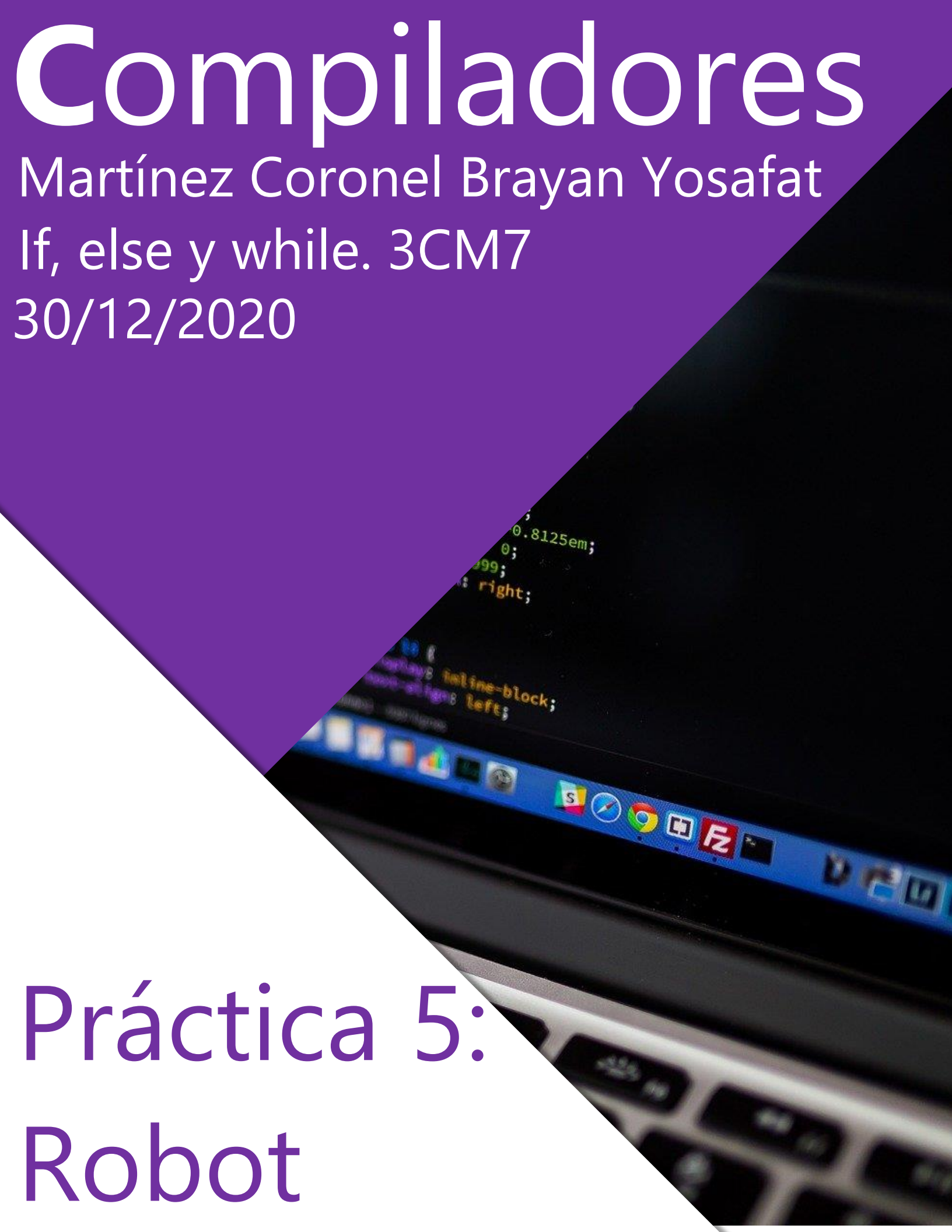
Compiladores

Martínez Coronel Brayan Yosafat

If, else y while. 3CM7

30/12/2020



Práctica 5: Robot

Robot con if, else y while

Introducción

En esta práctica se tiene como objetivo agregar algunas estructuras de control para hacer todavía comando más complejos, en este caso se desea agrega IF, WHILE y ELSE, para eso se tuvieron los siguientes objetivos específicos durante el desarrollo de esta práctica:

- Producciones de control (para el IF, el WHILE y el ELSE)
- Tokens para IF para ELSE y WHILE
- Tokens lógicos AND OR y NOT
- Tokens de comparación EQUALS, NEQUALS, CONTAINS, NCONTAINS
- Token para revisar si el dibujo está prendido
- Producción del bloque de código
- Unificar las producciones anteriores con las nuevas, en especial la del bloque

