1. Descripción

Traders DSL es un proyecto basado en el diseño de un lenguaje de dominio específico (DSL por sus siglas en inglés) que tiene como objetivo inicializar y ejecutar múltiples entornos con presencia de agentes y objetos de intercambio entre los mismos. Cada entorno es representado por un mundo con formato de una grilla rectangular, el cual será inicializado con una cantidad de filas y columnas de casillas donde podrán estar colocados estos agentes negociadores. Los entornos también tendrán funcionalidades que permitirán añadir agentes y objetos en ciertas posiciones.

Con el lenguaje será posible crear agentes negociadores, los cuales tendrán definidos una serie de objetos a vender y el precio al cual venderlos. Además, con el DSL propuesto se podrán definir comportamientos de estos en un entornos. Dichos comportamientos son definidos de forma independiente y permiten estar encapsulados e identificados correctamente de tal forma que puedan ser reutilizados para la lógica de comportamiento de otros agentes. Los comportamientos de los agentes incluyen operaciones para moverse, detectar la presencia de objetos en su posición en la grilla, saber quiénes son los agentes cercanos con los cuáles es posible iniciar un proceso de negociación y por su puesto la capacidad de negociar o no con un agente dado.

El lenguaje propuesto será Turing-Completo, es decir, podrá ser usado para resolver cualquier problema tratable en lenguajes de propósito general, lo que en este caso utilizando una sintaxis mucho más expresiva y acotada. Será posible realizar operaciones de declaraciones de variables, tipado de variables (int, float, string), expresiones de operaciones aritméticas y booleanas con estas, declaración de funciones (en este caso serían las funciones de comportamiento de los agentes) que pueden tener un comportamiento recursivo, control del flujo del código a partir de condicionales e implementación de ciclos.

1.1. Ejemplo de usos del DSL:

Para declara entornos bastaría en principio con la siguiente sintaxis:

```
environment e {
        rows: 10;
        columns: 10;
        number_iterations: 10;
        log: true;
}
```

Con el código anterior se logra declarar un entorno con dimensiones de 10x10, donde se realicen un total de 10 iteraciones y se muestren resultados de salida.

Para declarar un agente con un comportamiento y un conjunto de ofertas iniciales se podría realizar lo siguiente:

```
item apple {
        description: "Apple";
        price: 10;
}

behave b1 {
        move up;
        move left;

        let a = 20;
        repeat when me.items[0].price < a{
                say("hi");
                me.items[0].price++;
        }

        in case a %2 == 0 {
                stop;
        }
}

agent a1 {
        behavior: b1
        capacity: 10 // number of items that agent can store
}

a1.add_item(apple);
```

En el ejemplo anterior se muestra como crear un agente que contenga un objeto apple en inventario además de tener un comportamiento que consisten en que en cada iteración del entorno, este se moverá una casilla arriba, luego una casilla a la izquierda, luego declara una variable a, la cual utilizará en un bucle mientras que el precio al que está vendiendo su primer item sea menor que este. Al ejecutar este ejemplo, el agente debería decir "hi"diez veces, además de aumentar el precio al que vende su item. Finalmente, el comportamiento del agente termina ya que se cumple que la variable a guarda un número par y luego se ejecuta la funcionalidad stop, que detiene el funcionamiento interno del agente, y es equivalente a un return en un método de un lenguaje de propósito general.

Los comportamientos de los agentes solo tiene acceso a los items del mismo y los campos descripction y price de estos. Es posible hacer recursividad con estas lógicas de los agentes, lo que en dicho caso cuando se llama recursivamente se pasa como contecto el estado del agente y se devuelve la lista de items con el estado en que quedó esta después de llamar a dicho método. Para llamar

recursivo solo bastaría hacer call.

Veamos un ejemplo de cómo calcular Fibonacci utilizando un programa de Traders DSL. En este caso calcularemos el quinto valor de la sucesión de Fibonacci:

```
item n {
        description: "N";
        price: 5;
}

behave b1 {
        in case me.items[0].price < 2 {
                stop;
        }

        me.items[0].price -= 1;
        let fib1 = call()[0].price;

        me.items[0].price -= 1;
        let fib2 = call()[0].price;

        me.items[0].price = fib1 + fib2;
}

agent a1 {
        behavior: b1
        capacity: 10 // number of items that agent can store
}

a1.add_item(n);

environment e {
        rows: 10;
        columns: 10;
        number_iterations: 1;
        log: true;
}

e.add_agent(a1);

e.run();

print(e.agents[0].items[0].price);
```

2. Gramática

program → declarationList

Declarations:

declarationList → declaration declarationList | ϵ

declaration → envDecl | agentDecl | behaveDecl | varDecl | fieldAssign | envFunc

envDecl → "*env*" id "{" envBody "}"

agentDecl → "*agent*" id "{" agentBody "}"

behaveDecl → "*behave*" id "{" behaveBody "}"

varDecl → type id " = " expression ";"

fieldAssign → id "." id " = " expression ";"

envFunc → id "." "*reset*" ";" | id "." "*run*" expression ";"

Bodies:

envBody → varDeclList

agentBody → varDeclList

behaveBody → statementList

varDeclList → varDecl varDeclList | ϵ

statementList → statement statementList | ϵ

Statements:

statement → exprStmt | varDecl | repeatStmt | incaseStmt | primFuncStmt

exprStmt → expression ";"

repeatStmt → "*repeat*" "*when*" expression "{" statementList "}"

incaseStmt → "*in*" "*case*" expression "{" statementList "}"*inothercaseStmt*

inothercaseStmt → "*in*" "*other*" "*case*" expression "{" statementList "}" inothercaseStmt | "*otherwise*" "{" statementList "}" | ϵ

primFuncStmt → "*print*" expression ";" | "*move*" expression expression ";" | "*trade*" expression expression expression ";" | "*find*" | "*random*" expression expression

Expressions:

expression → logicExpr

logicExpr → logicAnd | logicAnd "*or*" logicExpr

logicAnd → equality | equality "*and*" logicAnd

equality → comparison | comparison equalityTail

equalityTail → "! = " comparison equalityTail | " == " comparison equalityTail | ϵ

comparison → term | term comparisonTail

comparisonTail → " < " term comparisonTail | " <= " term comparisonTail | " >= " term comparisonTail | " > " term comparisonTail | ϵ

term → factor | factor termTail

termTail → " + " factor termTail | " − " factor termTail | ϵ

factor → unary | unary factorTail

factorTail → " ∗ " unary factorTail | "/" unary factorTail | ϵ

unary → call | "!" call | " − " call

call → primary | id dotTail

dotTail → "." idTail | ϵ

idTail → id dotTail | listFunc dotTail

listFunc → "*get*" expression | "*push*" expression | "*size*" | "*pop*" | "*reverse*"

primary → "*true*" | "*false*" | number | string | "(" expression ")"

Lexical Grammar

type $\rightarrow$ "*number*" | "*bool*" | "*string*" | "*list*"