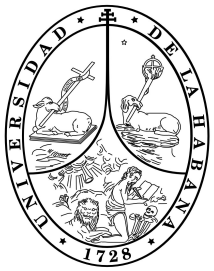


Universidad de La Habana

Facultad de Matemática y Computación

Proyecto de Compilación + Intenteligencia Artificial + Simulación

Simulador de un Jefe Técnico de MotoGP

Autores:

Arnel Sánchez Rodríguez Grupo: C312
arnelsanchezrodriguez@gmail.com

Samuel Efraín Pupo Wong Grupo: C312
s.pupo@estudiantes.matcom.uh.cu

Darián Ramón Mederos Grupo: C312
darianrm24@gmail.com

2021-2022

Índice

1. Motociclismo de Velocidad

El motociclismo de velocidad es una modalidad deportiva del motociclismo disputada en circuitos de carreras pavimentados. Las motocicletas que se usan pueden ser prototipos, es decir desarrolladas específicamente para competición, o derivadas de modelos de serie (en general motocicletas deportivas) con modificaciones para aumentar las prestaciones. En el primer grupo entran las que participan en el Campeonato Mundial de Motociclismo, y en el segundo las Superbikes, las Supersport y las Superstock.

Las motocicletas deben presentar una serie de características como son estabilidad, alta velocidad (tanto en recta como en paso por curva), gran aceleración, gran frenada, fácil maniobrabilidad y bajo peso.

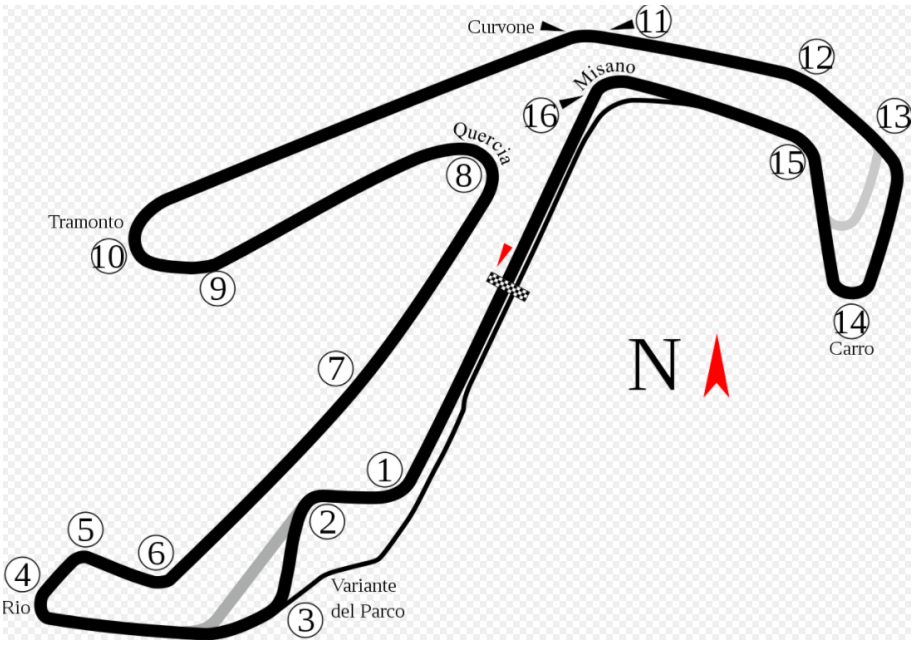
2. Estructura del Paddock

El **Jefe Técnico** de cada estructura se configura como una personalidad de bastante importancia dentro de un box, pues es quién se encarga de dirigir y controlar que todo funcione como un excelente engranaje que gane carreras. De igual importancia es la telemetría dentro de un box en MotoGP. Al fin y al cabo, los **Ingenieros Telemétricos** son las personas que se encargan de analizar, leer y comprender todos los datos proporcionados por el piloto, así como transmitirselos en boca al protago-

nista. Se trata de una figura de la que depende mucha de la información acerca de cualquier cambio realizado en la moto o asumir los puntos más fuertes de sus pilotos. Los **Mecánicos** también desempeñan un papel fundamental a la hora de construir la máquina perfecta.

