El lenguaje Tiny

El lenguaje **Tiny** es un pequeño lenguaje de programación imperativo, cuyo propósito es ejercitar las distintas técnicas descritas en la asignatura de "Procesadores de Lenguajes".
A continuación, se realiza una descripción completa, por niveles, de este lenguaje. También se muestra un programa de ejemplo, que lee una serie de nombres en un 'array', forma un árbol de búsqueda con dichos nombres, y utiliza dicho árbol para escribir los nombres ordenados y sin duplicados.

Léxico

- Identificadores: Comienzan necesariamente por una letra o subrayado (_), seguida de una secuencia de cero o más letras, dígitos, o subrayado (_)
- Literales *enteros*: comienzan, opcionalmente, con un signo + o -. Seguidamente debe aparecer una secuencia de 1 o más dígitos (no se admiten ceros no significativos a la izquierda).
- Literales *reales*: comienzan, obligatoriamente, con una parte entera, cuya estructura es como la de los números enteros, seguida de bien una parte decimal, bien una parte exponencial, o bien una parte decimal seguida de una parte exponencial. La parte decimal comienza con un ., seguido de una secuencia de 1 o más dígitos (no se permite la aparición de ceros no significativos a la derecha). Por último, y también opcionalmente, puede aparecer una parte exponencial (*e* o *E*, seguida de un exponente, cuya estructura es igual que la de los números enteros).
- Literales *cadena*: comienzan con comilla doble ("), seguida de una secuencia de 0 o más caracteres distintos de ", seguida de ". Los caracteres pueden incluir las siguientes secuencias de escape: retroceso (\b), retorno de carro (\r), tabulador (\t), y salto de línea (\n),
- Los siguientes símbolos de operación y puntuación: `+ - * / % < > <= >= == != ( ) ; = [ ] . ^, { } & && @`
- Las siguientes palabras reservadas: **int real bool string and or not null true false proc if else while struct new delete read write nl type call**

Las cadenas ignorables son las siguientes:

- Espacio en blanco, retroceso (\b), retorno de carro (\r), tabulador (\t), salto de línea (\n)
- Comentarios: Son comentarios *de línea*. Comienzan por **##**, seguido de una secuencia de 0 o más caracteres, a excepción del salto de línea.

El lenguaje no distingue entre minúsculas y mayúsculas en las palabras reservadas, pero sí en los identificadores.

Sintaxis

- Un *programa* es un *bloque*.
- Un *bloque* delimita entre llaves (`{..}`) una *sección de declaraciones* opcional, seguida de una *sección de instrucciones*, también opcional.
- La *sección de declaraciones* consta de una o más *declaracion*es separadas por ;, y termina con `&&`. Las *declaracion*es pueden ser de los siguientes tipos:
  - Declaración de *variable*. Comienza por el *tipo* de la variable, seguido del nombre de la variable que se está declarando (un identificador).
    - Ejemplo: **int** x
  - Declaración de *tipo*. Comienza por la palabra reservada **type**, seguida del del *tipo* que se está declarando, seguido del nombre que se está dando a este tipo (un identificador)
    - Ejemplo:

```
type struct{
        int x,
        int y
      } tPar
```

  - Declaración de *procedimiento*. Comienza por la palabra reservada **proc**, seguida del nombre del procedimiento (un *identificador*), seguida de los *parámetros formales*, seguidos de un *bloque*. Los parámetros formales comienzan, a su vez, por un paréntesis de apertura '(', seguido, opcionalmente, de una lista de uno o más *parámetro*s *formal*es separados por comas (,), seguida de ')'. Cada parámetro formal, por último, comienza por el *tipo* del parámetro, seguido opcionalmente por `&` (para indicar que el modo del parámetro es *por referencia*), seguido del nombre del parámetro.
    - Ejemplo:

```
proc incrementa(int & v, int delta) {
  @ v = v + delta
}
```

- En cuanto a los *tipo*s, pueden ser:
  - Un tipo básico: **int**, **real**, **bool** o **string**

- Un nombre de otro tipo (un identificador)
- Un tipo *array*: el *tipo base* del array, seguido del *tamaño* (un número entero encerrado entre corchetes `[…]`).
  - Ejemplo: **int** [20]
- Un tipo *registro*: **struct**, seguida de una lista de uno o más *campos* separados por ‘,’, y encerrada entre llaves (`{…}`). Cada *campo* consta de un *tipo*, seguido de un nombre (un identificador).
  - Ejemplo:

```
struct{
   int cont,
   int[20] lista
  }
```

