LENGUAJE DE PROGRAMACIÓN **SIUSPLAU**

DECLARACIÓN DE VARIABLES:

SiUsPlau cuenta con 3 distintos tipos de variables simples, enteros, booleanos y reales.

Para declarar cualquier variable, escribimos **var** seguido de el tipo de la variable y su identificador.

Los posibles tipos de variable son:

Tipo	Dominio	Valor por defecto
enter	…-2,-1,0,1,2...	0
bool	cert, fals	cert
real	**R**	0’0

Para asignar un valor a las variables emplearemos **=**. Varias variables pueden ir declaradas en la misma linea de codigo, separadas con una coma. Cada sentencia terminan con un punto.

Los números reales se escriben separando la parte entera y decimal con una apóstrofe.

var enter id1, id2 = 3 .

var bool id3 = **cert** .

var bool id4 = **fals** .

var real id5 = 4’25 .

EXPRESIONES ARITMÉTICAS

Definimos las siguientes operaciones, que toman 2 números enteros o reales y devuelven un tercero.

Operador	Tipo	Prioridad	Asociatividad
+	infijo	0	Asociativo por la izquierda
-	infijo	0	Asociativo por la izquierda
*	infijo	1	Asociativo por la izquierda
/	infijo	1	Asociativo por la izquierda

EXPRESIONES BOOLEANAS

Tanto la entrada como la salida son booleanos:

Operador	Tipo	Prioridad	Asociatividad
¬	Unario prefijo	5	Asociativo por la izquierda
&	Binario infijo	3	Asociativo por la izquierda
|	Binario infijo	2	Asociativo por la izquierda

La entrada puede ser cualquier tipo simple, y devuelven un booleano.

Operador	Tipo	Prioridad	Asociatividad
==	Binario infijo	4	No asociativo
!=	Binario infijo	4	No asociativo
<	Binario infijo	4	No asociativo
>	Binario infijo	4	No asociativo
<=	Binario infijo	4	No asociativo
>=	Binario infijo	4	No asociativo

ARRAYS DE VARIABLES:

Se pueden declarar arrays de cualquier tipo de variables añadiendo [] tras la declaración de tipo.

var enter[] lista .

Para reservar memoria, usamos la keyword **nou**, indicando como se indica en el siguiente ejemplo el tamaño del array:

var enter[] lista = **nou enter**[7].

var enter[][] matriz= **nou enter**[N][M].

Para inicializarlos, escribimos los elementos en cuestión separados por coma.

El operador de acceso a la posición de un array son los corchetes.

lista[1] = 4.

matrix[2][3] = 1.

Están implementados como arrays híbridos, se almacenan en la memoria dinámica de la pila, mientras que en memoria estática se guarda una referencia a su posición inicial, y sus respectivos tamaños. Es el programador el que debe asegurarse que el índice en el acceso a un array es válido, **SiUsPlau** no lanza excepciones en caso de acceder a memoria ajena al array.

OPERACIONES

Los distintos bloques de código quedan delimitados por un par de llaves **{ }**.

Los comentarios van precedidos por :: hasta el final de la línea.

La operación condicional que usaremos sigue la siguiente sintaxis:

¿ condición **?** {

    … se ejecuta si la condición evalúa a **cert**

} **resta** {

    … se ejecuta si la condición evalúa a **fals**

}

El bucle **mentre** es equivalente a un while. Ejecuta el código de dentro del bloque hasta que la condición deje de ser cierta.

```
mentre ( condicion ) fer {
        ...
}
```

El bucle **per** itera sobre todos los elementos de un array.

```
per elem en lista fer {
        …
}
```

Existe también la opción de realizar un filtrado sobre los elementos de la lista, con la sintaxis:

per elem **en** lista **amb** condición **fer** {

…

}

Las operaciones de escritura muestran la expresión por consola siguiendo el esquema :

    escriu expresion.

Las operaciones de lectura almacenan el valor que se lee en la variable indicada según el esquema :

    llegeix variable.

Ambas operaciones de entrada salida trabajan con los tipos primitivos enter, bool y real.

FUNCIONES

Para la declaración de funciones, se escribe la palabra reservada **funcio**, posteriormente el tipo, el identificador, y entre paréntesis los parámetros de entrada, separados por comas. El tipo de las funciones, además de los tipos ya existentes, puede ser **buit**, para las funciones sin valor de retorno.

No hay restricciones para hacer una llamada a función, ni hay restricciones en el tipo de la función ni de los parámetros. Los valores de estos se pasan por valor.

Para devolver el valor de retorno utilizamos la palabra **tornar**.

```
funcio buit fun1(enter a, real b) {
…
        }
funcio enter fun2(enter n, bool b) {
…
}
```

Para la llamada a estas funciones, escribimos el identificador de la función, y entre paréntesis sus parámetros, separados por comas.

```
fun1(4, num).
```

MEMORIA DE LA PRACTICA

Debido a la acumulación de entregas y exámenes en el mes de mayo no hemos tenido tiempo para implementar todas las funcionalidades que en un principio nos propusimos. Listamos aquí las instrucciones que, pese a ser reconocidas por el lenguaje, vinculadas y tipadas, su generación de código no está implementada:

Per, debido a que su funcionalidad semejante a un iterador la hacía menos intuitiva, y puede ser fácilmente sustituida con un bucle **mentre**.

Inicialización explícita de listas:

```
        var enter[] lista = [3, 7, 2, 4].
```

Las funciones pueden tener parámetros de cualquier tipo primitivo, por tanto, no se admiten variables de varias dimensiones.

