

Proyecto: Traductor de GCL a su Semántica

GCL (Guarded Command Language) es un lenguaje de programación imperativo que usa el comando de múltiples guardias, para la instrucción de selección e iteración, soporta tipo de datos entero, booleanos y arreglos de enteros. Se han omitido características normales de otros lenguajes, tales como llamadas a procedimientos y funciones, tipos de datos compuestos y números punto flotante, con el objetivo de simplificar su diseño y hacer factible la elaboración de un interpretador a lo largo de un trimestre.

El traductor que se debe implementar debe recibir un programa escrito en GCL y devolver su semántica. Un programa en GCL semánticamente hablando, es una relación sobre el espacio de posibles valores de las variables del programa. El traductor al recibir un programa, debe devolver la relación correspondiente, escrita con los símbolos de la teoría de conjuntos. A continuación se define el lenguaje GCL y la semántica de cada instrucción.

1. Estructura de un programa

Un programa en GCL tiene la siguiente estructura:

```
[[  
<declaración de variables>  
<instrucción>  
]]
```

Es decir, una declaración de variables y una instrucción cualquiera (simple o compuesta por varias instrucciones secuenciadas), que se encuentran dentro de un bloque que lo delimita los tokens `[[` y `]]`. Un programa en GCL podría verse así:

```
[[  
  declare x: int  
  x := 1;  
  print "Numero: " . x . " ";  
  for i in 2 to 10 -->  
    print i . " "  
  rof  
]]
```

2. Identificadores

Un identificador de variable es una cadena de caracteres de cualquier longitud compuesta únicamente de las letras desde la A hasta la Z (mayúsculas o minúsculas), los dígitos del 0 al 9, y el caracter `.`. Los identificadores no pueden comenzar por un dígito y son sensibles a mayúsculas; por ejemplo, la variable `var` es distinta a la variable `Var`, las cuales a su vez son distintas a `VAR`. No se exige que sea capaz de reconocer caracteres acentuados ni la ñ.

3. Tipos de datos

Se disponen de 3 tipos de datos en el lenguaje:

- **int**: representan números enteros.
- **bool**: representa un valor booleano, es decir, **true** o **false**.
- **array**[N..M]: representa un arreglo donde sus índices tienen un rango entre los enteros N y M (estos enteros pueden ser negativos pero deben cumplir que $N \leq M$). En la declaración de un arreglo, N y M deben ser literales enteros, es decir no se puede hacer una declaración de un arreglo, colocando N o M como variables de tipo enteras, sino que hay que colocar valores enteros fijos.

Las palabras **int**, **bool** y **array** están reservadas por el lenguaje y se usan en la declaración de tipos de variables.

La semántica de los tipos de datos anteriores es la siguiente $sem[\mathbf{int}] = \mathbb{Z}$, $sem[\mathbf{bool}] = \{true, false\}$ y $sem[\mathbf{array}[N..M]] = \mathbb{Z}^{[N..M]}$, donde $\mathbb{Z}^{[N..M]}$ es el conjunto de todas las funciones con dominio $\{N, \dots, M\}$ y rango \mathbb{Z} .

4. Instrucciones

4.1. Bloque

Permite secuenciar un conjunto de instrucciones y declarar variables locales. Puede usarse en cualquier lugar donde se requiera una instrucción. Su sintaxis es:

```
| [  
  <declaración de variables>  
  <instrucción 1> ;  
  <instrucción 2> ;  
  ...  
  <instrucción n-1> ;  
  <instrucción n>  
]|
```

El bloque consiste de una sección de declaración de variables, la cual es opcional, y 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 puede terminar con ;.

La sintaxis de la declaración de variables es:

```
declare  
  x1, x2, ... , xn1 : <tipo1> ;  
  y1, y2, ... , yn2 : <tipo2> ;  
  ...  
  z1, z2, ... , zm : <tipom>
```

Estas variables solo serán visibles a las instrucciones y expresiones del bloque. Se considera un error declarar más de una vez la misma variable en el mismo bloque.

