Algoritmos y Estructuras de Datos II - 2023

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 2022-10-4:

- Video: parte 1, parte 2
- Pizarra

Teórico 2020-10-20:

- Video
- Pizarra

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

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
```

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

Tipos de parámetros:

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

Observaciones:

- Los procedimientos no devuelven cosas (pero pueden escribir varios parámetros out).
- Las llamadas a procedimientos son sentencias del lenguaje.

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
```

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
```

Observaciones sobre las funciones:

- No hay sentencia “return”. Se devuelve lo que sea asignado a la variable declarada como “ret”. La variable se puede usar libremente en el cuerpo de la función (se puede leer, asignar varias veces, etc.).
- Las llamadas a funciones **no son sentencias**, 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.

Observaciones generales:

- Cada función y procedimiento define un estado propio llamado **“contexto”**. Las variables declaradas dentro de funciones y procedimientos no existen fuera de éstas.

- Cualquier función o procedimiento pueden llamar a cualquier otra función o procedimiento. Incluso pueden llamarse a sí mismas (recursión), y también mutuamente (recursión mutua).
- No importa el orden en que se declaran, un procedimiento puede llamar a otro que esté definido más adelante.
- **No se pueden** definir procedimientos ni funciones dentro de procedimientos o funciones (no hay anidamiento, todas las definiciones 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”)

Usaremos constantes que nos sirvan como infinito, -infinito, etc.

Tipos estructurados:

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

Arreglos

Sintaxis de Arreglos:

- Declaración de una variable de tipo arreglo:

  ```
  var a : array[N1..M1 , ... , Nk..Mk] of T
  ```

  donde:
  - a: es el nombre de la variable,
  - k: es la cantidad de dimensiones del arreglo.
  - N1,M1, … Nk,Mk: son números que indican el rango del arreglo para cada dimensión
  - T: es el tipo de los elementos del arreglo.

- Acceso (es una expresión):

  ```
  a[i , ... , ik]
  ```

- Asignación (es una sentencia):

  ```
  a[i1 , … , ik] := E
  ```

  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]
 -}
```

Definiciones de Tipos Nuevos

Más adelante veremos:

- Sinónimos de tipo
- Tipos enumerados
- Tuplas

Expresiones

Básicamente las mismas que usamos en Algoritmos 1, agregando la posibilidad de llamar a funciones.

Expresiones válidas en programación imperativa:

- valores constantes
- variables y constantes declaradas
- operaciones básicas (+, -, *, /, div, mod, max, min, $\wedge$, $\vee$, $\neg$, =, $\neq$, $\leq$, <, >, etc.)
- accesos a elementos de arreglos
- llamadas a funciones

Sentencias

skip

La sentencia que no hace nada.

Sintaxis: `skip`

Asignación (:=)

Sintaxis:

```
  v := E
```

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(e1, …, en)
```

Semántica: ya la vimos.

Condicional (if)

Sintaxis:

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

O también sin el else:

```
  if B then
    S1
  fi
```

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

Semántica: la usual.

Observaciones:

- 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 Algoritmos 1:

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

(no válido para algoritmos 2!!)

Ejercicio: Escribir este if en el lenguaje de Algoritmos 2.

Repetición (while)

Sintaxis:

```
while B do
  S
od
```

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
```

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
```
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 variable 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
Encabezado: `fun factorial(n: nat) ret f : nat`

2. Definición iterativa de la función factorial
Encabezado: `fun factorial(n: nat) ret f : nat`

3. Procedimiento para inicializar un arreglo en cero
Encabezado: `proc init_array(out a: array[N..M] of int)`

4. Procedimiento para incrementar en 1 los valores de un arreglo
Encabezado: `proc init_array(in/out a: array[N..M] of int)`

5. Función para encontrar el mínimo elemento de un arreglo
Encabezado: `fun min(a: array[1..N] of int) ret m : int`
`Hacer dos versiones: una con un while y otra con un for.`

—----------

SPOILERS
SOLUCIONES A LOS EJERCICIOS

Ejercicios

1. Definición recursiva de la función factorial

Encabezado de la función:

```
  fun factorial(n: nat) ret f : nat
    {- n = 0 -> 1
       n > 0 -> n * fact (n-1)
    -}
    if n = 0
    then f := 1
    else f := n * factorial (n-1)
    fi
  end fun
```

```
proc usarFactorial()
 var n, f : int
 n := 8
 f : = 28
 n : = factorial (n)
 {- El valor de la f va a ser 28 -}
end proc
```

2. Definición iterativa de la función factorial

Encabezado de la función:

```
  fun factorial(n: nat) ret f : nat

  end fun
```

```
fun factorial_for (n: nat) ret f : nat
 var aux : int
 aux := 1
 for i := n downto 1 do
   aux:= i*aux
 od
 f := aux
end fun
```

```
fun factorial (n: nat) ret f : nat
var a : int
a := 1
f := 1
  while (a <= n)  do
        f := f * a
        a := a+1
```

```
 od
end fun
```

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

Encabezado:

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

proc usarInit ()
  var b : int
  myArray : array[1..3] of int
  init_array(myArray)
  b := myArray[1]
  {- que valor tiene b? Tiene 0 -}
end proc

proc init_arrat(out a: array[N..M] of int)
            var y: int
            y:=N
            while y<=M do
                    a[y]:=0
                    y:=y+1
            od
end proc
```

Obs:

- el tamaño del arreglo (N y M) son parámetros implícitos que pueden ser utilizados en el cuerpo del procedimiento.

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

Encabezado:

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

5. Función para encontrar el mínimo elemento de un arreglo

Encabezado:

```
fun min(a: array[1..N] of int) ret m : int
  var aux : int
  aux := infinito
  for j := 1 to N do
     aux := minimo (aux, a[i])
  od
  m := aux
end fun
```

Tarea: Implementar minimo.