Algoritmos y Estructuras de Datos II - 2022

Introducción al lenguaje de programación de la materia

- Lenguaje inventado hace unos años específicamente para la materia.
- Inspirado remotamente en el lenguaje Pascal
- Definido informalmente en sintaxis y semántica
- Esfuerzos recientes en formalizarlo e implementarlo

Matías Federico Gobbi. “Semántica estática para un lenguaje Pascal-like”. Trabajo Especial de Licenciatura en Ciencias de la Computación. 2021
https://rdu.unc.edu.ar/handle/11086/17366

Repaso Algoritmos I

Teórico 2020-10-20

Video
https://drive.google.com/drive/u/2/folders/1RliCK9vCjsnekuZFYNTviurPY6JyQiL4

Pizarra
https://docs.google.com/document/d/1Na9aBD7AbvRrPbVdFn7zUGx9PmPTU1Oe75qkibQRprI/edit#

Sintaxis: ¿Cómo se escriben los programas? Un programa es un texto (una secuencia de letras). La sintaxis de un lenguaje me dice qué textos son programas válidos.

Semántica: ¿Qué significan? ¿Qué hacen? Un programa se puede ejecutar, y esa ejecución tiene efecto en un “mundo semántico”. En el caso de los programas imperativos, este mundo semántico es el estado.

Hacíamos programación a pequeña escala (“in the small”). Sólo escribíamos pequeños segmentos de código. En algoritmos 2, vamos a escalar un poco, introduciendo la posibilidad de definir procedimientos y funciones.

Procedimientos y Funciones

Un procedimiento encapsula un bloque de código con su respectiva declaración de variables (que define el estado).

Sintaxis:

```
proc nombre(<in|out|in/out> p1: T1, …, <in|out|in/out> pn: Tn)
  <declaraciones de variables>
  <sentencias>
end proc
```

a donde p1, p2, …, pn son nombres de variables (son los parámetros), y T1, T2, …, Tn son sus respectivos tipos.

- in: el parámetro es de entrada (no se puede modificar)
- out: el parámetro es de salida
- in/out: el parámetro de entrada/salida

Ejemplo de procedimiento:

```
proc abs(in x : int, out y : int)
  if x >= 0 then
    y := x
  else
    y := -x
  fi
end proc
```

(acá n=2, p1 es x, p2 es y, T1 es int, T2 es int)

Ejemplo de uso de este procedimiento:

```
proc llamadordeabs()   <- declaro un proc que tiene 0 parámetros
  var a, b : int
  a := -10
  abs(a, b)
  {- acá vale que b = 10 -} <- esto es un comentario
end proc
```

Obs:

- los proc no devuelven cosas (pero pueden escribir varios parámetros out)
- las llamadas a procedimientos son sentencias del lenguajes

Funciones: las funciones son como los procedimientos salvo que todos los parámetros son in, y devuelven algo.

Sintaxis:

```
fun nombre(p1: T1, p2: T2, …, pn : Tn) ret r : T
  <declaraciones de variables>
```

```
  <sentencias>
end fun
```

a donde p1, p2, …, pn y r son nombres de variables, y T1, T2, …, Tn, T son sus respectivos tipos.

Ejemplo de función:

```
fun abs(x : int) ret y : int
  if x >= 0 then
    y := x
  else
    y := -x
  fi
end fun
```

Ejemplo de uso de esta función:

```
proc llamadordeabs()   <- declaro un proc que tiene 0 parámetros
  var a, b : int
  a := -10
  b := 35
  b := abs(a)
  a := abs(a) + 10
  {- acá vale que b = 10 , a = 20 -} <- esto es un comentario
end proc
```

Obs sobre funciones:

- no tenemos sentencia “return”. se devuelve lo que sea asignado a la variable declarada como “ret”.
- las llamadas a funciones **no son sentencias** sin no que son **expresiones** (e.g. se puede usar en la parte derecha de una asignación, en una guarda, etc).
- las funciones se comprometen a que su llamada no tiene efectos colaterales en el estado.

Obs generales:

- cada función y procedimiento define un estado propio llamado **“contexto”**. Las variables declaradas dentro de funciones y procedimientos no existen fuera de éstas.
- las funciones y procedimientos pueden llamarse entre sí, e incluso mutuamente y a sí mismas.
- no importa el orden en que se declaran, un proc puede llamar a otro que esté definido más adelante
- **no se pueden** definir procedimientos ni funciones adentro de procedimientos o funciones (no hay anidamiento, todas las proc y func están al mismo nivel).

Tipos Nativos

Los tipos nativos son los tipos que trae predefinido el lenguaje de programación.

Tipos básicos:

- `bool`: booleanos (`true` y `false`)
- int: números enteros
- nat: números naturales (con el 0)
- real: números reales
- char: caracteres (‘a’, ‘j’)
- string: secuencias de caracteres (“pi”)

Nos permitiremos usar constantes que nos sirvan como infinito, -infinito, etc.

Tipos estructurados:

- array: arreglos
- pointer: punteros (para más adelante)

Sintaxis de Arreglos:

- Declaración:
  `var` `a : array[N1..M1] …. [Nk..Mk]` **`of`** `T`
  `a donde a es el nombre de la variable, N1,M1, … Nk,Mk son números, y T es el tipo (de los elementos del arreglo).`
- `Acceso (es una expresión):`
  `a[i1]...[ik]`
- `Asignación:`
  `a[i1]...[ik] := E`
  a donde E es una expresión de tipo T.

