Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Traductores e Interpretadores
Septiembre-Diciembre 2018

Proyecto: Interpretador de PascalUSB

PascalUSB es un lenguaje de programación imperativo inspirado en Pascal con postcondiciones e intervalos de enteros como tipo de dato. Se han omitido características normales de otros lenguajes, tales como 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 PascalUSB tiene la siguiente estructura:

```
program
<instrucción>
```

Es decir, la palabra clave `program` seguida de una instrucción cualquiera. Un programa en PascalUSB podría verse así:

```
program
begin
    declare x as inter;
    x := 1..5
    for i in x+(3..10) do println "Numero: ", i
end
```

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`.
- `inter`: representa un intervalo de enteros. Posee cota inferior y cota superior. Por ejemplo: `-1..5` es un intervalo con cota inferior -1 y cota superior 5.

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

4. Instrucciones

4.1. 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.

4.2. Bloque

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

```
begin
    <declaración de variables>
    <instrucción 1> ;
    <instrucción 2> ;
    ...
    <instrucción n-1> ;
    <instrucción n>
end
```

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 as <tipo1>, <tipo2>, ... , <tipon> ;
    y1, y2, ... , yn as <tipo1>, <tipo2>, ... , <tipon> ;
    ...
    z1, z2, ... , zn as <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 `as`, 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 as <tipo> ;
```

4.3. 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.

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

```
a..b
```

Donde `a` y `b` son las cotas del intervalo. Debe validarse que sean números enteros (con o sin signo) y que `a` sea menor o igual a `b`. Puede haber cualquier cantidad de espacio en blanco entre las cotas y el símbolo `..`

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

4.4. Salida

```
print x1, x2, ..., xn
println x1, x2, ..., xn
```

Donde `x1, x2, . . . , xn` pueden ser expresiones de cualquier tipo o cadenas de caracteres encerradas en comillas dobles. El interpretador debe recorrer los elementos en orden e imprimirlos en pantalla. La instrucción `println` imprime automáticamente un salto de línea después de haber impreso la lista completa de argumentos.

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 \\"
```

4.5. Condicional if then else

```
if <condición> then <instrucción 1> else <instrucción 2>
if <condición> then <instrucción 1>
```

La condición debe ser una expresión de tipo `bool`, de lo contrario debe arrojarse un error.

Ejecutar está instrucción tiene el efecto de evaluar la condición y si su valor es `true` se ejecuta la instrucción 1; si su valor es `false` se ejecuta la instrucción 2. Es posible omitir la palabra clave `else` y la instrucción 2 asociada, de manera que si la expresión es `false` no se ejecuta ninguna instrucción.

4.6. Condicional case

```
case <expresión int> of
  <expresión inter> ==> <instrucción>
  <expresión inter> ==> <instrucción>
  ...
  <expresión inter> ==> <instrucción>
end
```

La expresión del `case` debe tener tipo `int` y las expresiones de las guardias deben tener tipo `inter`.

Esta instrucción permite definir intervalos y asignar alguna instrucción a ejecutar cuando un entero pertenezca a alguno de ellos. Para ejecutarla se evalúa la expresión del `case`, luego se recorre en orden cada una de las expresiones `inter` y se evalúan. Cuando el resultado de alguna de las expresiones contenga el entero calculado anteriormente, se ejecuta la instrucción asociada. Siempre se evalúan todas las expresiones, es posible que los intervalos se solapen y que se ejecute cualquiera de las instrucciones asociadas. Por ejemplo:

```
case x of
    1..4 ==> <instrucción 1>
    3..10 ==> <instrucción 2>
end
```

Si x es igual a 1 o 2 sólo se ejecuta la instrucción 1. Si está entre 5 y el 10, sólo se ejecuta la instrucción 2. Pero si es igual a 3 o 4, como ambos valores se encuentran en los intervalos de ambas expresiones, se puede ejecutar tanto la intrucción 1 como la instrucción 2, pero sólo una de ellas, queda abierta al implementador del lenguaje esta decisión.

4.7. Iteración for

`for <identificador> in <inter> do <instrucción>`

Para ejecutar esta instrucción se evalúa la expresión `<inter>`, cuyo tipo debe ser `inter`, y a la variable `<identificador>` se le asigna la cota inferior de este intervalo. Si `<identificador>` está dentro del intervalo (incluyendo las cotas), se ejecuta `<instrucción>` y se incrementa en 1 la variable `<identificador>`.

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

4.8. Iteracion

`while <condición> do <instrucción>`

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.

