Proyecto Monopoly Deal

Simulación + Inteligencia Artificial + Compilación

Kevin Talavera Díaz C-311

Facultad de Matemática y Computación Universidad de La Habana

Curso 2021-2022

Link: https://github.com/KevinTD15/Monopoly_Deal-Project

Resumen

El proyecto desarrollado implementa un juego llamado Monopoly Deal en Python, que incluye aspectos de Inteligencia Artificial, Simulación y un DSL con su Intérprete. Existen 2 programas principales para su ejecución, uno que incluye el DSL y otro sin el DSL. Esta compuesto por 5 librerías.

Introducción

El juego de cartas Monopoly Deal reúne toda la diversión del juego de mesa Monopoly en un rápido juego de cartas. Está compuesto por 106 cartas que incluyen cartas de Propiedad, cartas de Renta, cartas de Acción, cartas de Casa y Hotel, Cartas de Dinero y cartas de Comodín de Propiedad. Las cartas de Comodín de Propiedad permiten a los jugadores formar grupos de propiedades. Las cartas de Acción permiten a los jugadores cobrar rentas y hacer hábiles negocios. Las cartas de Casa y Hotel suben el valor de la renta. Y las cartas de Dinero sirven para pagar las deudas. Para ganar completa 3 grupos de Propiedades de distinto color antes que los demás (ver reglas en pdf adjunto).

Estructura del proyecto

El proyecto implementado se encuentra estructurado de la siguiente manera:

Tipo	Nombre	Función
Modulo principal	mainConsola.py	Ejecuta Juego partiendo de la introducción de los datos de entrada por la consola.
	mainDSL.py	Ejecuta Juego a partir de fichero de entrada con código del DSL (ext .mpd)
Librería	Juego	Incluye el control del juego, las jugadas y las funciones del crupier.
	Jugador	Contiene los distintos tipos de jugadores con sus acciones.
	Mazo	Creación de las cartas básicas que componen el mazo del juego.
	PLN	Mecanismo para crear cartas nuevas y añadirlas al juego.
	DSL	Se explica en apartado de Compilación.

- **Simulación**

Este proyecto se basa en la simulación de partidas del juego Monopoly Deal, donde para comenzar su ejecución es necesario decidir si agregar nuevas cartas al juego (ver en el apartado de IA), luego hay que decidir la cantidad de jugadores, estos deben de estar entre dos y cinco de acuerdo a las reglas del juego, y acto seguido se deberán introducir los nombres junto a los tipos de jugadores (ver en apartado de IA como fueron implementados) y luego definir la cantidad de partidas a ejecutar, o sea, la cantidad de simulaciones a realizar.

Está basada en un enfoque de **agentes** (jugadores), los cuales pueden tener comportamientos tanto **proactivos** como **reactivos**, en dependencia del estado donde se encuentre el juego. Por otra parte, tenemos el **medio ambiente** donde se desarrollan los agentes, que es el juego, pero más concretamente es el tablero de cada uno de los jugadores junto al mazo de cartas y la pila de descartes, del cual los agentes obtienen información para realizar sus jugadas.

Explicación del comportamiento de los jugadores. Sea X un jugador tomado como ejemplo en una partida, sus posibles momentos de toma de decisión en el juego son:

1- X se encuentra en su turno y tiene que decidir que carta o cartas va a usar y como lo va a hacer en dependencia de que carta sea. Ejemplo: si es una carta de acción de renta tiene que decidir contra que jugador la va a usar, tomando en cuenta las cartas que tiene en el tablero tanto él como el adversario, o si es un comodín, debe decidir dado las cartas que tiene en su tablero en que grupo de color lo va a colocar, entre otros. Este es un comportamiento proactivo y la toma de decisiones anteriormente expuesta a modo de ejemplos, depende del tipo de jugador, el cual fue definido antes de comenzar la simulación.

2- El turno actual del juego no corresponde a X, pero aun así usan una carta que lo afecta, en ese momento debe decidir qué curso de acción tomar. Ejemplo: si usan contra él una carta de acción de renta o de robar dinero, debe decidir cómo pagar, ya sea usando propiedades, cartas de dinero, o una combinación de ambas, tomando en cuenta cómo quedaría su tablero después de devolverlas, o también puede darse el caso de tener un tipo de carta que directamente niega el efecto usado contra él, y tiene que decidir si usarla en ese momento vale la pena o no. Este es un comportamiento reactivo ya que está reaccionando a estímulos externos, los cuales en este caso son otros agentes (jugadores), y la toma de decisiones anteriormente expuesta a modo de ejemplos depende del tipo de jugador, el cual fue definido antes de comenzar la simulación.

