HULK INTERPRETE

El presente proyecto es un intérprete básico del lenguaje de programación **HULK**(***Havana University Language for Kompilers***). Su funcionamiento está contenido en los siguientes grupos de clases:

1-*Environment*: Se encuentran las clases relacionadas al almacenamiento de información para su posterior consulta, principalmente funciones y variables.

2-*Expression*: Contiene las clases relacionadas con la construcción de los componentes del AST(Abstract Syntax Tree), como la clase Node (Nodo), elemento principal del árbol, contiene también la evaluación de cada uno de los tipos de expresión.

3-*Lexer*: Contiene las clases relacionadas al proceso de análisis léxico y ,por tanto, tokenización de la línea de código introducida así como el desplazamiento por la estructura resultante de este proceso.

4-*Parser*: Clase individual que se ocupa de todo el proceso de análisis sintáctico de la expresión valiéndose y alterando las clases mencionadas anteriormente.Es la encargada de construir el AST y comprobar la validez sintáctica de la línea introducida.

5-*AST Evaluator*: Clase individual que toma el AST construido previamente por el Parser y procede a su evaluación y de existir, es decir ser válida semánticamente la expresión (no contener errores de este tipo) , devuelve el resultado de dicho proceso.

6-*Program*: Clase principal que se ocupa de dirigir todo el proceso de interpretación e interacción con el usuario.

A continuación se procede a describir con mayor profundidad cada clase y método contenidos en estos conjuntos:

Environment

1-Error

Es una clase que contiene la estructura Error, usada para guardar e informar sobre los errores ocurridos en las diversas fases de procesamiento de la entrada. Posee como atributos el tipo de error(***TypeError***),los cuales pueden ser léxicos, sintácticos o semánticos; característica general

del error (***ErrorCode***) , esperado (***Expected***) e inválido (***Invalid***) son las más comunes; argumento del error (***Argument***) expresa más detalles sobre el error.

2-Scope

Estructura para almacenar las variables válidas en cada entorno. Cada scope consta de un diccionario que relaciona el nombre de la variable declarada y el valor que esta va tomando en cada entorno.

3-Function

Estructura para crear y almacenar posteriormente una función declarada por el usuario. Guarda el nombre de la función (Name), un diccionario que contiene la información sobre los correspondientes parámetros necesarios para su ejecución (Functions_Arguments) y su cuerpo (Code).

4-Context

Estructura que contiene la información global, es decir aquellos datos que deben ser accesibles en todo momento. Contiene tanto la lista de funciones creadas durante cada sección, existentes hasta que se cierre la sección del intérprete, como las funciones por defecto del programa.

Las funciones por defecto disponibles son las siguientes:

•sin(seno)

•cos(cos)

•sqrt(raíz cuadrada)

•exp(e elevado al argumento)

•PI(valor númerico de π)

• E(valor númerico de e)

•rand(número racional aleatorio entre 0 y 1)

• log(logaritmo de base y argumento indicados)

Todas ellas están distribuidas en varios diccionarios en dependencia de la cantidad de parámetros que requieren. La estructura de estos es ***<string,Func<...>>***.

Expression

1-Node

Estructura que va a representar cada uno de los componentes del árbol. Tiene como atributos el tipo de nodo(***NodeType***),el contenido del nodo(***NodeExpression***) y la lista de subnodos que derivan de él (***Branches***).Los posibles tipos de nodos vienen dados por un enum.

2-Expression

Clase abstracta derivada de Node que declara el método de evaluación que van a heredar y sobreescribir todos los tipos de expresiones. Este método tiene dos versiones, una para las expresiones binarias y una para la expresión ternaria, las cuales se diferencia en la cantidad de parámetros que reciben.

3-Ternary

Clase que hereda de Expression.La evaluación de las expresiones ternarias recibe tres objetos que representan la condición, la expresión si la condición es cierta y la expresión si es falsa, en dependencia de la veracidad de la condición evalúa y devuelve una expresión o la otra.

4-Binary

Clase intermediaria entre ***Expression*** y los tipos de expresión binarias que hereda de Expression y da herencia a las demás clases binarias.

Clases binarias

Las clases binarias existentes son:

•Concatenation(Textual)(@)

•Sum(Numeric)(+)

•Substraction(Numeric)(-)

•Multiplication(Numeric)(*)

•Division(Numeric)(/)

•Power(Numeric)(^)

•And(Boolean)(&)

•Or(Boolean)(|)

•Minor(Boolean)(<)

•Major(Boolean)(>)

•Equal_Minor(Boolean)(<=)

•Equal_Major(Boolean)(>=)

•Equal(Boolean)(==)

•Diferent(Boolean)(!=)

En cada una de estas clases se sobreescribe el método de evaluación para ejecutar y devolver el valor de la operación correspondiente.

Lexer

1-Token

Clase que representa cada token mediante su tipo(***Tipo***) y su valor(***Value***). Los posibles tipos están expresados en el enum ***Type***.

2-Tokenizer

Clase que ejecuta todo el proceso de tokenización de la entrada y de detención de errores léxicos.

2.1-IdentifyType

Método que recibe un string y la lista de errores existentes hasta el momento, procede a determinar el tipo de Token que sería,asignarle como valor el string recibido y agregar el Token final a la lista de Tokens del lexer.

