Camilo Sodo

Programación II

Trabajo Práctico Final

Uso de los Programas

1. Uso del programa en C

Antes de poder ejecutarlo, es necesario compilar el código. Esto se puede hacer
ejecutando el siguiente comando desde una terminal ubicada en la dirección donde está el archivo *Main.c*:

```
gcc -Wall -Wextra -Werror Main.c Tablero.c Lectura.c -o salida.exe
```

Una vez terminado el proceso de compilación, se puede correr el programa con el comando:

```
./salida.exe [Archivo de Entrada] [Archivo de Salida]
```

2. Uso del programa en Python

El programa se puede correr ejecutando el siguiente comando en una terminal ubicada en el directorio donde está el archivo *Main.py*:

```
python3 Main.py [Archivo de Entrada] [Color del Jugador] [Nivel del Bot]
```

3. Uso de los tests

Ambos programas poseen tests que comprueban su correcto funcionamiento. Los tests de C se pueden correr ejecutando los siguientes dos comandos consecutivamente:

```
gcc -Wall -Wextra -Werror tests.c -o tests.exe
./tests.exe
```

Los tests de Python se pueden correr con el comando:

```
python3 tests.py
```

Programa en C

1. Representación de la Información

Antes de empezar a programar es necesario definir las estructuras de datos que van a usarse para representar la información. Hay que representar cuatro piezas principales de información: El nombre de los jugadores, el color de las fichas, las casillas y el tablero sobre el que se juega. Adicionalmente, es importante para la implementación representar las direcciones de movimiento en el tablero (por ejemplo “abajo a la izquierda”).

```c
/* Tipos */
typedef char Jugadores[51];
typedef char Color;                                    (*)
typedef char** Tablero;                                (**)
typedef struct __casilla {
    char fila;
    char columna;
} Casilla;                                             (***)
typedef struct __direccion {
    int deltaFila;
    int deltaColumna;
} Direccion;
```

Notas de diseño:
(*) Los char que representan colores solo pueden tomar los valores “N” ó “B”.
(**) El tablero se inicializa siempre como una matriz de 8x8 y cada una de sus casillas solo puede tomar el valor “N”, “B” ó “X” (si la casilla está vacía).
(***) El campo fila de una casilla sólo puede tomar los valores:

“1” “2” “3” “4” “5” “6” “7” “8”

y el campo columna sólo puede tomar los valores.

“A” “B” “C” “D” “E” “F” “G” “H”

si alguno de los dos campos tiene algo distinto a estos valores se considera que la casilla no es válida (pues está fuera del tablero).

2. Implementación

Antes de empezar a escribir código, me pareció conveniente trazar una hoja de ruta de cómo debería funcionar el programa.

```
1.  Procesar argumentos de entrada.
2.  Procesar la cabecera.
   2.1.  Encontrar jugadores y asociarlos con su color
      2.1.1.  Si los colores son válidos (B ó N) y los nombres
              no son iguales -> 2.2 si no -> 9
   2.2.  Determinar el color que empieza
      2.2.1.  Si es un color válido (B ó N) -> 3 si no -> 9
3.  Asignar memoria para un tablero e inicializarlo
4.  Procesar líneas
   4.1.  Si la línea está vacía -> 4.1.1 si no -> 4.2
      4.1.1.  Si el jugador no tiene movimientos disponibles ->
              4.5 si no -> 10
   4.2.  Si la línea tiene formato correcto -> 4.3 sino -> 9
   4.3.  Si la jugada es válida -> 4.4 si no -> 10
   4.4.  Aplicar la jugada en el tablero
   4.5.  Cambiar de jugador
5.  Repetir 4 hasta que no queden líneas
6.  Si ninguno puede hacer un movimiento -> 7 si no -> 8
7.  Dar por ganador al jugador con más fichas o declarar empate
    de tener igual cantidad -> 11
8.  Guardar progreso
   8.1.  Escribir el tablero al archivo de salida
   8.2.  Escribir jugador siguiente -> 11
9.  Imprimir guía de formato -> 11
10.  Manejar movimiento invalido
   10.1.  Imprimir Tablero
   10.2.  Imprimir jugador actual -> 11
11.  Finalizar
   11.1.  Liberar el tablero (de ser necesario)
   11.2.  Devolver 0 si salió por 7, 8 o 10, otro si salió por
          9.
```

