Lenguaje de programación GLUCK

(V1.3)

Introducción

GLUCK es un lenguaje de programación de próposito general, del tipo imperativo/procedural, fuertemente tipado, y con manejo nativo de contratos.

Para ejemplificar la estructura de un programa escrito en GLUCK, se muestra el famoso 'hola mundo':

```
use system.io;
start: function {
      print(“hola mundo”);
}
```

Siguiendo cada una de las partes, lo primero que se observa es la directiva *use* que instruye al compilador a incluir las definiciones de funciones que pertenencen al módulo io del sistema. Entre las definiciones, se encuentra la función print, que es la que nos interesa.

Luego sigue la declaración de función: *start: function*

Como se verá en las siguientes secciones, todas las declaraciones comienzan con un identificador (para GLUCK, los identificadores son secuencias de letras, números y _, con la restricción de que la secuencia comience con una letra), que será el nombre asociado a la definición. En este caso, la función se llamará start, que es además un nombre especial, porque será el que utilizará el compilador para designar el punto de arranque del programa.

La palabra clave function indica que start será una función. Entre llaves sigue la definición de la función, el código que será ejecutado al evaluar la función. En este caso, es simplemente la invocación de la función print, con el argumento hola mundo , que escribirá en la salida estándard el texto mencionado.

Variables y expresiones

En el siguiente ejemplo, se muestran las notaciones para declaración de variables, y notaciones utilizadas en expresiones:

```
use system.io;
start: function {
      num1: int;
      num2: int = 5;
```

```
        num1 = num2 * 3 – 5;
        print(num1);
}
```

Una declaración de variable comienza por un nombre, que será un identificador definido de manera similar al utilizado para nombrar funciones. Luego, sigue una identificación del tipo de dato que la variable va a llevar. Al ser fuertemente tipado, GLUCK exije que cada variable tenga asociado un tipo de dato, y verificará en tiempo de compilación si las asignaciones y operaciones realizadas tienen algún problema de tipos. Los tipos pueden ser bool, para representar valores de verdad; int, para representar enteros; real, para valores en punto flotante; o String para secuencias de texto. También podría especificarse un tipo de datos definido por el usuario (ver próximas secciones).

Una declaración de variable puede asignarle un valor a la misma (aunque esto es opcional). Dicha asignación puede ser cualquier tipo de expresión, siempre y cuando se resuelva al tipo de dato asignado a la variable. La utilización de una variable sin asignar es considerada un error. Por ejemplo, el siguiente fragmento generará un error de compilación:

```
x: int;
y:int = x;
```

En los casos donde el compilador no pueda determinar la correcta inicialización de una variable, asumirá que la variable puede no haber sido inicializada, y emitirá un error. Por ejemplo, el siguiente programa daría este tipo de error:

```
a: function x:bool {
        z: int;
        if (x) {
                z = 0;
  }
  print(z);
}
```

Las variables pueden ser declaradas dentro de la definición de una función. En este caso, su alcance será de función. Es decir, que no se las podrá utilizar fuera del cuerpo de la misma. Una variable también puede ser declarada fuera de una función. En este caso se dice que el alcance de la función es de módulo, y podrá ser utilizada por todas las funciones del módulo. Las variables solo pueden ser utilizadas después de ser declaradas. Se puede definir una variable de función que tenga el mismo nombre que una variable de módulo, pero no se pueden definir dos variables con el mismo alcance y el mismo nombre.

(nuevo en 1.2)

Las variables, y demas construcciones con nombre (funciones, tipos definidos y restricciones) son sensibles al uso de mayusculas y minúsculas. Es decir, que las variables hola, Hola y HOLA son tres

distintas.

Las expresiones son construcciones formadas por constantes, variables, invocaciones a funciones y operadores.

GLUCK reconoce los siguientes operadores:

`+ - * / %` para las operaciones aritméticas simples. Ambos parámetros deberán ser enteros o reales, o el compilador generará un error. El resultado de aplicar estos operadores es otro entero o real.

`!` negación lógica, que espera un único párametro de tipo bool, y devuelve otro valor bool representando su inverso.

`(nuevo en 1.2)`

`==` `!=` para comparaciones por igualdad. Ambos parámetros deberán ser del mismo tipo o promocionables al mismo tipo. El resultado de estos operadores es del tipo bool. El resultado de una comparación de parametros reales resuelve el problema de redondeo inerente a dichos formatos de manera automatica.

`<` `<=` `>` `>=` para comparaciones entre numeros y reales. Ambos parametrós deberán ser del mismo tipo, o promocionables al mismo tipo. El resultado de estos operadores es del tipo bool.

