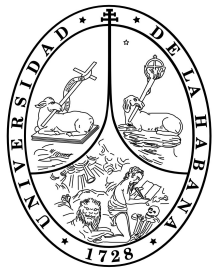


Universidad de La Habana

Facultad de Matemática y Computación

Proyecto de
Compilación + Inteligencia 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. Introducción **2**
- 1.1. Motociclismo de Velocidad . . . 2
- 1.2. Estructura del Paddock . . . 2
- 1.3. Pista . . . 3
- 1.4. Definición del Problema . . . 3

2. Diseño del Sistema **5**
- 2.1. Simulación [1][2] . . . 5
  - 2.1.1. Ambiente . . . 5
  - 2.1.2. Agentes . . . 7
  - 2.1.3. La carrera . . . 8
- 2.2. Inteligencia Artificial [3][4] . . . 10
  - 2.2.1. Configuración de la moto . . . 10
  - 2.2.2. Selección de acciones . . . 11
  - 2.2.3. Selección de la aceleración . . . 13

3. Definición del Lenguaje [5][6] **14**
- 3.1. Introducción a PySharp (P#) . . . 14
  - 3.1.1. Hello world! . . . 14
  - 3.1.2. Estructura del Programa . . . 14
  - 3.1.3. Tipos y Variables . . . 14
  - 3.1.4. Expresiones . . . 14
  - 3.1.5. Declaraciones . . . 15
  - 3.1.6. Tipos especiales . . . 17
  - 3.1.7. Métodos . . . 17
  - 3.1.8. Parámetros . . . 17
  - 3.1.9. Cuerpo del método y variables locales . . . 17
  - 3.1.10. Operadores . . . 18
  - 3.1.11. Análisis Léxico . . . 18
- 3.2. Explicación de la Implementación . . . 22
- 3.3. Conexión Simulación - Compilación . . . 24

1. Introducción

1.1. 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*.

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

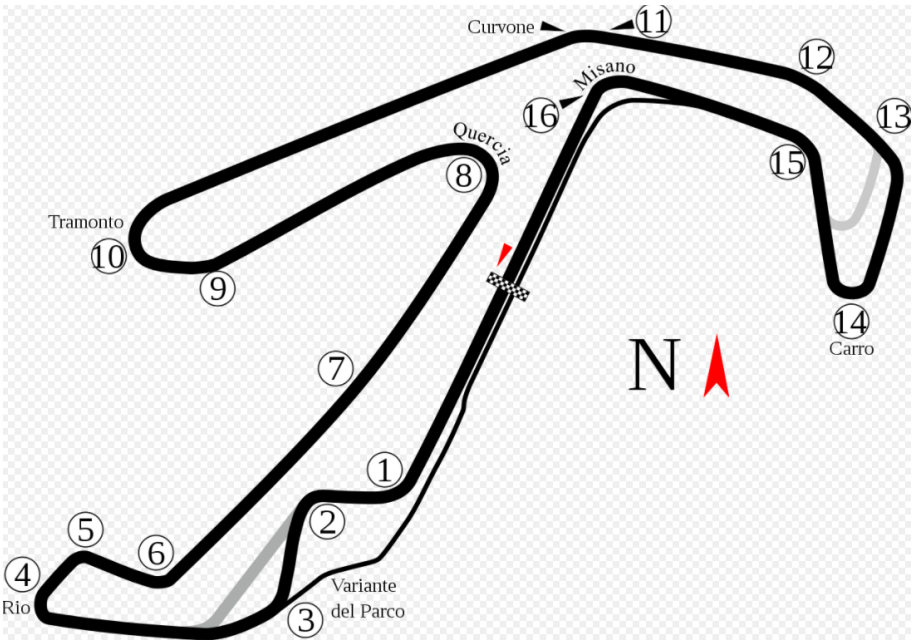
1.2. Estructura del Paddock

En MotoGP, el **Jefe Técnico** de cada estructura se configura como una personalidad de bastante importancia dentro de un box, pues es quien se encarga de dirigir y controlar que todo funcione como un excelente engranaje que gane carreras. De igual importancia son los **Ingenieros Telemétricos**, pues son las personas que se encargan de analizar, leer y comprender todos los datos proporcionados por el piloto,

así como transmitírselos al protagonista. 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.

1.3. 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.