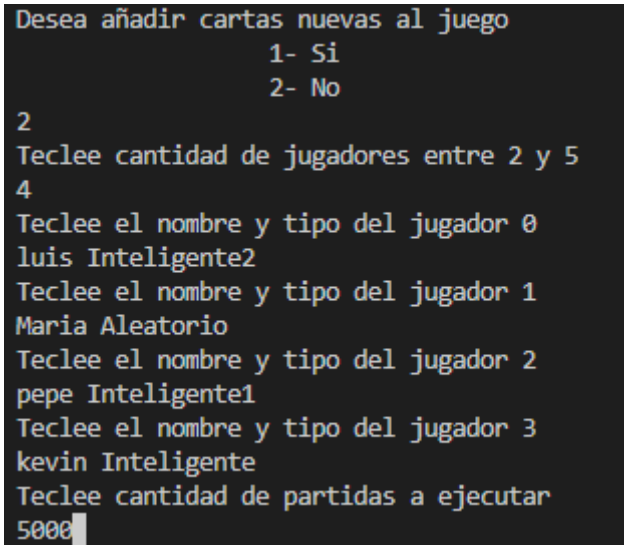
3- Es el final del turno de X (ya realizó todas sus posibles jugadas) pero la cantidad de cartas en su mano es inválida según las reglas (no se puede tener más de siete cartas en la mano al finalizar el turno) por tanto tiene que descartar tantas cartas como sea necesario hasta llegar a siete, en este punto tiene que decidir dado las cartas que posee, cuales serían las que menos le afecta perder, tomando en cuenta el tablero y el posible uso que tendría cada una. Estas decisiones se toman dependiendo del tipo de jugador, el cual fue definido antes de comenzar la simulación.

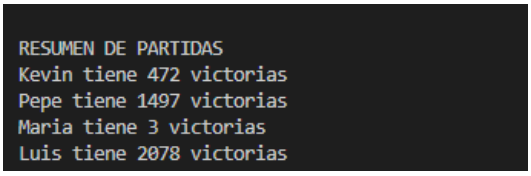
Como resultado de realizar estas simulaciones, variando la cantidad de jugadores por partidas y los propios tipos de estos, así como la cantidad de posibles jugadas generadas para ser evaluadas (ver más en detalle en Inteligencia Artificial), se pudo ir modificando las heurísticas para ir obteniendo cada vez mejores funciones de evaluación, y por ende mejorar la forma de juego de los jugadores implementados.

En este proyecto también se implementó la simulación de Montecarlo, que, a pesar de ser una herramienta de Inteligencia Artificial, realiza simulaciones de muchas partidas dado un estado específico de un juego. Su explicación se encuentra en el apartado de Inteligencia Artificial.

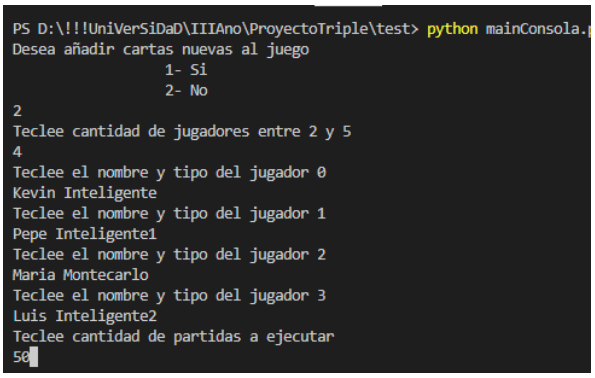
A continuación, se muestran algunos de los resultados del estado actual del proyecto:

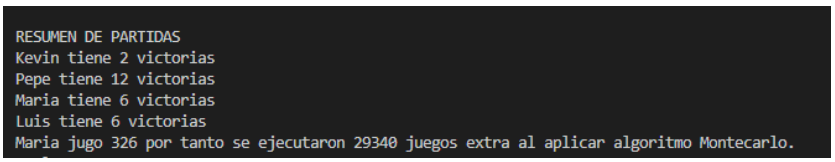
Resultados con 4 jugadores en 5000 partidas:





Resultados con 4 jugadores donde uno de ellos utiliza simulación de Montecarlo en 30 partidas:





Leyenda:
Inteligente: Jugador que hace uso de probabilidades en las heurísticas.
Inteligente1: Jugador que no toma en cuenta acciones del rival en sus heurísticas.
Inteligente2: Jugador como el Inteligente1 pero con valores más precisos en las heurísticas.
Montecarlo: Jugador que utiliza la simulación de Montecarlo para efectuar jugadas.
Aleatorio: Jugador que efectúa jugadas al azar.