- Un tipo *puntero*: ^ seguido del *tipo* base.
  - Ejemplo: ^tNodo

- El constructor de tipos *puntero* (^) tiene mayor prioridad que el constructor de tipos *array* ([…]). Ambos constructores son asociativos. El resto de constructores y tipos básicos tienen la máxima prioridad.
  - Ejemplo ^int[5] significa el tipo de los arrays de 5 punteros a int, en lugar del tipo de los punteros a arrays de 5 enteros. Para representar el segundo tipo es necesario utilizar una definición de tipo auxiliar. Por ejemplo:

```
type int[5] t5ints;
type ^t5ints;
```

- La *sección de instrucciones* consta de una lista de una o más instrucciones separadas por ;. Se contemplan los siguientes tipos de instrucciones:
  - Instrucción *eval*. Comienzan con `@`, seguida de una expresión.
  - Instrucción **if**. Comienza por **if**, seguida de una expresión, seguida de un *bloque*.
    - Ejemplo:

```
if x>0{
   @ x = x + 1;
   @ y = 10
  }
```

  - Instrucción **if-else**. Comienza por **if**, seguida de una expresión, seguida de un *bloque*, seguido de **else**, seguida de otro *bloque*.
    - Ejemplo:

```
if x>0{
   @ x = x + 1;
   @ y = 10
  }
else {
   @ x = x - 1;
   @ y = 20
  }
```

  - Instrucción **while**. Comienza por **while**, seguida de una expresión, seguida de un bloque..
    - Ejemplo:

```
while i>0 {
   @ s = s + 2*i;
   @ i = i - 1
  }
```

  - Instrucción de lectura. Comienza por **read**, seguida de una expresión.
    - Ejemplo:

```
read a.c[x + 1]
```

  - Instrucción de escritura. Comienza por **write**, seguida de una expresión.
    - Ejemplo:

```
write 2*x  + 1
```

  - Instrucción de nueva línea. **nl**.
  - Instrucción de reserva de memoria. Comienza por **new**, seguida de una expresión.
    - Ejemplo:

```
new l^.sig
```

  - Instrucción de liberación de memoria. Comienza por **delete**, seguida de una expresión.
    - Ejemplo:

```
delete r[45].x
```

  - Instrucción de invocación a procedimiento. Comienza por **call**, seguida del nombre del procedimiento (un identificador), seguido de los *parámetros reales*. Dichos parámetros

comienzan por (, seguido, opcionalmente, de una lista de una o más expresiones separadas por comas (,), seguida de).

- Ejemplo:
  call fact(x - 1,resul)

- Instrucción *compuesta*. Consiste en un bloque.

- Las expresiones pueden ser:
  - Expresiones *básicas*: literales enteros, reales, booleanos y cadena, nombres de variable (identificadores), y **null**.
  - Expresiones compuestas. Estas expresiones se construyen de acuerdo con las siguientes prioridades y asociatividades de los operadores contemplados (0 es el nivel de menor prioridad):
    - Nivel 0. Operador de asignación =. Operador binario, infijo, asociativo a derechas.
    - Nivel 1: Los operadores relacionales. Todos ellos son operadores binarios, infijos, asociativos a izquierdas.
    - Nivel 2: + asocia a izquierdas, - (binario) no asocia.
    - Nivel 3: **and** asocia a derechas, y **or** no asocia.
    - Nivel 4: *, / y %. Operadores binarios, infijos, asociativos a izquierdas.
    - Nivel 5: - (unario) y **not**. Operadores unarios, prefijos, asociativos.
    - Nivel 6: Operadores de *indexación*, de *acceso a registro* y de *indirección*. Todos estos operadores son operadores unarios posfijos, asociativos. Los operadores de *indexación* son operadores de la forma [E], con *E* una expresión. Los operadores de *acceso a registro* son operadores de la forma . *c*, con *c* un nombre de campo (un identificador). El operador de *indirección* es ^.

    Estas prioridades y asociatividades pueden modificarse mediante el uso de paréntesis, tal y como es habitual.

Semántica estática

Reglas de ámbito

En lo sucesivo:

- Por *ámbito* se entiende (i) cada bloque; (ii) cada procedimiento.
- Las *declaraciones* de un ámbito son (i) en el caso de un bloque, las que aparecen en su sección de declaraciones; (ii) en el caso de un procedimiento, cada uno de sus parámetros formales.