También me pareció que valía la pena hacer algo similar para la función que se encarga de poner una ficha en el tablero.

```
     Toma: tablero, posicion, color
1.   Si la posición está en el tablero y desocupada -> 2 si no
     -> 11
2.   Chequear inicio de direcciones
  2.1.   Si la primer casilla pertenece al tablero y contiene
         una ficha del color opuesto -> 3 si no -> 2.2
  2.2.   Marcar dirección como inválida
3.   Repetir 2 para todas las direcciones
4.   Chequear fin direcciones no invalidadas
  4.1.   Si la casilla pertenece al tablero -> 4.2 si no -> 4.6
  4.2.   Si la casilla no está vacía -> 4.3 si no -> 4.6
  4.3.   Si la casilla es del color opuesto -> 4.4 sino -> 4.5
  4.4.   Avanzar a la siguiente casilla en la dirección -> 4.1
  4.5.   Marcar la dirección como válida
  4.6.   Marcar la dirección como inválida
5.   Repetir 4 para todas las direcciones no invalidadas
6.   Si hay al menos una dirección válida -> 7 si no -> 11
7.   Marcar direcciones
  7.1.   Si la ficha es del color opuesto -> 7.2 si no -> 8
  7.2.   Cambiar el color de la ficha
  7.3.   Avanzar una posición -> 7.1
8.   Repetir 7 para todas las direcciones válidas
9.   insertar la ficha
10.  Devolver 1
11.  Devolver 0
```

3. ¿Por qué usar memoria dinámica?

Como el tablero se inicializa siempre a una 8x8 parecería en un principio que no hay motivo por el que usar memoria dinámica en este caso. Sin embargo, yo decidí usarlo para poder modularizar correctamente la estructura Tablero y sus funciones asociadas. Con mi implementación, la función *crearTablero* no toma ningún argumento y devuelve un Tablero nuevo.

```
Tablero crearTablero() {
     Tablero tablero = malloc(sizeof(char*) * 8);
```

```
        assert(tablero);
        for (int i = 0; i < 8; i++) {
        tablero[i] = malloc(sizeof(char) * 8);
        assert(tablero[i]);
              for(int j = 0; j < 8; j++) {
                    tablero[i][j] = CASILLA_VACIA;
              }
        }
        tablero[3][3] = tablero[4][4] = FICHA_BLANCA;
        tablero[3][4] = tablero[4][3] = FICHA_NEGRA;
        return tablero;
  }
```

Veamos ahora que, si quisiera evitar el uso de memoria dinámica, esta función debería recibir un puntero a un arreglo 8x8 ya creado (porque de crearse dentro de la función se liberaría al finalizar esta). Esto significa que el arreglo tendría que ser creado por una función externa a la estructura Tablero, lo que rompe un poco la modularización.

Programa en Python

1. Representación de la Información

A pesar de ser problemas totalmente distintos, el programa en Python trabaja con información muy similar al programa en C. En particular, hay que darle representación al color de las fichas, las casillas, el tablero y las direcciones. A esto se le agrega que, por motivos de la implementación, hay que representar el conjunto de las casillas vacías adyacentes a un casilla no vacía.

```
      ### Tipos
      Casilla = Tuple[int, int]
(*)
      ColorFicha = str
(**)
      Tablero = dict[Casilla, ColorFicha]
      JugadasPosibles = set[Casilla]
```

```
Direccion = Tuple[int, int]
```

Notas de diseño:
(*) Ambos componentes de la casilla deben estar entre 1 y 8.
(**) Sólo podrá tener “B” o “N”.

Salta a la vista que, a diferencia de *ColorFicha* y *Direccion* que tienen representaciones casi idénticas a las del programa en C, *Casilla* y *Tablero* recibieron algunos cambios.

El cambio de char a int en *Casilla* es para poder sacarle provecho a las operaciones de tuplas que Python tiene definida de base. Esto simplifica mucho en particular la operación de conseguir la casilla en una dirección de otra (por ejemplo, la casilla debajo de B3), que ahora se puede implementar con una suma de tuplas.

Tablero tiene más sentido como un diccionario de *Casillas* a *ColorFicha* pues se ahorra almacenar todas las casillas desocupadas y simplifica el código al no necesitar dos índices para acceder al contenido de una casilla. Sin embargo como en C los diccionarios no están definidos de base tuve que recurrir a usar un arreglo de arreglos.

