Proyecto Simulación, Compilación e Inteligencia Artificial

Simulador de Bots que operan activos financieros

Integrantes:
Karel Díaz Vergara
Frank Abel Blanco Gómez
Alben Luis Urquiza Rojas

Introducción:

Desde el surgimiento del dinero, la economía ha sido un renglón fundamental para la supervivencia y desarrollo de una persona, un pueblo o un país. El surgimiento de las criptomonedas han hecho tambalear el sistema económico tradicional y ha despertado un gran interés de cada vez más personas e instituciones.

Este proyecto tiene como objetivo ayudar a trabajar con bots que operen en mercados financieros, pueden ser los tradicionales como el mercado de divisas, de materias primas o de acciones, pero principalmente el mercado de los criptoactivos.

Para esto se crea un simulador de tres bots: Grid Bot, Rebalance Bot y Smart Trading Bot y una herramienta llamada Portafolio, que puede ser útil para configurar los parámetros de los bots, los cuales serán descritos en profundidad más adelante. Y para facilitar la interacción con los bots, un lenguaje en el que se pueda trabajar con activos y bots, configurándolo a gusto del usuario.

Install and run

```
$ git clone https://github.com/University-Projects-UH/ia-sim-cmp
$ cd ./ia-sim-cmp
$ python3 -m virtualenv env
$ source env/bin/activate
$ pip3 install < requirements.txt
$ python index.py code.botlang
```

Run tests:

```
$ pytest
```

Portafolio

Portafolio es una colección de diferentes activos con el objetivo de obtener una rentabilidad del mercado.

El problema de optimización de un portafolio, se basa en, dado un universo de activos y sus características, crear un método para repartir el capital entre ellos de forma que se maximice la rentabilidad del portafolio por unidad de riesgo asumido.

Se incluye en el proyecto esta funcionalidad, para contribuir a que el usuario escoja para los bots los activos que pueden llegar a tener mayor rendimiento en el futuro, especialmente para el bot de rebalanceo puede llegar a ser muy útil.

Para la solución del problema se usó el modelo de Markowitz consiste en la selección del portafolio más eficiente analizando varios posibles portafolios y se basa en el uso de rendimientos esperados(mean) y la desviación estándar(variance).

Se pueden utilizar dos tipos de portafolio, uno es el de *varianza mínima*(que vendría siendo el de menor riesgo), y el otro el de *mayor sharpe ratio*(que sería el portafolio de riesgo óptimo, pues maximiza la proporción entre el rendimiento esperado del portafolio y la desviación estándar, que se puede interpretar como el riesgo del portafolio).

La obtención de los portafolios mencionados anteriormente se realiza resolviendo un problema de optimización.

Bots

Grid Bot

Este bot solamente utiliza un par de monedas, por ejemplo BTC/USD. La estrategia es colocar una serie de compras y ventas en un cierto rango de precios (Lowest sell price, Highest sell price). Cuando una orden de venta es completamente ejecutada, el bot instantáneamente coloca otra orden de compra a un precio menor y viceversa. Esencialmente, el grid bot se aprovecha de pequeños cambios en el precio de un activo para obtener beneficios. (Figura 1)

Rebalance Bot

El rebalanceo de una cartera de inversión consiste en reajustar el peso de los diferentes activos que la componen, según la evolución del mercado y el perfil del inversor. Es decir, adaptar la inversión a los cambios que se van produciendo para obtener la rentabilidad más alta.

Este robot funciona de la siguiente manera. Se escoge la configuración, la cual está compuesta de un conjunto de activos, la proporción de cada activo, y la estrategia de reblanceo. Este programa pudiera ser muy útil, para ayudar a predecir cuáles pueden ser las mejores estrategias a seguir para crear bots y que estos sean lo más rentable posible. Así como para saber cuáles pueden ser los próximos movimientos de un activo en un futuro. Vamos a poner un ejemplo: Activos **BTC**(Bitcoin), **ETH**(Ether), **USD**; proporción: cada activo va a tener el 33.3% del valor del portafolio(cartera de inversiones), y la estrategia de rebalance: cuando cambie un 2%.