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 PascalUSB y pone en evidencia las reglas de alcance:

```
program
begin
  declare x, y as int

  begin
    declare x, y as inter
    x := 1..2;
    print "print 1", x // x será de tipo inter
  end;

  begin
    declare x, y as bool
    x := true;
    print "print 2", x // x será de tipo bool
  end;
```

```
  x := 10;
  print "print 3", x; // x será de tipo int

  for x in 1..5 do
    begin
      declare x as inter // Esconde la declaración de x hecha por el for
      x := 4..5;
      print "print 4", x // x será de tipo inter
    end
  end
end
```

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 inverso (`-` unario) y la construcción (`..`). 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`.

La operación construcción (`..`) se explica en la sección de expresiones con intervalos.

6.2. Expresiones con intervalos

Una expresión con intervalos está formada por números naturales, variables y operadores sobre intervalos. Los operadores binarios a considerar son los siguientes:

- construcción `(a .. b)` : toma dos expresiones de tipo `int` y produce un `inter` cuya cota inferior sea `a` y cota superior sea `b`. Si `a` es mayor que `b`, debe arrojarse un error.
- unión `(a + b)` : toma dos expresiones de tipo `inter` y produce un `inter` cuya cota inferior sea igual a la menor cota inferior entre `a` y `b` y cota superior igual a la mayor cota superior entre `a` y `b`. Si los intervalos `a` y `b` son disjuntos se debe arrojar un error.
- intersección `(a <> b)` : devuelve el subintervalo de mayor tamaño común a ambos operandos. Si la intersección es vacía se debe arrojar un error.

- escala `(a * b)` : sea `a` una expresión de tipo `inter` y `b` una expresión de tipo `int`, el resultado es el intervalo a cuyas cotas han sido multiplicadas por b. Si b es menor que 0, se invierten las cotas.

La precedencia del operador unión es mayor que la del operador intersección.

6.3. Expresiones booleanas

Una expresión booleana estará formada por constantes booleanas (`true` y `false`), variables 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`.

Además PascalUSB cuenta con operadores relacionales capaces de comparar enteros e intervalos. 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 `/=`.

Adicionalmente se provee el operador binario pertenece (`a in b`) el cual devueve `true` si la expresión `a`, de tipo `int`, está dentro del intervalo `b` (incluyendo sus cotas).

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:

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

Cuando se comparan tipos `inter`, los criterios son los siguientes:

- `a < b` : `true` si la cota superior de `a` es menor a la cota inferior de `b`.
- `a <= b` : `true` si la cota superior de `a` es menor o igual a la cota inferior de `b`.
- `a == b` : `true` si las cotas de `a` coinciden con las cotas de `b`.
- `a /= b` : `true` si las cotas de `a` no coinciden con las cotas de `b`.
- `a >= b` : `true` si la cota inferior de `a` es mayor o igual a la cota superior de `b`.
- `a > b` : `true` si la cota inferior de `a` es mayor a la cota superior de `b`.

6.4. Conversiones de tipo y funciones embebidas

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

- `itoi(a)` : convierte la expresión `a`, de tipo `inter`, a un entero, sólo si el intervalo tiene un solo elemento (es decir, sus cotas son iguales).
- `len(a)` : devuelve el tamaño del intervalo `a`.
- `max(a)` : devuelve la cota superior del intervalo `a`.
- `min(b)` : devuelve la cota inferior del intervalo `b`.

7. Comentarios y espacio en blanco

En PascalUSB 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. Postcondiciones

En PascalUSB opcionalmente al final de cada programa se puede colocar una aserción como postcondición a ser verificada. La postcondición siempre se coloca después del último `end` del programa y tiene la siguiente sintaxis:

```
{Post: <expresión booleana>}
```

Donde sólo en la expresión booleana de la postcondición se permiten usar cuantificadores $\forall$ y $\exists$, con la siguiente sintaxis:

```
(forall x|x in <inter>: <expresión booleana>)
```

y

```
(exists x|x in <inter>: <expresión booleana>)
```

Donde la variable `x` puede ser cualquiera (que respete las reglas de construcción sintactica de identificadores de la sección 2) y no necesita estar declarada. `<inter>` es un intervalo y `<expresión booleana>` es cualquier expresión booleana que también puede usar cuantificadores $\forall$ y $\exists$.

Si el estado final del programa no satisface la postcondición debe arrojarse un error, de lo contrario el programa finaliza satisfactoriamente.

9. Ejemplos

Ejemplo 1:

```
program
println "Hello world!"
```

Ejemplo 2:

```
program
begin
  declare count, value as int
  read count ;
  for i in 1..count do
    begin
      read value;
      println "Value: ", value
    end
end
```

Ejemplo 3:

```
program
begin
   declare x as int
   read x;
   case x of
      -5..0 -> println "Del -5 al 0"
      0..0 -> println "Tengo un cero"
      1..100 -> println "Del 1 al 100"
   end
end
```