Tarea 3. Árboles

Estructuras de datos

Semestre 2022-1

LCC — UNISON

Nadie es una isla. Congrégate con a lo más tres compañer@s de clase para colaborar en resolver la presente tarea.

P es un lenguaje de programación...
P es simple...
P es pequeño...
P es imperativo...

El programa P más simple es skip, una instrucción que no hace algo.

Todo programa P dispone de una memoria, tan infinita como los circuitos que habita.

Sean a_1 y a_2 expresiones aritméticas, el programa $x[a_1] := a_2$ almacena el valor numérico de a_2 en la posición a_1 de la memoria.

Antes de correr algún programa, el valor almacenado en las posiciones de la memoria es cero. Después de correr algún programa, este no necesariamente es el caso.

Existen programas que conforman programas, como la secuencia de dos, que cuando corre P_1 y luego P_2, es porque corrió $(P_1 ; P_2)$.

Los programas pueden decidir si si o si no, y correr por algún lado, pero no por los dos. Solo si b es Booleana y P_1, P_2 programas

$$(\text{if } b \text{ then } P_1 \text{ else } P_2)$$

será cómo P_1 si si, o cómo P_2 si no.

Algunos programas repiten y otros no terminan de repetir. Por lo que cuando un programa es

$$(\text{while } b \text{ do } p)$$

y b es tautológica, el programa correctamente nunca deja de correr. Por otra parte, cuando b es contradictoria, el programa no hace algo.

Lo interesante ocurre en el resto de los casos... pero únicamente si b se determina por lo que cambia p.

"¿Cómo puede ser esto posible? Si los Booleanos no cambian". Pues eso era antes de incorporar la memoria.

Como vimos en clases, una expresión Booleana puede ser true, false, o bien

$(a_1 = a_2)$, $(a_1 < a_2)$, $(b_1 \wedge b_2)$, $(b_1 \vee b_2)$, $\neg b_1$.

Cuando a_1 y a_2 son aritméticas
Cuando b_1 y b_2 son Booleanas.

Esos símbolos tienen una interpretación usual: "igual qué", "menor qué", conjunción, disyunción y negación.

"¿Y la memoria?" está escondida en las Booleanas con hijas aritméticas.

En clases vimos que ellas podían ser naturales, sumas, restas y multiplicaciones. Ahora pueden ser valores guardados en posiciones de la memoria.

Recuerda que estos valores son naturales, pero pueden cambiar.

Lo descrito anteriormente es una mezcla de lo sintáctico y lo semántico, la forma y el contenido, lo superficial y lo real. Para que "suene" conocido esto es P, pero sin el sentido:

$$P \to \texttt{skip} \mid x[A] := A \mid (P \,;P) \mid (\texttt{while}\ B\ \texttt{do}\ P) \quad (\texttt{if}\ B\ \texttt{then}\ P\ \texttt{else}\ P)$$

$$B \to \texttt{true} \mid \texttt{false} \mid (A = A) \mid (A < A) \mid (B \wedge B) \mid (B \vee B) \mid \neg B$$

$$A \to N \mid x[A] \mid (A + A) \mid (A - A) \mid (A \times A)$$

$$N \to \text{cualquier entero excepto los negativos}$$

Veamos un programa en concreto, calcula el factorial de cinco:

```
(x[0] := 1;
 (while (x[1] < 5) do
   (x[1] := (x[1] + 1);
    x[0] := (x[0] × x[1]))))
```

Y así fue construido:

① P ② (P ; P) ③ (x[A] := A ; P)

④ (x[N] := A ; P) ⑤ (x[0] := A ; P)

Termina de derivar el programa utilizando las reglas de la gramática.

