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)

- **Simulación**

Está basada en agentes donde estos son los jugadores y pueden tener comportamientos tanto proactivos como reactivos en dependencia de estado del juego donde se encuentren. El medio ambiente es el tablero de juego, del cual los agentes pueden obtener la información necesaria para la toma de decisiones. Para un jugador, se encuentran implementados 3 tipos de comportamiento:

1- Jugar la carta (o cartas) que decida en dependencia de una heurística en su propio turno. Este es un comportamiento proactivo.
2- Reaccionar ante el efecto de cartas de adversarios en dependencia de una heurística, este es un comportamiento reactivo ya que es en respuesta a un estímulo externo, que, en este caso, viene de otro agente.
3- Descartar cartas de la mano.

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 mejorar las heurísticas.

- **Inteligencia Artificial**

Este proyecto cuenta con dos de sus las aplicaciones: **heurísticas** y **procesamiento de lenguaje natural.**

El juego consta de 4 tipos de jugadores, de los cuales 3 tienen funciones heurísticas y uno 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 que pueda llegar a perjudicarlo junto a qué tan buena sería hacer la jugada, otro calcula que 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 pruebas 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.

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 sus producciones son las siguientes:

Producciones no propias

```
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
```

Producciones que derivan en producciones no terminales donde se definirán instrucciones

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

Definidas para el tratamiento de variables simples y elementos de lista

(t[0]=t[1] o t[0]=ExpresionLista(t[1],t[3],t[5]) respectivamente)

```
Rule 5     variable -> ID
Rule 6     variable -> ID CORCHEIZQ expresion CORCHEDER
Rule 7     variable -> idjuego
Rule 8     variable -> idjuego CORCHEIZQ expresion CORCHEDER atributo
```

Identificadores del juego. Para saber en la ejecución si es de juego, es que se asigna al nodo una instancia que define esta produccion (t[0]=IdJuego(t[1]))

```
Rule 9     idjuego -> JUGADORES
Rule 10    idjuego -> NOTIFICACIONES
Rule 11    idjuego -> GANADOR
Rule 12    idjuego -> FINALJUEGO
```

El proceso ejecución de las instrucciones, reconoce el identificador a quien pertenece, al leer el 1er parámetro, si es una instancia de la clase idJuego (en este caso), realiza la acción con el identificador del juego (que se encuentra como parámetro 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 13     atributo -> PUNTO ID
Rule 14     atributo -> vacio
```

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

```
Rule 15     declFuncion -> FUNC ID PARIZQ argument_list_decl PARDER
LLAVIZQ instrucciones LLAVDER
Rule 16     argument_list_decl -> argument_list_decl COMA argument_decl
Rule 17     argument_list_decl -> argument_decl
Rule 18     argument_list_decl -> vacio
Rule 19     argument_decl -> variable
```

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

```
Rule 20     argument_list -> argument_list COMA argument
Rule 21     argument_list -> argument
Rule 22     argument_list -> vacio
Rule 23     argument -> expresion
Rule 24     llamaFuncion -> ID PARIZQ argument_list PARDER
```

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

```
Rule 25     llamaFuncion -> REESTABLECER JUEGO
Rule 26     llamaFuncion -> EJECUTAR TURNO
```

Produccion que define la ejecucion de un retorno

```
instrRetorno -> RETURN expresion
```

Produccion que corresponde a imprimir (anteriormente definido)

```
Rule 28     instrImprimir -> IMPRIMIR expresion
```

Definida para calcular longitud de una expresion

```
Rule 29     instrLen -> LEN PARIZQ expresion PARDER
```

Definicion de producciones para admitir asignación (regla 30 definida anteriormente)

Se creó producción para admitir asignación de funciones y de elementos de lista

```
Rule 30    instrAsignacion -> variable IGUAL expresion
Rule 31    instrAsignacion -> variable IGUAL llamaFuncion
Rule 32    instrAsignacion -> variable IGUAL CORCHEIZQ CORCHEDER
```

Producciones definidas parcialmente con anterioridad pues se eliminaron los paréntesis. Al igual que al resto de las producciones que se utilizaron del proyecto del que se partio, se elimino el simbolo “;”.

```
Rule 33    instrWhile -> WHILE expresion_logica LLAVIZQ instrucciones
LLAVDER
Rule 34    instrIf -> IF expresion_logica LLAVIZQ instrucciones LLAVDER
Rule 35    instrIfElse -> IF expresion_logica LLAVIZQ instrucciones
LLAVDER ELSE LLAVIZQ instrucciones LLAVDER
R
```

Producciones que posibilitan añadir elementos a una lista (definida en el juego o en el codigo de entrada). Se aplica el mismo principio para saber donde esta definido el id.

```
Rule 36    instrAgregar -> AGREGAR expresion ID
Rule 37    instrAgregar -> AGREGAR expresion idjuego
```

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

```
Rule 38    instrCrear -> CREAR JUEGO
Rule 39    instrCrear -> CREAR CARTA CADENA
Rule 40    instrCrear -> CREAR JUGADOR tipo ID
```

Producciones donde se definen los tipos de jugadores

```
Rule 41    tipo -> ALEATORIO
Rule 42    tipo -> INTELIGENTE
Rule 43    tipo -> INTELIGENTE1
Rule 44    tipo -> INTELIGENTE2
```

Definidas con anterioridad

```
Rule 45    expresion_numerica -> expresion_numerica MAS
expresion_numerica
Rule 46    expresion_numerica -> expresion_numerica MENOS
expresion_numerica
Rule 47    expresion_numerica -> expresion_numerica POR
expresion_numerica
Rule 48    expresion_numerica -> expresion_numerica DIVIDIDO
expresion_numerica
Rule 49    expresion_numerica -> PARIZQ expresion_numerica PARDER
Rule 50    expresion_numerica -> ENTERO
Rule 51    expresion_numerica -> DECIMAL
```

Permitir las variables y la función len como expresión

```
Rule 52    expresion_numerica -> variable
Rule 53    expresion_numerica -> instrLen
```

Definidas anteriormente con algún cambio pues existía la produccion de expresión de cadenas y fue cambiado por expresión .OJOOO no me acuerdo por que

```
Rule 54    expresion -> expresion CONCAT expresion
Rule 55    expresion -> CADENA
Rule 56    expresion -> expresion_numerica
Rule 57    expresion_logica -> expresion_numerica MAYORQUE
expresion_numerica
Rule 58    expresion_logica -> expresion_numerica MENORQUE
expresion_numerica
Rule 59    expresion_logica -> expresion_numerica IGUALQUE
expresion_numerica
Rule 60    expresion_logica -> expresion_numerica NIGUALQUE
expresion_numerica
```

Para permitir las producciones “vacios”

```
Rule 61    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 (se lee como parámetro las producciones necesarias si lo lleva) 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 funcion 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 anadir a la lista un subconjunto de los creados. La lista puede cambiarla en cualquier momento. El crear un jugador no quiere decir que este 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 ejecucion :

La ejecucion 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 asi sucesivamente.

En cuanto a las funciones como no se implementó las funciones anidadas, se creo una tabla de simbolo 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/