¨Estos tipos de jugador se verán explicados más en detalle en el apartado de Inteligencia Artificial¨

- **Inteligencia Artificial**

Este proyecto cuenta con tres de sus aplicaciones: **heurísticas**, **procesamiento de lenguaje natural y simulación** de **Montecarlo.**

El juego consta de **5 tipos de jugadores**, de los cuales **3** tienen **funciones heurísticas**, uno utiliza **simulación** de **Montecarlo** y el otro es totalmente **aleatorio**. De los 3 jugadores "inteligentes" implementados, uno hace uso de probabilidades basadas en qué tan probable es que cartas que tenga algún contrario pueda llegar a perjudicarlo junto a qué tan buena sería hacer la jugada; otro calcula qué tan buena es una jugada pero sin tomar en cuenta acciones que pueda realizar el rival, este jugador fue tomado como base debido a su buen rendimiento en las simulaciones que se realizaron para probar diferentes heurísticas e ir variando sus valores, y el mejor resultado obtenido fue el último de los jugadores inteligentes, el cual funciona justo como el anteriormente explicado pero con valores más certeros y un rendimiento aún mejor.

Las **heurísticas** fueron desarrolladas como parte de un problema de búsqueda donde se generan aleatoriamente una cantidad (definida por parámetros) de posibles jugadas y estas escogen la mejor de todas y son las que se ejecutan. Específicamente son 3, una que se centra en elegir la mejor jugada que le toca hacer a un jugador en su turno, otra que devuelve cuales serían las cartas que al descartarlas afecta menos al jugador (ver en las reglas cuando un jugador debe descartar cartas), y por último tenemos una que devuelve cuál es la mejor acción a tomar cuando tenemos que reaccionar ante la jugada de algún adversario.

Con respecto al algoritmo **Expectiminimax** debido a que la cantidad de jugadas que se pueden hacer por turno son: $n^3 + n^2 + n + 1 : "n"\ es\ la\ cantidad\ de\ cartas\ en\ la\ mano$ (todas las formas de escoger 3 cartas del total de cartas en la mano más todas las formas de escoger 2 cartas del total de cartas en la mano más todas las formas de escoger 1 carta del total de cartas en la mano, o sea **n,** más 1, el cual se refiere a la jugada donde no se realiza ninguna acción), si tomamos como caso promedio donde $n = 7$ (este es el caso más usual por las estrategias del juego) serían unas 400 jugadas posibles a realizar en un turno. Por otra parte el mazo de cartas tiene 106 cartas (sin contar las cartas que se pueden añadir al juego), por lo que implementar el algoritmo de Expectiminimax tendría que crear $\binom{106}{2}$ nodos CHANCE debido a que por turno se toman dos cartas del mazo y por cada uno ver todas las jugadas posibles, que como se explicó anteriormente ronda las 400. Debido a esto el árbol generado por Expectiminimax, tomando en cuenta que solo participen 2 jugadores sería demasiado costoso desde su primer nivel como para crearlo, por lo que no fue implementado.

La **simulación de Montecarlo** implementada se basa en que se creó un nuevo tipo de jugador, el cual utiliza las heurísticas de los otros tres tipos de jugadores inteligentes implementados y toma la mejor jugada que devuelve cada uno de ellos, esto se realizó de esta forma debido a que como se explicó anteriormente se pueden hacer alrededor de 400 jugadas diferentes en un turno y para cada una de estas habría que realizar una cantidad X de simulaciones para ver cuál es la que devuelve el mayor porcentaje de victorias, por lo que si se toma el valor de X = 30, se tendrían que realizar 12000 partidas para cada jugada antes de saber cuál es la mejor, y teniendo esto, en un juego relativamente corto donde cada jugador tenga 10 turnos, se tendrían que ejecutar 120000 partidas, lo cual es demasiado costoso. Por tanto, si solo tomamos 3 como se aclaró al principio se reduce considerablemente la cantidad de partidas a ejecutar a 900 tomando el mismo valor de X y la misma cantidad de turnos.