`Ejemplos:`

- var precios : array[1..10] of int
  {- arreglo de 10 elementos precios[1], precios[2], … , precios[10] -}
- var matriz : array[0..25][5..10] of char
  {- arreglo de caracteres de dos dimensiones (26 x 6):
  matriz[0,5], matriz[0,6], matriz[0,7], … , matriz[0,10]
  matriz[1,5], matriz[1,6], matriz[1,7], … , matriz[1,10]
  …
  matriz[25,5], matriz[25,6], matriz[25,7], … , matriz[25,10]
  -}

Definición de Tipos

Más adelante veremos:

- sinónimos de tipo
- tipos enumerados
- tuplas

Sentencias

skip

La sentencia que no hace nada.
Sintaxis: `skip`

Asignación (:=)

Sintaxis:

```
  v := E
```

a donde v es una variable y E es una expresión.
Semántica: la saben.

Obs:

- no tenemos más la asignación múltiple

Llamada a procedimiento

Sintaxis:

```
  nombreproc(v1, …, vn)
```

Semántica: ya la vimos.

Condicional (if)

Sintaxis:

```
  if B then
    S1
  else
    S2
  fi
```

a donde B es una expresión booleana, y S1, S2 son sentencias.

Semántica: la usual.

Obs:

- no tenemos if multi-guarda como el que había en Algo I
- este es más tipo C

Ejercicio:

- ¿Cómo se simula un if multi-guarda con este if?

If multiguarda en Algo1

```
if [] (x > 0 && x < 10) -> Sentencia1
   [] (x >= 10 && x < 20) -> Sentencia2
   [] (x >= 20) -> Sentencia3
fi
```

Ejercicio: escribir ese if en el lenguaje de Algo2

```
if (x > 0 && x < 10) then
    Sentencia1
else
    if (x >= 10 && x < 20) then
        Sentencia2
    else
        if (x >= 20) then
            Sentencia3
        else
            skip
        fi
    fi
fi
```

Repetición (while)

Sintaxis:

```
  while B do
    S
  od
```

a donde B es una expresión booleana y S es una sentencia.

Semántica: la usual.

Otra repetición (“for to” y “for downto”)

Sintaxis del “for to”:

```
  for i := N to M do
    S
  od
```

a donde N y M son expresiones de tipo int y S es sentencia

Semántica:

1. se declara la variable i (sólo exisitirá dentro de la sentencia S)
2. se le asigna a i el valor N
3. se ejecuta S

4. **se incrementa i en 1**
5. **si i > M termina**, si no vuelve al punto 3.
6. i deja de existir al terminar

Sintaxis del "for downto":

```
for i := N downto M do
  S
od
```

a donde N y M son expresiones de tipo int y S es sentencia
Semántica: igual que el "for to" pero restando 1.

Observaciones:

- no hace falta declarar i (el for mismo ya la declara)
- si había otra i afuera, esta i la tapa.
- **no se puede modificar i** con asignaciones en el cuerpo del ciclo (S)
- no agrega expresividad al lenguaje (todo se puede hacer con while).
- itera desde N hasta M **inclusive**

Ejemplos: Declaramos un arreglo y lo llenamos de ceros de izq. a derecha:

```
var precios: array[1..100] of int
for i := 1 to 100 do
  precios[i] := 0
od
```

Ejemplo sin arreglos: El factorial de un número n.

```
var n, fac : int
n := 10
fac := 1
for i := 1 to n do
  fac := fac * i
od
```

Ejemplo con el downto: recorro un arreglo de der. a izq.

```
var precios: array[1..100] of int
precios[100] := 35
for i := 99 downto 1 do
  precios[i] := precios[i+1] * 2
od
{- pregunta: cuánto vale precios[98] ?? -}
```

Secuenciación

No hay secuenciación explícita como teníamos en Algoritmos I con el ";". Acá simplemente ponemos una sentencia después de la otra y se asumen secuenciadas. Ejemplo:

a := 10
b := 20

Otras

Veremos más adelante otras sentencias como alloc y free para punteros.

Ejercicios

1. Definición recursiva de la función factorial

```
fun factorial(n: nat) ret f: nat
    { - no hace falta inicializar f ya que es una función - }
    if(n = 0) then
        f := 1
    else
        f := x * factorial(n - 1)
    fi
end fun
```

2. Definición iterativa de la función factorial

```
fun factorial(n: nat) ret f: nat
    if(n = 0) then
        f := 1
    else
        for i := 1 to n do
            f := f * i
        od
    fi
end fun
```

3. Procedimiento para inicializar un arreglo de una dimensión en cero

```
proc init_array(out a: array[N..M] of int)
    for i := N to M do
        a[i] := 0
    od
end fun
```

4. Procedimiento para incrementar en 1 los valores de un arreglo

```
proc init_array(in/out a: array[N..M] of int)
    for i := N to M do
        a[i] := a[i] + 1
    od
end fun
```

5. Función para encontrar el mínimo elemento de un arreglo

```
fun min(a: array[1..N] of int) ret i : int
    i := a[1]
    for i =: 2 to N do
        if(a[i] < i) then
            i := a[i]
        else
            skip
        fi
    od
end fun
```