Trabajo Práctico 1
OBDDs

Organización del Computador 2

Segundo Cuatrimestre 2017

1. Introducción

El *teorema de expansión de Boole* también conocido como *expansión de Shannon* permite definir una función que predica sobre variables bi valuadas como descomposición de otras dos funciones que predican sobre un conjunto estrictamente más pequeño de variables. Si contamos con un operador de restricción $f|_{x=b}$ que reemplaza todas las ocurrencias de x en f por el valor $b \in true, false$, la función f que predica sobre el conjunto $\mathcal{X} = \{x_1, \ldots, x_n\}$ puede reescribirse como:

$$f = x \wedge f|_{x=true} \vee \neg x \wedge f|_{x=false}$$

Por ejemplo, la función $f_1 = x \vee y$ puede escribirse como:

$$f_1 = x \wedge f_1|_{x=true} \vee \neg x \wedge f_1|_{x=false}$$

$$f_1 = x \wedge (true \vee y) \vee \neg x \wedge (false \vee y)$$

Más allá de presentar una interpretación recursiva para la semántica de las fórmulas esta descomposición abre la puerta para utilizar estructuras más compactas y eficientes al modelar expresiones booleanas. Una primer propuesta sería utilizar tablas de verdad. Es un formato exhaustivo para representar las fórmulas ya que para cada posible valuación en las variables del dominio se guarda registro del valor que corresponde a evaluar la función en esa instancia. Para el ejemplo anterior corresponde a (notamos el valor de verdad con $\top$ y su complemento como $\bot$):

x	y	$f(x, y)$
$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\top$

Una representación alternativa es la de los árboles binarios de decisión (BDD que dan un orden (arbitrario) a las variables, asocian cada nodo con una variable y sus sucesores (se asigna un sucesor para el valor de verdad y otro para el complemento) con sub árboles que corresponden a la función

restringida. Estas estructuras están claramentes vinculadas a la descomposición de shannon. Si para una variable cualquiera en $\mathcal{X}$ suponemos que el sucesor izquierdo corresponde al valor $x_i = \bot$ y el derecho a $x_i = \top$ el ejemplo sobre el que venimos trabajando se representaría como:

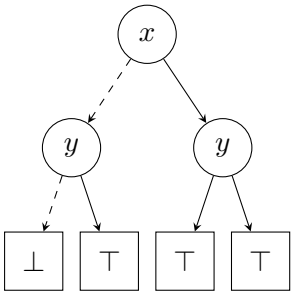

Figura 1: Diagrama de decisión de $f = x \vee y$

Es interesante notar que hay tres terminales redundantes ($\top$) y que si vale $x = \top$ no hace falta evaluar y para conocer el resultado de la evaluación. Podemos compactar nuestro diagrama (eliminando terminales duplicados y sub-evaluaciones redundantes) obteniendo lo siguiente:

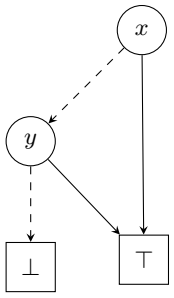

Figura 2: Reducción de $f = x \vee y$

En su trabajo de 1992 Randal Bryant estudia la forma de representar, minimizar y operar sobre fórmulas binarias utilizando lo que denomina diagramas de decisión binarios ordenados (OBDDs). Postula también que si se fija un orden (arbitrario) sobre las variables las estructuras tienen una representación canónica (criterio de unicidad) y esto facilita operaciones cerradas sobre OBDDs. Vamos a definir morfismos entre las fórmulas y los OBDDs.

2. Operaciones sobre OBDDs

La primera operación a definir sobre OBBDs es la de restricción, ya que nos va a permitir expresar otras más complejas sobre ella.

