Examen final de CL Enero de 2011

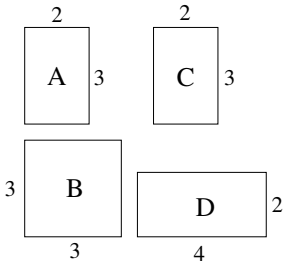
Fecha de publicación de notas: 19-1-2011 Fecha de revisión: 20-1-2011

Sin apuntes. Tiempo: 3h. Nombre y Apellidos:

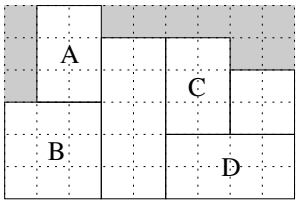
Problema de análisis léxico, sintáctico, intérpretes y análisis semántico [5 puntos]

Deseamos diseñar una calculadora sencilla para representar distribuciones de objetos *rectangulares* en superficies bidimensionales. Para ello utilizaremos el concepto de *diseño seccionable* (DS). Un DS puede definirse recursivamente de la siguiente forma:

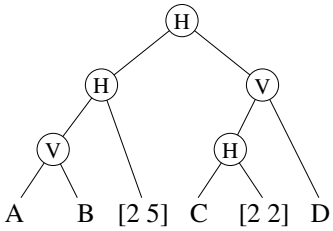
- Un bloque rectangular es un DS.
- Si un diseño se puede seccionar en dos partes (con un corte horizontal o vertical) y cada parte es un DS, entonces el diseño completo también es un DS.



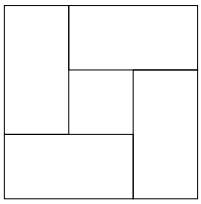
(a)



(b)



(c)


(d)

Para especificar DSs disponemos de una gramática que describimos con ejemplos:

```
A = [2 3];    # Describe el objecto A de la figura (a) con las dimensiones [x y]
B = [3 3]; C = [2 3]; D = [4 2]; # Describe los otros objetos de la figura (a)
```

Con la misma gramática podemos describir composiciones de DSs, por ejemplo:

```
Block1 = A / B;   # Un DS donde A está por encima de B
Block2 = C | D;   # Un DS donde C está a la izquierda de D
Block3 = Block1 | C / [3 4]; # Un bloque mas complejo con diversas composiciones
```

Se puede observar que en una expresión de un DS pueden aparecer DSs definidos anteriormente o nuevos rectángulos (sin nombre) declarados en la misma expresión (por ejemplo `[3 4]`). El DS de la figura (b) puede ser descrito con la siguiente expresión:

```
A/B | [2 5] | (C | [2 2]) / D
```

En esta gramática, el operador `/` tiene mas prioridad que el operador `|`. Además, ambos operadores son asociativos por la izquierda. Es necesario utilizar paréntesis para modificar estas reglas.

Observación: no todos los diseños son seccionables. Por ejemplo, el diseño de la figura (d) no lo es, dado que no existe ninguna disección que lo divida en dos partes.

Nos interesa realizar cálculos con los DSs. Los dos operadores que disponemos para realizar cálculos son `BoundingBox` y `Utilization`. El primero escribe (por el canal de salida) las dimensiones del rectángulo mas pequeño que incluye un DS. El segundo escribe el porcentaje de utilización del rectángulo. Este porcentaje se puede calcular dividiendo el área total de los rectángulos de la expresión por la del "BoundingBox"de la expresión. La gramática para estos dos operadores es la siguiente:

`BoundingBox` *expr*`;`
`Utilization` *expr*`;`

Ejemplos, utilizando las definiciones anteriores:

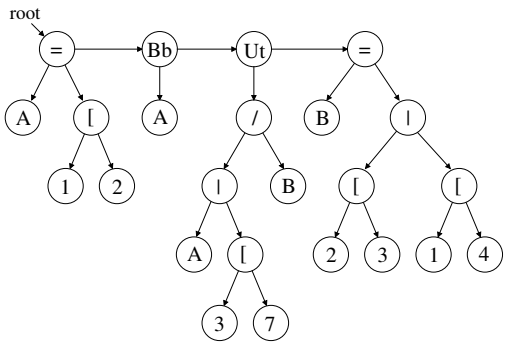
```
BoundingBox A;
[2 3]
BoundingBox Block1;
[3 6]
Utilization A;
100%
Utilization Block1;
83.33%
BoundingBox A/B | [2 5] | (C | [2 2]) / D;
[9 6]
Utilization A/B | [2 5] | (C | [2 2]) / D;
79.63%
```

A partir de su definición recursiva, todo DS se puede representar como un árbol donde los nodos representan disecciones (horizontales o verticales) y las hojas representan rectángulos. Como ejemplo se puede observar el árbol de la figura (c) que representa el DS de la figura (b).

