mismo nombre que otra variable en un bloque o for exterior. En este caso se dice que la variable interior esconde a la variable exterior y cualquier instrucción del bloque será incapaz de acceder a la variable exterior.

Dada una instrucción o expresión en un punto particular del programa, para determinar si existe una variable y a qué bloque pertenece, el interpretador debe partir del bloque o for más cercano que contenga a la instrucción y revisar las variables que haya declarado, si no la encuentra debe proceder a revisar el siguiente bloque que lo contenga, y así sucesivamente hasta encontrar un acierto o llegar al tope.

### **Expresiones**

Las expresiones consisten de variables, constantes numéricas y booleanas, y operadores. Al momento de evaluar una variable ésta debe buscarse utilizando las reglas de alcance descritas, y debe haber sido inicializada. Es un error utilizar una variable que no haya sido declarada ni inicializada.

Los operadores poseen reglas de precedencia que determinan el orden de evaluación de una expresión dada. Es posible alterar el orden de evaluación utilizando paréntesis, de la misma manera que se hace en otros lenguajes de programación.

#### Expresiones con enteros

Una expresión aritmética estará formada por números naturales (secuencias de dígitos del 0 al 9), llamadas a funciones, variables y operadores ariméticos convencionales. Se considerarán la suma +), la resta-), la multiplicación \*), la división entera/), módulo '%) y el inverso (- unario). Los operadores binarios usarán notación infija y el menos unario usará notación prefija.

La precedencia de los operadores (ordenados comenzando por la menor precedencia) son:

- +, binario
- \*, /, %
- - unario

Para los operadores binarios +, -, \*, / y % sus operandos deben ser del mismo tipo. Si sus operandos son de tipo Int, su resultado también será de tipo Int.

### Expresiones con booleanos

Una expresión booleana estará formada por constantes booleanas (true y false), variables, llamadas a funciones y operadores booleanos. Se considerarán los operadores and, or y not. También se utilizará notación infija para el and y el or, y notación prefija para el not. Las precedencias son las siguientes:

- or
- and
- not

Los operandos de and, or y not deben tener tipo Bool, y su resultado también será de tipo Bool.

También hay operadores relacionales capaces de comparar enteros. Los operadores relacionales disponibles son menor <, menor o igual <=, igual ==, mayor o igual >=, mayor >) y desigual /=. Ambos operandos deben ser del mismo tipo y el resultado será de tipo Bool. También es posible comparar expresiones de tipo Bool utilizando los operadores == y /=. Los operadores relacionales no son asociativos, a excepción de los operadores == y /= cuando se comparan expresiones de tipo Bool. La precedencia de los operadores relacionales son las siguientes:

- <, <=, >=, >
- ==, /=

### Conversiones de tipos

Las siguientes funciones están embebidas en el lenguaje para convertir tipos:

• toInt :: Float -> Int

• toFloat :: Int -> Float

Ambas funciones tienen una sola entrada y una salida.

## Comentarios y espacios en blanco

En SUPERCOOL se pueden escribir comentarios de una línea al estilo de Haskell. Al escribir -- se ignorarán todos los caracteres que lo proceden en la línea.

El espacio en blanco es ignorado de manera similar a otros lenguajes de programación, es decir, el programador es libre de colocar cualquier cantidad de espacio en blanco entre los elementos sintácticos del lenguaje.