2.1. Restricción ($f|_{x=v}$)

Basicamente si O_1 es la estrucutura que representa f_1 sobre el dominio $\mathcal{X} = \{x_1, \ldots, x_n\}$ luego $restrict(O_1, x, v)$ donde $v \in \{\top, \bot\}$ será el OBDD que representa $f_1|_{x=v}$ y se calcula reemplazando en O_1 todos los nodos etiquetados con x por los de su sucesor v. Por ejemplo, en el caso anterior $f|_{x=\bot}$ se representa con el siguiente OBDD:

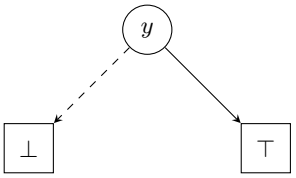

Figura 3: OBDD que representa $f|_{x=\bot}$

2.2. Cuantificaciones (∀,∃)

A partir de esto podemos definir operaciones para la lógica de primer orden bajo la siguiente semántica:

$$\forall x f = (f|_{x=\top}) \wedge (f|_{x=\bot})$$
$$\exists x f = (f|_{x=\top}) \vee (f|_{x=\bot})$$

2.3. Operaciones sobre funciones ($f \bullet f'$)

Podemos resolver de forma general la aplicación de operaciones lógicas entre funciones al operar con OBDDs, sea $\bullet$ una operación sobre funciones (por ej. $\wedge$,$\vee$ o $\oplus$) buscamos una forma de resolver la aplicación de estas operaciones. La base del procedimiento se basa en la aplicación de la descomposición de Shannon para la composición de fórmulas:

$$\forall x \in \mathcal{X} : f \bullet f' = (f|_{x=\top} \bullet f'|_{x=\top}) \wedge (f|_{x=\bot} \bullet f'|_{x=\bot})$$

El pseudocódigo de la figura 4 muestra la forma de obtener el OBDD equivalente a la aplicación de la operación sobre los OBBD que representan a las funciones compuestas. Cabe aclarar el significado de la estructura *mgr* que contiene las defiiciones de las variables junto con su orden y el de las siguientes funciones:

- `is_constant` indica si el OBDD en cuestión representa un valor constante $\top$ o $\bot$
- `is_true` indica si se trata en particular de $\top$
- `obdd_mk(mgr,var,high,low)` construye un nuevo OBDD donde el nodo raíz representa la evaluación de la variable `var` y sus OBDD sucesores `high` que corresponde al cofactor asociado a una asignación $var = \top$ y `low` que corresponde al cofactor asociado a la asignación $var = \bot$

Basicamente se trata de una aplicación recursiva dividida en casos de la descomposición de Shannon. En (A) se evalúa el caso base, en el cual ambos OBDD son constantes, esto quiere decir que no tienen sucesores y representan o bien el valor $\top$ o su complemento $\bot$. Aquí se aplica directamente la operación sobre los valores correspondientes. En (B) se evalúa el caso en el que el operando de la izquierda es constante pero el de la derecha no, por lo que la recursión prosigue fijando f al evaluar los cofactores *high* y *low*, (C) es análogo para el caso en que el operando de la derecha es constante. A continuación se evalúan los casos en que ambos operandos corresponden a OBDDs de altura mayor a uno, quiere decir, no constantes. Aquí entra en juego el orden fijado sobre las variables, porque asegura que sobre todo par de OBDDs que respeten el orden fijado en `mgr` podemos ordenar su exploración. (D) aplica la operación en el caso en que ambos OBDDs comparten la misma variable en la raíz. (E) y (F) prosiguen la exploración desenvolviendo el OBDD con mayor prioridad en su raíz.

```
1  algorithm obdd_node_aply(•, mgr, f, f′) is
2     input: • la operacion a aplicar
3     input: mgr la estructura que contiene la informacion de 𝒳 y su orden
4     input: f el OBDD de la formula que figura como primer operando
5     input: f′ el OBDD de la formula que figura como segundo operando
6     output: op(f) el OBDD que representa f • f′
7     if (is_constant(f) && is_constant(f′)): (A)
8        if(is_⊤(f) • is_⊤(f′)):
9           return obdd_mk(mgr,'⊤',NULL,NULL)
10       else:
11          return obdd_mk(mgr,'⊥',NULL,NULL)
12    if (is_constant(f)): (B)
13       return obdd_mk(mgr, var(f′),obdd_node_apply(•,mgr,f,f′->high)
14          ,obdd_node_apply(•,mgr,f,f′->low))
15    else if(is_constant(f′)): (C)
16       return obdd_mk(mgr, var(f),obdd_node_apply(•,mgr,f->high,f′)
17          ,obdd_node_apply(•,mgr,f->low,f′))
18    else if(var(f) == var(f′)): (D)
19       return obdd_mk(mgr,var(f),obdd_node_apply(•,mgr,f->high,f′->high)
20          ,obdd_node_apply(•,mgr,f->low,f′->low))
21    else if(var(f) < var(f′)): (E)
22       return obdd_mk(mgr,var(f),obdd_node_apply(•,mgr,f->high,f′)
23          ,obdd_node_apply(•,mgr,f->low,f′))
24    else if(var(f) > var(f′)): (F)
25       return obdd_mk(mgr,var(f),obdd_node_apply(•,mgr,f,f′->high)
26          ,obdd_node_apply(•,mgr,f,f′->low))
```

