Lenguaje de programación SAPPHIRE

SAPPHIRE es un lenguaje de programación de uso general desarrollado para la cadena de Lenguajes de Programación de la USB. El lenguaje está parcialmente inspirado en Ruby.

SAPPHIRE es imperativo, fuertemente tipado, con funciones, iteraciones indeterminadas y acotadas, recursividad, comentarios, con soporte para bloques anidados de instrucciones, unioness y estructuras arbitrariamente anidadas y más.

Adicionalmente tiene un tipo de datos que representa un rango de enteros, selector n-ario (case) y arreglos multidimensionales de segunda clase.

A CONSIDERAR: Potencialmente funciones de segunda clase por medio de apuntadores

Programa

Ejemplos:

```
write "hello world!\n"
```

```
main
    write "hello world!\n"
end
```

Sintaxis:

```
<stats..>
```

```
[<funcs..>]

main
    <stats..>
end

[<funcs..>]
```

Es una lista de instrucciones a ejecutar una tras otra. Cada instrucción está terminada por punto y comas (`;`) o saltos de línea, *newlines*. **Qué pasa si ponemos *backslash* al final de una línea?**

Estructura lexicográfica

Identificadores

Ejemplos:

```
foo
fooBar_baz
```

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

Comentarios

Ejemplos:

```
-- esto es un comentario
```

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

Palabras reservadas y Símbolos

Las palabras reservadas son las siguientes

```
main, begin, def, as, end, return
true, false, or, and, not
if, then, else, unless, case, when
for, in, while, do, until, break, continue
print, read
Void, Int, Bool, Float, Char, String, Array, Range
toInt, toFloat, toString, length

::, ->
--
;
,
=
[, ]
+, -, *, /, %, ^
(, )
```

```
==
/= | !=
>, <, >=, <=
```

COMPLETAR Y ORDERNAR Union??? Struct???

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, esencialmente `[Char]`
- **`[Array]`** arreglos, no se permiten `[Void]`. Se permiten arreglos de arreglos.
- **`def id ::  firma`** funciones, debe especificarse los tipos de entrada y salida.
- **`Union`** unions arbitrarimente anidados, equivalentes a los unions de `C`.
- **`Struct`** structs arbitrarimente anidados, equivalentes a los structs de `C`.
- **`Range`** rangos de enteros.

- **`{Range}`** enums, si es de enteros o elementos naturamente ordenados se puede usar `..`, sino se especifica el orden listando cada elemento.

El espacio de nombres definido para los tipos de datos es disjunto del espacio de nombres de los identificadores, además todos los tipos de datos empiezan por una letra mayuscula.

Instrucciones

Instrucción vacia

Ejemplos:

```
;;
```

Instrucción que no hace nada, *noop*. En el ejemplo hay dos operaciones *noop*, una al principio y la otra entre los dos punto y coma (`;`).

Asignación

Ejemplos:

```
foo = bar
foo[0] = bar
```

Sintaxis:

```
<var> = <expr>
```

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.

Asignación propia

Ejemplos:

```
foo += 42
```

Sintaxis:

```
<var> <op>= <expr>
```

Esta instrucción es equivalente a `<var> = <var> <op> <expr>`. El `<op>` puede ser cualquiera de los siguientes: `+`, `-`, `*`, `/`, `%`, `^`, `and`, `or`.

Bloque

CAPAZ NO SE NECESITA LITERALMENTE UNA INSTRUCCIÓN BLOQUE

Ejemplos:

```
begin
    x = 2
    begin
        y = 3; print x + y
    end
end
```

Sintaxis:

```
begin
    <stats..>
end
```

Permite colocar una secuencia de instrucciones donde se requiera *una* instrucción. Permite anidarlos arbitrariamente.

Declaración

Ejemplos:

```
Bool valid
Int num, index
```

Sintaxis:

```
<Tipo> <id> [, <ids..>]
```

Declara variables para el *alcance* actual.

Se escribe primero el `Tipo` de las variables a declarar y luego una lista de identificadores.