Ahora, como base tenemos el HOC número 5, que contiene incluso la parte de la práctica siguiente, pero, nos concentraremos en los tokens y producciones que nos importan. La parte más importante es que se hace un apartado de lugar en la máquina para indicar 3 cosas: dónde está la condición lógica, dónde está el bloque y dónde termina la función. Para fines de claridad se cambiaron muchos nombres, por ejemplo, en el HOC original se usa STOP para todo, incluso si se refiere a una función. Sin embargo, en la práctica se decidió utilizar STOPFUN para indicar que no es un STOP de todo lo que se ingresó, sino que sólo es un fragmento del código que contiene la máquina.

Otro lugar en donde se cambiaron los nombres fue en IFCODE, que para que sea más claro porque usamos Java, es IFFUN, WHILEFUN, y la clase Símbolo fue cambiada para que use mucho menos espacio en la escritura, al igual que la función code en la máquina, donde aprovechamos dos de las características de Java: el ciclo foreach y los argumentos variables:

```
int code(Object ...f) {
        for (Object o : f) {
                System.out.println("Añadido (" + o + ") en " + prog.size());
                prog.addElement(o);
        }

        return prog.size() - 1;
}
```

Además, la forma de apartar lugares en el HOC es mediante la inserción de STOP en la máquina, sin embargo, aquí se prefirió usar la palabra QUEUE para hacer referencia a que está en espera de ser reemplazado con un símbolo. Aquí un ejemplo de ambas cosas que se hablaron antes:

```
{ $$ = new ParserVal(new Simbolo(maq.code("iffun")));
  maq.code("queue", "queue", "queue", "queue"); }
```

Como se observa, se usa la misma función de dos formas diferentes, esto usa mucho menos espacio y es de una claridad alta. Además, la clase Union que estaba en el archivo Y fue eliminada porque Símbolo puede contener los dos tipos de datos que usamos: String y Entero.

Desarrollo

A la hora de seleccionar cómo hacer la escritura de los comandos, se pensó directamente en el uso de Java, y es que no estamos comparando números, porque usamos cadenas de caracteres todo el tiempo. Por lo que una comparación se escribe como: EE equals EE, lo cual mete un valor TRUE en la pila.

```
ctrl    : if '(' logic endlog ')' '{' block '}' end
                                        {
                                          $$ = $1;
                                          int ifIndex = (int) ((Simbolo) $1.obj).getValor();
                                          maq.replace(ifIndex + 1, new Simbolo(((Simbolo) $3.obj).getValor()));
                                          maq.replace(ifIndex + 2, new Simbolo(((Simbolo) $7.obj).getValor()));
                                          System.out.println("Block apunta a: " + ((Simbolo) $7.obj).getValor());
                                          maq.replace(ifIndex + 4, new Simbolo(((Simbolo) $9.obj).getValor()));
                                        }
        | if '(' logic endlog ')' '{' block '}' ELSE '{' block '}' end
                                        {
                                          $$ = $1;
                                          int ifIndex = (int) ((Simbolo) $1.obj).getValor();
                                          maq.replace(ifIndex + 1, new Simbolo(((Simbolo) $3.obj).getValor()));
                                          maq.replace(ifIndex + 2, new Simbolo(((Simbolo) $7.obj).getValor()));
                                          maq.replace(ifIndex + 3, new Simbolo(((Simbolo) $11.obj).getValor()));
                                          maq.replace(ifIndex + 4, new Simbolo(((Simbolo) $13.obj).getValor()));
                                        }
        | whl '(' logic endlog ')' '{' block '}' end
                                        {
                                          $$ = $1;
                                          int whileIndex = (int) ((Simbolo) $1.obj).getValor();
                                          maq.replace(whileIndex + 1, new Simbolo(((Simbolo) $3.obj).getValor()));
                                          maq.replace(whileIndex + 2, new Simbolo(((Simbolo) $7.obj).getValor()));
                                          maq.replace(whileIndex + 3, new Simbolo(((Simbolo) $9.obj).getValor()));
                                        }
        ;
```