1.4. Definición del Problema

Se tienen varios pilotos con sus respectivas motos, las cuáles difieren entre sí en cuanto a sus prestaciones. Cada uno posee su propio método de manejo, siendo algunos más cuidadosos y otros más agresivos. 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 la combinación de su moto, su modo de conducción, la pista y el clima.

Sin embargo, durante la carrera las condiciones pueden variar y es necesario realizar los ajustes necesarios para que el piloto mejore su rendimiento, los cuales

se harán al pasar de una sección a otra de la pista o al finalizar cada vuelta. Esto podrá hacerse utilizando un lenguaje imperativo, mediante el uso de palabras claves, para que el piloto no necesite analizar situaciones complejas y pueda concentrarse en pilotar de la forma más eficiente posible.

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, el cuál podrá ser una persona o una IA.

2. Diseño del Sistema

2.1. Simulación [1][2]

Se desea simular una carrera de MotoGP. Con tal fin, se utiliza una estructura agente conformada por un piloto y una moto, y una estructura ambiente formada por el clima y la pista. Se trata de un sistema multiagentes en el que dos de estos pueden interactuar entre sí, ya sea en modo ofensivo o en modo defensivo. Las variables de las distintas entidades se manipulan utilizando una escala del 1 al 10, donde sus valores se clasifican como: bajos (menores que 4), medios (entre 4 y 6), y altos (mayores que 6); excepto en los casos en los que se utilizan estructuras *Enums* para definir los datos.

2.1.1. Ambiente

El ambiente se compone por la combinación del clima y la pista.

El clima es simulado utilizando variables aleatorias con distribución normal, que responden a:

- El estado del clima
- La humedad
- La temperatura
- La visibilidad
- La dirección e intensidad del viento

El uso de la distribución normal se debe a que son menos comunes los climas extremos. Nótese que la escala utilizada permite esta configuración.

Una función probabilística es la encargada de lanzar un evento de cambio de clima, generando nuevo estado de este. Además, sus parámetros se varían cada vez que termina una vuelta de la carrera. Es importante señalar que estas variables se encuentran enlazadas bajo ciertas circustancias, como es el caso de un aumento de la humedad cuando comienza a llover.

La pista posee:

- El nombre

- La longitud total
- La lista de las secciones que la componen

Donde cada **sección** está formada por:

- El nombre
- La longitud
- La velocidad máxima alcanzable (sin accidentarse)
- La orientación cardinal
- El tipo de sección (curva o recta)
- Si tiene un equivalente en boxes y, en caso de tenerlo, su longitud.

El circuito de Misano es el escogido por defecto para la competición. Sin embargo, es posible reestructurar la pista para crear una experiencia diferente o, incluso, generar una nueva pista de manera aleatoria automáticamente. Mas debemos asegurarnos de que estas nuevas creaciones sean estructuras válidas.

El método de reestructuración intercambia las curvas o rectas con otras secciones de su mismo tipo y su misma orientación cardinal. De este modo, se asegura de que solo se varíe el orden de estas y no se pierdan las propiedades que hacen al circuito cerrado.

Por otro lado, generar una nueva pista de manera aleatoria automáticamente es una problemática topológica de un mayor nivel de complejidad. Como solución factible simplificada, se genera una línea poligonal de secciones, que parte de un punto y se expande variando su orientación.

La longitud de cada sección es generada utilizando una variable aleatoria, donde los mayores valores tienen una menor probabilidad. Luego, atentiendo a dicha longitud, la velocidad máxima alcanzable se calcula por rangos, agregando incertidumbre mediante probabilística con distribución uniforme.

Cuando la longitud total excede el mínimo necesario, se redirige la orientación de las nuevas secciones en sentido cardinal opuesto al inicial. Esta línea es abierta, pero su punto final está orientado en sentido inverso al punto inicial. Luego, tomando una copia exacta de esta y rotándola 180^0, se puede crear una figura poligonal cerrada con simetría central, la cual responde a las condiciones necesarias de una pista.

2.1.2. Agentes

Un agente está compuesto por un piloto y su moto.

El piloto está conformado por:

- El nombre
- La habilidad de pasar por la curva y por la recta
- La probabilidad de caerse de la moto
- La independencia (probabilidad de ignorar órdenes)
- La agresividad
- La pericia

Los valores de estas características oscilan entre 1 y 10, pero son divididos por potencias de 10, con el objetivo de que respondan a probabilidades generadas por variables aleatorias con distribución uniforme.

