Final TLP

WHILE20 es una extensión del lenguaje WHILE. Se trata de un lenguaje de programación imperativo, estructurado en bloques y con ámbito estático.

· Problema 1

Las expresiones aritméticas de WHILE20 incluyen un operador *let...in* que permite definir variables locales a una expresión. La nueva sintaxis es:

$$a ::= \text{let } D_v \text{ in } a \mid \ldots$$
$$D_v ::= \text{var } x := a; D_v \mid \varepsilon$$

donde 'a' es una expresión aritmética, D_v son las variables locales utilizadas en la definición de 'a' y ε la cadena vacía.
Por ejemplo:

```
n := let
        var x := 1;
        var y := 2;
     in
        x + y;

// n = 3
```

```
z := let
        var a := 2;
        var b := 2*a;
        var c := let
                    var a := 6;
                 in
                    b + a;
     in
        a * c;

// z = 20
```

Las variables locales D_v sólo están disponibles durante la evaluación 'a'. Estas variables locales deben gestionarse mediante ámbito estático, usando un entorno de variables (locations) y store:

a) (0.5) Especifica formalmente la semántica natural de Aexp. Sólo es necesario que especifiques la semántica de los numerales, variables, suma y *let...in*.
b) (1) Define las funciones aVal y bVal que implementan la semántica natural de las expresiones aritméticas y booleanas.

· Problema 2

Los lenguajes estructurados en bloque que hemos visto en clase carecen de variables globales. Esto implica que dado un programa S, el judgement inicial tendrá la forma:

$$env_v \vdash < S, sto > \rightarrow sto'$$

donde env_v y sto están vacíos. El estado final queda reflejado en sto' pero, dado que env_v está vacío, solo podemos obtener un volcado de memoria para consultarlo.

Para paliar ese problema, el lenguaje WHILE20 permite definir variables globales. Para ello se define una nueva categoría sintáctica *Prog* cuya definición formal es:

$$P ::= \text{program pname } D_v\ S$$

donde *pname* es el nombre del programa, D_v las declaraciones de variables globales (accesibles desde todo el programa) y S es el cuerpo del programa. Por ejemplo:

```
program Factorial
        var x := 3;   // global var.

        var y := 1;
        while (x != 0) do {    //body
                y := y*x;
                x := x-1;
}
```

```
program FactorialWithLocal
        var x := 3;   // global var.

var y := 1;
begin
        var z := 1;
        while (z <= x) do {
                y := y*z;
                z := z+1;
        }
end
```

Observa que el cuerpo del programa puede o no ser un bloque. El programa *Factorial* realiza todos los cómputos con variables globales (que realmente no forman parte de ningún bloque); el programa *FactorialWithLocal* realiza los cómputos usando una variable local z.

Las variables globales permiten definir un estado inicial no vacío en el que ejecutar el cuerpo del programa,

$$S: \quad env_{gv} \vdash \langle S, sto_{gv} \rangle \rightarrow sto'$$

donde env_{gv} y sto_{gv} contienen las variables globales. De nuevo el estado final queda reflejado en sto' pero ahora podemos utilizar env_{gv} para consultar la localización y valor de una variable global.

a) (1) Define la función *execProgram* que procesa las variables globales de un programa para computar su estado inicial (env_{gv} y sto_{gv}) e invoca a la semántica natural (nsStm) para obtener el estado final. Utiliza la función *showFinalState* para ejecutar los programas de ejemplo y obtener un informe sobre el estado final.

b) (1) Observa un problema de la semántica de WHILE20. Si comparamos las salidas de los programas *Factorial* y *FactorialWithLocal*

```
Program Factorial executed.
Global variables:
        var: x   location: 1   value: 0
        var: y   location: 2   value: 6
Number of cells used: 2
Memory dump: (1,0) (2,6)
```

```
Program FactorialWithLocal executed.
Global variables:
        var: x   location: 1   value: 3
        var: y   location: 2   value: 6
Number of cells used: 3
Memory dump: (1,3) (2,6) (3,4)
```

podemos apreciar que en estado final la variable local z aún sigue presente en el store, ocupando la posición 3. Esto no tiene sentido.

