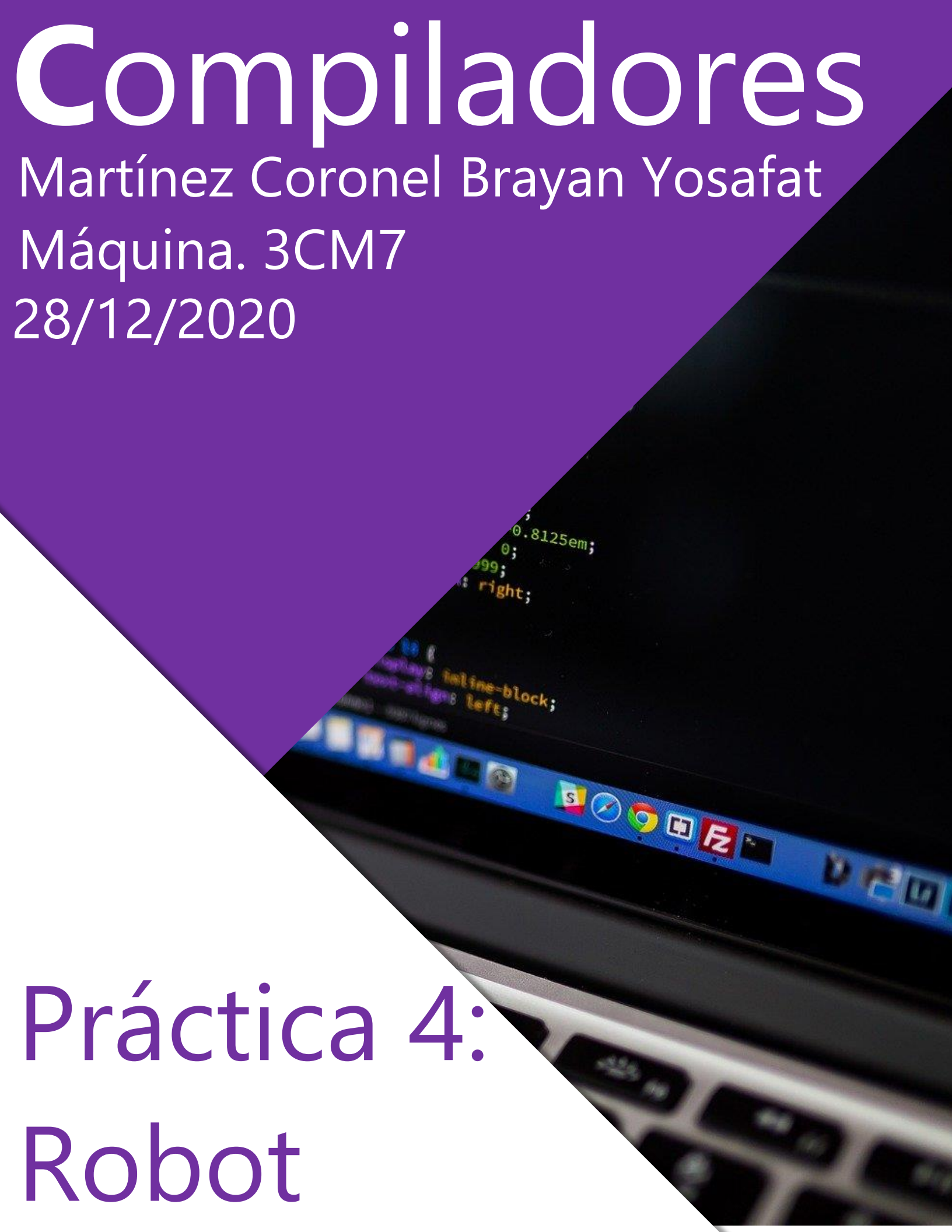
Compiladores

Martínez Coronel Brayan Yosafat
Máquina. 3CM7
28/12/2020



Práctica 4: Robot

Robot con máquina

Introducción

Como lo hicimos en la práctica anterior, vamos a agregar todavía más funciones a lo que hace. En general tenemos que cubrir los siguientes puntos:

- Máquina
- Función para ingresar una constante numérica
- Función para ingresar una secuencia variable
- Función para sumar
- Función para multiplicar
- Función para negar una secuencia
- Función para imprimir el contenido de una variable
- Token de número
- Producciones del comando de dibujo
- Unificar más las producciones

Comenzando con la máquina, esa ya estaba parcialmente implementada por la práctica 2, pero, faltaba agregar muchas funciones y producciones, además de hacer funcionar la máquina como nosotros requerimos es algo que también se debe hacer.