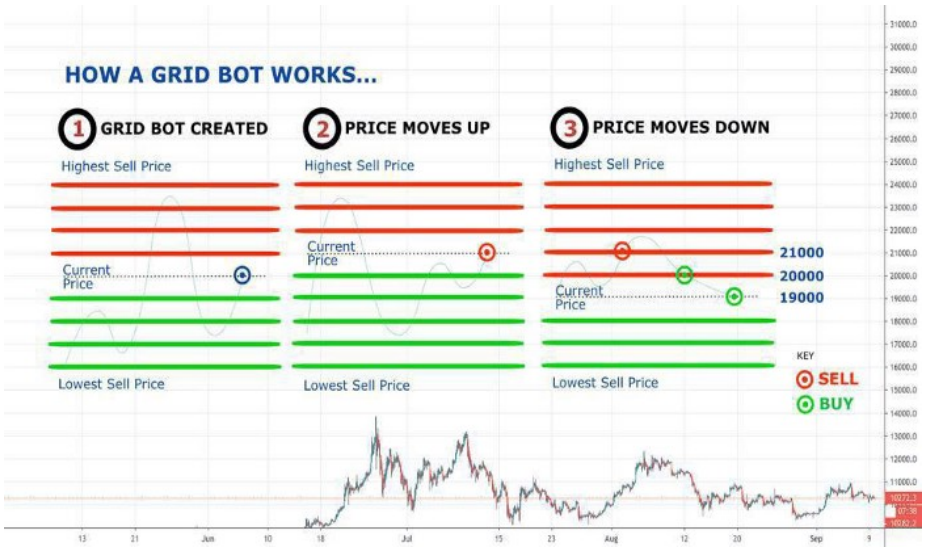

Figure 1: Grid Trading Bot explicación

Funcionamiento: Si el BTC llegara ocupar en algún momento el 36% del valor total de la cartera de inversión, superaría el 2% de margen(36-33.3=2.7% > 2), por lo tanto, se vendería BTC con tal de volver a balancear la cartera, en el caso que la proporción actual sea menor que la deseada, se compraría BTC. Así pasaría de igual forma con el resto de los activos.

Manteniendo la premisa de vender caro y comprar barato de forma periódica y constante.

Smart Trading Bot

Este bot funciona con un par de activos al igual que el Grid Bot, utilizando el 2do activo para comprar y vender el 1ero, si solo se le proporciona un activo, se asume que el 2do es el USD. El funcionamiento del bot se basa en la premisa fundamental del mercado, comprar barato y vender caro, para lograr este objetivo se apoya en varios indicadores de trading como la media móvil(MA), media móvil exponencial(EMA), MACD, índice de fuerza relativa(RSI), entre otros, además del precio anterior de los activos y las compras hechas anteriormente.

La estrategia del bot es la siguiente: si la media móvil de 20 periodos está por encima del precio actual consideramos que el mercado está en un momento bajista, si a su vez el índice de fuerza relativa está por debajo de 30, es que el mercado está en un buen momento para comprar; y si la diferencia de precio con respecto a la última compra abierta es de -10% o menos, es decir, el precio ha disminuido al menos un 10% desde la última compra realizada, si todo esto se cumple, se compra. El bot dedica el 10% de la inversión en cada compra, por

lo tanto, puede llegar a tener 10 posiciones abiertas.

Por el contrario, cuando la media móvil de 20 periodos está por debajo del precio actual, se considera que estamos en un movimiento alcista a mediano plazo, si esto viene aparejado de que el RSI está por encima de 70, pues entramos en una buena zona de venta. Solo se tiene que compararse que el precio actual este al menos 5% por encima de la última compra que no se haya vendido todavía, el proceso se repite hasta que no queden órdenes de compra abierta o no se cumplan las condiciones para vender.

Este proceso se repite cada vez que el precio cambia.

Compilador

Gramática

En la carpeta grammar tenemos un fichero de python con el mismo nombre en el cual hay un conjunto de clases que se refieren a elementos de las gramáticas para los lenguajes. Nos referimos a clases como **NonTerminal**, **Terminal**, **Production**, **Grammar** entre otras.