Bajo este supuesto, las reglas de ámbito de este lenguaje son las de *ámbito léxico* explicadas en clase, que pueden resumirse como sigue:

- Las declaraciones que *afectan* a un uso de identificador *i* son todas aquellas en los ámbitos que lo contienen, y que preceden a dicho uso. En caso de que *i* esté inmediatamente precedido por ^ (declaración de tipo *puntero*), también se considera que le afectan todas las declaraciones del ámbito en el que se produce el uso. En el caso de que *i* se esté usando en una declaración de tipo, y no esté inmediatamente precedido de ^, queda excluida dicha declaración de las que afectan a *i*. La declaración de un procedimiento afecta a todos los usos de identificadores en el ámbito del procedimiento (esto permite tener procedimientos recursivos). Nótese que la declaración del procedimiento en sí no pertenece al ámbito del procedimiento, sino al ámbito padre de dicho procedimiento[1].
- Para determinar la declaración asociada al uso de *i* (es decir, para establecer el *vínculo* de *i* en dicho uso), se ordenan todas las declaraciones que lo afectan por orden inverso de aparición, y se localiza la primera declaración de *i* en dicho orden[2]. Si tal declaración no existe, se dice que se hace un uso de un identificador *i no declarado*.

La existencia de identificadores no declarados son errores de ámbito, como también lo es el declarar más de una vez un identificador en un mismo ámbito (es decir, en las declaraciones de un mismo ámbito no se permiten identificadores duplicados). Los programas semánticamente correctos están libres de errores de ámbito.

Reglas de tipo

Las reglas de tipo del lenguaje asumirán que:

---

[1] Esto permite, por ejemplo, ocultar la declaración de un procedimiento mediante alguno de sus parámetros formales o mediante otra declaración en el bloque asociado al mismo.

[2] Obsérvese que, bajo esta perspectiva, primero se consideran las declaraciones en el ámbito de uso, seguidamente las declaraciones en el ámbito padre, y así sucesivamente. Esto permite, por ejemplo, que una declaración de un identificador en un ámbito oculte otra declaración del mismo identificador en otro ámbito que contiene al primero.

(1) El programa está libre de errores de ámbito.
(2) El programa, además, cumple con las siguientes restricciones relativas a las declaraciones (*restricciones pre-tipado*):
   - Los vínculos de los nombres de tipo utilizados en las declaraciones de tipo deben ser declaraciones **type**.
   - El tamaño de los tipos *array* es siempre un entero no negativo.
   - Las definiciones de tipos registro no tienen campos duplicados.

Bajo estos supuestos, para enunciar las reglas de tipo del lenguaje, dado un tipo τ, denotaremos por **ref!**(τ) a:

- El propio tipo τ si τ no es un nombre de tipo.
- **ref!**(τ') si τ es un nombre de tipo y su vínculo es una declaración **type** cuyo tipo asociado es τ'.

Así mismo, definiremos reglas de *compatibilidad de tipos*. Más concretamente, el tipo τ' es *compatible para asignación* con el tipo τ (se denota como $\tau \leftarrow \tau$') cuando τ' es *compatible para asignación con* τ *en el contexto* $\varnothing$. Por su parte, τ' es *compatible para asignación con* τ *en el contexto* Γ (se denota como «$\tau \leftarrow \tau$' | Γ») cuando ocurre alguno de los siguientes supuestos:

- $\tau \leftarrow \tau' \in \Gamma$
- **ref!**(τ) es **int** y **ref!**(τ') es **int**.
- **ref!**(τ) es **real** y **ref!**(τ') es, bien **int** o bien **real**.
- **ref!**(τ) es **bool** y **ref!**(τ') es **bool**.
- **ref!**(τ) es **string** y **ref!**(τ') es **string**.
- **ref!**(τ) es un tipo *array* τ'' [*n*], **ref!**(τ') es otro tipo *array* τ''' [*n*], y «τ'' $\leftarrow \tau$''' | $\Gamma \cup \{\tau \leftarrow \tau'\}$»
- Tanto **ref!**(τ) como **ref!**(τ') son tipos **struct**, y ambos tienen el mismo número de campos (sea este número *n*). Además, «$\tau_i \leftarrow \tau'_i$ | $\Gamma \cup \{\tau \leftarrow \tau'\}$», con $1 \leq i \leq n$, donde τ_i es el tipo del campo *i*-esimo de **ref!**(τ) y τ'_i es el tipo de campo *i*-esimo de **ref!**(τ')
- **ref!**(τ) es $\hat{}\tau$'', y τ' es **null**.
- **ref!**(τ) es $\hat{}\tau$'', **ref!**(τ') es $\hat{}\tau$''' , y «τ'' $\leftarrow \tau$''' | $\Gamma \cup \{\tau \leftarrow \tau'\}$».