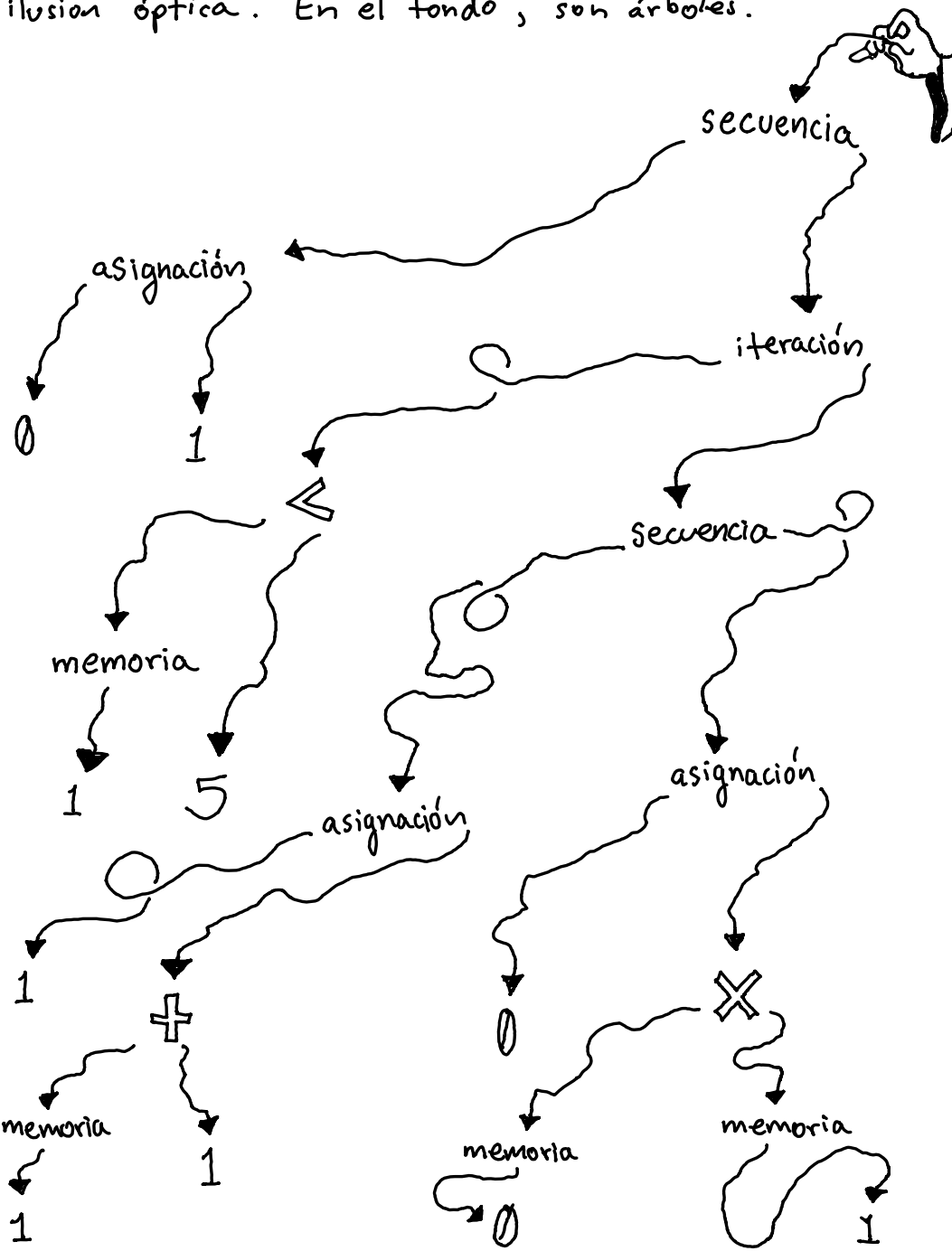
Y así corre:

Al inicio, hay cero en x[i] para toda i natural. Luego hay uno en x[0]. ¿Lo que hay en x[1] es menor a cinco? ¡SI! Ya que en x[1] hay cero, entonces repetimos:

En x[1] ahora hay uno y en x[0] ahora hay uno. ¿De nuevo? ¡SI!
En x[1] ahora hay dos y en x[0] ahora hay dos. ¿De nuevo? ¡SI!
En x[1] ahora hay tres y en x[0] ahora hay seis. ¿De nuevo? ¡SI!
En x[1] ahora hay cuatro y en x[0] ahora hay...