'`and`' y '`or`' para expresiones booleanas. Ambos parámetros deberán ser de tipo bool. El resultado será de tipo bool, resolviendo la table de verdad de dichas operaciones.

Si los operandos no fuesen del tipo requerido, el compilador intentará realizar una conversión de tipos de la siguiente manera:

Si se necesita un real y se tiene un entero, el entero se promueve a real

Si se necesita un entero y se tiene un real, el real se trunca a entero, y se emite una advertencia

Si se necesita un string y se posee un entero o real, se convierten a su representación en base 10.

Si se necesita un string y se posee un bool, se convierte a '`true`' o '`false`' dependiendo de su valor.

(nuevo en 1.2)

Cuando un operador necesita iguales operandos, y en el codigo se encuentran distintos tipos, se promocionan de manera implicita siguiendo las siguiente reglas:

Si uno de los operadores es string, el otro se promueve a string

Si uno de los operadores es real, cualquier int se promueve a real

Sino, es un error y se reporta como tal.

Cualquier otro tipo de incongruencia será reportada como un error.

Los operadores que pueden aparecer en una expresión son:

- constantes: Las constantes pueden ser enteras (números), reales (números decimales), strings (secuencia de texto entre comillas), o bool (keywords true y false).

- Variables: utilizando su nombre
- invocaciones a funciones que retornen parámetros: utilizando su nombre seguido de los parámetros de entrada entre paréntesis

Ejemplos:

- `5 + 4`
- `6 == 7 or 4 == var1`
- `var2 * var1 > 5`
- `fn1(var2, true) != fn2()`

Funciones

Las funciones se declaran mediante la palabra clave function. Siguen la siguiente secuencia:

```
nombre: function [<param1> {, <paramn>} ] [ -> <param_out1> {, <param_outn>} ]
'{' ... '}'
```

Las funciones se declaran y definen en el mismo momento. A diferencia de las variables, el orden en que son definidas no es importante en cuanto al uso. Es decir, se puede utilizar una función que será definida subsecuentemente.

Las funciones pueden tener cero o mas parámetros de entrada y cero o mas parámetros de salida. Cada párametro esta caracterizado por un nombre formal y un tipo. Por ejemplo:

```
fn1: function param1: int, param2:bool -> out1:bool, out2:string { ... }
```

Dentro del cuerpo de la función, lo parámetros se comportan como si fueran variables declaradas del tipo dado. El compilador considera un error intentar definir una variable de alcance función que poseea el mismo nombre que un párametro formal. Asimismo, no puede haber dos parámetros con el mismo nombre.

Las funciones solo pueden ser declaradas a nivel de módulo. No es posible declarar funciones dentro de funciones, o dentro de alguna otra definición.

Es posible utilizar funciones declaradas en otro módulo, mediente el uso de la directiva *use*.

```
use  <nombre de modulo>
```

Al declarar la utilización de otro módulo, el compilador se encargara de procesar dicho módulo, además del actual. Si bien cada módulo es compilable independientemente de otro, para poder realizar la verificacion de declaraciones, es necesario contar con ambos compilados. El compilador determinará si la dependencia ha sido modificada desde su ultima compilación, y procedera a recompilarla o tomar los datos directamente según el caso. (nuevo en 1.2) Para mas detalles ver la sección de dependencias.

Los parámetros de entrada son considerados como de solo lectura. Cualquier intento de modificar un párametro de entrada será indicado con un error.

Si algún párametro de salida no fuese asignado, el compilador emitirá un error, de manera similar a la situacion de variables no inicializadas.

Cuando una función retorna mas de un párametro, no podrá ser utilizada en expresiones, a menos que se opere con arreglos (ver próximas secciones).

Arreglos

Un arreglo es una lista finita de elementos que poseen el mismo tipo. Cualquier tipo, sea básico o definido, puede ser utilizado como arreglo.

Para decir que una variable es un arreglo, se utilizan corchetes: `arr: int[]`

Los arreglos tienen tamaño variable, e inicialmente estan vacios, a menos que se indique entre los corchetes el tamaño inicial del mismo.

Intentar acceder a un elemento que no existe dentro de un arreglo, generará un error en tiempo de compilación o ejecución, dependiendo del caso.

Para modificar la cantidad de elementos que posee un arreglo, se puede realizar una asignación directa de una expresión que retorne un arreglo, o se puede asignar explicitamente un valor a una posición.