Desarrollo

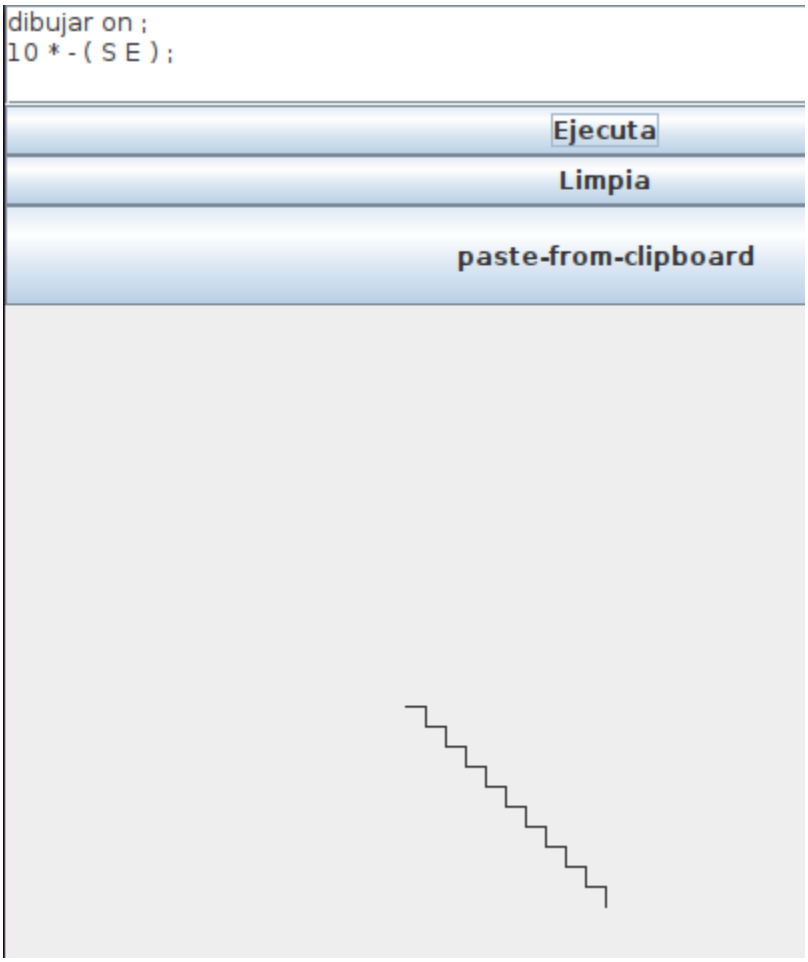
Comencemos con las producciones, aunque son más, ahora se ven mucho más unificadas y con una comprensión superior de lo que pasa por detrás en la ejecución del código. En la siguiente página se muestra la captura de las producciones desarrolladas para esta práctica. Ahora se cuenta con producciones con mucha más funcionalidad que antes, como las producciones que sólo indican la dirección, hacen que sea mucho más claro lo que hace la producción de secuencia, y con ella las producciones de expresión son tan detalladas que se entiende de forma escrita lo que se hace cuando están trabajando juntas. Además, la suma, la multiplicación y la negación de secuencia hace que tengamos mucho más control en el dibujo, por ejemplo, ahora para hacer una escalera podemos usar 10 * NE, en vez de repetirlo.

```
        | list ';'
        | list cmd ';'                    { maq.code("STOP"); return 1; }
        | list expr ';'                   { maq.code("draw"); maq.code("STOP"); return 1; }
        | list assign ';'                 { maq.code("STOP"); return 1; }
        | list PRINT VAR ';'              { maq.code("print"); maq.code(((Union) $3.obj).simb); maq.code("STOP"); return 1; }
        ;
cmd     : DRAW ON                         { maq.code("cmd"); maq.code(new Simbolo("", (short)0, 1)); }
        | DRAW OFF                        { maq.code("cmd"); maq.code(new Simbolo("", (short)0, 0)); }
        ;
assign  : VAR '=' expr                    { maq.code("assign"); maq.code(((Union) $1.obj).simb); }
        ;
expr    : sec                             { maq.code("varpush"); maq.code(((Union) $1.obj).simb); }
        | VAR                             { maq.code("varpush"); maq.code(((Union) $1.obj).simb); }
        | '-' '(' expr ')'                { maq.code("negate"); }
        | expr '+' expr                   { maq.code("add"); }
        | INT '*' expr                    { maq.code("constpush"); maq.code(((Union) $1.obj).simb); maq.code("mul"); }
        ;
sec     : sec dir                         {
                                            String sec1 = ((Union) $1.obj).simb.getSecuencia();
                                            String sec2 = ((Union) $2.obj).simb.getSecuencia();
                                            $$ = new ParserVal(new Union(new Simbolo("", (short)0, sec1 + sec2)));
                                          }
        | dir                             { $$ = $1; }
        ;
dir     : NORTE                           { $$ = new ParserVal(new Union(new Simbolo("", (short)0, "N"))); }
        | SUR                             { $$ = new ParserVal(new Union(new Simbolo("", (short)0, "S"))); }
        | ESTE                            { $$ = new ParserVal(new Union(new Simbolo("", (short)0, "E"))); }
        | OESTE                           { $$ = new ParserVal(new Union(new Simbolo("", (short)0, "O"))); }
        ;
```