Las funciones que devuelven arrays pueden dar problemas en memoria. Estas devolverían solo la posición dinámica en la que se encuentra el array, pero dichas posiciones de memoria pueden ser pisadas con nuevas reservas de memoria dinámica, borrando por lo tanto el array devuelto por la función.

GUÍA DE PRUEBA

De los cinco ejemplos de abajo se pueden realizar pruebas con todos ellos, salvo de generación de código para el quinto por lo comentado sobre la inicialización de listas y el per.

Para ello se necesita incluir el código en el fichero **input.txt**, o en otro siempre y cuando se ejecute el **constructoraast.Main** para ese fichero.

Para probar los ejemplos no será necesario modificar el archivo de **main.js** . Sin embargo, si se desea probar código con un número de lecturas por consola mayor que 8 (recordamos es la instrucción **llegeix** lee solo enteros) se deberá cambiar manualmente el **main.js**.

Si se realiza la prueba desde windows, contamos con 2 archivos ejecutables, **Ejecutar y Compilar.bat** que compila y ejecuta el código de java, tomando los datos de **input.txt** y generando el archivo **codigo.wat**; y **Ejecutar wasm.bat**, que obtiene **codigo.wasm** y llama al **main.js**, leyendo desde el fichero **datos.txt.txt**.

No será necesario su uso; pero también aparecen dos ejecutables que obtienen los analizadores léxicos y sintácticos a partir de los archivos .L y .CUP.

EJEMPLOS

Cálculo de los primeros N números primos.

```
var enter N = 100, p = 2.
var bool[] tabla = nou  bool[N+1].

var enter i = 0.
mentre (i <= N) fer {
   tabla[i] = cert.
   i = i + 1.
}
tabla[0] = fals.
tabla[1] = fals.
mentre (p <= (N/2)) fer {
   ¿ tabla[p] ? {
      var enter i = 2*p.
      mentre (i <= N) fer {
         tabla[i] = fals.
         i = i + p.
      }
   }
   p = p + 1.
}
i = 0.
mentre (i <= N) fer {
   ¿ tabla[i] ? {
      escriu i.
   }
   i = i +1.
}
```

Factorial de un numero

```
funcio enter fact(enter n) {
        var enter sol = 1.
        ¿ n > 0 ? {
                sol = n * fact(n-1).
        }
        tornar sol.
}
var enter n.
llegeix n.
escriu fact(n).
```

Algoritmo de la burbuja.

```
var enter n = 4.
llegeix n.
var enter[] lista = nou enter[n].
::Puede ser necesario cambiar el apostrofe al pasar de word a .txt
var enter i = 0.
mentre (i < n) fer {
   llegeix lista[i].
   i = i + 1.
}
i = 1.
mentre (i < n) fer {
   var enter j = 0.
   mentre ( j < (n - i )) fer {
      ¿ lista[j] > lista[j+1] ? {
         var enter aux = lista[j].
         lista[j] = lista[j+1].
         lista[j+1] = aux.  }
      j = j + 1.  }
   i = i + 1.
}
i = 0.
mentre (i < n) fer {
   escriu lista[i].
   i = i +1.
}
```

Números combinatorios(ineficiente).

```
funcio enter comb(enter n, enter k) {
var enter sol = 0.
   ¿ k == 0 ? {
         sol = 1.
   } resta {
      ¿ n == 0 ? {
         sol = 0.
      } resta {
         sol = comb(n-1,k-1) + comb (n-1,k).
      }
   }
   tornar sol.
}
var enter n,m.
llegeix n.
llegeix m.
escriu comb(n,m).
```

Algoritmo de Kruskal. (No es posible generar su código)

```
var enter v = 8, aux = 1, i = 0, incluidas = 0.
var real[] distancia = [2, 4, 5, 2, 7, 1, 4, 4, 3, 1, 5, 7].
var enter[] extremo1 = [0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5], extremo2 = [1, 2, 3, 3, 4, 3, 5, 4, 5, 5, 6, 6].
:: ordenar segun la distancia
var enter[] p= [0, 0, 0, 0, 0, 0, 0, 0].
var bool[] incluida= [fals, fals, fals, fals, fals, fals, fals, fals, fals, fals, fals, fals].
mentre (incluidas < (v-1)) fer {
 ¿ p[extremo1[i]] == 0 & p[extremo2[i]] == 0? {
    p[extremo1[i]] = aux.
    p[extremo2[i]] = aux.
    aux = aux +1.
    incluidas = incluidas+1.
    incluida[i] = cert.
  } resta{
   ¿ p[extremo1[i]] == 0 & p[extremo2[i]] != 0 ? {
      p[extremo1[i]] = p[extremo2[i]].
      incluidas = incluidas+1.
      incluida[i] = cert.
    } resta{
     ¿ p[extremo1[i]] != 0 & p[extremo2[i]] == 0 ? {
        p[extremo2[i]] = p[extremo1[i]].
        incluidas = incluidas+1.
        incluida[i] = cert.
      } resta{
       ¿ p[extremo1[i]] < p[extremo2[i]] ? {
          per elem en p amb(elem == p[extremo2[i]]) fer {
            elem = p[extremo1[i]].
          }
          incluidas = incluidas+1.
          incluida[i] = cert.
        } resta{
         ¿ p[extremo1[i]] > p[extremo2[i]] ? {
            per elem en p amb(elem == p[extremo1[i]]) fer {
              elem = p[extremo2[i]].
            }
            incluidas = incluidas+1.
            incluida[i] = cert.
          } resta{ }
          ::no se incluye ninguna, p[extremo[i]]==p[extremo2[i]]
       }
      }
    }
  }
  i = i + 1.
}
```