La declaración de variables puede verse como una secuencia de declaraciones de diferentes tipos separadas por el caracter ;. Nótese que por lo tanto la última declaración no puede terminar con ;. El tipo de dato

después del token `:`, es el tipo de dato que corresponde a cada variable de la lista precedente a dicho token.

El bloque de declaración desde el punto de vista semántico define un espacio producto. Concretamente, si \bar{x} es la lista de todas las variables declaradas (en el orden que fueron declaradas en las diferentes líneas) y T_1, T_2, \dots, T_n es la lista de la semántica de los tipos que corresponde a cada una de las variables anteriores, entonces el bloque de declaración denota el espacio producto

$$\prod_{i=1}^n T_i.$$

A este espacio producto se le llama “Espacio de Estados” Esp y a cada tupla de Esp se le llama “Estado”. Por ejemplo la declaración

```
declare
  x, y : int ;
  arr : array[1..3] ;
  bo1, bo2 : bool
```

denotaría el espacio de estados $Esp = sem[\mathbf{int}] \times sem[\mathbf{int}] \times sem[\mathbf{array}[1..3]] \times sem[\mathbf{bool}] \times sem[\mathbf{bool}] = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}^{[1..3]} \times \{true, false\} \times \{true, false\}$

Por otro lado. Sea un bloque B_2 que tiene un espacio de estados $\prod_{i=n+1}^m T_i$ cuando se calcula de forma

aislada y B_1 un bloque con espacio de estados $\prod_{i=1}^n T_i$. Se define que cuando B_2 se encuentra anidado dentro de otro bloque B_1 , entonces el espacio de estados de B_2 es

$$\prod_{i=1}^m T_i.$$

Se le llamará espacio de estados extendido Esp' , al conjunto de estados que se le agrega el estado *abort*. Es decir:

$$Esp' := Esp \cup \{abort\}$$

Luego del bloque de declaración una instrucción denota una relación total sobre el espacio de estados (extendido) Esp' . A continuación se explica la sintáxis y semántica de cada instrucción.

4.2. Skip

La frase **skip** es una instrucción de GCL. Si dicha instrucción se encuentra en un bloque donde el espacio de estados extendido es Esp' , entonces se define su semántica como:

$$sem[\mathbf{skip}] := id_{Esp'}$$

donde $id_{Esp'}$ es la función identidad de Esp' a Esp' .

4.3. Secuenciación

```
<instrucción1> ;  
<instrucción2>
```

Una secuenciación de instrucciones es una instrucción que se construye con dos instrucciones `<instrucción1>` y `<instrucción2>`. El operador de secuenciación es el `;`, el cuál es un operador binario, por lo tanto, la última instrucción de varias instrucciones secuenciadas, nunca lleva punto y coma.

A la instrucción de secuenciación se le asigna la siguiente semántica:

$$sem[\llbracket \text{instrucción1} \gg; \llbracket \text{instrucción2} \gg] = sem[\llbracket \text{instrucción2} \gg] \circ sem[\llbracket \text{instrucción1} \gg]$$

4.4. 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 arrojar un error.

Si x_1, \dots, x_n es la lista de todas las n variables que son visibles dentro de un bloque, listadas en el orden que fueron declaradas y x_i es la i -ésima variable de esa lista, entonces la semántica de la instrucción $x_i := \text{Exp}$, cuando se encuentra dentro de ese bloque, es la siguiente:

$$\{\langle (x_1, \dots, x_{i-1}, x_i, \dots, x_n), x' \rangle \in Esp \times Esp \mid x' = (x_1, \dots, x_{i-1}, \text{Exp}, \dots, x_n) \} \cup \{ \langle abort, abort \rangle \}$$

4.5. Salida

```
print <Expresión>
```

Donde `<Expresión>` puede ser una expresión de cualquier tipo o cadenas de caracteres encerradas en comillas dobles, separadas por el operador de concatenación de cadenas “punto” (`.`).

Las cadenas de caracteres deben estar encerradas entre comillas dobles (`"`) y solo debe contener caracteres imprimibles. No se permite que tenga saltos de línea, comillas dobles o backslashes (`\`) a menos que sean escapados. Las secuencias de escape correspondientes son `\n`, `\"` y `\\`, respectivamente.