Para la interpretación de esta calculadora, se representarán las definiciones de rectángulos y operaciones con un árbol sintáctico similar al que se muestra a continuación (el dibujo es solo una representación gráfica aproximada de la estructura real del árbol en PCCTS):

```
A = [1 2];
BoundingBox A;
Utilization (A | [3 7]) / B;
B = [2 3] | [1 4];
```

$\Longrightarrow$



---

1. **Análisis léxico [0.5 puntos]**. Escribir las descripciones léxicas de los siguientes tokens: `INTCONST`, `IDENT`, `LPAR`, `RPAR`, `LBRACKET`, `RBRACKET`, `HORCUT`, `VERCUT`, `BBOX`, `SEMICOLON`, `UTIL` y `ASSIGN`. Suponer que los tokens `WHITESPACE` y `EOF` ("@") ya han sido definidos y que los comentarios ya han sido tratados adecuadamente (se pueden ignorar). Indicar además cuál debería ser el orden de las declaraciones de tokens para que el análisis léxico fuese correcto.

2. **Análisis sintáctico [1.5 puntos]**. Completar la gramática de la calculadora a partir de las siguientes reglas:

   ```
   calc: ((assig_instruction | bb_instruction | ut_instruction) SEMICOLON!)* EOF!;
   assig_instruction: IDENT ASSIGN^ expr;
   bb_instruction: BBOX^ expr;
   ut_instruction: UTIL^ expr;
   ```

3. **Interpretación [1.5 puntos]**. Suponer que se han realizado las comprobaciones semánticas adecuadas (no existen múltiples definiciones de un mismo símbolo y no existen definiciones cíclicas de DSs). Completar las funciones `BoundingBox` y `Utilization` en el código que se detalla a continuación. Utilizar la función `FindAssign` para localizar el nodo del AST que define la expresión para un DS con un determinado nombre.

Sin apuntes. Tiempo: 3h. Nombre y Apellidos:

```
int stringtoint(string s); // Converts string into integer
AST* child(AST* T, int n); // Returns the n-th child of T

// Defines the attributes of an AST node
void create_attr(Attrib *attr, int token, char *text) {
  switch (token) {
    case IDENT: attr->kind="ident"; attr->text=text; break;
    case INTCONST: attr->kind="intconst"; attr->text=text; break;
    default: attr->kind=lowercase(text); attr->text=""; break;
  }
}

// Returns the AST node of the ASSIGN statement
// that defines the DS with the specified Name
AST* FindAssign(string Name);

struct BBox {int x, y;}; // Structure to represent a bounding box

BBox BoundingBox(AST* a) {
    BBox r;
    if (a->kind == "[") {
        r.x = stringtoint(child(a,0)->text);
        r.y = stringtoint(child(a,1)->text);
    } else
        ...
        ... // Completar
        ...
    return r;
}

double Utilization(AST *a) {
   ...
   ...  // Completar
   ...
}

void Calculator(AST* Root) {
    for (AST* a = root; a; a = a->right) {
        if (a->kind == "boundingbox") {
            BBox b = BoundingBox(a->down);
            cout << "[" << b.x << " " << b.y << "]" << endl;
        } else if (a->kind == "utilization") {
            double u = Utilization(a->down);
            cout << u * 100.0 << "%" << endl;
        }
    }
}
```

4. **Análisis semántico [1.5 puntos]**. Deseamos detectar si una definición de rectángulo es cíclica (genera un árbol infinito). Por ejemplo:

```
A = [2 5]; B = [1 2] | C; C = B / A;
```

es una definición cíclica. Sin embargo,

```
A = [2 5] | B; B = C | D; C = [3 4]; D = C / [2 3];
```

no es cíclica. Diseñar la siguiente función:

```
bool IsCyclic (AST *Tassign);
```

que indica si un nodo del tipo ASSIGN contiene una definición cíclica de un DS. Indicar los atributos de los nodos del AST o las estructuras de datos adicionales necesarias para implementar la función `IsCyclic`. Utilizar la función `FindAssign` para localizar el nodo del AST que define la expresión para un DS con un determinado nombre.