El **procesamiento de lenguaje natural** (PLN) esta implementado para poder añadir nuevas cartas al juego de cualquiera de los tipos predefinidos con sus respectivos parámetros. Esto se logra mediante la implementación de una gramática para poder procesar los textos de forma eficiente y

elegante. Luego al obtener el AST (Árbol de Sintaxis Abstracta) que este devuelve se verifica si es una carta válida, se crea y se añade al mazo de cartas para poder usarse en la simulación.

- **Gramática**

  Implementada para permitir al usuario crear cartas de todo tipo, exceptuando cartas de acción rápida.

  Para añadir cartas nuevas el usuario puede escribir en lenguaje natural sus especificaciones, pero con las siguientes restricciones:

1- Para crear una carta de Propiedad válida, el usuario debe definirle un nombre, este tiene que tener la letra inicial en mayúscula; también debe de especificarse el color, de cuantas propiedades de ese color estará conformado el grupo (Ej. Las propiedades azules tienen grupo de tamaño 2), cuales son las rentas, las cuales son el resultado de tener varias propiedades de un mismo color en el tablero, estas tienen que estar en consonacia con lo dicho en el punto anterior (Ej. Como las azules tienen grupo de tamaño 2, este grupo solo podrá tener 2 rentas, estos deben ser ingresados por coma de la forma: <num><,><' '><num> …), y por último se de debe definir un valor a la carta, para esto de debe teclear la palabra "valor" o "valga" seguido de cualquier palabra o grupo de palabras y luego el valor que queremos asignarle seguido de la letra "M" (esta hace referencia a la moneda del juego). Todo esto puede ser tecleado en cualquier orden.

2- Para crear una carta de Comodín válida, el usuario debe definirle un nombre, este tiene que tener la letra inicial en mayúscula; también debe definir los colores por los que se podrá jugar esta carta, y en el caso que desee crear el comodín maestro, no escribir ningun color específico o escribirlos todos, y por último definirle un valor, análogo a como se hace en la creación de propiedades.

3- Para crear una carta de Dinero basta con poner el valor desead.o seguido de "M".

4- Para crear una carta de Acción Renta el usuario debe definirle un nombre, este tiene que tener la letra inicial en mayúscula; también debe teclear todos los colores para los que sea válida esa renta y en caso de no definir un color se tomará como si valiera por todos, también hay que especificar si es un efecto contra todos los jugadores o no, en caso de ser contra todos hay que escribir "todos" o "cada" en caso contrario es teclear la cantidad la cantidad de jugadores contra los que se usa la carta de la palabra "jugadores" o "participantes" y por último el valor de esta carta, análogo a como se hace en la creación de propiedades.

5- Para añadir una carta de Acción Construcción el usuario debe definirle un nombre, este tiene que tener la letra inicial en mayúscula; el tipo de construcción ya sea "casa" u "hotel", luego definir el monto por el que nos tendrán que pagar tecleando el valor seguido de "M" y por último el valor de esta carta, para esto de debe teclear la palabra "valor" o "valga" seguido de cualquier palabra o grupo de palabras y luego el valor que queremos asignarle seguido de la letra "M".

6- Para crear una carta de Accion para tomar cartas del mazo el usuario debe definirle un nombre, este tiene que tener la letra inicial en mayúscula; luego debe poner la cantidad de cartas que desea tomar seguido de la palabra "carta", "tarjeta" o "naipe", y en caso de ser más de una poner esas palabras en plural y por último el valor de esta carta, análogo a como se hace en la creación de propiedades.

7- Para crear una carta de Acción que robe propiedades, el usuario debe definirle un nombre, este tiene que tener la letra inicial en mayúscula; también debe decidir si será una carta con la cual 2 jugadores intercambien cartas o si las roba directamente, y esto le logra poniendo la palabra "intercambio" o "intercambiar" y en caso que lo que desee sea robar, no teclear ninguna de estas 2 palabras. Luego hay que definir cuantas cartas se van a tomar del rival tomar seguido de la palabra "carta", "tarjeta" o "naipe", y en caso de ser más de una poner esas palabras en plural y en caso de que no se defina una cantidad se definira que la carta roba un grupo completo de color y por último el valor de esta carta, análogo a como se hace en la creación de propiedades.