El código de alguna de las funciones es presentado inmediatamente:

```
void varpush() {
        Simbolo s = (Simbolo) prog.elementAt(pc);
        pila.push(s.getSecuencia());
        pc++;
}

void assign() {
        String sec = (String) pila.pop();
        Simbolo s = (Simbolo) prog.elementAt(pc);
        s.setSecuencia(sec);
        pc++;
}

void cmd() {
        Simbolo s = (Simbolo) prog.elementAt(pc);
        pc++;
        dibuja = s.getValor() == 1;
        System.out.println("Dibujar: " + dibuja + "");
}

void draw() {
        String s = (String) pila.pop();

        if (dibuja)
                for (int i = 0; i < s.length(); i++)
                        comando(s.charAt(i));
}
```

Los resultados son mucho más notables cuando corremos el programa, en este caso hacemos una escalera con algo sumamente sencillo, tan solo una línea de menos de diez caracteres es suficiente para hacerlo, en vez de tener que crear una secuencia tan repetitiva a mano.



Además, como ese comando es parte de una expresión, su resultado puede ser asignado a una variable en vez de dibujarlo, y, con la negación de una variable, podemos hacer que cambié la dirección de la escalera. Las negaciones siempre van con paréntesis para hacer mucho más entendible qué se está negando, además, la multiplicación siempre se hace con un entero a la izquierda, para que quede mucho más claro qué se está multiplicando. Ya que la división, la potencia, raíz, entre muchas otras, no están definidas con las secuencias, realmente no podemos tener esas funciones como parte del repertorio de las funciones preprogramadas. Sin embargo, podemos agregar todavía más funciones útiles, como una función que sólo niegue en un solo eje.

Conclusiones

Agregar aún más funciones a las que teníamos y hacer todavía que tengan más funcionalidades las que ya teníamos, nos deja en claro el potencial de YACC. En lo personal me gusta mucho el programar en Java porque permite una escritura muy directa entre el modelado UML y el código producido. Me siento satisfecho con cómo se escriben los comandos hasta ahora, sin embargo, aunque tiene potencial, todavía no podemos preguntar por condiciones, o crear secuencia a partir de un ciclo, o crearlas a partir de decisiones, como la longitud. Sin embargo, tenemos algunos mismos problemas sobre la definición de algunas operaciones lógicas, como mayor que o menor que, no están realmente definidas. Podríamos comparar que sean exactamente iguales dos secuencias, o tener más funciones de utilidad como una función que verifique que contiene una subsecuencia.