3. Pista

Se utilizará como referencia el circuito de Misano, Misano World Circuit Marco Simoncelli, autódromo localizado en la fracción de Santa Mónica, comuna de Misano Adriático (provincia de Rímini), región de Emilia-Romaña, Italia.



4. Definición del Problema

Existirán varios pilotos con sus respectivas motos, las cuáles difieren entre sí en cuanto a sus prestaciones. Cada piloto posee sus propias habilidades para circular por curvas y rectas, de acuerdo a ellas es la velocidad que le será posible alcanzar en el tipo de sección. La pista se encuentra influenciada por el accionar del clima, puesto que no es lo mismo el manejo durante un día soleado que bajo la lluvia. Por tanto, el resultado de un piloto se verá condicionado por su moto, su habilidad de conducción y el clima.

Sin embargo, durante la carrera las condiciones pueden variar y el Jefe Técnico será el encargado de señalar los ajustes necesarios que el piloto deberá hacer para mejorar su rendimiento. Dicho intercambio de información se hará al finalizar la vuleta mediante el uso del DSL en el caso que sea el piloto controlado, los demás pilotos se hará la comunicación mediante el uso de la Inateligencia Artificial.

De esta manera, la simulación de la carrera será dinámica, puesto que entre las vueltas podrán existir variaciones provocadas por los ajustes propuestos por el Jefe Técnico.

5. Definición del Lenguaje

5.1. Introducción a PSharp (P)

5.1.1. Hello world!

```
1       method void main() {
2         print("Hello World!");
3       }
```

Los archivos de P# suelen tener la extensión de archivo .pys.

P# permite almacenar el texto fuente de un programa en varios archivos fuente. Cuando se compila un programa P# de varios archivos, todos los archivos de origen se procesan juntos y los archivos de origen pueden hacer referencia libremente entre sí; conceptualmente, es como si todos los archivos de origen estuvieran concatenados en un archivo grande antes de ser procesados.

5.1.2. Estructura del Programa

Los conceptos organizativos clave en P# son programas, tipos y miembros. Los programas P# constan de uno o más archivos fuente. Los programas declaran tipos y miembros. Las motocicletas y los motociclistas son ejemplos de tipos. Los métodos y propiedades son ejemplos de miembros.

5.1.3. Tipos y Variables

Hay dos tipos de tipos en P#: tipos de valor y tipos de referencia. Las variables de tipos de valor contienen directamente sus datos, mientras que las variables de tipos de referencia almacenan referencias a sus datos. Con los tipos de referencia, es posible que dos variables hagan referencia al mismo objeto y, por lo tanto, las operaciones en una variable afecten al objeto al que hace referencia la otra variable. Con los tipos de valor, cada una de las variables tiene su propia copia de los datos, y no es posible que las operaciones en una afecten a la otra.

Categoría	Tipo	Descripción
Tipos por Valor	Tipos Simples	Entero con signo: int
		Punto flotante IEEE: double
		Booleanos: bool
		Cadenas Unicode: string
	Tipos que aceptan valores NULL	Extensiones de todos los demás tipos de valor con un valor nulo
Tipos por Referencia	Tipos de matrices	Unidimensionales, por ejemplo, int []

5.1.4. Expresiones

Las expresiones se construyen a partir de operandos y operadores. Los operadores de una expresión indican qué operaciones aplicar a los operandos. Los ejemplos de operadores incluyen +, -, * y /. Los ejemplos de operandos incluyen literales, variables y expresiones.

Categoría	Expresión	Descripción
Primaria	x(...)	Invocación de método
	x[...]	Acceso a matrices e indexadores
Unaria	+x	Identidad
	-x	Negación
	!x	Negación lógica
Multiplicativa	x * y	Multiplicación
	x / y	División
	x % y	Resto
	x ** y	Exponenciación
Aditiva	x + y	Adición y concatenación de strings
	x - y	Substracción
Relacionales	x <y	Menor que
	x >y	Mayor que
	x <= y	Menor o igual que
	x >= y	Mayor o igual que
Igualdad	x == y	Igual
	x != y	Distinto
Condicionales AND	x && y	Evalúa y si y sólo si x es verdadera
Condicionales OR	x || y	Evalúa y si y sólo si x es falsa
Condicionales XOR	x ^ y	
Asignación	x = y	Asignación
	x op= y	Asignación compuesta; los operadores admitidos son *= /= %= **= += -= &&= ||= ^=

5.1.5. Declaraciones

Las acciones de un programa se expresan mediante declaraciones. P# admite varios tipos diferentes de declaraciones, algunas de las cuales se definen en términos de declaraciones integradas.