Figura 4: Pseudo código obdd_node_apply($f \bullet f'$)

2.4. Negación ($\neg f$)

Vamos a implementar la operación $\neg f$ como una aplicación de $\top \oplus f$ donde $\oplus$ representa la disyunción exclusiva.

3. Estructuras

A continuación presentamos las estructuras que emplearemos para representar los diagramas de decisión binarios ordenados.

```
typedef struct obdd_node_t{
uint32_t  var_ID;
uint32_t node_ID;
uint32_t ref_count;
struct obdd_node_t* high_obdd;
struct obdd_node_t* low_obdd;
} __attribute__((__packed__)) obdd_node;

typedef struct obdd_t{
struct obdd_mgr_t* mgr;
struct obdd_node_t* root_obdd;
} __attribute__((__packed__)) obdd;
```

obdd es la estructura que contiene un puntero a un nodo con la información relevante a la representación de la fórmula (root_obdd) y un puntero a la estructura (mgr) que mantiene el mapeo entre variables literales y su lugar en el orden dado. obdd_node es donde se almacena la información efectiva del OBDD, var_ID contiene el índice numérico de la variable referenciada en la raíz según su orden, node_ID es un identificador único

numérico para el nodo, `ref_count` es un contador de referencias al nodo, ya que más de un predecesor puede hacer referencia al nodo en la representación reducidad de OBDDs es necesario llegar registro de las referencias para saber cuando eliminar efectivamente la estructura (`node->ref_count == 0`). `high_obdd` es un puntero (potencialmente nulo) al nodo que representa el cofactor de la fórmula la valuación $v = \top$ y `low_obdd` es un puntero (potencialmente nulo) al nodo que representa el cofactor de la fórmula la valuación $v = \bot$.

```
typedef struct obdd_mgr_t {
uint32_t ID;
uint32_t greatest_node_ID;
uint32_t greatest_var_ID;
struct obdd_t* true_obdd;
struct obdd_t* false_obdd;
struct dictionary_t* vars_dict;
} __attribute__((__packed__)) obdd_mgr;
```

`obdd_mgr` representa el contexto por medio el que se opera con los OBDDs, definiendo un orden particular para las variables, para esto cuenta con una referencia a un diccionario `vars_dict` que mapea expresiones literales de las variables con su índice en el orden citado ($vars\_dict : \Sigma^* \longrightarrow \mathbb{N}$). ID identifica unívocamente a la estructura `obdd_mgr`, `greatest_node_ID` mantiene registro del índice del último nodo creado, `greatest_var_ID` mantiene registro del índice de la última variable ingresada al diccionario. `true_obdd` es un puntero al OBDD que representa el valor constante $\top$ y `false_obdd` es un puntero al OBDD que representa el valor constante $\bot$.

```
struct dictionary_entry_t {
    char *key;
    uint32_t value;
} __attribute__((__packed__)) dictionary_entry;

struct dictionary_t{
uint32_t size;
uint32_t max_size;
struct dictionary_entry_t* entries;
} __attribute__((__packed__)) dictionary;
```

El diccionario se ha de representar con dos estructuras, `dictionary` que representa el diccionario como colección de entradas ($\langle key, value \rangle \in \Sigma^* \times \mathbb{N}$) empleando un arreglo `entries`. `size` representa el tamaño actual y `max_size` representa la cantidad máxima de entradas que puede almanacer en ese momento el diccionario.