Ejemplo:

```
a: int[];
(nuevo en 1.2)
b: int[1 + leeCantidad()]; // la inicialización puede ser una expresion
a[9] = 0;   // asigna a a[9] el valor 0; hace que el arreglo tenga 10 elementos (0 a 9)
a[6] = 1; // asigna a a[6] el valor 1; el arreglo permanece con el mismo tamaño
print(a[6]); // imprime 1
print(a[0]); // imprime 0  (ver a continuación por que)
print(a[10]); // error
```

El acceso a elemento de un arreglo se realiza utilizando corchetes y una expresión cuyo resultado debera ser entero, para determinar el índice del elemento a accesar. El primer elemento de un arreglo tiene índice 0, el segundo 1, y asi sucesivamente.

La expresión resultante del acceso a un elemento puede ser utilizada a la izquierda o a la derecha de una asignación. Si es usada a la derecha, el elemento debe existir (el arreglo debe tener el tamaño adecuado), o se generará un error en tiempo de compilación o ejecución. Si es usada a la izquierda, el valor podrá superar el tamaño actual del arreglo, y esto producirá una expansión del tamaño del mismo.

Al expandir un arreglo, es probable que queden espacios entre el último elemento y el elemento actual. En esos casos, los valores serán completados con 0 para arreglos numéricos, false para arreglos de booleanos, para arreglos de strings, y con el resultado de aplicar estas reglas a cada campo para los valores definidos por el usuario.

Una expresión puede construir un arreglo utilizando llaves: '{' <expr1> { , <exprn> } '}'. Como caso especial, se permite la construcción de arreglos de esta forma para el lado izquierdo de asignaciones. De hecho ésta construcción es una de las formas de recibir parámetros de las funciones que retornan mas de uno:

```
a:int;
b:string;
{a, b} = fn();
```

Notar que la construcción, si bien tiene sintaxis similar, no es un arreglo según la definición dada, porque posee elementos de distinto tipo. En la misma interpretacion, una función que retorna mas de una variable no esta retornando un arreglo.

De un arreglo, se pueden obtener subconjuntos utilizando llaves. Por ejemplo x{2,3} significa que retornaria un arreglo que contiene los elementos 3ro y 4to del arreglo x. Si x no fuera un arreglo o una expresión que resuelve a un arreglo, se generará un error. Los subconjuntos resueltos de esta forma son alias del arreglo original y pueden ser asignados, con la salvedad de que deben ser asignados con exactamente el mismo tamaño del subarreglo.

Ejemplos:

```
a: string[10];
a{3, 4} = { “dd”, “aa”};   // equiv. A a[3]=”dd”; a[4]=”aa”;
a{5, 6} = a{3, 4}; // equiv. A a[5] = a[3]; a[6]=a[4];
```

(nuevo en 1.2)

```
a{0, 3} = a{1, 4} // esta operación mueve los primeros 4 elementos a la derecha, y deja el quinto sin modificar
a{1, 4} = a{0, 3} // esta operación mueve los elementos 1 a 4 a la izquierda, y deja el primero sin modificar
a[5] = a{5, 5}; // error, dado que el lado izquierdo es un elemento y el derecho un arreglo
a{5,6} = { “s”, “d”, “dd” }; // error, dado que los tamaños son distintos.
```

Sentencias

La definición de una función esta compuesta por una serie de sentencias.

Las sentencias que poseee Gluck son:

Asignacion: `ID = <expr> ;`

Asigna el resultado de una expresión a una variable o párametro. El destino de la asignación debe haber sido definido con anterioridad (antes en el archivo). Caso contrario, el compilador emitirá un error.

Condicional simple: `if <expr> then '{' ... '}' { elseif <expr> then '{'...'}' } [else '{' ... '}' ]`

Permite dividir el flujo de ejecución en uno o mas caminos de acuerdo a las expresiones utilizadas. Las expresiones deben resultar en un valor de tipo bool, o el compilador generará un error. Los bloques elseif se procesarán en orden, solo si la condición anterior fue evaluada a false. El bloque else, opcional, se ejecutará si todas las expresiones anteriores fueron false.

(Modificado en 1.3) Iteracion: `for ID in <expr> loop '{' ... '}'[ ifnever '{' ... '}' ] [ifquit '{' ... '}' ]`

Permite repetir un conjunto de sentencias sobre cada elemento de un arreglo. El identificador podrá ser utilizado dentro de las sentencias para referirse al elemento que se esta procesando en ese momento. La expresión deberá devolver un arreglo o el compilador emitirá un error.