8- Para crear una carta de Acción que robe dinero, el usuario debe definirle un nombre, este tiene que tener la letra inicial en mayúscula; luego se debe definir si se va a usar contra todos o contra 1 solo jugador, en caso de ser contra todos hay que escribir "todos" o "cada" en caso contrario es teclear la cantidad la cantidad de jugadores contra los que se usa la carta de la palabra "jugadores" o "participantes", también tiene que definir el monto que el/los participantes deberán pagar seguido de la letra "M" y por último el valor de esta carta, análogo a como se hace en la creación de propiedades.

- **Compilación**

El intérprete ejecuta código fuente de un lenguaje programación diseñado para ejecutar el juego de "Monopoly Deal" (MPD en lo adelante). El DSL se llama Mpd, de (Monopoly Deal), y la extensión de los programas de este lenguaje es 'mpd'.

El DSL creado no resuelve un problema específico, lo que permite es, de una manera sencilla configurar la ejecución del MPD, tanto los datos de entrada (creación de jugadores y/o cartas, el manejo de turnos de juego a ejecutar y cantidad de simulaciones a realizar), así como el tratamiento de los resultados y la posibilidad de reiniciar el juego.

Para implementación del DSL se utilizaron las librerías lex (para el analizador léxico) y yacc (para el análisis sintáctico) de PLY y se partió un link[i] que tenía desarrollado un compilador básico que ejecutaba las tablas de multiplicar, en él se encontraban implementadas las instrucciones de: ciclos while, bloques if-else, declaracion de números (enteros) e impresión por consola (print).

Las funcionalidades incluidas fueron las siguientes:

1. **Declaración de funciones** no anidadas (con y sin retorno). Permite incluir **lista de argumentos**.
2. **Ejecución de funciones**, tanto las **definidas** en el **DSL**, como **definidas** en el **MPD**.
3. **Tratamiento de listas** (crear lista, añadir elementos, asignar valor a una posición, obtener el valor de un elemento). Se **reconocen también listas creadas** en el **MPD**, por tanto, se implementaron todas las funcionalidades del tratamiento de listas. excepto la creación.
4. **Asignación** a **variables simples definidas** en MDP.
5. Permite **añadir jugadores** a la **lista de jugadores** del **MPD**, cuyos **elementos** son **instancias** de **clases** en dependencia del **tipo** de **jugador**.
6. Permite **obtener valor** de **atributos** de una **instancia** de **clase** que es **elemento** de **un arreglo definido** en el **MPD**.
7. **Crear cartas** nuevas haciendo uso del módulo de procesamiento de lenguajes implementado en Python.
8. Las **funciones** pueden formar parte de las **expresiones**.
9. **Crear** un **juego** (crea instancia del juego).

10. Permite **obtener** la **longitud** de una **cadena o lista.**
11. **Tratamiento** de **errores** en la **fase** de **ejecución** del código (se muestra mensaje de error y la fila del programa fuente que provocó el error), que es donde se realiza el "análisis semántico".

- **Sintaxis del lenguaje**

➢ **Imprimir**:
<*imprimir*> < (> < expresión > <) >
Se permite concatenas expresiones de cadena con "&".

➢ **Mientras**:
<*mientras*> < (> < expresión lógica > <) > < { > < instrucciones > < } >

➢ **Si**:
< *si* > < (> < expresión lógica > <) > < { > < instrucciones > < } >

➢ **Sino**:
< *sino* > < { > < instrucciones > < } >

➢ **Declaración de funciones**:
< *func* > < identificador > < (> < [parámetros] > <) > < { > < instrucciones > < } >

➢ **Llamado de función**:
< identificador > < (> < [parámetros] > <) >

➢ **Retorno**:
< *retorno* > < expresión >

➢ **Asignación**:
< variable > < = > < expresión >

➢ **Agregar elemento a una lista**:
< *agregar* > < expresión > < identificador >

➢ **Crear el juego**:
< *crear* > < *juego* >

➢ **Crear jugador**:
< *crear* > < *jugador* > < tipo de jugador > < identificador >

➢ **Crear carta**:
< *crear* > < *carta* > < cadena >

➢ **Ejecutar turno del juego**:
< *ejecutar* > < *turno* >

➢ **Reestablecer parámetros del juego**:
< *reestablecer* > < *juego* >

- **Característica de la gramática**

La gramática implementada es regular libre de contexto, el método de parseo es el de PLY (LALR) donde las reglas del lenguaje son las siguientes:

Reglas del lenguaje definidas anteriormente excepto la regla 2.