Sin apuntes. Tiempo: 3h. Nombre y Apellidos:

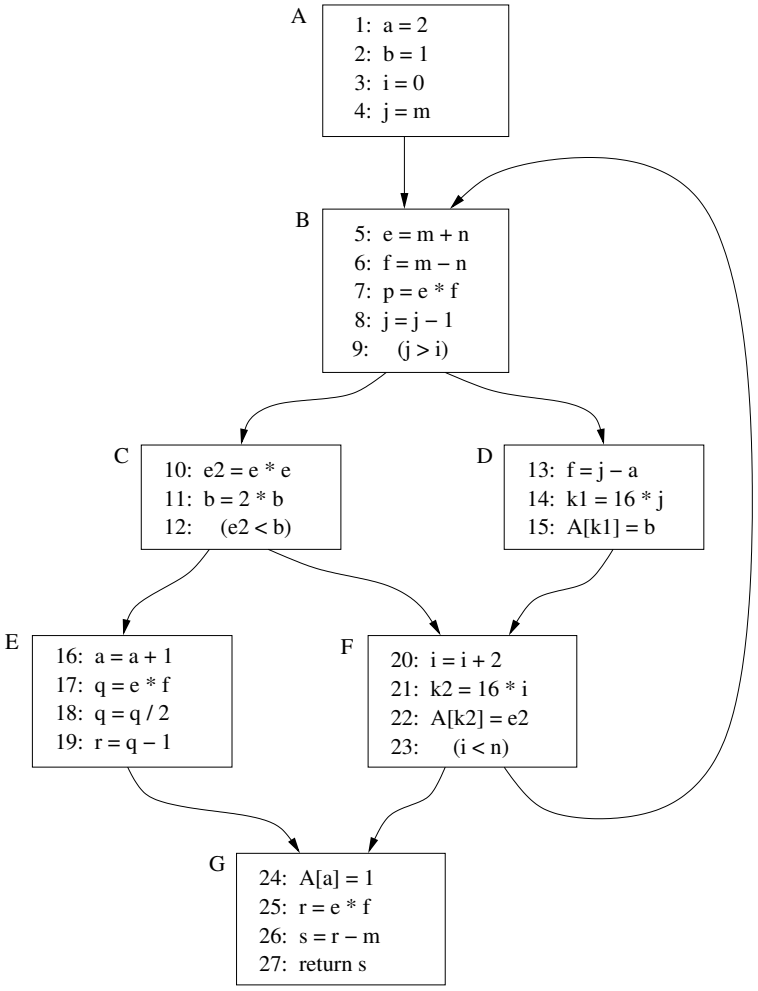
Problema de optimización [3 puntos]

Dado el grafo de control de flujo de una cierta función que vemos más abajo, resolver los siguientes apartados. Considerar que tras la instrucción `return` las únicas variables vivas son `A`, `p` y `q`.

- [0.6 puntos] Dibujar el árbol de dominadores

- [0.6 puntos] Enumerar las variables vivas al principio del bloque `F`.

- [1.8 puntos] Aplicar, de una en una, todas las técnicas de optimización posibles para obtener un código mejorado.



Sin apuntes. Tiempo: 3h. Nombre y Apellidos:

Problema de generación de código [2 puntos]

En este problema tratamos la traducción de la instrucción de control `switch`, parecida a la de lenguajes como C o Java, con diferentes alternativas para algunos valores de la expresión evaluada, y en la que al final puede haber un caso `default`.
A continuación vemos un pequeño ejemplo:

```
switch (A[i+1]) {
  case 11: x = 1; y = 1; endcase;
  case 12: x = 1; y = y/2; endcase;
  case  3: x = 0; y = y*y; endcase;
  case  7: x = 7; y = x; endcase;
  default: x = 0; y = 1;
}
```

En primer lugar vemos [parte de] una versión preliminar del esquema de traducción estándar, es decir traduciendo el `switch` mediante una secuencia de `if-else-if`. El temporal `t0` contiene inicialmente el valor de la expresión *expr*.