Las habilidades de manejo son tomadas en cuenta para el paso por las distintas secciones de la pista. La independencia da la probabilidad de que el piloto tome sus propias decisiones, ignorando las orientaciones dadas por el **Jefe Técnico**. Mientras, la agresividad es la tendencia a ejercer acciones más riesgosas y la pericia, la capacidad para salir airoso de ellas.

La moto posee:

- La marca
- El modelo
- La velocidad máxima alcanzable
- El peso
- El tipo de neumáticos (lisos o de lluvia)
- Los frenos
- La rigidez del chasis

- La probabilidad de romperse
- La probabilidad de que exploten los neumáticos por desgaste

Las propiedades de las motos influyen en el manejo de sus pilotos y, por tanto, en el resultado de estos en la carrera. Sobreexplotarlas lleva consigo un mayor desgaste, lo que provoca un aumento en la probabilidad de una caída.

Además del piloto y su moto, un **agente** está compuesto por variables que responden a su desempeño en la carrera:

- La velocidad y la aceleración actuales
- El tiempo de la vuelta y el tiempo total
- La sección de la pista en la que se encuentra
- La posición en el ranking
- La entrada a boxes (el piloto cambia de moto)
- Si sufrió o provocó un accidente (se cayó o derribó a otro piloto)

2.1.3. La carrera

El *ranking* de la carrera permite conocer las posiciones de los pilotos mientras corren en sus motos, para así premiar al ganador.

Inicialmente, los agentes parten en un orden por defecto calculado de manera aleatoria, asumiendo que este se alcanzó el día anterior en las clasificaciones. El comienzo de la carrera se hace desde la primera sección de la pista, y tomando un piloto a la vez se calculan sus interacciones, haciendo que este avance.

Las acciones que el agente puede ejecutar en un instante determinado dependen de las variables del ambiente, las del propio agente, y las de sus rivales inmediatos hacia adelante y hacia atrás. Analizando las consecuencias de la acción escogida (acelerar, frenar, doblar, ir a boxes o luchar por la posición), se considera si fue correcta y sobrepasó la sección o, en caso contrario, sufrió un accidente.

De efectuar ataque (intentar superar a otro piloto) o defensa (evitar ser superado), se provoca una interacción con otros agentes. Si un ataque es satisfactorio, se intercambian las posiciones en el *ranking*, sino su tiempo en la carrera sufre una penalización. Existen variables probabilísticas que calculan si el piloto comete un error

y debe abandonar la carrera, provoca la salida de otro piloto, o ambos. Tomando su aceleración, se puede calcular su nueva velocidad y el tiempo que demora en llegar a una nueva sección.

Sin embargo, es necesario decidir la forma más acertada de otorgarle el turno al próximo piloto. Para ello se utiliza una cola con prioridad, donde los agentes están ordenados por sus tiempos en la carrera. Esto se debe a que, al pasar su turno, el tiempo que poseen es el que demorarán en llegar a la nueva sección. Por tanto, esto permite darle el turno al más rápido primero, pues se supone que debe avanzar más veloz que los que demoran más.

Dicho sistema (cola con prioridad por tiempo) permite tener varios agentes en diferentes secciones, corriendo a distintas velocidades. Cada vez que un piloto tiene su turno y logra continuar, se actualizan las variables que lo componen. En caso de que no esté solo en su sección y su tiempo teórico sea menor que quien le antecede, se reajustan las posiciones utilizando ataque/defensa. Al pasar por la meta, se muestra el *ranking* según van completando las vueltas, hasta que se completa la carrera.

Los parámetros de las distintas entidades pueden interactuar entre ellos, lo que trae determinadas consecuencias, algunas positivas y otras negativas. Dado que dichas interacciones se producen en mayor o menor medida, después de varias pruebas se obtuvo una escala que se puede considerar funcional, puesto que se acerca a los resultados de una carrera verdadera.

Para lograr un mayor realismo, se hace una primera variación de parámetros para adaptar a las entidades a las condiciones climatológicas del momento; luego, en cada cambio de vuelta se modifica el clima y se reajustan los parámetros de acuerdo las nuevas condiciones climatológicas.