De esta forma, las siguientes reglas rigen para la determinación del tipo de las expresiones:

- El tipo de un literal *entero* es **int**.
- El tipo de un literal *real* es **real**.
- El tipo de un literal *booleano* es **bool**.
- El tipo de un literal *cadena* es **string**.
- El tipo de una variable es:
  - El que aparece en la declaración asociada, siempre y cuando dicha declaración sea una declaración de variable.
  - **error** en otro caso.
- El tipo de **null** es **null**.
- El tipo de las expresiones de la forma $E_0 \circ E_1$ se determina a partir del tipo τ_0 de E_0, del tipo τ_1 de E_1 y del operador $\circ$ como sigue:
  - Si $\circ$ es un operador de asignación:
    - Si E_0 es un *designador* (un identificador, o una expresión de la forma *E*[*E*'], *E.c* o *E*^), y $\tau_0 \leftarrow \tau_1$ (es decir, los tipos son compatibles para asignación), entonces el tipo de la expresión de asignación es τ_0.
    - **error** en cualquier otro caso.
  - Si $\circ$ es un operador binario aritmético (+, -, *, /):
    - Si **ref!**(τ_0) y **ref!**(τ_1) son ambos **int**, entonces el tipo resultante es **int**
    - Si uno de los tipos **ref!**(τ_0), **ref!**(τ_1) es **real**, y el otro es, bien **real**, o bien **int**, entonces el tipo resultante es **real**
    - En cualquier otro caso, el tipo resultante es **error**
  - En caso de que $\circ$ sea % (operador *módulo entero*):
    - Si **ref!**(τ_0) y **ref!**(τ_1) son ambos **int**, entonces el tipo resultante es **int**
    - En cualquier otro caso, el tipo resultante es **error**
  - Si $\circ$ es un operador lógico (**and**, **or**)
    - Si **ref!**(τ_0) y **ref!**(τ_1) son ambos **bool**, entonces el tipo resultante es **bool**
    - En cualquier otro caso, el tipo resultante es **error**
  - Si $\circ$ es un operador relacional (<, >, <=, >=, ==, !=)
    - Si **ref!**(τ_0) es **int** o **real** y **ref!**(τ_1) es también **int** o **real**, entonces el tipo resultante es **bool**

        - Si tanto **ref!**(τ_0) como **ref!**(τ_1) son ambos **bool**, el tipo resultante es también **bool** (de esta forma, se supone que el tipo **bool** está ordenado: para ello, se considera **false** < **true**)
        - Si tanto **ref!**(τ_0) como **ref!**(τ_1) son ambos **string**, el tipo resultante es **bool** (los *strings* se suponen ordenados alfabéticamente)
    - Si ° es == o != se permite, además, que:
        - Tanto **ref!**(τ_1) como **ref!**(τ_2) sean tipos *puntero* (no se tiene en consideración sus tipos base, pudiéndose comparar entre sí punteros a valores de cualquier tipo)
        - Uno de ellos sea de tipo *puntero* τ y el otro de tipo **null**
        - Ambos sean de tipo **null**
- El tipo de las expresiones de la forma °E se determina a partir del tipo τ de E, y del operador ° como sigue:
    - Si ° es el operador *menos unario* (-), entonces:
        - Si **ref!**(τ) es **int**, entonces el tipo resultante es **int**
        - Si **ref!**(τ) es **real**, entonces el tipo resultante es **real**
        - En cualquier otro caso, el tipo resultante es **error**
    - Si ° es el operador **not**, entonces:
        - Si **ref!**(τ) es **bool**, entonces el tipo resultante es **bool**
        - En cualquier otro caso, el tipo resultante es **error**
- El tipo de las expresiones de la forma E° se determina a partir del tipo τ de E, y del operador ° como sigue:
    - Si ° es un operador de indexación [*E*'], y τ' es el tipo de *E*':
        - Si **ref!**(τ) es un tipo *array* τ''[*n*], y **ref!**(τ') es **int**, entonces el tipo resultante es el tipo base del tipo *array* τ''.
        - En cualquier otro caso, el tipo resultante es **error**
    - Si ° es un operador de acceso de la forma *.c*:
        - Si **ref!**(τ) es un tipo **struct**, *c* es un campo declarado en dicho tipo, y τ' es el tipo de dicho campo, entonces el tipo resultante es τ'.
        - En cualquier otro caso, el tipo resultante es **error**
    - Si ° es el operador de indirección ^:
        - Si **ref!**(τ) es un tipo ^τ', entonces el tipo resultante es τ'
        - En cualquier otro caso, el tipo resultante es **error**.