Las producciones de control se hicieron con el fin de ser muy formales en su escritura, incluso el token de negación se escribe como: !(expresión). Además, se escribieron para que cualquier persona pueda leerlas y comprenderlas de forma rápida, en caso de que se quiera modificar el proyecto en algún momento diferente, e incluso, por qué no, con una persona diferente.

```
logic   : bool AND logic                    { $$ = $1; maq.code("and"); }
        | bool OR logic                     { $$ = $1; maq.code("or"); }
        | NOT logic ')'                     { $$ = $2; maq.code("not"); }
        | bool                              { $$ = $1; }
        ;
endlog  : /*NADA*/                          { maq.code("STOPFUN"); }
        ;
bool    : expr EQUALS expr                  { $$ = $1; maq.code("eq"); }
        | expr NEQUALS expr                 { $$ = $1; maq.code("neq"); }
        | expr CONTAINS expr                { $$ = $1; maq.code("con"); }
        | expr NCONTAINS expr               { $$ = $1; maq.code("ncon"); }
        | DRAW                              { $$ = new ParserVal(new Simbolo(maq.code("checkdraw"))); }
        ;
```

El apartado de la condición está hecho para que sea rápido el entender qué hace, se trató de ser muy directo con lo que hace cada apartado, procurando que cada bloque de código quedara dentro de la misma línea, e incluso la más complicada en este caso, también está en la misma línea de la producción.

Por otra parte, fue algo complicado hacer lo mismo para la parte del bloque de código de la estructura de control.

```
block   : /*Bloque vacio*/                  { $$ = new ParserVal(new Simbolo(maq.code("STOPFUN"))); }
        | line block                        { $$ = $1; }
        ;
line    : expr ';'                          { $$ = $1; maq.code("draw"); }
        | assign ';'                        { $$ = $1; }
        | cmd ';'                           { $$ = $1; }
        | ctrl                              { $$ = $1; }
        ;
```

Pero, después de varias ideas se logró una gran simplificación de lo que puede contener el bloque, tan solo unas cuantas líneas y el potencial de los bloques es de gran tamaño, además, se hizo de forma intencional que el árbol quedara cargado a la derecha, porque se requiere obtener la dirección en la máquina de la primera instrucción del bloque, por lo que, si está o no vacío el bloque, no cambia tanto el cómo se calcula la dirección.

```
void iffun() {
        int ifIndex = pc;
        Simbolo condSimb = (Simbolo) prog.elementAt(ifIndex);
        Simbolo blockSimb = (Simbolo) prog.elementAt(ifIndex + 1);
        Simbolo endSimb = (Simbolo) prog.elementAt(ifIndex + 3);

        execute((int) condSimb.getValor());
        boolean shouldExec = (boolean) pila.pop();

        try {
                if (shouldExec) {
                        execute((int) blockSimb.getValor());
                } else {
                        Simbolo elseSimb = (Simbolo) prog.elementAt(ifIndex + 2);
                        execute((int) elseSimb.getValor());
                }
        } catch (Exception e) {

        } finally {
                pc = (int)endSimb.getValor();
        }
}
```

Incluso las funciones se pensaron para ser sumamente claras y utilizar características del lenguaje, como el try, catch, finally de Java, ya que siempre queremos que cambie de posición pc.

```
void whilefun() {
        Simbolo condSimb = (Simbolo) prog.elementAt(pc);
        Simbolo blockSimb = (Simbolo) prog.elementAt(pc + 1);
        Simbolo endSimb = (Simbolo) prog.elementAt(pc + 2);
        boolean shouldExec;

        do {
                execute((int) condSimb.getValor());
                shouldExec = (boolean) pila.pop();
                if (shouldExec) execute((int) blockSimb.getValor());
        } while (shouldExec);

        pc = (int)endSimb.getValor();
}
```

Dos ejemplos que me gustaron mucho fueron estos, en el primero se usa un if con else para hacer alternante el resultado, es decir, como modifica con el comando de dibujo en el else y en el bloque del if se dibuja y se apaga el dibujo, entonces alterna en cada ejecución:

```
if ( dibujar ) {
    S S ;
    dibujar off ;
} else {
    dibujar on ;
}
```

El otro fragmento del código es cuando se usa el while, que va agregando una E cada vez que entra:

```
var = E ;
dibujar on ;
while ( var !equals E E E E ) {
    var = var + E ;
}
var ;
```

Conclusiones

Aunque fueron pocas líneas de código, ahora podemos hacer todavía mucho más de lo que en la práctica anterior, el while y el if son mucho más potentes de lo que había imaginado. Con esta materia ahora aprecio mucho más la potencia de estas estructuras, e incluso mucho más la del for, que es para la siguiente práctica. Es bastante divertido utilizar el conocimiento que tengo sobre Java para hacer una sintaxis nueva, una sintaxis formal y que me gusta mucho escribir.