4. Semántica de las funciones

4.1. Diccionario

- `struct dictionary_t* dictionary_create();`
  Reserva memoria para la estructura del diccionario e inicializa su estructura con valores consistentes.
- `void dictionary_destroy(struct dictionary_t* dict);`
  Destruye estructuras anidadas del diccionario y libera su memoria.
- `bool dictionary_has_key(struct dictionary_t* dict, char *key);`
  Devuelve `true` si el diccionario `dict` contiene una entrada de valor `key`.
- `uint32_t dictionary_add_entry(struct dictionary_t* dict, char* key);`
  Agrega al diccionario `dict` una entrada de valor `key` y devuelve un índice nuevo, las llamadas sucesivas con claves nuevas deben generar índices estrictamente mayores. Si el tamaño del diccionario llega a ser igual que el tamaño, la capacidad debe duplicarse y los elementos preexistentes debe persistir.
- `uint32_t dictionary_value_for_key(struct dictionary_t* dict, char *key);`
  Devuelve el índice de una entrada de valor `key` en el diccionario `dict`.
- `char* dictionary_key_for_value(struct dictionary_t* dict, uint32_t value);`
  Devuelve el valor de una entrada de índice `value` en el diccionario `dict`.

4.2. Globales

- `uint32_t get_next_mgr_ID();`
  Devuelve un valor estrictamente mayor luego de cada llamada.

4.3. Orden sobre $\mathcal{X}$ (Manager)

- `obdd_mgr* obdd_mgr_create();`
  Reserva memoria para la estructura del manager e inicializa su estructura con valores consistentes.
- `void obdd_mgr_destroy(obdd_mgr* mgr);`
  Destruye estructuras anidadas del manager y libera su memoria.
- `void obdd_mgr_print(obdd_mgr* mgr);`
  Imprime los contenidos del manager de la siguiente manera:

  ```
  [OBDD MANAGER]
  Mgr: 3
  Mgr.Dict:
  [x1]:0
  [x2]:1
  ```

  Donde 3 corresponde al `ID` del manager y luego los valores `[x1]:0` corresponden a la entrada del diccionario con clave `x1` e índice `0`.
- `obdd* obdd_mgr_true(obdd_mgr* mgr);`
  Devuelve la referencia a `true_obdd` del manager.
- `obdd* obdd_mgr_false(obdd_mgr* mgr);`
  Devuelve la referencia a `false_obdd` del manager. Devuelve el índice (u orden) de la variable `var` según lo contenido dentro del diccionario del manager `mgr`.
- `obdd* obdd_mgr_var(obdd_mgr* mgr, char* name);`
  Crea un nuevo OBDD que representa a la variable `name` en relación al orden del manager `mgr`.
- `bool obdd_mgr_equals(obdd_mgr* mgr, obdd* left, obdd* right);`
  Devuelve `true` si los OBDDs `left` y `right` son equivalente según el orden definido en `mgr`.
- `uint32_t obdd_mgr_get_next_node_ID(obdd_mgr* mgr);`
  Incrementa el valor de `mgr->greatest_node_ID` y lo devuelve.
- `obdd_node* obdd_mgr_mk_node(obdd_mgr* mgr, char* var, obdd_node* high, obdd_node* low);`
  Construye un nuevo OBDD según el orden de las variables dictado por `mgr` con la variable `var` a la cabeza, `high` como sucesor para la asignación $var = \top$ y `low` como sucesor para la asignación $var = \bot$.
- `void obdd_node_destroy(obdd_node* root);`
  Destruye estructuras anidadas del OBDD y libera su memoria.

4.4. OBDD

- `obdd* obdd_create(obdd_mgr* mgr, obdd_node* root);`
  Reserva memoria para la estructura del OBDD e inicializa su estructura con valores consistentes (según los valores provistos por parámetro).
- `void obdd_destroy(obdd* root);`
  Destruye estructuras anidadas del OBDD y libera su memoria.