De esta forma:

- El tipo de una instrucción `@` *E* es **error** si el tipo de *E* es **error**, y es **ok** en cualquier otro caso (es decir, cuando *E* está correctamente tipada).
- El tipo de una instrucción de lectura **read** E es:
    - **ok** cuando el tipo de E es τ, **ref!**(τ) es **int**, **real** o **string**, y E es un designador.
    - **error** en cualquier otro caso.
- El tipo de una instrucción de escritura **write** E es:
    - **ok** cuando el tipo de E es τ, y **ref!**(τ) es **int**, **real**, **bool** o **string**.
    - **error** en cualquier otro caso.
- El tipo de una instrucción de reserva de memoria **new** E es:
    - **ok** cuando el tipo de E es τ, y **ref!**(τ) es un tipo *puntero*.
    - **error** en cualquier otro caso.
- El tipo de una instrucción de liberación de memoria **delete** E es:
    - **ok** cuando el tipo de E es τ, y **ref!**(τ) es un tipo *puntero.*
    - **error** en cualquier otro caso.
- El tipo de una instrucción de llamada a procedimiento **call** p (E_0, …, E_k) con *k* parámetros reales es:
    - **ok** cuando (i) p es un identificador vinculado a una declaración de procedimiento **proc** p(PF_1, …, PF_k) {…} con el mismo número *k* de parámetros formales; (ii) el tipo τ_i de cada parámetro real E_i es compatible para asignación con el tipo τ'_i del correspondiente parámetro formal PF_i, con la excepción de que el parámetro formal sea un parámetro por referencia de tipo 'real' (en este caso, el parámetro real debe ser un designador de tipo 'real') y (iii) para todos aquellos parámetros formales PF_j de la forma τ&*v* (parámetros *por variable*), el correspondiente parámetro real E_j debe ser un designador.
    - **error** en cualquier otro caso.

- El tipo de una instrucción **if** E B:
  - **ok** cuando el tipo de E es τ, **ref!**(τ) es **bool**, y el tipo del bloque B es **ok**.
  - **error** en cualquier otro caso.
- El tipo de una instrucción **if** E B_1 **else** B_2 es:
  - **ok** cuando el tipo de E es τ, **ref!**(τ) es **bool**, el tipo de B_1 es **ok**, y el tipo de B_2 es **ok.**
  - **error** en cualquier otro caso.
- El tipo de una instrucción **while** E B:
  - **ok** cuando el tipo de E es τ, **ref!**(τ) es **bool**, y el tipo del bloque B es **ok.**
  - **error** en cualquier otro caso.
- El tipo de una instrucción *compuesta* es el tipo del bloque asociado

En cuanto al tipo de un bloque B, es **ok** cuando (i) el tipo del bloque de cada procedimiento en su sección de declaraciones es **ok**; (ii) el tipo de cada instrucción en su sección de instrucciones es **ok**. En cualquier otro caso, el tipo del bloque es **error**.

Por último, un programa correctamente vinculado, y que cumple las *restricciones pre-tipado*, está *correctamente tipado* cuando el tipo de su bloque es **ok**. En otro caso, el programa tendrá *errores de tipado*, y no se considerará semánticamente correcto.

Semántica operacional

En lo que sigue se asumen programas semánticamente correctos, de acuerdo con las reglas de la semántica estática.

Consideraremos un modelo de ejecución basado en una *memoria* en la que pueden almacenarse y consultarse valores. Dicha memoria se organiza en celdas, cada una de las cuáles puede almacenar un valor atómico (un entero, un real, un booleano, una cadena, o un puntero). El manejo de dicha memoria se abstraerá mediante las siguientes operaciones:

- **alloc**(τ), con τ un tipo. Se reserva una zona de memoria adecuada para almacenar valores del tipo τ. La operación en sí devuelve la dirección de comienzo de dicha zona de memoria.
- **dealloc**(*d*,τ), con *d* una dirección, y τ un tipo. Se notifica que la zona de memoria que comienza en *d* y que permite almacenar valores del tipo τ queda liberada.
- **fetch**(*d*), con *d* una dirección. Devuelve el valor almacenado en la celda direccionada por *d*.
- **store**(*d*,*v*), con *d* una dirección, y *v* un valor. Almacena *v* en la celda direccionada por *d*.
- **copy**(*d*, *d*', τ), con *d* y *d'* direcciones, y τ un tipo. Copia el valor del tipo τ que se encuentra almacenado a partir de la dirección *d*' en el bloque que comienza a partir de la dirección *d*.
- **indx**(*d*,*i*,τ), con *d* una dirección, *i* un valor, y τ un tipo. Considera que, a partir de *d*, comienza un array cuyos elementos son valores del tipo τ, y devuelve la dirección de comienzo del elemento *i*-esimo de dicho array.
- **acc**(*d*,*c*,τ), con *d* una dirección, *c* un nombre de campo, y τ un tipo **record**. Considera que, a partir de *d*, está almacenado un registro de tipo τ, que contiene un campo *c*. Devuelve la dirección de comienzo de dicho campo.