El bloque de sentencias que sigue a ifnever se ejecutará solo si el arreglo a utilizar estaba vacio. El bloque de sentencias que sigue a ifquit se ejecutará solo si se ha salido abruptamente del ciclo mediante la sentencia quit.

(Modificado en 1.3) Ciclos: `(while | until) <expr> loop '{' ... '}'  [ ifnever '{' ... '}' ] [ifquit '{' ... '}' ]`

Repite el bloque de sentencias mientras o hasta que se cumpla la condición expresada. Notar que las construcciones loop while x y loop until !x son equivalentes. El bloque que sigue a ifnever se ejcutara sólo si el ciclo no pudo ejecutarse ni siquiera una vez. El bloque ifquit se ejecutará solo si el ciclo fue terminado abruptamente por la sentencia quit.

Quit: `quit [if <expr>] ;`

Permite salir de una iteración o un ciclo sin que se cumplan las condiciones normales de salida. Si se utiliza la parte opcional if, solo se ejecutará el quit si la expresión, que debe ser booleana (o el compilador emitirá un error) es evaluada como true.

Tipos de datos definidos

Además de los tipos basicos, se pueden definir nuevos tipos compuestos.

La sintaxis de esa definición es :

```
nombre: type  '{'  <declar1> {, <declarn> } '}'
```

(nuevo en 1.2) Los tipos deben definirse a nivel modulo. No pueden definirse dentro de una función, tipo definido o restriccion.

Ejemplo:

```
composite:type {
   x:int,
   y:int,
   z:int,
   parent: composite
```

```
}
```

Para utilizar un tipo definido, el tipo deberá haber sido declarado anteriormente, mediante una declaración explicita, o el uso del módulo donde fue declarado.

Un tipo compuesto puede tener otros tipos compuestos como datos, pero los nuevos tipos no pueden declarase adentro de otro tipo o función.

Desde el punto de vista de uso, los tipos compuestos son similares a los valores: pueden utilizarse a la derecha e izquierda de una asignación, pueden formarse arreglos de tipos compuestos, y pueden ser utilizados como parámetros de entrada y salida de funciones.

Además, existen operadores que permiten acceder a los componentes individuales del tipo:

`var{p var rop}` → referencia a la parte del tipo llamada prop

`var{prop1, prop2}` → referencia a las partes del tipo llamas prop1 y prop2.

El segundo tipo de construcciones tiene dos razones de ser: primero, permite asignar rapidamente varios valores:

```
var{prop1, prop2} = {expr1, expr2}
```

Segundo, provee una forma natural de asignar funciones que retornen mas de un valor:

```
var{prop1, prop2} = fn_retorna_2_valores();
```

De hecho, se puede considerar que una función que retorna n valores esta conceptualmente definiendo un tipo nuevo de datos (sin nombre), y retornandolo.

Notar que según las definiciones, `var{prop1, prop2}` es equivalemente a `{var{prop1}, var{prop2}}`.

Los tipos definidos por el usuario, poseen un valor especial, null, que indica que no fueron inicializados. Intentar acceder a los valores de una variable cuyo valor sea null, causará un error.

Restricciones

Las restricciones son un conjunto de condiciones que pueden ser adosadas a declaraciones de variables, parámetros (nuevo en 1.2) o elementos de un tipo definido. Su inclusión restringe los posibles valores de dichos parámetros, y hace que se genere un errror en tiempo de compilación o ejecución si alguna de las restricciones no se cumple.

(nuevo en 1.2) Las restricciones deben definirse a nivel modulo. No pueden definirse dentro de una función, tipo definido u otra restriccion.

La declaración de las restricciones simplemente involucra a las condiciones:

```
positive:constraint x:int {   x > 0   }
```

El ejemplo crea una restricción llamada positive, que pide que al ser aplicada sobre una variable o párametro de tipo entero, su valor sea siempre mayor que 0.

Con esta definición, es posible hacer las siguientes construcciones:

```
fn:function in:int positive {
   b:int positive;
    b=in;
    b=-b;  // generara un error!!!
}
fn2:function -> ticks:int positive { ... }
```

El primer ejemplo define una función donde su párametro de entrada debe cumplir la restricción. Esto hara que el programa verifique que se cumple dicha condicion antes de cada invocacion, y genere un error si falla.

También una variable ha sido definida con dicha restricción. Por ello el programa revisara en cada asignación que se cumpla dicha restricción. De esta misma manera, se puede restringir el valor de un componente de un tipo definido de datos.