Un **bloque** permite escribir múltiples declaraciones en contextos donde se permite una sola declaración. Un bloque consta de una lista de declaraciones escritas entre

los delimitadores{y}.

La **declaración de inclusión** se utiliza para obtener y luego disponer de un recurso.

```
include test;
```

Las sentencias de **declaración** se utilizan para declarar variables, valga la redundancia.

```
method void example() {
  int a = 1;
}
```

Las **declaraciones de expresión** se utilizan para evaluar expresiones. Las expresiones que se pueden usar como declaraciones incluyen invocaciones de métodos, asignaciones que usan = y los operadores de asignación compuesta.

```
method void example() {
  int a = 1;
  print(a + 2);
}
```

La **instrucción de selección** se utiliza para seleccionar una de varias declaraciones posibles para su ejecución en función del valor de alguna expresión. Este es el caso de la sentencia **if**.

```
method void example(int a) {
  if (a < 5) {
    print(a);
  }
  else {
    print(a % 5);
  }
}
```

La **instrucción de iteración** se utiliza para ejecutar repetidamente una instrucción incorporada. Este es el caso de la instrucción **while**.

```
method void example(int a) {
  while (a > 5) {
    print(a);
    a -= 1;
  }
}
```

Las **sentencias de salto** se utilizan para transferir el control. En este grupo están las declaraciones de **break**, **continue** y **return**.

```
method int example() {
  while (true) {
    x = input()
    if (x == "x") {
      continue;
    }
    elif (x == "") {
      break;
    }
    print(x);
  }
  return 0;
}
```

5.1.6. Tipos

Los **tipos** son los elementos más importantes de P#. Un tipo es una estructura de bloques compuesta por acciones (métodos). Un tipo proporciona una definición para casos de, por ejemplo, motociclistas o motocicletas. Su declaración comienza con un encabezado que especifica qué tipo se va a crear y el nombre que se le dará a esta instancia. El encabezado va seguido del cuerpo del tipo, que consiste en una lista de declaraciones de miembros escritas entre los delimitadores{y}.

```
rider Rossi() {
  method int example() {
    return 0;
  }

  method void curves() {
    ...
  }
  ...
}
```

5.1.7. Métodos

Un **método** es un miembro que implementa un cálculo o acción que se puede realizar por tipo. Los métodos tienen una lista (posiblemente vacía) de **parámetros**, que representan valores o referencias de variables pasadas al método, y un tipo de retorno, que especifica el tipo de valor calculado y devuelto por el método. El tipo de retorno de un método es nulo si no devuelve un valor.

5.1.8. Parámetros

Los **parámetros** se utilizan para pasar valores o referencias de variables a métodos. Los parámetros de un método obtienen sus valores reales de los **argumentos** que se especifican cuando se invoca el método. Las modificaciones de un valor de parámetro no afectan el argumento que se pasó para el parámetro.

5.1.9. Cuerpo del método y variables locales

El cuerpo de un método especifica las declaraciones que se ejecutarán cuando se invoca el método. El cuerpo de un método puede declarar variables que son específicas de la invocación del método. Estas variables se denominan variables locales. Una declaración de variable local especifica un nombre de tipo, un nombre de variable y un valor inicial.

5.1.10. Operadores

Un **operador** es un miembro que define el significado de aplicar un operador de expresión particular.

Operadores Binarios:

```
1       3+5;
2       true && false;
```

5.1.11. Análisis Léxico

input
: input_element* new_line
| directive
;
input_element
: whitespace
| comment
| token
;
Terminadores de línea
new_line
: '<Caracter de retorno (U+000D)>'
| '<Caracter de avance de línea (U+000A)>'
;

whitespace
: '<Cualquier personaje con clase Unicode Zs>'
| '<Caracter de tabulación horizontal (U+0009)>'
;

Comentarios

comment
: '#' comment_section '#'
;

Tokens

token
: identifier
| keyword
| literal
| operator_or_punctuator
;