También a estas clases hemos implementado un conjunto de funciones básicas para que trabajar con ellas sea más fácil y legible. A continuación mostraremos un fragmento de código de la clase Terminal:

```
class Terminal(Symbol):

    def __init__(self, name, grammar):
        super().__init__(name, grammar)

    @property
    def is_terminal(self):
        return True

    @property
    def is_nonTerminal(self):
        return False
```

Las clases **Production**, **NonTerminal**, **Terminal** tiene una propiedad **grammar** que hace referencia a la gramática a donde pertenecen. De igual manera la gramática tiene propiedades para acceder a sus terminales, no terminales y producciones. Debido a esto no aconsejamos no crear las instancias de estas tres últimas clases aisladas. La forma adoptada para ello es inicializar la gramática y luego con las funciones **add_terminals** y **add_non_terminals** añadirle los no terminales a dicha gramática. Luego una vez que tenemos los terminales y

no terminales, generamos las producciones utilizando el operador %= que lo redefinimos para los no terminales.

Se creó la clase **Sentence** para modelar una secuencia de símbolos (no terminales y terminales) y otra **SentenceList** para modelar una lista **Sentence**, estas clases nos ayudan a ganar más comodidad y hacer nuestro código más legible.

Además, se crea la clase **AttributedProduction** que hereda de **Production** a la que se le incluye además una propiedad **attributes** para poder crear gramáticas atributadas.

En el proyecto tuvimos que crear dos gramáticas:

1- La primera fue una gramática sencilla, parceable con un parser ll, que la usamos solo para el lenguaje de las expresiones regulares.

2- la segunda gramática mucho más compleja que la anterior es la del lenguaje de los bots. Esta gramática la podemos encontrar en la carpeta compiler en un fichero llamado bots_grammar.py

Automata

Automaton class

En nuestro proyecto en la carpeta automaton tenemos básicamente dos clases Automaton y DFA, la primera hace referencia a la representación de los autómatas en general, es decir podemos crear lo mismo uno determinista que no determinista con esta clase, veamos un poco como funciona:

```
def __init__(self, states, start_state, finals_states,
transitions):
    self.states = states
    self.tags = [None] * states
    self.items = [[] for _ in range(states)]
    self.start_state = start_state
    self.finals_states = finals_states
    self.transitions = {}
    self.universe = set()
    for state, c, ends_array in transitions:
        if(c != ''):
            self.universe.add(c)
        if(self.transitions.__contains__(state) ==
False):
            self.transitions[state] = {}
        if(self.transitions[state].__contains__(c) ==
False):
            self.transitions[state][c] = []
        self.transitions[state][c] += ends_array
```

Nuestra función constructor recibe:

• **states**: un entero que representa la cantidad de estados/nodos que tendrá nuestro autómata.

• **start_state**: un entero que representa el estado inicial del cual se parte en el autómata a la hora de reconocer una cadena.

• **finals_states**: un arreglo de enteros que del cual se cumple que si x pertenece a este, entonces x es un estado terminal del autómata.

• **transitions**: un array de tuplas (**state**, **ch**, **state_array**), lo cual significa que partiendo del estado state hay una arista con valor ch dirigida hacia cualquier estado perteneciente al **state_array**

Dentro se definen otros atributos que más adelante veremos su utilidad.

DFA class

Luego tenemos nuestra clase **DFA** que hace ilusión a un autómata determinista la cual hereda de **Automaton**, tiene los mismos atributos que los explicados anteriormente solo con la condición que desde cualquier estado existirá a lo más 1 transición a otro estado con el símbolo **ch**.

En la carpeta automaton hay un archivo llamado **utils.py** el cual contiene las operaciones que definimos.

Unión de autómatas: básicamente cuando unimos dos autómatas **aut_a** y **aut_b** este debe reconocer lo mismo las cadenas de uno que las del otro, entonces el algoritmo constructivo que utilizamos se basa en crear un nodo inicial para el nuevo autómata y conectarlo al nodo inicial de **aut_a** y **aut_b** respectivamente usando épsilon transición, y ya luego creamos un nuevo nodo final y para cada nodo final de **aut_a** y **aut_b** ponemos una épsilon transición a este nuevo nodo final que creamos.

