Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Traductores e Interpretadores
Trimestre Septiembre-Diciembre 2019

Proyecto: Interpretador de GuardedUSB

GuardedUSB 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.

1. Estructura de un programa

Un programa en GuardedUSB tiene la siguiente estructura:

```
|[
<instrucción>
]|
```

Es decir, una instrucción cualquiera se encuentra dentro de un bloque que lo delimita los tokens `|[` y `]|`. Un programa en GuardedUSB podría verse así:

```
|[
  declare x: int
  x := 1;
  println "Numero: " || x || " ";
  for i in 2 to 10 -->
    |[
      println 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 con signo de 32 bits.
- `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`≤`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.

4. Instrucciones

4.1. 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.

4.2. 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.

4.3. 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, ... , xn : <tipo1>, <tipo2>, ... , <tipon> ;
  y1, y2, ... , yn : <tipo1>, <tipo2>, ... , <tipon> ;
  ...
  z1, z2, ... , zn : <tipo1>, <tipo2>, ... , <tipon>
```

Estas variables sólo 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 `;`. Cada tipo de dato después del token `:`, se listan en el orden correspondiente a la secuencia de variables, es decir `x1` es de tipo `<tipo1>`, `x2` es de tipo `<tipo2>`, `xn` es de tipo `<tipon>`.

En la sección de declaraciones es posible declarar varias variables de un mismo tipo en una misma línea haciendo:

```
x1, x2, ... , xn : <tipo>
```

4.4. Entrada

```
read <variable>
```

Permite obtener entrada escrita por parte del usuario. Para ejecutar esta instrucción, el interpretador debe solicitar al usuario que introduzca un valor, dependiendo del tipo de la variable, y posteriormente se debe validar y almacenar lo que sea introducido. Si la entrada es inválida se debe repetir el proceso. Puede haber cualquier cantidad de espacio en blanco antes o después del dato introducido.

La variable debe haber sido declarada o sino se arrojará un error.

Para los tipos `array` el interpretador debe aceptar el siguiente formato de entrada:

```
<entero1>, <entero2>, ..., <enteron>
```

Donde `<entero1>`, `<entero2>`,...,`<enteron>` son enteros de 32 bits escritos en base decimal, y el total de enteros separados por comas, debe ser igual a la longitud del arreglo declarado. Es decir si `A` es un arreglo declarado de la forma `array[-1..3]`, entonces este arreglo `A` tiene longitud 5 y por lo tanto una instrucción `read A` debe recibir una lista de 5 enteros.

4.5. Salida

```
print <Expresión>
println <Expresión>
```

Donde `<Expresión>` puede ser una expresion de cualquier tipo o cadenas de caracteres encerradas en comillas dobles, separadas por el operador de concatenación de cadenas `||`. La instrucción `println` imprime automáticamente un salto de línea después de haber impreso el argumento.

Las cadenas de caracteres deben estar encerradas entre comillas dobles (`"`) y sólo 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 concateción de las cadenas de caracteres es `||` 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 arrojarse un error. Las guardias que tienen las condiciones e instrucciones de la 2 a la N son opcionales.

Ejecutar esta instrucción tiene el efecto de evaluar la condición 1 y si su valor es `true` se ejecuta la instrucción 1; si su valor es `false`, entonces se pasa a evaluar la condición 2 (si existe, sino no se ejecuta ninguna instrucción). Si el valor de la condición 2 es `true`, se ejecuta la instrucción 2; si su valor es `false` se sigue el proceso en la siguiente guardia con la condición 3 y así sucesivamente.

4.7. Iteración for

```
for <identificador> in <expresión1> to <expresión2> -->
   <bloque>
rof
```

Para ejecutar esta instrucción se evalúan las expresiones `<expresión1>` y `<expresión2>`, cuyo tipo debe ser `int`, y en la primera iteración a la variable `<identificador>` se le asigna el valor de `<expresión1>`. Si `<identificador>` es menor o igual a `<expresión2>`, se ejecutan las instrucciones del `<bloque>` y se incrementa en 1 la variable `<identificador>`, para luego repetir el proceso.

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

4.8. Iteración

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

La condición debe ser una expresión de tipo `bool`. Para ejecutar la instrucción se evalúa la condición, si es igual a false termina la iteración; si es `true` se ejecuta la instrucción del cuerpo y se repite el proceso.

4.9. 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 el mismo comportamiento que la instrucción siguiente:

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

El siguiente ejemplo es válido en GuardedUSB 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 ariméticos convencionales. Se considerarán la suma (`+`), la resta (`-`), la multiplicación (`*`), la división entera (`/`), módulo (`%`), 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 sólo pueden contener elementos de tipos enteros, no existen en GuardedUSB 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. Es un error de ejecución que el valor de `exp` sea un entero fuera del rango de índices del arreglo.
- 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. Es un error de ejecución que el valor de `exp1` sea un entero fuera del rango de índices del 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, A: int, 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 GuardedUSB cuenta con 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 (ordenados comenzando por la menor precedencia) son las siguientes:

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

6.4. Conversiones de tipo y funciones embebidas

GuardedUSB provee las siguientes funciones embebidas para convertir tipos y otra tareas:

- `atoi(a)`: convierte la expresión `a`, de tipo `array[N..M]`, a un entero, sólo si el arreglo tiene un sólo elemento (es decir, si `N=M`).
- `size(a)`: devuelve la longitud del arreglo `a`.
- `max(a)`: devuelve el máximo índice del arreglo `a`.
- `min(a)`: devuelve el mínimo índice del arreglo `a`.

7. Comentarios y espacios en blanco

En GuardedUSB 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:

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

Ejemplo 2:

```
|[
  declare count, value: int
  read count;
  for i in 1 to count -->
    |[
      read value;
      println "Value: " || value
    ]|
  rof
]|
```

Ejemplo 3:

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