Examen final de CL 11 de Enero de 2012

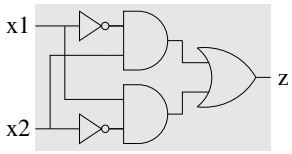
Fecha de publicación de notas: 19-1-2012 Fecha de revisión: 20-1-2012

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

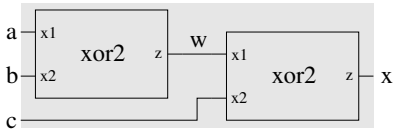
Problema de analisis léxico, sintáctico e intérpretes [5 puntos] Deseamos crear una herramienta de diseño de circuitos que permita especificar su lógica con un lenguaje de descripción de hardware.

Cada circuito se representará como un *módulo* con un interfaz de señales lógicas de entrada/salida. Cada módulo estará implementado con un conjunto de equaciones lógicas y otros módulos.

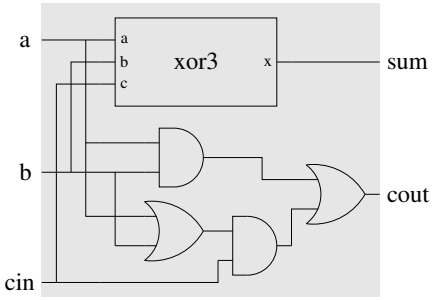
Así por ejemplo, los circuitos de la figura podrán ser descritos de la siguiente forma:



xor2



xor3



full_adder

```
module xor2 (input x1, input x2, output z)
  z = not x1 and x2 or x1 and not x2;
endmodule

module xor3 (input a, input b, input c, output x)
  wire  w;
  xor2 (a, b, w);
  xor2 (w, c, x);
endmodule

module full_adder (input a, input b, output sum, input cin, output cout)
  xor3 (a, b, cin, sum);
  cout = a and b or cin and (a or b);
endmodule
```

Dentro de cada módulo pueden declararse señales internas (*wire*) para conectar diversos módulos o ecuaciones. Así por ejemplo, el módulo `full_adder` podría también describirse de la siguiente forma:

```
module full_adder (input a, input b, output sum, input cin, output cout)
  wire x, y;
  xor3 (a, b, cin, sum);
  x = a and b;
  y = a or b;
  cout = x or cin and y;
endmodule
```

1. Gramática

Diseñar la gramática del lenguaje utilizando EBNF. Para ello, suponer que:

- Los operadores para expresiones lógicas son *not*, *and* y *or*, utilizando este orden de prioridad en la evaluación, tal como se define en los lenguajes de programación convencionales.
- Solo existe una declaración de *wire* con todas las señales internas del módulo.

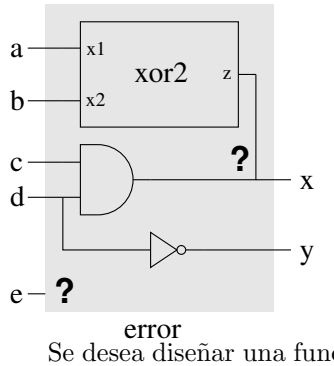
Utilizar las siguientes definiciones y completar la gramática:

```
Circuit : (Module)* ;

Module : "module" IDENT '(' FormalParams ')'
             WireDeclaration
             Components
         "endmodule" ;

Components : (ModuleOrEquation ';')+ ;

ModuleOrEquation : ModuleInstance | Equation ;

Equation : IDENT "=" Expression

...
```

2. First y Follow

Calcular *First* y *Follow* de los siguientes símbolos no terminales: `FormalParams`, `WireDeclaration` y `Expression`.

3. Comprobaciones semánticas




```
module error (input a, input b, input c, input d, input e,
              output x, output y)
    xor2 (a, b, x);
    x = c and d;
    y = not d;
endmodule
```

Se desea diseñar una función que compruebe ciertas propiedades en un circuito. La figura previa muestra un ejemplo de los dos errores que deberán comprobarse.

El primero consiste en que una de las señales del circuito no sea utilizada en ninguno de los componentes o ecuaciones. Este es el caso de la señal e en el circuito. Esta comprobación deberá realizarse tanto para la señales de entrada (*input*) como las internas (*wire*).

El segundo error consiste en que alguna de las señales esté conectada a mas de una salida de una función lógica, ya sea generada por el propio módulo o por otro módulo conectado a este. Este es el caso de la señal x en el circuito, que está generada por la instancia *xor2* y por la ecuación `x = c and d`. Un error similar ocurriría si una de las entradas del circuito estuviese conectada a la salida de una función lógica.

Diseñar las siguientes dos funciones:

```
bool AllSignalsConnected (Module &M);

bool SingleDrivenSignals (Module &M);
```

La primera función comprueba que todas las señales de un módulo están conectadas. La segunda función comprueba que no hay mas de una función lógica generando una señal.

Para diseñar dichas funciones, se puede suponer que el circuito está representado con un árbol semántico utilizando las siguientes estructuras:

```
struct Module {
  vector<string> Interface;   // input/output signals (in order of declaration)
  vector<bool>   InOut;       // same size as Interface (true if input, false if output)
  vector<string> Wires;       // Set of internal signals
  vector<Equation> Equations; // Equations inside the module
  vector<ModuleInstance> Modules;  // Module instances
};

struct Equation {
  // Represents:  signal = expression
  string signal;
  Expression expr;
};

struct Expression {
  // Structure to represent expressions as trees.
  // Leaves only represent identifiers.
  int type;          // 0: ident, 1: not, 2: and, 3: or
  Expression* Left;  // used for not, and, or (NULL for ident)
  Expression* Right; // used for and, or (NULL for ident, not)
  string Ident;      // only used for ident
};

struct ModuleInstance {
  // Represents: name (ActualParams)
  string name;
  vector<string> ActualParams;
};
```

Todos los módulos declarados en un circuito están almacenados en una tabla a la que puede accederse por el nombre del módulo:

```
map<string, Module*> Circuit;
```

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

Problema de generación de código [2 puntos]

1. Asserts

Explica formalmente (mediante GenRight, GenLeft, etc...) como sería la generación de código para instrucciones del estilo `assert(boolean_expression)` cuya semántica es: la expresión Booleana es evaluada y en caso de no ser cierta el programa suspenderá su ejecución.

2. Iteradores sobre arrays

Explica formalmente (mediante GenRight, GenLeft, GenCode, etc...) como sería la generación de código en el caso de disponer de iteradores de arrays simples de enteros. Disponemos de los siguientes operadores, funciones e instrucciones nuevas del lenguaje:

Operadores	**Comportamiento**
`!i`	Devuelve el valor apuntado por el iterador `i`.
`i1 != i2`	Devuelve `true` si ambos iteradores apuntan a diferentes posiciones del *array*.

Instrucciones	**Comportamiento**
`i++`	Avanza en una posición el iterador `i`.

Funciones	**Comportamiento**
`first(a)`	Dado un *array* `a`, devuelve un iterador apuntando al primer elemento.
`last(a)`	Dado un *array* `a`, devuelve un iterador apuntando al elemento **posterior** al último elemento.

En concreto, el siguiente programa describe un ejemplo:

```
Program
 Vars
   a array [10] of Int
   it IntIterator
   max Int
 EndVars
```

```
 it := first(a)
 max := !it
 it++
 While (it != last(a)) Do
   If (max < !it) Then max := !it EndIf
   it++
 EndWhile
 WriteLn(max)
EndProgram
```

Describe formalmente la generación de código para cada una de las nuevas construcciones descritas en la tabla anterior.

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

Problema de optimización [3 puntos]