Identificadores

identifier
: '<Un identificador que no es una palabra clave>'
| identifier_start_character identifier_part_character*
;

identifier_start_character
: letter_character
| '<Caracter guión bajo (U+005F)>'
;

identifier_part_character
: letter_character
| decimal_digit
| '<Caracter guión bajo (U+005F)>'
;

letter_character
: uppercase_letter_character
| lowercase_letter_character
;

uppercase_letter_character
: 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H'
| 'I' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q'
| 'R' | 'S' | 'T' | 'V' | 'X' | 'Y' | 'Z'
;

lowercase_letter_character
: 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'
| 'i' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
| 'r' | 's' | 't' | 'v' | 'x' | 'y' | 'z'
;
decimal_digit
: '0' | '1' | '2' | '3' | '4'
| '5' | '6' | '7' | '8' | '9'
;
Palabras Claves
keyword
: 'bool'
| 'break'
| 'continue'
| 'double'
| 'elif'
| 'else'
| 'false'
| 'if'
| 'include'
| 'int'
| 'method'
| 'null'
| 'return'
| 'string'
| 'true'
| 'void'
| 'while'
| 'bike'
| 'rider'
| 'brakes'
| 'max_speed'
| 'weight'
| 'chassis_stiffness'
| 'speed'
| 'tyres'
| 'cornering'
| 'step_by_line'

;
Literales
literal
: boolean_literal
| integer_literal
| double_literal
| string_literal
| null_literal
;
Literales Booleanos
boolean_literal
: 'true'
| 'false'
;
Literales Enteros
integer_literal
: decimal_digit
;
Literales flotantes
double_literal
: decimal_digit+ '.' decimal_digit+
;
Literales de Cadenas
string_literal
: '"' string_literal_character* '"'
;
string_literal_character
: '<Cualquier caracter, excepto "(U+0022)'
;
Literales Nulos
null_literal
: 'null'
;
Operadores y signos de puntuación
operator_or_punctuator
: '{'
| '}'
| '['

| ']'
| '('
| ')'
| ':'
| ','
| ':'
| ';'
| '+'
| '-'
| '*'
| '/'
| '%'
| '**'
| '='
| '<'
| '>'
| '&&'
| '||'
| '=='
| '!='
| '<='
| '>='
| '+='
| '-='
| '*='
| '/='
| '%='
| '**='
| '&&='
| '||='
| '^='
;

Directivas

directive

: 'include' '<Nombre_del_archivo.pys>' ';'
;

5.2. Explicación de la Implementación

Para crear nuestra gramatica nos apoyamos en los lenguajes Python y CSharp , de ahi el nombre de nuestro DSL PSharp.

Cuando se recibe el código se tokeniza, luego de tener los tokens resultantes acordamos que tendríamos como una línea e implementamos el método $split_linese$, el cual recibe los tokens y los convierte en líneas , estas líneas son pasadas al parser.

El parser que decidimos desarrolar fue el parser LL , lo primero que hicimos fue las producciones , donde decidimos como serían correctas sintácticamente nuestras líneas, las producciones generan todas las posibles cadenas válidas para nuestro lenguaje y no existen cadenas que son generadas por nuestra gramática que no pertenezcan a nuestro lenguaje. De esta forma con una gramática válida empezamos al proceso de parsing. Nuestro parser necesita un método $hacer_first$ primeramente para hacer los first de cada cadena posible de nuestra gramática, luego llamamos un método auxiliar $calcular_first_restantes$ el cual tiene la función de calcular los first de los no terminales que aún no lo tienen calculado , nos hace falta guardar los first de los no terminales pq los necesitamos para hallar los follows en el método $hacer_follow$ .Luego de invocar a $hacer_follow$ debemos invocar un metodo auxiliar $completar_follows$ el cual se encarga de satisfacer la regla de los follows que dice que el follow de la cabeza de la producción es subconjunto del follow del último no terminal , si el último no terminal puede ser el último elemento de la producción.

Teniendo los first y los follows construimos la tabla LL(1) mediante el metodo $construir_tabla_LL$, al tener la tabla ya podemos comprobar que nuestra gramática no es ambigua , siempre existe solo una producción que aplicar. Luego creamos el metodo parsear al que hay que pasarle todas las líneas de nuestro código una por una y este realiza la comprobacion sintáctica y en este mismo método vamos a ir creando nuestro AST para luego hacer el chequeo semántico. Para crear el AST utilizamos metodos como $CreaNododExpresion$, $CreaNodoCondicion$, $CreaNododFuncion$, $EligeTipoDdeclaracion$, entre otros, declarados en la clase Parser.