En el segundo ejemplo, una variable de salida posee una restricción. En este caso el programa emitirá un error si al finalizar la ejecución de la función la variable no cumple con la restricción. (NOTA: esto significa que la restricción puede ser incumplida DURANTE la ejecución de la función)

Las restricciones pueden acumularse.

Por ejemplo:

```
pair: constraint x:int { x % 2 == 0 }
natural: constraint {x >= 0 }

duplicate: function in:int natural -> out:int natural pair {
   out = int * 2;
}
```

Las restricciones pueden utilizarse sobre mas de un párametro:

```
greaterThan:constraint x:int,  y:int {
   x > y
}
fn:function in1:int, in2:int greaterThan(in1) { ... }
```

(nuevo en 1.2) Dado que el contenido de una restricción es una expresión, pueden incluirse llamadas a funciones para implementar restricciones mas complejas:

```
withinList x:mytype, validItems:mytype[], allowEmpty:bool {
    (x == null && allowEmpty) || isInList(x, validItems)
```

```
}
isInList:function x:mytype, list:mytype[] -> result:bool { ... }
```

Comentarios (nuevo en 1.1)

Los archivos de código fuente gluck pueden estar comentados siguiendo el estanrd de comentarios de C ++. Esto es:

Comentarios multilinea.

Comienzan con los caracteres / y *, y continuan hasta encontrar la secuencia * y /, independientemente de la cantidad de lineas que transcurran entre ambas secuencias.

Las secuencias pueden comenzar y terminar en cualquier lugar del texto, pero no pueden partir a una palabra reservada, identificador o constante en dos. Por ejemplo: sss/* comment */eee NO se interpreta como el identificador ssseee, sino como dos identificadores sss y eee seguidos.

Los comentarios no pueden comenzar en el interior de una constante de texto. Por ejemplo: a = /* NO inicia un bloque de comentario.

Un comentario multilinea no cerrado es un error de compilación, y es reportado consecuentemente.

Los comentarios multilinea pueden anidarse, esto es, abrir un comentario dentro de otro. El número de comentarios cerrados debe igualar al de comentarios abiertos para considerar que el comentario finalizó

Comentarios inline

Comienzan con la secuencia / y /, pueden comenzar en cualquier parte del archivo, excepto dentro de una constante literal

Convierten en comentario el resto de la línea. El fin de linea indica el fin del comentario.

Dependencias (nuevo en 1.2)

Cuando un archivo debe hacer uso de funciones, restricciones o tipos definidos en otro, debe indicar esto mediante la palabra clave *use*.

Dicha palabra clave deberá aparecer fuera de otra definición, y al principio del archivo, antes de cualquier otra definicion.

Por ejemplo:

```
use a; // valido
use b; // valido
```

```
    b:int;   // valido, el nombre de modulo b no entra en conflicto con la variable
    use c;  // invalido, dado que ya aparecio otra definicion antes de 'use'
```

El nombre que se coloca a continuación del use representa el nombre del archivo que contiene el codigo fuente/objeto utilizado.

Los archivos fuentes de gluck tienen extension .gluck, mientras que los archivos compilados tienen extension .class

Los archivos pueden encontrarse en otros directorios, diferentes al del archivo que esta siendo procesado. Para poder lozalizar los archivos se utiliza resuelve el nombre utilizado reemplazando los '.' por separadores de directorio.

Asi use a.b.c; va a buscar el archivo a/b/c.gluck o a/b/c.class y utilizar sus definiciones.

Para resolver el problema de como referenciar dependencias cuando el archivo no se encuentra en la raiz de directorios, se puede utilizar la palabra clave *location.*

La palabra clave location indica el lugar relativo del archivo con respecto al proyecto, y es utilizado para resolver dependencias, que siempre son referencias absolutas con respecto a la carpeta que aloja el proyecto.

Por ejemplo, si se tienen los archivos a.gluck, b.gluck y c.gluck en las siguientes carpetas:

```
<prj>
    a.gluck
    dir1/
        b.gluck
        dir2/
            c.gluck
```

Posibles definiciones de ellos serían:

```
a.gluck:
use dir1.dir2.c;
...

b.gluck:
location dir1;
use a;
...

c.gluck:
```

```
location dir1.dir2;
use dir1.b;
...
```

Es posible que dos archivos se utilizen mutuamente de manera directa o a traves de una relación transitiva. El compilador resolverá correctamente las dependencias en dichos casos.

Al utilizar un modulo, se accede a sus funciones, restricciones y tipos. las variables definidas a nivel modulo NO son visibles afuera del mismo.