Termina la ejecución, ¿En dónde está el resultado?

Al implementar el lenguaje, su sintaxis es una ilusión óptica. En el fondo, son árboles.



⚠ Considera que la entrada al programa factorial es `x[1]` y que su salida termina en `x[0]`. Escribe el programa con esta nueva convención.

⚠ ¿Conoces otros algoritmos simples? Implementa otro en el lenguaje P siguiendo la convención anterior.

Mejoremos el código de la clase:

⚠ Documenta lo que falta documentar

⚠ Escribe casos de prueba que faltan.

Presta atención a las técnicas que utilizamos:

- Estructuras opacas
- Por cada tipo de expresión se tienen:
  - Predicados
  - Selectores
  - Constructores
  - Destructores (free)
  - Evaluadores

Para las expresiones aritméticas, nos falta un predicado `aexp_is_mem` y un constructor `aexp_make_mem`.

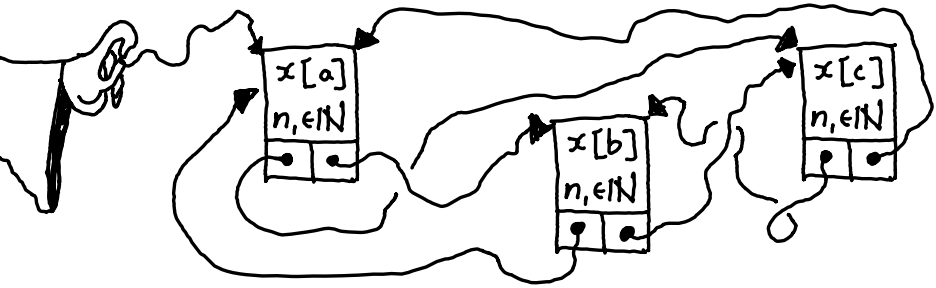
⚠ Impleméntalos, documéntalos, pruébalos.

El evaluador de expresiones Booleanas y aritméticas debe adaptarse... 

⚠ Adapta los evaluadores

... para tomar como segundo argumento una memoria. Quizá piensen en implementar la memoria con un arreglo dinámico, no lo hagan, utilicen mejor una lista.

Si un programa utiliza la memoria de `x[a]`, `x[b]` y `x[c]` tal que $a < b < c$. Puedes representar la memoria "infinita" como



¿Es necesario que sea circular?
¿Es necesario que sea doblemente enlazada?

¡NO!, pero es parte de la tarea que `x[i]` preceda a `x[j]` si $i < j$ y que al buscar el valor de una `x[K]` no pasemos por más de un nodo `x[r]` donde $r > K$.

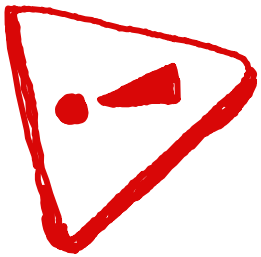
⚠ Implemenenta la memoria como se indica arriba.

Lo que nos falta por implementar es una estructura:

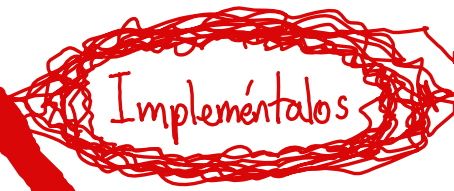
```
struct pexp_t;
typedef struct pexp_t pexp_t;
```

Los predicados:

```
bool pexp_is_skip(pexp_t *);
bool pexp_is_ass(pexp_t*);
bool pexp_is_seq(pexp_t *);
bool pexp_is_while(pexp_t *);
bool pexp_is_if(pexp *);
```



Los selectores:



Los constructores:

Los destructores:

Los evaluadores:

Cada grupo presentará el producto de su trabajo frente al resto de la clase (o por video conferencia).

Se decide el dia de la presentación qué integrante presenta qué parte de la tarea.

El resto de la clase podrá influir la calificación del grupo.

DISFRUTEN
EL
HACKEO
INTENSO

1er día de presentación: 25 de Abril del 2022