Nuestro AST tiene un nodo por cada declaración que se puede realizar en el código. Tiene un nodo para una definición de función, una definición de variables, redefinición de variables, If, WHile, Rider, Bike, Return. En cada uno de estos nodos excepto Rider y Bike sí existe un ámbito como es el caso del nodo If, el While y la definición de función, cada uno de estos nodo tiene como atributo un tipo de nodo Program, el cuál posee una lista de declaraciones y por lo tanto en él se pueden guardar la lista de declaraciones que se haga en el ámbito.

Explicada la estructura del AST pasamos al chequeo semántico sobre este. Hacemos 3 recorridos sobre el AST, el primero para validar cada nodo. Un nodo es válido si todo lo que tiene guardado en sus atributos que es dependiente del contexto puede

ser utilizado desde ese contexto y de la forma que se quiere. Decimos esto porque por ejemplo las variables solo se pueden redefinir en el contexto en que fueron definidas. Decir que cada vez que creamos una función o un tipo creamos un contexto que responde a dicho ámbito. Todo lo que se defina en dicho ámbito pertenece a su contexto, no importa si se define dentro de un While o dentro de un If. En resumen los contextos en nuestro programa funcionan como en python con la particularidad de que no tenemos variables globales, si quieres redefinir una variable debes hacerlo en el contexto donde fue definida y solo se crea un nuevo contexto cuando se crea una funcion fuera de un tipo o cuando se crea un tipo. Destacar que las funciones definidas dentro de los tipos solo tienen el mismo context, y no se le pueden pasar parámetros.

Volviendo al AST, hacemos una segunda pasada, en esta pasada verificamos los tipos donde en los nodos en que hay expresiones inducimos el tipo. Decir que una expresión para nosotros puede ser una expresión aritmética , un bool, o un string, podemos incluir variables y llamado a función. La tercera pasada la hacemos para evaluar nuestros nodos. Si encontramos un nodo Def_Fun no lo evaluamos, una definición de función se evalua cuando se llama a la función.

Cuando se crea un tipo se importan las variables que puede tener ese tipo en la simulación, dentro de un tipo Bike solo se puede redefinir una función , la función $select_configuration$ que debe ser void , aunque dentro de la funcion si se pueden redefinir las variables de Bike, en este caso el objetivo de $select_configuration$ es seleccionar el tipo de gomas dadas las características de la moto y el ambiente y por tanto modificar la variable tires del contexto del tipo, la cuál utilizará el simulador.

Dentro de un tipo Rider existen en el contexto, igual que en Bike, variables que pertenecen a Rider que fueron importadas desde la simulación, en este caso se pueden redefinir dos de ellas *cornering* y $step_by_line$ que posteriormente serán pasadas al simulador, estas variables tienen mucha influencia en la simulación ya que son la habilidad en recta y en curva de un piloto. Son variables enteras que lo maximo que pueden ser es 10 , en caso de que se entre un valor mayor se supondra que se quiso entrar el maximo de habilidad y la variable será igual a 10 .En cuanto a las funciones se podrán definir $select_action$ que debe retornar un valor entero y es la encargada de elegir que acción realizar, para hacer este metodo podemos tener en cuenta las características del piloto que pueden ser llamadas y están actualizadas. También podemos definir la funcion $select_acceleration$ la cual debe ser void y su función debe ser actualizar la aceleracion de un agente.

5.3. Conexion Simulacion - Compilacion

El resultado del DSL si se hacen los 3 recorridos del AST sin error, son dos listas, una con todos los pilotos que fueron creados y otra con todas las motos , a partir de estas listas se crean los pilotos y las motos en la simulacion. En la simulacion los pilotos que no fueron creados en el código ejecutan sus métodos como normalmente, en el caso de un piloto que fue creado desde el DSL se ejecuta la función definida en el DSL, pero antes de ejecutarla se actualiza el contexto de la funcion para que dicha función pueda apoyarse en la situación actual. Luego de la ejecución de la función dependiendo que función se ejecutó se importa a la simulación la variable que se quiere desde el contexto del método.

6. Simulación

7. Inteligencia Artificial