Los agentes reciben cada variación del ambiente y analizan qué modificaciones realizar para adaptarse. Asimismo, las interacciones entre dos de estos provocan cambios en sus condiciones. Nótese que en un momento dado, una interacción puede generar una reacción en cadena que provoque una serie de cambios en varios agentes que compiten directamente entre sí. Una variable probabilística, influida por las habilidades de los pilotos en cuestión, sirve para obtener el resultado de un ataque o una defensa.

2.2. Inteligencia Artificial [3][4]

Una inteligencia artifical es la encargada de elegir las acciones a realizar por cada agente, con el fin de tomar las mejores decisiones para alcanzar el máximo rendimiento. Para su implementación se utilizan: **Sistemas Expertos** generados mediante el uso de un procedimiento declarativo/imperativo, utilizando la biblioteca **PyKE**[7] de **Python**; y **heurísticas**. También se emplean variables probabilísticas para crear incertidumbre, es decir, la respuesta de la IA ante la misma situación puede variar: es **no determinista**.

Se emplea como base de conocimiento el conjunto de variables, antes expuestas, que forman parte de la simulación y que influyen en el desempeño:

- Las condiciones del clima
- Las características de la sección
- Las habillidades del piloto
- Las propiedades de la moto
- El estado del agente

2.2.1. Configuración de la moto

El primer sistema se encarga de escoger la mejor configuración de la moto. De este modo, atendiendo a las condiciones iniciales de la carrera (que serán los hechos), la IA podrá tomar decisiones atendiendo a las reglas definidas. Al ser declarativas, las determinaciones tomadas se obtendrán mediante *match* hechos-reglas.

Los hechos se presentan mediante lógica difusa y n-valente:

```
1  weather(Sunny)
2  humidity(2)
3  wind_intensity(7)
4  wind_direction(3)
```

Las reglas se formulan del modo:

```
1  slick
2      use select_type(Slick)
3      when
4          moto_facts.weather($ans_1)
5          check $ans_1 != "Rainy"
6          moto_facts.humidity($ans_2)
```

```
            check $ans_2 <= 6

    soft
        use select_tires(Soft)
        when
            moto_facts.wind_intensity($ans_1)
            check $ans_1 > 6
            moto_facts.wind_direction(3)
```

Tomando este ejemplo, el sistema analiza los hechos y las reglas permitiendo llegar a la decisión de tomar neumáticos *Slick_Soft* (lisos suaves). Luego, consultando los resultados obtenidos por PyKE, mediante Python de manera imperativa será posible la interacción con la simulación de la carrera. En este caso, se utiliza para generar la selección un *Enum*:

```
    class Tires(Enum):
        Slick_Soft = 0
        Slick_Medium = 1
        Slick_Hard = 2
        Rain_Soft = 3
        Rain_Medium = 4
```

El resto de los parámetros de la moto no son modificados, puesto que se asume que la configuración por defecto es la óptima. No así con los neumáticos, de los cuáles depende mucho el rendimiento, atendiendo a las variaciones del clima.

2.2.2. Selección de acciones

Un segundo sistema se emplea para escoger la acción que debe ejecutar el piloto, atendiendo a las condiciones de la carrera:

- Aumentar/disminuir/mantener la velocidad
- Doblar
- Ir a *pits*
- Atacar/defender
- Combinaciones de todas las anteriores

(Ejemplo: Aumentar la velocidad + Doblar + Atacar)

En este caso, se utiliza el mismo método antes expuesto, declarativo/imperativo con el uso de PyKE.

Hechos:

```
    speed(Lower)
    section(Straight)
    slick_tires(True)
    weather(Cloudy)
    humidity(6)
    nearest_forward(60)
    nearest_behind(-60)
```

Reglas:

```
    speed_up
        use select_action(SpeedUp)
          when
            action_facts.speed("Lower")

    keep_speed
        use select_action(KeepSpeed)
        when
            action_facts.speed("Same")

    brake
        use select_action(Brake)
        when
            action_facts.speed("Higher")

    turn
        use select_action(Turn)
        when
            action_facts.section("Curve")
```