Concatenación de autómatas: cuando concatenamos dos autómatas **aut_a** y **aut_b** este debe reconocer las cadenas que tengan un patrón reconocible de **aut_a** al principio, y un patrón reconocible de **aut_b** en el resto de la cadena, nuestro nodo inicial será el inicial de **aut_a**, luego conectamos todos los nodos terminales de **aut_a** al nodo inicial de **aut_b** usando épsilon transiciones y para terminar conectamos los nodos terminales de **aut_b** a un nuevo nodo terminal usando también épsilon transiciones.

Clausura de autómatas: la clausura de un autómata **aut** no es más que un autómtata que reconoce los patrones de **aut** ya sea 0 o infinitas veces, por tanto, para hallar esta básicamente creamos un nodo inicial y uno terminal, ponemos una épsilon transición desde el inicial al terminal para permitir que se reconozca la cadena vacía, luego ponemos una transición del nodo inicial al nodo inicial de **aut** y desde cualquier nodo terminal de **aut** ponemos una épsilon transición al nodo inicial de **aut** y otra al nuevo nodo terminal.

Convertir un autómata no determinista a determinista: la transformación de un autómata no determinista a determinista se basa en simular las transiciones entre los posibles estados en los que puede estar a la vez el no determinista durante el reconocimiento de cualquier patrón. Por lo que los nuevos estados en el autómata determinista serán supernodos disjuntos conformados por conjuntos de nodos del autómata original. Básicamente, lo que hacemos es buscar la épsilon clausura del estado inicial del autómata, y esto lo marcamos como un supernodo, luego para cada supernodo sacado de la pila miramos por cada símbolo del universo de nuestro autómata a cuantos estados puede llegar utilizando este símbolo y épsilon transiciones y a partir del nuevo conjunto de nodos creamos un supernodo que metemos en la pila y ponemos una transición con el símbolo que analizamos.

Expresiones Regulares

Para poder identificar los tokens de nuestro lenguaje, creamos nuestro propio lenguaje que reconoce expresiones regulares que siguen una estructura definida por nosotros. Donde los símbolos son:

- "|" funciona como divisor y significa un "o" de la expresión regular que se encuentra a la izquierda y la que se encuentra a la derecha.
- "{ }" funciona como agrupador de una expresión regular.
- "ϵ" significa la cadena vacía.

• “symbol” hace ilusión a cualquier carácter no definido en los anteriores, es decir symbol ! = “|”, “{”, “}”, “ϵ”

Luego podemos definir nuestra clase Regex:

```
def __init__(self, regex, skip_spaces = False):
    self.regex = regex
    self.automaton = self.build_automaton(regex,
skip_spaces)
```

La clase recibe una expresión regular con la estructura de nuestro lenguaje regex, y con el método **build_automaton** pasamos a construir el autómata equivalente a esta expresión regular. Podemos ver el argumento **skip_spaces** esto como tal es para que si nuestra expresión regular tiene caracteres espacios no tenerlos en cuenta.

```
def build_automaton(self, regex, skip_spaces):
    G = build_grammar()
    tokens = self.regex_tokenizer(regex, G, skip_spaces)
    parser = non_recursive_descending_parser_fixed(G)
    left_parse = parser(tokens)
    ast = evaluate_left_parse(left_parse, tokens)
    automaton = ast.evaluate()
    dfa = nfa_to_dfa(automaton)
    return dfa
```