4.5. Operaciones sobre OBDDs

- `obdd* obdd_restrict(obdd* root, char* var, bool value);`
  Aplica la operación de restricción $restrict(root, var, value)$ que equivale a $f_{root}|_{var=value}$.
- `obdd_node* obdd_node_restrict(obdd_mgr* mgr, obdd_node* root, char* var, uint32_t var_ID, bool value);`
  Función que aplica recursivamente la restricción invocada por `obdd_restrict` comenzando por una llamada sobre su nodo inicial. Ha de seguir iterando hasta encontrar un nodo `v` que cumpla `v->var_ID == var_ID` donde `var_ID` es el indice de la variable `var` en `mgr->vars_dict`, y una vez encontrado reemplazara la ocurrencia del mismo por el sucesor que corresponda al valor obtenido en `value`.
- `obdd* obdd_exists(obdd* root, char* var);`
  Es el equivalente sobre OBDDs a resolver la aplicación $(var \wedge f_{root}|_{var=\top}) \vee (\neg var \wedge f_{root}|_{var=\bot})$.
- `obdd* obdd_forall(obdd* root, char* var);`
  Es el equivalente sobre OBDDs a resolver la aplicación $(var \wedge f_{root}|_{var=\top}) \wedge (\neg var \wedge f_{root}|_{var=\bot})$.
- `void obdd_print(obdd* root);`
  Imprime el contenido del OBDD `root`, exponiendo primero el ID del `mgr` y luego el contenido haciendo una llamada a `obdd_node_print` donde pasa el nodo raíz como parámetro y el valor de `spaces` (que representa cuando espacios dejar a izquierda) como 0.
- `void obdd_node_print(obdd_mgr* mgr, obdd_node* root, uint32_t spaces);`
  En conjunción con la función anterior imprime el contenido del OBDD, por ejemplo para el caso de la función $x1 \vee x2$ será:

```
[OBDD]
Mgr_ID:0
Value:
x1 &
 x2->1
 |
 (!x2)->1
|
(!x1) &
 x2->1
 |
 (!x2)->0
```

  Si la variable del nodo actual es `x1` y `spaces` es igual a 4, la función va a imprimir primero, para la rama de asignación `x1 = true` (*high_obdd*) el nombre de la variable del nodo, con tantos espacios a la izquierda como haya sido indicado en `spaces`, por ejemplo, va a imprimir:

```
    x1
```

  Luego, si el nodo no es constante pondrá un `&`
  `n` que simboliza que vamos a continuar imprimiendo el obdd que cuelga por el lado `x1 = true`,

```
    x1&
```

  Luego de esto se llama recursivamente a `obdd_node_print` con el nodo por la rama correspondiente e incrementando el valor de `spaces` en 1. Si el nodo es constante, va a imprimir `->` y el valor correspondiente al terminal por el lado de `x1=true` (1 si el terminal el `true`, 0 si el terminal es `false`) y luego un salto de línea
  `n`,

```
    x1->1
```

  A continuación se imprime la cadena

```
|
```

que indica el contenido de la rama opuesta `x1=false` (low_obdd). El procedimiento es análogo al del caso `x1=true` pero al escribir el nombre de la variable se hace prefijándola con un `!` y encerrando la expresión entre paréntesis.

```
(!x1)->
```