Las acciones obtenidas son combinadas con el fin de generar una respuesta compuesta mediante un *Enum*:

```
    class AgentActions(Enum):
        SpeedUp = 0
        KeepSpeed = 1
        Brake = 2

        SpeedUp_Turn = 3
        KeepSpeed_Turn = 4
        Brake_Turn = 5

        SpeedUp_Pits = 6
        KeepSpeed_Pits = 7
        Brake_Pits = 8

        SpeedUp_Turn_Pits = 9
        KeepSpeed_Turn_Pits = 10
```

```
16          Brake_Turn_Pits = 11
17
18          SpeedUp_Attack = 12
19          KeepSpeed_Attack = 13
20          Brake_Attack = 14
21
22          (...)
23
24          SpeedUp_Turn_Pits_Defend = 33
25          KeepSpeed_Turn_Pits_Defend = 34
26          Brake_Turn_Pits_Defend = 35
```

[Explicar como se toman las decisiones (valores de las variables)]

2.2.3. Selección de la aceleración

Una tercera heurística es la encargada de escoger la mejor aceleración en una sección dada de la pista, con el objetivo de alcanzar la mayor velocidad posible sin causar un accidente que obligue a abandonar la carrera. Primeramente, se calcula la aceleración máxima alcanzable, tomando en cuenta que no se exceda la velocidad máxima admitida por la moto y la sección que se recorre. $a_{max} = (V_{max}^2 - V^2)/(2 * X)$ Luego, se establece un sistema de penalizaciones que disminuyen dicha aceleración si no se poseen las condiciones óptimas del ambiente (clima, humedad, temperatura) y del piloto (destreza en curvas y rectas). El resultado obtenido es incorporado a la simulación para continuar la carrera. De su valor depende mucho si el piloto obtiene un buen tiempo o sufre un accidente que lo saque de la competencia.

3. Definición del Lenguaje [5][6]

3.1. Introducción a PySharp (P#)

3.1.1. Hello world!

```
1    method int main() {
2      return 0;
3    }
```

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

3.1.2. Estructura del Programa

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

3.1.3. Tipos y Variables

En P# solo existen tipos de valor, no hay de referencia. Por tanto, todas las variables contienen directamente sus datos, cada una tiene su propia copia y no es posible que las operaciones en una afecten a otra.

Categoría	Tipo	Descripción
Tipos	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

3.1.4. 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
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 *= /= %= **= += -= &&= ||= ^=

3.1.5. 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}.

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

```
1    method void example() {
2      int a = 1;
3    }
```

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 int example() {
  int a = 1;
  return 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 int example(int a) {
  if (a < 5) {
    return a;
  }
  else {
    return a % 5;
  }
}
```

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

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

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

```
method int example() {
  x = 10
  while (true) {
    if (x < 100) {
      x += 10;
      continue;
    }
    else {
      break;
    }
  }
  return 0;
}
```

3.1.6. Tipos especiales

Los **tipos especiales** son los elementos más importantes de P#, constituyen estructuras de bloques compuestas 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}.