El *estado de ejecución* del programa asocia direcciones con variables. Mediante **dir**(*u*) denotaremos la dirección de la variable *u*. Inicialmente todas las celdas de la memoria almacenarán un valor indeterminado ⊥. Cuando se utiliza un valor indeterminado en una operación que espera un valor de un determinado tipo, dicho valor podrá concretarse, aleatoriamente, en cualquier valor de dicho tipo. Obsérvese que, aparte de conducir a comportamientos no deterministas, este efecto puede ser particularmente lesivo cuando afecta a la interpretación de punteros. Efectivamente, si ⊥ se interpreta como un puntero, y se accede a la región apuntada por el mismo, dicha región puede ser *cualquier* región del espacio de memoria del programa.

La evaluación de las expresiones da lugar, bien a valores, bien a direcciones (en el caso de que las expresiones se traten de designadores). Esta evaluación obedece a las siguientes reglas:

- La evaluación de un literal da como resultado el valor que representa dicho literal (en concreto, **null** se representará mediante el entero -1).
- La evaluación de una variable *u* de tipo τ da como resultado:
  - **dir**(*u*), si dicha variable no está asociada a un parámetro formal de un procedimiento de la forma **var** *u*:τ (parámetro por *variable*).
  - **fetch**(**dir**(*u*)), en caso de que la variable se corresponda con un parámetro por variable **var** *u*:τ. Efectivamente, en este caso *u* contendrá, en realidad, un puntero al parámetro real (los punteros se representarán mediante valores enteros no negativos, reservándose -1 para **null**)

- Supongamos que en una expresión de la forma $E_0 \circ E_1$, con $\circ$ un operador binario distinto del de asignación (=), τ_0 es el tipo de E_0, τ_1 es el tipo de E_1, τ es el tipo de $E_0 \circ E_1$, r_0 es el resultado de evaluar E_0 y r_1 es el resultado de evaluar E_1. Para determinar el valor de la expresión:
  - Si E_0 es un designador, $v_0 \leftarrow$ **fetch**(r_0). En otro caso, $v_0 \leftarrow r_0$
  - Si E_1 es un designador, $v_1 \leftarrow$ **fetch**(r_1). En otro caso, $v_1 \leftarrow r_1$
  - Si $\circ$ es un operador aritmético (+, -, *, /), y τ es **real**:
    - Si τ_0 es **int**, v_0 se convierte a un valor real equivalente. Si no, se deja como está. Sea v'_0 el valor que resulta de estas consideraciones.
    - Si el tipo de E_1 es **int**, v_1 se convierte a un valor real equivalente. Si no, se deja como está. Sea v'_1 el valor que resulta de estas consideraciones.
    - El valor de la expresión se obtiene realizando la operación $v'_0 \circ v'_1$.
  - En otro caso, el resultado de evaluar la expresión es el valor que se obtiene realizando la operación $v_0 \circ v_1$
- Supongamos que en una expresión de la forma $\circ$E, τ es el tipo de E, y el resultado de evaluar *E* es *r*. Entonces:
  - Si *E* es un designador, $v \leftarrow$ **fetch**(r). Si no, $v \leftarrow r$.
  - El resultado es $\circ v$
- Supongamos que en una expresión de la forma E$\circ$, el tipo de *E* es τ, y el resultado de evaluar *E* es *r*. En este caso:
  - Si $\circ$ es [*E*'], τ debe ser un tipo **array** [*n*] **of** τ'. Sea *r*' el resultado de evaluar *E*'. Entonces:
    - Si *E*' es un designador, $i \leftarrow$ **fetch**(*r*'). Si no, $i \leftarrow r$'.
    - El resultado es **indx**(*r*, *i*, τ')
  - Si $\circ$ es *.c*, τ debe ser un tipo **record**, que contiene un campo *c*. De esta forma, el resultado será **acc**(*r*, *c*, τ)
  - Si $\circ$ es $\wedge$, τ deberá ser $\wedge\tau$'. En este caso, el resultado será **fetch**(*r*) siempre y cuando $r \neq -1$. Si $r=-1$ se producirá un error de ejecución (intento de acceso a través de **null**).
- Por último, consideremos una expresión de *asignación* de la forma $E_0 = E_1$. Sea τ_0 el tipo de E_0 (que debe ser un designador), y τ_1 el tipo de E_1. Sean, así mismo, r_0 el resultado de evaluar E_0 y r_1 el resultado de evaluar E_1. Entonces:
  - Si τ_1 es **int** y τ_0 es **real**:
    - $v_1 \leftarrow$ **fetch**(r_1) si E_1 es un designador. Si no, $v_1 \leftarrow r_1$
    - v_1 se convierte al valor real equivalente: sea v_1' dicho valor.
    - **store**(r_0, v_1')
  - En otro caso:
    - Si E_1 es un designador, **copy**(r_0, r_1, τ_0)
    - En otro caso, **store**(r_0, r_1)
  - El valor de la expresión es r_0