Lo primero es crear nuestra gramática, definida en el método **build_grammar** la cual formalmente expone los nodos terminales y no terminales, y las producciones de nuestro lenguaje de expresiones regulares:

```
def build_grammar():
    G = Grammar()

    E = G.add_non_terminal('E', True)
    T, F, A, X, Y, Z = G.add_non_terminals('T F A X Y
Z')
    pipe, star, opar, cpar, symbol, epsilon =
G.add_terminals('| ^ [ ] symbol ε')

    E %= T + X, lambda h, s: s[2], None, lambda h,s:
s[1]

    T %= F + Y,lambda h, s: s[2], None, lambda h, s:
s[1]

    X %= pipe + T + X, lambda h,s: s[3] , None, None,
lambda h, s: UnionNode(h[0], s[2])
    X %= G.epsilon, lambda h, s: h[0]

    F %= A + Z, lambda h, s: s[2], None, lambda h, s:
s[1]

    Y %= F + Y, lambda h, s: s[2], None, lambda h, s:
ConcatNode(h[0], s[1])
    Y %= G.epsilon, lambda h, s: h[0]

    A %= symbol, lambda h, s: SymbolNode(s[1]), None
    A %= opar + E + cpar, lambda h, s: s[2], None, None,
None
    A %= epsilon, lambda h, s: EpsilonNode(s[1]), None

    Z %= star + Z, lambda h, s: s[2], None, lambda h, s:
ClosureNode(h[0])
    Z %= G.epsilon, lambda h, s: h[0]
```

Entonces luego llamamos al método **regex_tokenizer**:

```
def regex_tokenizer(self, text, G, skip_spaces):
    tokens = []
    regex_tokens = {symbol: Token(symbol, G[symbol]) for
symbol in ['|', '^', '[', ']', 'ε']}
    for c in text:
        if(c == " " and skip_spaces):
            continue
        if(regex_tokens.__contains__(c)):
            tokens.append(regex_tokens[c])
        else:
            tokens.append(Token(c, G['symbol']))

    tokens.append(Token('$', G.eof))
    return tokens
```

A este le pasamos nuestra expresión regular y nos devuelve un arreglo de **Token** donde fácilmente tenemos identificado a que tipo de símbolo hace referencia en nuestro lenguaje.

Luego usamos el parser ll para obtener el ast equivalente a nuestra expresión.

```
left_parse = parser(tokens)
ast = evaluate_left_parse(left_parse, tokens)
```

Luego que tenemos, el **ast** el cual está conformado por nodos que realizan las operaciones de union, clausura y concatenación en autómatas, podemos llamar **ast.evaluate()** y esto hará un recorrido recursivo por estos nodos y nos retornará un autómata no determinista, que haciendo uso del método **nfa_to_dfa** lo convertimos a determinista. De esta forma para cualquier expresión regular comprendida en nuestro lenguaje de expresiones regulares obtenemos un autómata equivalente.

Lexer

Nuestra clase Lexer recibe un arreglo de tuplas (type, regex) donde type es el nombre que le ponemos a los patrones que se reconocen en la expresión regular regex.

```
# regex array contains tuples (type, regex)
def __init__(self, regex_array, eof):
    self.eof = eof
    self.regexs_aut = self._build_regexs(regex_array)
    self.automaton = self._build_automaton()
```

Usando el método _build_regexs pasamos a construir los autómatas equivalentes a cada expresión regular de este array.

```
def _build_regexs(self, regex_array):
    regexs_aut = []
    for i in range(len(regex_array)):
        token_type, regex = regex_array[i]
        # this is an dfa
        regex_aut = Regex(regex).automaton
        for final_state in regex_aut.finals_states:
            regex_aut.put_tag(final_state, (i,
token_type))
        regexs_aut.append(regex_aut)
    return regexs_aut
```

Entonces a cada nodo final del autómata le ponemos un **tag**, que básicamente es un par (**i, token_type**), esto es para que si dos patrones diferentes terminan en el mismo nodo final el de menor valor i nos dice que tiene más prioridad ante el otro patrón, dentro de automaton tenemos una función **put_tag** que le pasas un **tag** y compara si el tag que tiene el nodo actualmente tiene mayor prioridad, en caso contrario actualiza el tag del nodo.