Declaración de funciones

Ejemplos:

```
def iguales(a, b) :: (Int, Int) -> Bool as
    return a == b
end
```

es equivalente a:

```
-- definición
def iguales :: (Int, Int) -> Bool

    -- ...código...

-- implementación
def iguales(a, b) as
    return a == b;
end
```

Sintaxis:

```
def <id> :: <firma>

def <id>(<lista de entradas>) :: <firma> as
    <stats..>
end
```

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

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

Nótese que la definir una función no obliga la implementación inmediata, pero debe ser implementada luego, en caso de no hacerlo se lanzaria un error si intenta hacerse una llamada a dicha funcion. 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:

Retorno de valores

Ejemplos:

```
return 3
```

Sintaxis:

```
return <expr>
```

Instrucción `return` típica.

Entrada

Ejemplos:

```
read valid, num
```

Sintaxis:

```
read <ids..>
```

Instruccion encargada de la lectura de datos. Los `<ids..>` sería una o más variables previamente declaradas. Dichas variables solo pueden ser de alguno de los tipos de datos primitivos del sistema (`String`, `Char`, `Int`, `Float`, `Bool`, `Range????`).

Salida

Ejemplos:

```
print index, num
```

Sintaxis:

```
write/print <exprs..>
```

Instruccion encargada de la escritura de datos hacia la salida estandar. Las `<exprs..>` se evalúan completamente antes de imprimir los valores por pantalla.

Condicional

Ejemplos:

```
if x%2==0 then
    print "even\n"
else if x%3 == 0 then
    print "threeven"    -- esto no existe.
else
    print "I dunno\n"
end
```

Sintaxis:

```
if <expr Bool> then
    <stats..>
[else
    <stats..>]
end
```

Condicional típico. La condición debe ser la `<expresion>` de tipo `Bool` y en caso de ser cierta, se ejecuta la `<stat>` , sino se ejecuta la `<stat>` despues del `else` (en caso de haber).

Condicional invertido

Ejemplos:

```
unless tired then
    work()
end
```

Sintaxis:

```
unless <expr Bool> then
    <stats..>
end
```

Es opuesto a un condicional `if`. Es equivalente a:

```
if not (<expr Bool>) then
    <stats..>
end
```

Condicional por casos

Ejemplos:

```
case age
when 0 .. 3 do
    print "bebé"
when 4 .. 12 do
    print "niño"
when 12 .. 17 do
    -- notar que el 12 está en "niño" y "joven"
    print "joven"
else
    print "adulto"
end
```

Sintaxis:

```
case <expr>
when <expr> do <stats..>
when <expr> do <stats..>
...
[else <stats..>]
end
```

Condicinal por casos, *case*.

Iteración determinada

Ejemplos:

```
for i in 1 .. 10 do
    print i*2
    print ","
end
```

Sintaxis:

```
for <id> in <rango> do
    <stats..>
end
```

El campo para `<rango>` debe ser del estilo `Int..Int`, puede ser con identificadores o expresiones. El `<id>` puede ser modificado dentro del `for`. Vale la pena mencionar que dicho identificador es alcanzable unicamente en el cuerpo de la iteración, al finalizar la iteración éste deja de existir.

Iteración indeterminada

Ejemplos:

```
while money > 0 do
    spend(money)
    print money
end
```

Sintaxis:

```
while <expr Bool> do
    <stats..>
end
```

Mientras la `<expr Bool>` evalue a `true`, ejecutar el cuerpo `<stats..>`.

Iteración indeterminada invertida

Ejemplos:

```
until entiende("recursión") do
    estudia("recursión")
end
```

Sintaxis:

```
until <expr Bool> do
    <stats..>
end
```

Hasta que la `<expr Bool>` evalue a `true`, ejecutar el cuerpo `<stats..>`. Es equivalente a:

```
while not (<expr Bool>) do
    <stats..>
end
```

Terminación de iteración

```
break
```

Instrucción `break` típica.

Continuación de iteración

```
continue
```

Instrucción `continue` típica.

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:

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

Expresiones con rangos

Primero llevar a cabo todo lo demas. Después nos preocupamos por rangos

Conversiones de tipos

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

- `def toInt ::  Float -> Int`
- `def toFloat ::  Int -> Float`
- `def toString ::  _ -> String`
- `def length ::  [_] -> Int`