Rule 0 S' -> inicio
Rule 1 inicio -> instrucciones
Rule 2 inicio -> vacio
Rule 3 instrucciones -> instrucciones instruccion
Rule 4 instrucciones -> instruccion
Rule 5 instruccion -> instrImprimir
Rule 6 instruccion -> instrAsignacion
Rule 7 instruccion -> instrWhile
Rule 8 instruccion -> instrIf
Rule 9 instruccion -> instrIfElse

Reglas nuevas.

Rule 10 instruccion -> declFuncion
Rule 11 instruccion -> llamaFuncion
Rule 12 instruccion -> instrRetorno
Rule 13 instruccion -> instrAgregar
Rule 14 instruccion -> instrEliminar
Rule 15 instruccion -> instrLen
Rule 16 instruccion -> instrCrear

Definidas para el tratamiento de variables simples y elementos de lista.

Rule 17 variable -> ID
Rule 18 variable -> ID CORCHEIZQ expresion CORCHEDER
Rule 19 variable -> idjuego
Rule 20 variable -> idjuego CORCHEIZQ expresion CORCHEDER atributo

Definen los identificadores del juego.

Rule 21 idjuego -> JUGADORES
Rule 22 idjuego -> NOTIFICACIONES
Rule 23 idjuego -> GANADOR
Rule 24 idjuego -> FINALJUEGO

Para saber en la ejecución si es de juego el identificador, es que se utiliza la clase IdJuego para construir la instrucción que va a componer el AST. (t[0]=IdJuego(t[1])).
El proceso ejecución de las instrucciones, reconoce el identificador a quien pertenece, al leer el 1er parámetro del método que ejecuta la instrucción, si es una instancia de la clase idJuego (en este caso), realiza la acción con el identificador del juego (que se encuentra como atributo de idJuego), de lo contrario realiza la acción que corresponda con el identificador definido en el código de entrada.

Para el tratamiento de atributo cuando la producción corresponde a un elemento de lista simple o de tipo clase.

Rule 25 atributo -> PUNTO ID
Rule 26 atributo -> vacio

Reglas relacionadas con la declaración de función y la lista de argumentos

Rule 27 declFuncion -> FUNC ID PARIZQ argument_list_decl PARDER LLAVIZQ instrucciones LLAVDER
Rule 28 argument_list_decl -> argument_list_decl COMA argument_decl
Rule 29 argument_list_decl -> argument_decl
Rule 30 argument_list_decl -> vacio
Rule 31 argument_decl -> variable

Reglas relacionadas con el llamado a la función y la lista de argumento que es diferente a la de declaraciones pues en el llamado se pasan expresiones

Rule 32 argument_list -> argument_list COMA argument
Rule 33 argument_list -> argument
Rule 34 argument_list -> vacio
Rule 35 argument -> expresion
Rule 36 llamaFuncion -> ID PARIZQ argument_list PARDER

Reglas que derivan en ejecución de funciones del MPD, para ello se siguió la misma política de las variables, en el lugar del identificador se creó instancia de clase que me dice a quién corresponde la función, si está definida en el programa mdp o en el juego
(t[0] = Funcion(t.slice[1].lineno,IdFuncionJuego(t[1]), t[2]))

Resaltar que en las producciones donde se completa la instrucción, se captura la línea del código fuente para su posterior uso en la ejecución en caso de ser necesario.

Rule 37 llamaFuncion -> REESTABLECER JUEGO
Rule 38 llamaFuncion -> EJECUTAR TURNO

Regla que define la para ejecutar retorno.

Rule 39 instrRetorno -> RETURN expresion

Reglas que corresponde a imprimir

Rule 40 instrImprimir -> IMPRIMIR expresion

Definida para calcular longitud de una expresion

Rule 41 instrLen -> LEN PARIZQ expresion PARDER

Definición de reglas para admitir asignación.Se creó regla para admitir asignación de funciones y de elementos de lista.

Rule 42 instrAsignacion -> variable IGUAL expresion
Rule 43 instrAsignacion -> variable IGUAL llamaFuncion
Rule 44 instrAsignacion -> variable IGUAL CORCHEIZQ CORCHEDER

Reglas definidas parcialmente con anterioridad pues se eliminaron los paréntesis y se permite que las expresiones lógicas sean cualquier tipo de expresión. Al igual que al resto de las reglas que se utilizaron del proyecto del que se partió, se eliminó el símbolo “;”.