Un ejemplo de `print` válido es el siguiente:

```
print "Hola mundo! \n Esto es una comilla escapada \" y un backslash \\"
```

El operador de concatenación de las cadenas de caracteres es “punto” (`.`) el cual convierte automáticamente cualquier argumento entero en cadena de caracteres. Por ejemplo el siguiente `print` imprime `Hola mundo! 1` con un salto de línea al final:

```
print "Hola " . "mundo! " . 1 . "\n"
```

El formato para la impresión de un arreglo es como en el siguiente ejemplo. Un arreglo de tipo `array[-1..1]` en donde la casilla `-1` toma el valor de 1, la casilla 0 el valor 6 y la casilla 1 el valor `-3` debe imprimirse de la siguiente manera:

```
-1:1, 0:6, 1:-3
```

4.6. Condicional y Guardias

```
if <condición1> -->
  <instrucción1>
[] <condición2> -->
  <instrucción2>
.
.
[] <condiciónN> -->
  <instrucciónN>
fi
```

La condición debe ser una expresión de tipo `bool`, de lo contrario debe arrojar un error. Las guardias que tienen las condiciones e instrucciones de la 2 a la N son opcionales.

Si $T_i := \{x \in Esp \mid \langle \text{condición}_i \rangle\}$, entonces la semántica de la instrucción condicional es:

$$\left(\bigcup_{i=0}^n (sem[\langle \text{instrucción}_i \rangle] \circ id_{T_i}) \right) \cup \left(\left(\bigcup_{i=0}^n T_i \right)^c \times \{abort\} \right)$$

4.7. Iteración

```
do <condición> -->
  <instrucción>
od
```

La condición debe ser una expresión de tipo `bool`.

La semántica de la instrucción de iteración se le entregará más adelante en el trimestre.

4.8. Iteración con múltiples guardias

```
do <condición1> -->
  <instrucción1>
[] <condición2> -->
  <instrucción2>
.
.
[] <condiciónN> -->
  <instrucciónN>
od
```

El comportamiento de esta instrucción se describe en base a las anteriores instrucciones que se han definido. Esta instrucción se describe como aquella que tiene la misma semántica que la instrucción siguiente:

```
do <condición1> \/\ <condición2> \/\ ... \/\ <condiciónN> -->
  if <condición1> -->
    <instrucción1>
  [] <condición2> -->
    <instrucción2>
  .
  .
  .
```

```

        .
    [] <condiciónN> -->
        <instrucciónN>
    fi
od

```

4.9. Iteración for

```

for <identificador> in <expresión1> to <expresión2> -->
    <instrucción>
rof

```

La instrucción declara automáticamente una variable llamada `<identificador>` de tipo `int` y local al bloque de la iteración. Esta variable es de solo lectura y no puede ser modificada.

La semántica de esta instrucción es la misma que la semántica de:

```

| [
    declare
        <identificador> : int
        <identificador> := <expresión1>;
    do <identificador> <= <expresión2> -->
        <instrucción>;
        <identificador> := <identificador> + 1
    od
] |

```

5. Regla de alcance de variables

Para utilizar una variable primero debe ser declarada al comienzo de un bloque o como 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 de 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 traductor 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.

El siguiente ejemplo es válido en GCL y pone en evidencia las reglas de alcance:

```

| [
    declare x, y: int

    | [
        declare x, y: array[2..3]
        x := 1, 2;
        print "print 1 " . x // x será de tipo array
    ] |;

    | [

```

```

    declare x, y: bool
    x := true;
    print "print 2 " . x // x será de tipo bool
  ];
  x := 10;
  print "print 3 " . x; // x será de tipo int

  for x in 1 to 5 -->
    |[
      declare x: array[-1..0] // Esconde la declaración de x hecha por el for
      x := 4, 5;
      print "print 4 " || x // x será de tipo array
    ]|
  rof
]|

```

6. 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.