Luego de que tenemos un arreglo con los autómatas equivalentes a cada expresión regular pasamos a llamar al método **_build_automaton** que ejecuta la unión de estos, con el objetivo de tener un solo autómata que me reconozca cualquiera de los patrones reconocibles por las expresiones regulares.

```
def _build_automaton(self):
    aut_result = self.regexs_aut[0]
    aut_count = len(self.regexs_aut)
    for i in range(1, aut_count):
        aut = self.regexs_aut[i]
        aut_result = union_automatas(aut_result, aut)
    return nfa_to_dfa(aut_result)
```

Luego entonces, definimos **___call___** para nuestra clase Lexer el cual recibe un texto y retorna un arreglo de tokens, con el patrón reconocido y a que tipo hace ilusión.

Básicamente, es dame el texto voy a caminar por mi autómata anotando el último nodo terminal que visito, hasta que llega el momento que ya leí toda la cadena o que llegué a un nodo del cual no puedo moverme con un símbolo de la cadena. Por tanto, pueden pasar dos cosas:

• Nunca llegué a un nodo terminal (el patrón no se puede obtener con la gramática

definida)

• Llegué a un nodo terminal, cojo el último al cual llegué, guardo todo el patrón que se reconoció y el tipo de patrón lo saco del tag del nodo terminal. Entonces voy a repetir todo el proceso pero solo con la cadena restante(resultante de eliminar el prefijo reconocido).

Parsers

Parser ll

Se implementó un parser ll para el lenguaje de las expresiones regulares que nos devuelve un pequeño ast (ya que solo tiene unos pocos nodos para evaluar reconocer las expresiones regulares: concat, union,etc) que luego es utilizado en el lexer para construir los autómatas que reconocen cada una de las expresiones regulares que va a tener nuestro lenguaje.

Para poder parsear el lenguaje de las expresiones regulares con el parser ll, las producciones no deben tener prefijos comunes y recursividad izquierda, como la gramática para las expresiones regulares era muy pequeña y sencilla lograr esto no fue muy difícil.

Además, para poder usar este parser implementamos los algoritmos para calcular los conjuntos *firsts* y *follows* de la gramática. Para ello nos creamos una clase **ContainerSet** que tiene una propiedad set que es el conjunto de valores y otra contains_epsilon que es para si épsilon pertenece o no al conjunto.

La idea que seguimos para computar estos conjuntos fue ir iterando por las producciones aplicando las reglas de pertenencia de un elemento a estos conjuntos y a medida que se fueran cumpliendo actualizábamos los conjuntos y cuando llegaba el momento en que no hubo cambio terminar de iterar.

A continuación mostraremos un fragmento del código de computar los **follows**:

```python
def compute_local_first(firsts, alpha):

    first_alpha = ContainerSet()

    try:
        alpha_is_epsilon = alpha.is_epsilon
    except:
        alpha_is_epsilon = False

    # Rule 2
    if alpha_is_epsilon:
        first_alpha.set_epsilon()

    # Rules 1, 3, 4, 5
    else:
        for symbol in alpha:
            first_alpha.update(firsts[symbol])
            if not firsts[symbol].contains_epsilon:
                break
        else:
            first_alpha.set_epsilon()

    return first_alpha
```

También implementamos dos métodos: el primero **non_recursive_descending_parser** el cual devuelve una función **parser** que recibe una lista de tokens y devuelve la secuencia de producciones a aplicar para parsearla; el segundo **evaluate_left_parse** es el encargado de evaluar los atributos de las producciones de la gramática para contruir el ast.

Parser shift reduce

Dentro de la carpeta de parser se creó una clase **ShiftReduceParser** que tiene un método **build_parsing_table** que no está implementado, pues será implementado más adelante por los pareser lr(1) y slr(1). Además, tiene la función **___call___** que se le pasa una secuencia de tokens y devuelve las producciones a aplicar para parsear dicha secuencia. Además, esta función tiene un parámetro **get_operations** que cuando se lo pasamos con valor **True** nos devuelve además las secuencias de operaciones SHIFT y REDUCE que hay que hacer.

En el mismo fichero donde está esta clase también un método **evaluate_reverse_parse** en donde le pasamos la secuencia de producciones y operaciones para parsear una lista de tokens y, la lista de tokens en sí; y se encarga de aplicar las evaluaciones de los atributos de la gramática correspondiente.