- `void obdd_node_destroy(obdd_node* node);`
  Destruye estructuras anidadas del nodo OBDD y libera su memoria..
- `bool obdd_apply_equals_fkt(bool left, bool right);`
  Es una implementación trivial del operador de igualdad sobre los valores `left` y `right`.
- `bool obdd_apply_xor_fkt(bool left, bool right);`
  Es una implementación trivial del operador de disyunción exclusiva sobre los valores `left` y `right`.
- `bool obdd_apply_and_fkt(bool left, bool right);`
  Es una implementación trivial del operador de conjunción sobre los valores `left` y `right`.
- `bool obdd_apply_or_fkt(bool left, bool right);`
  Es una implementación trivial del operador de disyunción sobre los valores `left` y `right`.
- `obdd* obdd_apply_not(obdd* value);`
  Es una implementación del operador de negación siguiendo lo postulado arriba, donde $\neg f = \top \oplus f$.
- `obdd* obdd_apply_equals(obdd* left, obdd* right);`
  Llama a la función `obdd_apply` pasando la función `obdd_apply_equals_fkt` como primer parámetro.
- `obdd* obdd_apply_xor(obdd* left, obdd* right);`
  Llama a la función `obdd_apply` pasando la función `obdd_apply_xor_fkt` como primer parámetro.
- `obdd* obdd_apply_and(obdd* left, obdd* right);`
  Llama a la función `obdd_apply` pasando la función `obdd_apply_and_fkt` como primer parámetro.
- `obdd* obdd_apply_or(obdd* left, obdd* right);`
  Llama a la función `obdd_apply` pasando la función `obdd_apply_or_fkt` como primer parámetro.
- `obdd* obdd_apply(bool (*apply_fkt)(bool,bool), obdd *left, obdd* right);`
  Es la implementación de la aplicacion de operadores por composición que se resuelve utilizando la función recursiva sobre nodos `obdd_node_apply`.
- `obdd_node* obdd_node_apply(bool (*apply_fkt)(bool,bool), obdd_mgr* mgr, obdd_node* left_node, obdd_node* right_node);`
  Funcion recursiva que opera como implementación del pseudocódigo presentado en la figura 4.
- `void obdd_remove_duplicated_terminals(obdd_mgr* mgr, obdd_node* root, obdd_node** true_node, obdd_node** false_node);`
  Función que recibe manager `mgr`, una referencia a un nodo OBDD `root` y dos referencias a referencias a nodos `true_node` y `false_node`. Recorriendo el OBDD de forma recursiva y si encuentra la primera ocurrencia de un nodo que represente el valor $\top$ o $\bot$ lo referencia en el doble puntero correspondiente. Caso contrario reemplaza esa ocurrencia por la referida en `true_node` o `false_node` y elimina el nodo duplicado.
- `void obdd_merge_redundant_nodes(obdd_mgr* mgr, obdd_node* root);`
  Reemplaza aquellos nodos dentro de `root` cuyos sucesores sean equivalentes, vinculando a su antecesor con su sucesor y eliminando el nodo en cuestión.
- `void obdd_reduce(obdd* root);`
  Función que reduce el OBDD `root` haciendo eliminando terminales duplicados y nodos redundantes.
- `bool is_true(obdd_mgr* mgr, obdd_node* root);`
  Devuelve `true` si el nodo pasado como parámetro es igual al nodo constante `true`, la comparación debe hacerse utilizando `mgr->true_obdd->root_obdd->var_ID == root->var_ID`.

- `bool is_constant(obdd_mgr* mgr, obdd_node* root);`
  Devuelve `true` si el nodo equivale al nodo que representa `true` o `false`.
- `bool is_tautology(obdd_mgr* mgr, obdd_node* root);`
  Devuelve `true` si todas las hojas de `root` equivalen al nodo que representa `true`.
- `bool is_sat(obdd_mgr* mgr, obdd_node* root);`
  Devuelve `true` si alguna de las hojas de `root` equivale al nodo que representa `true`.

4.6. Auxiliares

- `uint32_t str_len(char* a);`
  Devuelve el largo de la cadena `a` contando la cantidad de caracteres que se suceden hasta encontrar el caracter de fin de archivo (0).
- `char* str_copy(char* a);`
  Devuelve una copia de `a` sin destruir la cadena pasada por parámetro.
- `int32_t str_cmp(char* a, char* b);`
  Realiza una comparación alfanúmerica entre `a` y `b`, devuelve 0 si son iguales, 1 si `a` es menor que `b` o -1 si `a` es mayor que `b`.