2. Implementación

Hice primero para el programa en Python una traza de cómo resolver el problema a grandes rasgos.

```
1.  Procesar argumentos de entrada
2.  Leer archivo de C
   2.1.  Crear el diccionario
   2.2.  Leer línea
      2.2.1.  Si el carácter no es 'X' -> 2.2.2 si no -> 2.2.3
      2.2.2.  Agregar la Ficha correspondiente en la casilla
      2.2.3.  Repetir 2.2.1 hasta llegar al final de la línea
   2.3.  Repetir 2.2 para todas las líneas
   2.4.  Leer el último carácter para decidir quién empieza
3.  Imprimir el tablero
4.  Jugar
   4.1.  Si empieza el jugador -> 4.2 si no -> 4.5
   4.2.  Turno del jugador
   4.3.  Imprimir el tablero
   4.4.  Si no termino el juego -> 4.5 si no -> 5
```

```
   4.5.  Turno del bot
   4.6.  Si no termino el juego -> 4.2 si no -> 5
   4.7.  Imprimir el tablero
5.  Fin del juego
```

También me fue útil hacer una hoja de ruta similar para las funciones principales del programa como el turno del jugador y el turno de la máquina (el diseño de la función que pone una ficha en el tablero la omito porque es casi idéntica a la del programa en C)

```
Turno del jugador. Toma el tablero, el color del jugador y el
conjunto de jugadas posibles (no necesariamente válidas)
1.  Si el jugador tiene jugada posible -> 2 si no -> 5
2.  Pedir una jugada
   2.1.  Si el formato de la jugada es válido -> 2.2 si no -> 3
   2.2.  Si la jugada es válida -> 2.3 si no -> 4
   2.3.  Hacer la jugada
   2.4.  Eliminar la casilla jugada de las jugadas posibles y
         agregar sus casillas adyacentes vacías -> 5
3.  Imprimir mensaje de error de formato -> 2
4.  Imprimir mensaje de jugada invalida -> 2
5.  Finalizar

Turno del Bot 0. Toma el tablero, el color del bot y el conjunto
de jugadas posibles (no necesariamente válidas)
1.  Si hay jugadas posibles no vistas -> 2 si no -> 6
2.  Elegir una jugada de las jugadas posibles
3.  Si la jugada es válida -> 4 si no -> 1
4.  Hacer la jugada
5.  Eliminar la casilla jugada de las jugadas posibles y
    agregar sus casillas adyacentes vacías
6.  Finalizar

Turno del Bot 1. Toma el tablero, el color del bot y el conjunto
de jugadas posibles (no necesariamente válidas)
1.  Si hay jugadas posibles no vistas -> 2 si no -> 5
2.  Elegir una jugada de las jugadas posibles
3.  Si la jugada es válida -> 4 si no -> 1
4.  Contar cuántas fichas esa jugada capturaria -> 1
5.  Si encontro alguna jugada válida -> 6 si no -> 8
6.  Jugar la jugada que capturará la mayor cantidad de fichas
```

7. Eliminar la casilla jugada de las jugadas posibles y agregar sus casillas adyacentes vacías
8. Finalizar

3. El Conjunto de las Casillas Vacías Adyacentes a una Casilla no Vacía

Durante el programa de Python es necesario en varias ocasiones recorrer el conjunto de las jugadas válidas. Sin embargo, encontrar ese conjunto para un tablero cualquiera es complicado. Por eso es que vale la pena considerar el conjunto de las casillas vacías adyacentes a una casilla no vacía. Pues si bien no todas las casillas dentro del conjunto son jugadas válidas, se sabe que toda jugada válida está dentro del conjunto.

Esto es particularmente útil para el turno de la máquina (sobre todo en su máxima dificultad) donde hay que encontrar una o todas las jugadas válidas. Recorrer este conjunto en vez de todas las casillas vacías reduce bastante la búsqueda.

Además, a diferencia del conjunto de jugadas válidas, el conjunto de jugadas posibles es muy fácil de obtener. Una vez que tengo el conjunto asociado a un tablero, si se le agrega una ficha, simplemente hay que eliminar esa casilla y agregar sus adyacentes.