En lo referente a las instrucciones:

- La ejecución de una instrucción `@` *E* consiste en evaluar *E* y desechar su resultado.
- La ejecución de una instrucción de lectura **read** E supone:
  - Evaluar E. Sea *r* el resultado (será una dirección).
  - Leer un valor apropiado *v* de la entrada estándar, dependiendo del tipo de E.
  - **store**(r_0,*v*)
- La ejecución de una instrucción de escritura **write** E supone:
  - Evaluar E para obtener un resultado *r*
  - Si E es un designador, $v \leftarrow$ **fetch**(*r*). Si no, $v \leftarrow r$.
  - Mostrar *v* por la salida estándar.
- La ejecución de una instrucción de reserva de memoria **new** E, con E un designador de tipo $\wedge\tau$, supone:
  - Evaluar E para obtener una dirección *d*.
  - **store**(*d*, **alloc**(τ))
- La ejecución de una instrucción de liberación de memoria **delete** E, con E un designador de tipo $\wedge\tau$, supone:
  - Evaluar E para obtener una dirección *d*.
  - **dealloc**(*d*, τ), si $d \neq -1$. Error de ejecución si $d = -1$

- La ejecución de una instrucción de llamada a procedimiento **call** p (E_0, …, E_k) con *k* parámetros reales supone:
  - Para cada parámetro formal con nombre *u*, si ya existe una variable *u*, debe salvaguardarse **dir**(u).
  - Para cada parámetro formal en modo valor τ_i u_i:
    - **dir**(u_i) ← **alloc**(τ_i)
    - Evaluar el correspondiente parámetro real E_i. Sea τ^R_i el tipo de E_i y sea r_i el resultado de la evaluación:
      - Si τ_i es **real** y τ^R_i es **int**:
        - Si E_i es un designador, v_i ← **fetch**(r_i). Si no, v_i ← r_i
        - Convertir v_i a real. Sea v'_i el resultado de dicha conversión
        - **store**(**dir**(u_i), v'_i)
      - en otro caso:
        - Si E_i es un designador, **copy**(**dir**(u_i), r_i, τ_i)
        - En otro caso, **store**(**dir**(u_i), r_i)
  - Para cada parámetro formal en modo referencia τ_i & u_i :
    - **dir**(u_i) ← **alloc**(**int**) (efectivamente, los parámetros por variable se tratan como punteros, por lo que únicamente requieren una celda).
    - **store**(**dir**(u_i), r_i) (el valor del parámetro formal será la dirección de comienzo del parámetro real)
  - Se ejecuta el bloque asociado al procedimiento.
  - Para cada parámetro formal τ_i u_i:
    - **dealloc**(**dir**(u_i),τ_i)
    - Si, al comienzo de la ejecución del procedimiento se salvaguardó **dir**(u_i), restaurar dicha dirección.
  - Para cada parámetro formal τ_i & u_i :
    - **dealloc**(**dir**(u_i),**int**)
    - Si, al comienzo de la ejecución del procedimiento se salvaguardó **dir**(u_i), restaurar dicha dirección.
- La ejecución de una instrucción **if** E B supone:
  - Evaluar *E*. Sea *r* el resultado de dicha evaluación.
  - Si *E* es un designador, *v* ← **fetch**(*r*). Si no, *v*←*r*.
  - Si *v* es **true**, entonces debe ejecutarse B.
  - Si *v* es **false**, no se hace nada más.
- La ejecución de una instrucción **if** E B_1 **else** B_2 supone:
  - Evaluar *E*. Sea *r* el resultado de dicha evaluación.
  - Si *E* es un designador, *v* ← **fetch**(*r*). Si no, *v*←*r*.
  - Si *v* es **true**, entonces debe ejecutarse B_1.
  - En otro caso, debe ejecutarse B_2.
- La ejecución de una instrucción **while** E B supone:
  - Evaluar *E*. Sea *r* el resultado de dicha evaluación.
  - Si *E* es un designador, *v* ← **fetch**(*r*). Si no, *v*←*r*.
  - Si *v* es **true**, entonces debe ejecutarse el blque B, y, seguidamente, ejecutar de nuevo la instrucción **while** E B
  - Si *v* es **false**, no se hace nada.
- La ejecución de una instrucción compuesta asociada al bloque B supone ejecutar este bloque.