Rule 45 instrWhile -> WHILE expresion_logica LLAVIZQ instrucciones LLAVDER
Rule 46 instrIf -> IF expresion_logica LLAVIZQ instrucciones LLAVDER
Rule 47 instrIfElse -> IF expresion_logica LLAVIZQ instrucciones LLAVDER ELSE LLAVIZQ instrucciones LLAVDER

Regla para añadir elementos a una lista (definida en el juego o en el código de entrada). Se aplica el mismo principio para saber dónde está definido el id.

Rule 48 instrAgregar -> AGREGAR expresion ID
Rule 49 instrAgregar -> AGREGAR expresion idjuego

Para eliminar elemento de una lista.

Rule 50 instrEliminar -> ELIMINAR expresion ID
Rule 51 instrEliminar -> ELIMINAR expresion idjuego

Reglas que inserta la posibilidad de crear el juego, crear jugadores, crear cartas

Rule 52 instrCrear -> CREAR JUEGO
Rule 53 instrCrear -> CREAR CARTA CADENA
Rule 54 instrCrear -> CREAR JUGADOR tipo ID

Reglas para definir los tipos de jugadores

Rule 55 tipo -> ALEATORIO
Rule 56 tipo -> INTELIGENTE
Rule 57 tipo -> INTELIGENTE1
Rule 58 tipo -> INTELIGENTE2

Definidas con anterioridad

Rule 59 expresion_numerica -> expresion_numerica MAS expresion_numerica
Rule 60 expresion_numerica -> expresion_numerica MENOS expresion_numerica
Rule 61 expresion_numerica -> expresion_numerica POR expresion_numerica
Rule 62 expresion_numerica -> expresion_numerica DIVIDIDO expresion_numerica
Rule 63 expresion_numerica -> PARIZQ expresion_numerica PARDER
Rule 64 expresion_numerica -> ENTERO
Rule 65 expresion_numerica -> DECIMAL

Permitir variable y función len como expresión

Rule 66 expresion_numerica -> variable
Rule 67 expresion_numerica -> instrLen

Definida las reglas de expresiones

Rule 68 expresion -> expresion CONCAT expresion
Rule 69 expresion -> CADENA
Rule 70 expresion -> expresion_numerica
Rule 71 expresion_logica -> expresion MAYORQUE expresion
Rule 72 expresion_logica -> expresion MENORQUE expresion
Rule 73 expresion_logica -> expresion IGUALQUE expresion
Rule 74 expresion_logica -> expresion NIGUALQUE expresion

Regla “vacío”

Rule 75 vacio -> <empty>

- **Arquitectura**

El intérprete del DSL de MPD, analiza un archivo de entrada que contiene una especificación en lenguaje MPD, si es correcto, ejecuta el código, de lo contrario, envía mensaje de error propio del DSL y aborta la ejecución.

La arquitectura del intérprete está compuesta por 3 fases diferenciadas: análisis léxico, análisis sintáctico, y el “análisis semántico” se realiza a la par de la ejecución de las instrucciones.

Las dos primeras fases, el Analizador Léxico y el Analizador Sintáctico, han sido generados automáticamente a partir de la gramática del lenguaje mediante la herramienta PLY.

Cuando se inicia la interpretación de un programa fuente de entrada, este es procesado por el **analizador léxico**, generando una secuencia de componentes léxicos mínimos (tokens).

Esta secuencia es procesada por el **analizador sintáctico** quien se encarga de definir el orden de las instrucciones en relación a la precedencia en que estas se van a ejecutar aplicando las reglas del lenguaje, y para la construcción del AST, se implementó una clase para cada tipo de nodo, lo que permite que, en el momento de su creación, se le asigne una nueva instancia de clase que indica la acción a realizar en cada caso en tiempo de ejecución. La ventaja de este enfoque es que puede facilitar la incorporación de semánticas más complicadas, verificación de tipos, generación de código y otras funciones a las clases de nodos, aunque en este interprete no se aplicó estas facilidades. En ambas fases se muestra error en dependencia del momento en que ocurrió.

Una vez construido el **AST** que representa al programa fuente, se procede a ejecutar la 3ra fase de este intérprete, la cual comienza a **procesar instrucción** por instrucción mediante la ejecución de procedimientos implementados asociados a cada acción representada en la clase instanciada correspondiente al nodo en curso. Durante el proceso de ejecución se realiza el “**análisis semántico”** donde se valida índices fuera de rango, uso incorrecto de variables (obtener valor de una variable que no existe, obtener un elemento de una lista que no lo es), etc., al mismo tiempo esta fase es responsable de controlar los ámbitos del programa fuente para asegurar el correcto tratamiento de los símbolos (identificadores) que se definen en dicho entorno, esto se garantiza creando una instancia de la tabla de símbolos antes de ejecutar el conjunto de instrucciones asociadas a la función llamada, por tanto, solo la función tiene acceso a su tabla de símbolo. La tabla de símbolos es la clase implementada que permite el almacenamiento y recuperación de los valores de las variables.