La tabla **ACTION** y **GOTO** se llena en el método **build_parsing_table**

que dejamos en blanco

Parser slr(1)

Se creó un parser slr(1) que hereda del parser anterior y sobreescribe la función **build_parsing_table**.

Pero antes, tuvimos que crear una clase **Item** para representar los ItemsLR0 y ItemsLR1 los cuales van a tener asociados una producción, un entero *position* que representa la posición del punto (.) y un lookaheads que será el lookahead de los ItemsLR1. A continuación mostraremos un fragmento de código de esta clase

```
# Return True if the entire production was viewed
@property
def CanReduce(self):
    return len(self.production.right) == self.position

@property
def NextSymbol(self):

    if self.position < len(self.production.right):
        return self.production.right[self.position]

    return None
```

Se creó un método **build_LR0_automaton** para construir el autómata con los ItemsLR0 con el que trabajará el parser slr1 para construir la tabla **ACTION** y **GOTO**, para ello se obtiene primero el autómata finito no determinista empezando por el ítem inicial. Luego una vez que terminemos de construir este autómata transformamos a determinista con el método **nfa_to_dfa**

Luego para llenar la tabla se hace de la misma manera que se explica en conferencia. Si en algún momento se va a guardar en la tabla un valor y en la posición correspondiente ya hay un valor que no corresponde con el que queremos guardar entonces hay un conflicto y la gramática no puede ser parseable con un parser slr(1).

Parser LR(1)

Implementamos también un parser LR(1) que nuevamente sobreescribe la función **build_parsing_table**

También se usó la clase Item para modelar los ItemLR1 esta vez haciendo uso del parámetro lookaheads.

El método **build_LR1_automaton** construye el autómata finito determinista que luego nos ayudará a construir la tabla. Esta vez se construyó directo a de-

terminista como se explica en la conferencia. Para ello tuvimos que implementar un conjunto de funciones auxiliares que nos ayuden en esta tarea. En el siguiente fragmento de código mostraremos la función closure_lr1 que utilizamos para modelar la función clausura planteada en la conferencia:

```python
def closure_lr1(items, firsts):
    closure = ContainerSet(*items)

    changed = True
    while changed:
        changed = False

        new_items = ContainerSet()
        for item in closure:
            new_items.extend(expand(item, firsts))

        changed = closure.update(new_items)

    return compress(closure)
```

Para llenar la tabla se hace casi igual que en el caso anterior, esta vez considerando solo el lookahead de los ítems pertinentes en lugar de todo el **follow**. Igualmente, al registrar un valor chequeamos que en esa posición no halla un valor previo diferente al que queremos registrar, en ese caso hay conflicto.

En el momento en que implementamos el parser slr(1) lo siguiente que hicimos fue probar si la gramática nuestra era parseable con este, pero obtuvimos que había conflicto, y luego de esto fue que implementamos el parser lr(1). No obstante resultó que el conflicto se debía a errores en la gramática que posteriormente fueron corregidos. Al final con ambos parsers slr(1) y lr(1) decidimos usar el lr(1), ya que aunque el autómata de este parser tiene muchos más estados también es más poderoso que slr(1) para parsear gramáticas más complejas, y la nuestra puede seguir creciendo ;).

Sintaxis y Semántica de nuestro lenguaje

Sintaxis

El lenguaje nuestro cuenta con los tipos int, bool, float, string y date; para los números enteros, las expresiones booleanas, los números flotantes, string y fechas (el formato de estas es yyyy-mm-dd). La manera de instanciarlos es la siguiente:

```
int x = 5;
```

De manera análoga para el resto. También en lugar del número solamente se puede poner una expresión aritmética:

```
int x = 5*6 + 2;
```

También contiene una serie de operaciones para binarias y unarias: operaciones aritméticas (suma, resta, ...) y operaciones de comparación (mayor, igual, negación, ...)