$$
\begin{aligned}
&\mathit{GenCode}(\ \texttt{switch}(\ \mathit{expr}, \langle\, v_1 \times \mathit{l\_instrs}_1 \rangle, \ldots, \langle\, v_n \times \mathit{l\_instrs}_n \rangle, \mathit{l\_instrs}_d\ )\ ) \ \equiv \\
&\qquad /*\ k = \mathit{newLabelSwitch}();\ */ \\
&\qquad\quad \ldots \quad \texttt{||} \\
&\texttt{etiq sw\_}k\texttt{\_cas\_num1 ||} \\
&\quad \texttt{equi t0 }v_1\texttt{ t0 ||} \\
&\quad \texttt{fjmp t0 sw\_}k\texttt{\_cas\_num2 ||} \\
&\quad \mathit{GenCode}(\mathit{l\_instrs}_1)\ \texttt{||} \\
&\quad \texttt{ujmp sw\_}k\texttt{\_end ||} \\
&\texttt{etiq sw\_}k\texttt{\_cas\_num2 ||} \\
&\qquad \ldots \quad \texttt{||} \\
&\texttt{etiq sw\_}k\texttt{\_cas\_num}n\ \texttt{||} \\
&\quad \texttt{equi t0 }v_n\texttt{ t0 ||} \\
&\quad \texttt{fjmp t0 sw\_}k\texttt{\_def ||} \\
&\quad \mathit{GenCode}(\mathit{l\_instrs}_n)\ \texttt{||} \\
&\quad \texttt{ujmp sw\_}k\texttt{\_end ||} \\
&\texttt{etiq sw\_}k\texttt{\_def ||} \\
&\quad \mathit{GenCode}(\mathit{l\_instrs}_d)\ \texttt{||} \\
&\texttt{etiq sw\_}k\texttt{\_end}
\end{aligned}
$$

Responder breve y razonadamente a las siguientes preguntas:

- Identifica el error que aparece en el esquema de traducción.

- ¿Es correcto el esquema cuando los `switch`s son anidados?

- ¿Teniendo en cuenta este esquema de traducción, qué consejo podemos dar a un programador cuando escribe un `switch`?

Alternativa: salto calculado. Muchos compiladores permiten traducir la instrucción `switch` de una forma más eficiente que la que resulta de utilizar el esquema anterior. Se trata de utilizar una tabla para realizar un salto directamente hasta la lista de instrucciones correspondiente al valor de la expresión.

Más abajo vemos parte del t-código resultado de traducir el `switch` de la página anterior mediante salto calculado. En él aparece una nueva instrucción `cjmp t0 sw_tab` (computed `jump`) con dos operandos, un temporal y una etiqueta, y cuya semántica es la siguiente: hace un salto incondicional a la instrucción situada `t0` instrucciones más allá de la correspondiente a la etiqueta `sw_tab`. Por ejemplo, si `t0` contiene el valor 8, `cjmp t0 sw_tab` salta a la instrucción `ujmp sw_cas_val_11`.

```
      ...
    cjmp t0 sw_tab
  etiq sw_tab          // Etiqueta de comienzo de la tabla de salto calculado
    ujmp sw_cas_val_3
    ujmp sw_def
    ujmp sw_def
    ujmp sw_def
    ujmp sw_cas_val_7
    ujmp sw_def
    ujmp sw_def
    ujmp sw_def
    ujmp sw_cas_val_11
    ujmp sw_cas_val_12
  etiq sw_cas_val_11
      ...
```

Especificar ahora el esquema de traducción de la instrucción `switch` con esta técnica, **comentando como siempre entre `/* ... */` las acciones que realiza el compilador**.

$$GenCode(\ switch(\ expr,\ \langle\ v_1 \times l\_instrs_1 \rangle\ ,\ \ldots,\ \langle\ v_n \times l\_instrs_n \rangle\ ,\ l\_instrs_d\ )\ )\ \equiv$$

Responder breve y razonadamente a las siguientes preguntas:

- **Disminución sólo del coste temporal.** ¿Cuál de los t-códigos generados por ambas versiones tiene un menor coste temporal en el caso peor?

- **Disminución sólo del coste espacial.** Define los parámetros que determinan el coste espacial *específico* de cada opción, considerando como tal el número de instrucciones del t-código generado pero sin tener en cuenta las instrucciones comunes a las dos opciones —como por ejemplo las instrucciones de cada `case`—.
  Suponiendo que el compilador intenta generar el t-código más reducido posible, calcula en función de esos parámetros cuándo se usará un esquema y cuándo el otro.