Para el **tratamiento de los objetos asociados a los módulos del juego implementado**, a la hora de construir el AST, en lugar de enviar como parámetro de la clase a instanciar, el identificador de la producción, se crea una nueva instancia (con el identificador como parámetro) que identifica a la hora de la ejecución, que el objeto a tratar tengo que buscarlo en el juego. Para ello se accede mediante la variable que contiene la instancia creada del juego, que fue inicializada cuando se ejecutó la línea de código “crear juego”.

Componentes del interprete (carpeta ‘**dsl’** dentro del MPD):

Módulo	**Función**	**Librerías externas**	**Librería interna**

mainDSL.py	Pedir archivo de entrada ‘mpd’ (archivo implícito ‘juegoMD.mpd’) y al interprete		Interprete
mainConsola.py	Pide los datos de entrada por consola y ejecuta el juego		
gramatica.py	Análisis léxico y sintáctico y obtiene AST	Ply (lex y yacc)	
interprete.py	Ejecuta la gramática y ejecuta nodos del AST, a la par realiza “análisis semántico”		Las 4 definidas en la tabla de abajo

Para la ejecución de interprete se hace uso de las clases y métodos definidos en la Librerías de Entorno de Ejecución.

Librerías	Función	Observación
expresiones.py	Contiene las clases de expresiones a instanciar	
instrucciones.py	Contiene las clases de instrucciones a instanciar	Se añadieron clases no definidas con anterioridad
TS.py	Contiene la clase de símbolo para el tratamiento de identificadores. Y métodos para la gestión de las variables	Se añadieron y modificaron métodos, principalmente para tratamiento de listas y errores
Juego.py	Contiene clases para el tratamiento de objetos asociados al MPD	Nuevo

Adicionalmente esta la carpeta ‘Ejemplos’ que contiene ficheros de entrada ‘mpd’ para probar la ejecución del intérprete.

Para poder ejecutar un juego se tienen que definir los siguientes elementos:

Crear juego -> momento en que se instancia la clase juegoMD en la variable “jmpd”,
esta variable tiene que ser pasada como parámetro a cualquier función que dentro haga referencia a algún objeto del juego.

Crear jugadores -> crear los jugadores seleccionado el tipo de cada uno y dándole nombre, usted puede crear los que desee, y después añadir a la lista un subconjunto de los creados. La lista puede cambiarla en cualquier momento. El crear un jugador no quiere decir que esté listo para jugar, para ello tiene que añadirlo a la lista de jugadores.
Ejemplo de crear: crear jugador inteligenteProbabilistico luis

Agregar jugador a l lista de jugadores-> Pone en línea a los jugadores que van a participar en el/los juego(s) a efectuar.
Ejemplo: agregar luis jugadores

Definir la cantidad de juegos a efectuar, la variable no pertenece al juego.

En el mpd que ejecuta el juego, se implementaron los métodos ejecutarSimulacion y EjecutarJuego que están definidas en el juego, pero se hizo para ampliar la posibilidad de

acceder a elementos del juego desde el programa mpd. Esto permite maniobrar con la forma en que quiero ejecutar el juego.

Aspectos de la ejecución:

La ejecución del retorno se concibió de la siguiente manera:

La instrucción de retorno según lo definido en este DSL, siempre retorna un valor distinto de *None*. Por otro parte, el "retorno" puede formar parte del conjunto de instrucciones perteneciente a función, a un "si" o a un "mientras", por ello, en cada una de ellas, se puede recibir un retorno de la última instrucción ejecutada, valor que es capturado y si este es distinto de *None* se retorna del procedimiento el valor. Lugar donde también se captura el valor y se evalua si no es *None* …, y así sucesivamente.

En cuanto a las funciones como no se implementó las funciones anidadas, se creó una tabla de símbolo de funciones a nivel global a la cual se va a tener acceso en todo momento.

[i] https://ericknavarro.io/2020/03/15/26-Interprete-sencillo-utilizando-PLY/