Para imprimir en la consola se utiliza la funcion *print*, de la siguiente manera:

```
print x;
print "Hola Mundo";
```

También contaremos con un conjunto de objetos específicos de nuestro lenguaje:

• Los bots: que pueden ser de varios tipos (smart bot, grid bot, rebalance bot) y con varias configuraciones. Mostraremos con un ejemplo como luce la sintaxis para crear los bots:

```
grid_bot x = grid_bot(p1, ..., pn);
```

-El grid_bot recibe 7 parámetros que son (en ese orden): stop_loss take_profit, investment, grids, limit_low, limit_high y assets_array.

-El rebalance bot recibe 6 parámetros (los dos últimos son opcionales): stop_loss, take_profit, investment, assets_array.

-El smart bot recibe 4 parámetros: stop_loss, take_profit, investment, assets_array.

• Los assets: son objetos que representan activos específicos. Mostraremos con dos ejemplos la sintaxis para crear activos:

```
asset x = CreateAsset("BTC"); # para crear un solo asset

array assets = [CreateAsset("A1"), CreateAsset("A2"),...]; # para crear un array de asset
```

• Los portfolios:

```
array x = PortfolioSDMin(p1, p2)
array y = PortfolioMSR(p1, p2)
```

-El portfolio recibe 2 parámetros: assets array y date.

También tendremos un conjunto de funciones propias de la lógica de los bots:

- CreateAsset: Recibe el string con el nombre del activo y se le asocia a una variable
- PrintInfo: Imprime la información del bot, ya sea los activos con los que opera, y alguno de sus atributos.
- Run: Se usa para hacer un llamado al bot a que inicie la simulación.
- PrintHistory: Para imprimir el historial de operaciones del bot.

Semántica

A continuación definiremos un conjunto de reglas semánticas que se deben cumplir en nuestro lenguaje:

1- Al crear un objeto de un tipo el valor que se le asigne tiene que ser del tipo que le corresponde (un int no puede tener un valor flotante por ejemplo).

2- Al crear instancias de los objetos específicos de nuestro lenguaje, estos deben tener la cantidad de parámetros correcta.

3- Al crear instancias de los objetos específicos de nuestro lenguaje, cada uno de los parámetros que se le pasan tienen que ser del tipo que le corresponde en dicha posición.

4- No pueden existir dos variables con el mismo identificador.

5- No se puede asignar un valor a una variable que no ha sido instanciada anteriormente en el código.

6- Al llamar a las funciones específicas de nuestro lenguaje los parámetros que se le pasan deben corresponder con el tipo de parámetros que espera la función.

Ast

En la carpeta ast hay un fichero .py con el mismo nombre donde definimos el conjunto de nodos del ast.

Además, en la misma carpeta hay otros ficheros que cada uno implementa un recorrido por el ast para un propósito distinto (imprimirlo, chequear reglas semánticas, transpilar a código de python).

Estos recorridos fueron implementados con el patrón visitor, de ese modo desacoplamos de los nodos de ast todas las posibles funciones que debemos ejecutar en cada uno ellos.

Conclusiones

Las criptomonedas son conocidas por ser muy volátiles, su precio fluctúa dramáticamente en cuestión de minutos o incluso segundos en ocasiones. El mercado está cambiando en cada momento de las 24 horas del día, incluso los fines de semana, limitando así la efectividad de un trader humano en ocasiones. Para ayudar al trader, aparecen los bots de trading, los cuales con unos pocos parámetros, pueden estar operando en los mercados ininterrumpidamente durante días, incluso meses.

La configuración de un bot va a influir directa y decisivamente en su rendimiento y su correcto funcionamiento. Una incorrecta o pobremente elaborada estrategia para un bot, puede hacer que se pueda perder buena parte o todo de la inversión empleada. De ahí que sea importante estudiar las diferentes estrategias a utilizar antes de que un bot empiece a operar en un mercado real.

Para aliviar o mitigar este problema, se ha creado este programa. En este se puede configurar cualquier cantidad de bots, probándolo en momentos anteriores del mercado y valorando su rendimiento y correcta operativa, sin ningún riesgo, pues solo se estaría probando la estrategia, no se estaría operando en los mercados financieros realmente. Razón por lo que pudiera ser muy útil, para ayudar a los traders a predecir cuáles pueden ser las mejores estrategias a seguir para crear bots y que estos sean lo más rentable posible.