Modifica la semántica natural de WHILE20 de forma que las variables locales se eliminen del store. Explica en el fichero *NaturalSemanticswhile20* en qué consiste la modificación y qué reglas se ven

afectadas. Para cada regla afectada, define la semántica natural de la nueva regla en un comentario, implementa la nueva regla y anula la anterior implementación.

· Problema 3

El lenguaje WHILE20 dispone de una sentencia iterativa con la siguiente sintaxis:

$$S ::= \text{loop } S_1 \text{ exit when } b\ S_2$$

y se ejecuta de nuevo el bucle.

a) (0.5) Especifica formalmente la semántica natural de loop.
b) (1.5) Implementa la semántica natural de loop.

· Problema 4

Se desea añadir a WHILE20 una nueva sentencia selectiva no determinista cuya sintaxis es:

$S ::= \text{if } G_s \text{ fi} \mid \ldots$
$G_s ::= b \rightarrow S; G_s \mid \varepsilon$

donde b → S es una sentencia guardada: S sólo se puede ejecutar si la guarda b que la precede es cierta. Para ejecutar una selección no determinista se procede de la siguiente manera:
- Si hay varias sentencias guardadas cuya guarda sea cierta, se selecciona una de ellas de manera no determinista y se ejecuta esta sentencia; el resto de sentencias guardadas se ignora.
- Si no hay ninguna sentencia guardada cuya guarda sea cierta, el programa aborta la ejecución.

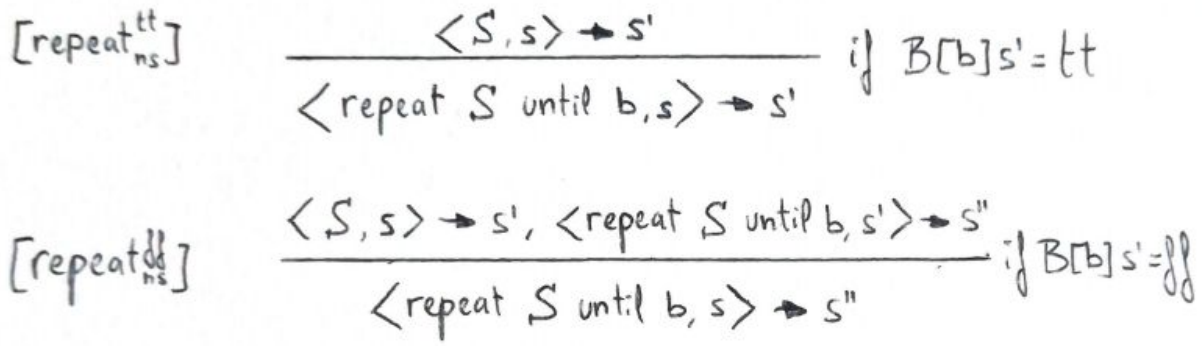
Por ejemplo:

```
if
        x>0 → y := 1;
        x>5 → y := 2;
        x>10 → y := 3;
fi
```

Si x=3, el único resultado posible es y=1. Si x=7 hay 2 resultados posibles: y=1 y y=2, pero no podemos predecir cual. Si x = 15 habrá hasta 3 respuestas posibles. Si x = 0 el programa interrumpe la ejecución.
(1) Define formalmente la semántica natural de la sentencia selectiva no determinista en un comentario del fichero *NaturalSemanticsWhile20*.

· Problema 5

Semántica natural de *repeat*:



(2.5) Demuestra que repeat S until b es semánticamente equivalente a S; while (¬b) do S según la semántica natural. Sugerencia: usa inducción sobre el número de veces que se ejecuta la sentencia S.

· Problema 6

Supongamos que $< S_1; S_2 , s > \Rightarrow^* < S_2 , s' >$.

(1) Muestra que no es necesariamente cierto que $< S_1 , s > \Rightarrow^* s'$.