6.1. Expresiones con enteros

Una expresión aritmética estará formada por números naturales (secuencias de dígitos del 0 al 9), variables y operadores aritméticos convencionales. Se considerarán la suma (+), la resta (-), la multiplicación (*), el - unario, la inicialización de arreglos (,) y la modificación de arreglos (:). 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 de tipo `int` y su resultado también será de tipo `int`.

La operación inicialización y modificación de arreglos (, y :) se explica en la sección de expresiones con arreglos.

6.2. Expresiones con arreglos

Una expresión con arreglos está formada por números enteros, variables y operadores de inicialización, consulta y modificación de arreglos. Los arreglos solo pueden contener elementos de tipos enteros, no existen en GCL arreglos de booleanos o arreglos de arreglos. Los operadores a considerar son los siguientes:

- inicialización `exp1, exp2, ..., expn`: un arreglo `A` de longitud `n` se inicializa asignando una lista de enteros separados por comas, de la siguiente manera `A := exp1, exp2, ..., expn`. Es un error asignar una lista de enteros de diferente longitud a la longitud del arreglo.
- consulta `[exp]`: un elemento del arreglo `A`, es consultado por medio de la expresión `A[exp]`, en donde `exp` es una expresión entera y su valor indica el índice del elemento a consultar.
- modificación `exp1:exp2`: una modificación al arreglo `A` es denotado por la expresión `A(exp1:exp2)`, donde `exp1` y `exp2` son expresiones enteras. El valor de `exp1` corresponde al índice que se quiere modificar en el arreglo, y el valor de `exp2`, es el valor que tendrá el elemento de la posición `exp1` en el arreglo.

La expresión `A(exp1:exp2)` se considera como un nuevo arreglo, por lo tanto puede ser consultado o nuevamente modificado con las expresiones `A(exp1:exp2)[exp]` y `A(exp1:exp2)(exp3:exp4)` respectivamente, y así sucesivamente.

Una expresión del tipo `A(exp1:exp2)` no genera un cambio persistente en el arreglo `A`, al menos que se haga una asignación de la forma `A:=A(exp1:exp2)`. Se ejemplifica con el siguiente programa:

```
[[
  declare a : int;
           A : array[0..2]
  A := 3, 2, 1;
  a := A(2:4)[2]; // a toma el valor de 4 pero no ocurren cambios persistentes en A
  a := A[2];      // a toma el valor de 1
  A := A(2:4);    // A es modificado por el arreglo 0:3, 1:2, 2:4
  a := A[2];      // a toma el valor de 4
]]
```

6.3. Expresiones booleanas

Una expresión booleana estará formada por constantes booleanas (`true` y `false`), variables y operadores booleanos. Se considerarán los operadores `/\`, `\/` y `!` (and, or y not). También se utilizará notación infija para el `/\` y el `\/`, y notación prefija para el `!`. Las precedencias son las siguientes (ordenados comenzando por la menor precedencia):

- `\/`
- `/\`
- `!`

Los operandos de `/\`, `\/` y `!` deben tener tipo `bool`, y su resultado también será de tipo `bool`.

Además GCL cuenta con predicados capaces de comparar enteros. Los predicados 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 predicados no son asociativos, a excepción de los operadores == y != cuando se comparan expresiones de tipo bool.

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

- ==, !=
- <, <=, >=, >

7. Comentarios y espacios en blanco

En GCL se pueden escribir comentarios de una línea, estilo *C*. Al escribir // se ignorarán todos los caracteres hasta el siguiente salto de 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.

8. Ejemplos

Ejemplo 1:

```
| [  
  print "Hello world!"  
]|
```

Ejemplo 2:

```
| [  
  declare count, value: int  
           arr : array[1..4]  
  arr := 2, 3, -1, -2;  
  count := 4;  
  for i in 1 to count -->  
    value := arr[i];  
    print "Value: " . value  
  rof  
]|
```

Ejemplo 3:

```
| [  
  declare x: int  
  x = 0;  
  if -5 <= x /\ x < 0 -->  
    print "Del -5 al 0"  
  [] x == 0 -->  
    print "Tengo un cero"  
  [] 1 <= x /\ x < 100 -->  
    print "Del 1 al 100"  
  fi  
]|
```