5. Enunciado

Ejercicio 1

Implementar las siguientes funciones según lo descripto anteriormente:

En lenguaje assembler

- `obdd_node* obdd_mgr_mk_node(obdd_mgr* mgr, char* var, obdd_node* high, obdd_node* low);`
- `void obdd_node_destroy(obdd_node* root);`
- `obdd* obdd_create(obdd_mgr* mgr, obdd_node* root);`
- `void obdd_destroy(obdd* root);`
- `obdd_node* obdd_node_apply(bool (*apply_fkt)(bool,bool), obdd_mgr* mgr, obdd_node* left_node, obdd_node* right_node);`
- `bool is_tautology(obdd_mgr* mgr, obdd_node* root);`
- `bool is_sat(obdd_mgr* mgr, obdd_node* root);`
- `uint32_t str_len(char* a);`
- `char* str_copy(char* a);`
- `int32_t str_cmp(char* a, char* b);`

En lenguaje C

- `uint32_t dictionary_add_entry(struct dictionary_t* dict, char* key);`
- `void obdd_mgr_destroy(obdd_mgr* mgr);`
- `obdd* obdd_exists(obdd* root, char* var);`
- `obdd* obdd_forall(obdd* root, char* var);`
- `void obdd_node_print(obdd_mgr* mgr, obdd_node* root, uint32_t spaces);`
- `bool is_true(obdd_mgr* mgr, obdd_node* root);`
- `bool is_constant(obdd_mgr* mgr, obdd_node* root);`

Debe definir el orden que crea más conveniente para implementar las funciones (reflexione sobre las dependencias) y crear un test para cada una incluso antes de escribir el código.

Ejercicio 2

Escriba un programa de prueba (modificando `main.c`) que construya las fórmulas presentadas a continuación, las imprima y evalúe si son satisfacibles y/o tautologías. El resultado debe guardarse en el archivo "formulas.txt"

1- $x1 \vee x2$

2- $x1 \wedge x2$

3- $x1 \wedge \neg x1$

4- $(x1 \wedge x2) \rightarrow x1$

5- $\exists x2.(x2 = (x1 \wedge \neg x1))$

Testing

Se entregan un conjunto de tests que permiten verificar su implementación.

Luego de compilar, puede ejecutar `./tester.sh` para probar el código. El test realizará una serie de operaciones, estas generarán archivos que serán comparados las soluciones provistas por cátedra. Además, el test realizará pruebas sobre la correcta administración de la memoria dinámica.

Archivos

Se entregan los siguientes archivos:

- `obdd.asm`: Archivo a completar con su código assembler.
- `obdd.c`: Archivo a completar con su código C.
- `main.c`: Archivo para completar la solución del ejercicio 2.
- `Makefile`: Contiene las instrucciones para compilar `tester` y `main`. No debe modificarlo.
- `obdd.h`: Contiene la definición de la estructura y funciones. No debe modificarlo.
- `tester.c`: Codigo de testing. No debe modificarlo.
- `tester.sh`: Script que realiza todos los test. No debe modificarlo.
- `Catedra.salida.caso.`X`.txt` : Resultados de la cátedra para cada caso X. No deben modificarlos.

Notas:

a) Para todas las funciones hechas en lenguaje ensamblador que llamen a cualquier función extra, ésta también debe estar hecha en lenguaje ensamblador. (Idem para código C).

b) Toda la memoria dinámica reservada por la función `malloc` debe ser correctamente liberada, utilizando la función `free`.

c) Para el manejo de archivos se recomienda usar las funciones de C: `fopen`, `fread`, `fwrite`, `fclose`, `fseek`, `ftell`, `fprintf`, etc.

d) Para poder correr los test, se debe tener instalado *Valgrind* (En ubuntu: `sudo apt-get install valgrind`).

e) Para corregir los TPs usaremos los mismos tests que les fueron entregados. El criterio de aprobación es que el TP supere correctamente todos los tests y no contenga errores de forma.

6. Informe y forma de entrega

Para este trabajo práctico no deberán entregar un informe. En cambio, deberán entregar el mismo contenido que el dado para realizarlo, habiendo modificado los archivos `obdd.asm`, `obdd.c` y `main.c`.

La fecha de entrega de este trabajo es 12/09. Deberá ser entregado a través de un repositorio GIT almacenado en `https://git.exactas.uba.ar` respetando el protocolo enviado por mail y publicado en la página web de la materia.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes `orga2-doc@dc.uba.ar`.

Por favor no comparta código con compañerxs, ante cualquier incoveniente o demora comuníquese con su tutorx o con el JTP.

Para quien quiera revisar la publicación de Bryant se encuentra en:
`http://repository.cmu.edu/cgi/viewcontent.cgi?article=1217&context=compsci`