En lo referente a la ejecución de un bloque:

- Para cada declaración de variable τ *u* en la sección de declaraciones del bloque:
  - Si existe otra variable con el mismo nombre *u*, debe salvaguardarse **dir**(u).
  - **dir**(u) ← **alloc**(τ)
- Ejecutar en orden las instrucciones de la sección de instrucciones de B.
- Para cada declaración de variable τ *u* en la sección de declaraciones del bloque:
  - **dealloc**(**dir**(u),τ)
  - Si, al comienzo de la ejecución se salvaguardó **dir**(u), restaurar dicha dirección.

La ejecución del programa consiste en ejecutar su bloque asociado.

Programa de ejemplo

```
{
type ^tNodo tArbol;
type struct{
       string nombre,
       tArbol izq,
       tArbol der
     } tNodo;
type struct {
       string [50] nombres,
       int cont
      } tListaNombres;

tListaNombres nombres;   ## Aquí se guardarán los nombres leídos (max. 50)
tArbol arbol;            ## Aquí se construirá un árbol de búsqueda que contendrá
                         ## los nombres leídos, sin duplicados

   ## Lee los nombres a ordenar (max. 50 nombres)
proc lee_nombres(tListaNombres & nombres) {
  int i
  &&
      write "Introduce el número de nombres a ordenar (max 50): ";
      nl;
      read nombres.cont;
      while (nombres.cont < 0) or (nombres.cont > 50) {
         write "Introduce el número de nombres a ordenar (max 50): "; nl;
         read nombres.cont
      };
      @ i=0;
      write "Introduce un nombre en cada línea: "; nl;
      while i < nombres.cont {
        read nombres.nombres[i];
        @ i = i + 1
      }
 };  ## fin del procedimiento lee_nombres


  ## Construye un árbol de búsqueda sin duplicados con los nombres leídos
  ## Hace un uso global de las variables 'nombres' y 'arbol' declaradas en
  ## el programa principal
proc construye_arbol() {
  int i; ## para iterar sobre la lista de nombres

     ## Inserta el nombre actual en el árbol de búsqueda que recibe como parámetro.
     ## Hace un uso global de la variable 'nombres' declarado en el programa principal,
     ## y en del contador 'i' declarado en el subprograma contenedor 'construye_arbol'
  proc inserta_nombre(tArbol & arbol) {
    if arbol == null {
        new arbol;
       @ arbol^.nombre = nombres.nombres[i];
       @ arbol^.izq = arbol^.der = null
    }
    else {
       tArbol padre;  ## apuntará al nodo padre del nuevo nodo a insertar
       tArbol act;    ## para recorrer la rama al final de la cuál debe realizarse
                      ## la inserción.
       bool fin       ## para controlar el final del recorrido de la rama
       &&
        @ fin = false;
        @ padre = null;
        @ act = arbol;
         while not fin {
            @ padre = act;
            if act^.nombre < nombres.nombres[i] { ##insertar en el hijo derecho
               @ act = act^.der
            }
            else {
              if act^.nombre > nombres.nombres[i] { ##insertar en el hijo izquierdo
                 @ act = act^.izq
              }
            };
            if act == null { ## se ha alcanzado el punto de inserción
                 @ fin = true
            }
            else {
```

```
            if act^.nombre == nombres.nombres[i] {  ## el nombre está duplicado
               @ fin = true
            }
          }; ## fin del while


         if act == null { ## se ha alcanzado un punto de inserción
                          ## hay que insertar un nuevo nodo
             if padre^.nombre < nombres.nombres[i] { ## insertar como hijo izquierdo
                new padre^.der;
                @ padre^.der^.nombre = nombres.nombres[i];
                @ padre^.der^.izq = padre^.der^.der = null
             }

            else {                                  ## insertar como hijo derecho
                new padre^.izq;
                @ padre^.izq^.nombre = nombres.nombres[i];
                @ padre^.izq^.izq= padre^.izq^.der = null
            }
          }
        }
       }
      } ## del procedimiento anidado inserta_nombre
   &&
      ## cuerpo del procedimiento construye_arbol
     @ arbol = null;
     @ i=0;
     while i < nombres.cont {
       call inserta_nombre(arbol);
       @ i = i + 1
     }
  };  ## del procedimiento construye_arbol

    ## Escribe los nombres almacenados en el árbol de búsqueda, recorriendo
    ## dicho árbol en inorden.
    ## Por tanto, los nombres se listan ordenados alfabéticamente,
    ## y sin duplicados
  proc escribe_nombres(tArbol arbol) {
      if arbol != null {
         call escribe_nombres(arbol^.izq);
         write arbol^.nombre; nl;
         call escribe_nombres(arbol^.der)
      }
  } ## del procedimiento escribe_nombres
 &&
      ## Programa principal

    call lee_nombres(nombres);
    call construye_arbol();
    write "Listado de nombres ordenado"; nl;
    write "---------------------------"; nl;
    call escribe_nombres(arbol)

} ## fin programa
```