```
1    rider Rossi() {
2      method int select_action() {
3        return 0;
4      }
5      method void select_acceleration(){
6        ...
7      }
8      ...
9    }
```

3.1.7. Métodos

Un **método** es un miembro que implementa un cálculo o acción que se puede realizar por tipo. Los métodos tienen una lista de **parámetros**, que representan valores 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.

3.1.8. Parámetros

Los **parámetros** se utilizan para pasar valores 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.

3.1.9. 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.

3.1.10. Operadores

Un **operador** es un miembro que define el significado de aplicar un operador de expresión particular. Se pueden definir solamente operadores binarios.

Operadores Binarios:

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

3.1.11. 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'
| '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>' ';'
;

3.2. Explicación de la Implementación

Para crear nuestra gramática nos apoyamos en los lenguajes Python y CSharp, de ahi el nombre de nuestro DSL PSharp. Luego de tener los tokens resultantes del tokenizer, acordamos quée tendríamos como una línea e implementamos el metodo split_lines , 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 fueron 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 grámatica que no pertenezcan a nuestro lenguaje.De esta forma con una gramática válida empezamos al proceso de parsing. Nuestro parser necesitó un método "hacer_first"primeramente para hacer los first de cada cadena posible de nuestra gramatica, luego llamamos un método auxiliar çalcular_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 porque los necesitamos para hallar los follows en el método "hacer_follow". Luego de invocar a "hacer_follow"debemos invocar un método auxiliar çompletar_follows.[el] cual se encarga de satisfacer la regla de los follows que dice que el follow de la cabeza de la produccion es subconjunto del follow del ultimo 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 método çonstruir_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 método parsear al que hay que pasarle todas las líneas de nuestro código una por una , este realiza la comprobación sintáctica y en este mismo método vamos a ir creando nuestro AST para luego hacer el chequeo semántico. Para crear el AST utilizamos métodos como ÇreaNododExpresion", ÇreaNodoCondicion", ÇreaNododFuncion", .[El]igeTipoDdeclaracion", entre muchos 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 si existe un ámbito como es el caso del nodo If, el While, los tipos rider, bike, environment y la definición de función cada uno de estos nodos tienen como atributo un tipo de nodo Program, el cual 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 nodo. Todo lo que se defina en dicho ámbito pertenece a su contexto específicamnete, 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 función fuera de un tipo o cuando se crea un tipo. Destacar que las funciones definidas dentro de los tipos no crean un contexto específico para ellas, sino que su contexto es el mismo que el de el tipo, y no se le pueden pasar parámetros a las que pueden ser utilizadas luego en la simulación.

Volviendo al AST, hacemos una segunda pasada, en esta pasada verificamos los tipos,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 en una expresión. La tercera pasada la hacemos para evaluar nuestros nodos. Si encontramos un nodo Def_Fun no lo evaluamos, una definición de función se evalúa cuando se llama a la función.

Cuando se crea un tipo se importan las variables que puede tener ese tipo en la simulación y las que podría utilizar. Estas variables están predefinidas, por lo tanto se pueden redefinir, pero una definición de otra variable con un nombre igual al de alguna de estas variables predefinidas arrojaría un error, ya que existe una variable en ese contexto con ese nombre dentro del tipo. Dentro de cada tipo se pueden definir las funciones que el jefe técnico decida pero hay algunas funciones con nombres claves como son la función "select_configuration.^{en} un tipo Bike , las funciones "select_accelerationz "select_action.^{en} un tipo rider y la función probability_change_weather en un tipo environment. El objetivo de select_configuration es seleccionar el tipo de gomas dadas las características de la moto y por tanto modificar la variable "tires"del contexto del tipo, la cual utilizará el simulador, esta función debe ser void, "select_action"debe retornar un valor entero y es la encargada de elegir que acción realizara el piloto. La función "select_acceleration"debe ser void y su objetivo es actualizar la aceleración de un agente. Las variables dentro de los tipos se actualizan justo antes de que la simulación utilice las funciones claves, para eso cada tipo tiene un método refreshContext que es quien se encarga de actualizar las variabes. Las variables que se pueden utilizar dentro de un tipo Bike son los atributos de dicha moto y las características del clima, dentro del tipo Rider podemos utilizar las características del piloto, algunas características del agente como son speed, acceleration y time_lap, los atributos del clima y de la sección en la que esta el piloto en ese momento.Dentro de un tipo environment podemos utilizar solo las variables propias del clima y la variable track que es de tipo string y se utiliza si dicho tipo environment es a partir del cual se crea el clima al principio de la simulación , la variable track sirve para crear la pista de diferentes formas.

3.3. Conexión Simulación - Compilación

El resultado del DSL si se hacen los 3 recorridos del AST sin error, son 3 listas, la primera lista con todos los pilotos que fueron creados, la segunda con todas las motos y la tercera con diferentes ambientes .A partir de estas listas se crean los pilotos, las motos y el ambiente en la simulación, puede definirse un ambiente o no en el DSL. Los pilotos que no fueron creados en el DSL ejecutan sus métodos normalmente. En el caso de un piloto que fue creado desde el DSL se ejecuta la función definida en el DSL ,antes de ejecutarla se actualiza el contexto de la función para que esta 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.

Referencias

[1] Conferencias de Simulacion. Curso 2021-2022. Facultad de Matemática y Computación, Universidad de la Habana.

[2] L. Garcia, L. Marti, and L. Perez. *Temas de Simulación. Facultad de Matemática y Computación, Universidad de la Habana.*

[3] Conferencias de Inteligencia Artificial. Curso 2021-2022. Facultad de Matemática y Computación, Universidad de la Habana.

[4] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach.* Fourth edition edition, 2021.

[5] Conferencias de Compilacion. Curso 2021-2022. Facultad de Matemática y Computación, Universidad de la Habana.

[6] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools (Dragon Book).* Second edition edition, 1986.

[7] Documentación oficial de pyke. http://pyke.sourceforge.net.