Los tipos de Tokens que se identifican son:

•print

•sqrt

•exp

•rand

•sin

•cos

•log

- PI
- E
- sum(+)
- substraction(-)
- multiplication(*)
- division(/)
- power(^)
- keyword(let,in, function)
- conditional(if,else)
- equal(==)
- diferent(!=)
- symbol({,},{;},{:},{=>},{=})
- left_bracket { (}
- right_bracket {) }
- concatenate(@)
- minor(<)
- major(>)
- equal_minor(<=)
- equal_major(>=)
- Or(|)
- And(&)
- boolean(true,false)
- number
- text("...")

•identifier

•not_id(Token inválido)

2.2-Tokens

Método que recibe la entrada, y utilizando expresiones regulares la divide en un conjunto de posibles tokens(los patrones se declaran separados y se juntan a la hora de buscar las coincidencias, hay uno para los números (negativos incluidos),textos, comillas separadas, palabras claves y identificadores).

¡Debido al patrón de búsqueda de los números negativos a la hora de declarar una operación de resta se debe separar del número derecho, si no lo tomará como un número negativo y no contará el operador!.

Una vez se tiene el grupo de coincidencias, utilizando el método ***IdentifyType***, se analiza cada uno y se va conformando la lista de tokens que se devolverán al final.

2.3-Lexic_Errors

Método que devuelve el conjunto de errores que se hayan identificado durante la etapa de análisis léxico.

Por ejemplo la ausencia del (;) al final de la línea o un identificar inválido (3number).

3-TokenStream

Clase que contiene diversos métodos para acceder a los elementos de la lista de tokens, como MoveForward que avanza n posiciones indicadas o Position que devuelve el índice de la posición actual.

Parser

Clase que ejecuta todo el proceso de análisis sintáctico, elabora el AST y identifica los errores sintácticos presentes.

1-Parse

Método que indica el inicio del proceso de análisis y devuelve el nodo principal(raíz del AST) del que se derivan todos los demás.

2-Syntactic_Errors

Método que devuelve la lista de errores sintácticos que se hayan identificado, por ejemplo paréntesis incompletos o ausencia de palabras claves.

3-ParseExpression

Método que dependiendo del primer Token indica que tipo de expresión parsear:

if->condicional

print->función print

let->declaración de variable y evaluación en un contexto de la misma

function->declaración de una función

En cualquier otro caso se parsea una expresión genérica (es decir matemática, booleanas o concatenación de textos, así como números, true,false o textos individuales).

4-Print

Método que comprueba que una posible expresión print sea válida y construye el conjunto de nodos referentes a la expresión, en este caso el nodo principal de tipo Print y como subnodo su argumento.

5-IF_ELSE

Método que comprueba que una posible expresión condicional sea válida y construye el conjunto de nodos referentes a la expresión, en este caso el nodo principal de tipo Condicional y como subnodos la condición, la expresión if y la else.

6-Function

Método que comprueba que la declaración de una función sea válida. Como nodos tiene el nombre de la función,los argumentos y el cuerpo de la función.

7-Let_In

Igualmente comprueba la validez de la declaración de variables. Cómo nodos tiene la lista de variables que se declaren,sus argumentos y la expresión donde ellas existen.

8-Unit

Reconoce y crea el nodo si este es un número,true,false ,un texto, una función por defecto, función existente, variable declarada y otros casos. También si es una expresión entre paréntesis indica y recibe su parseo mediante nuevamente el método ***ParseExpression***.

9-ParsePower

Parsea una potencia que como parte izquierda llama a ***Unit*** y por la derecha parsea más potencias de existir y ser consecutivas sino parsea unidades. Devuelve un nodo Power, de no existir ninguna potencia devuelve la unidad izquierda.

10-ParseMul_O_Div

Igualmente trata de parsear una multiplicación o una división, como parte izquierda trata de parsear una potencia y llama a ***ParsePower*** y por la derecha trata de parsear las multiplicaciones y divisiones existentes. De no existir ninguna devuelve la parte izquierda solamente.

11-ParseSum_O_Sub

Sigue la misma idea de ***ParseMul_O_Div*** y ***ParsePower***.

12-ParseComparation

Parsea una comparación mediante las operaciones (<,>,<=,>=,!= y ==) y como los métodos anteriores devuelven valores numéricos parsea como sus partes izquierda y derecha a ***ParseSum_O_Sub***. No llama por la derecha a ***ParseComparation*** porque está durante la evaluación devolverá un booleano y no se pueden comparar un número y un booleano.

13-ParseOr_O_And

Parsea las operaciones (&,|) y sigue la misma idea de ***ParseSum_O_Sub***, en este caso por la parte derecha si se intentaría parsear otro ***ParseOr_O_And*** porque en este caso los dos valores siempre serían solo booleanos.

14-ParseOP

Método que inicia la cadena de llamadas que desciende hasta ***Unit***, intenta parsear una concatenación. En este interprete la concatenación solo está definida entre *textos* pero la validez de la parte derecha y izquierda se comprueban durante la evaluación, lo mismo se comprueba para cada una de las otras operaciones.

AST Evaluador

Clase que ejecuta el proceso de análisis semántico, evaluación del AST y identificación de errores semánticos

1-AST_Evaluator

Constructor de la clase que inicializa el contexto, el scope principal, la lista de errores y un AST vacío que posteriormente recibirá el árbol creado durante el parser.

2-Tree_Reader

Asigna el AST del parser a la propiedad AST del evaluador.

3-StartEvaluation

Comienza el proceso de evaluación de un nodo llamando al método ***GeneralEvaluation*** y devolviendo su valor.

4-Semanti_Errors

Método que devuelve la lista de errores semánticos identificados, como por ejemplo una suma de dos textos o parámetros incorrectos al llamar a una función.

5-GeneralEvaluation

Evalúa cada tipo de nodo, en caso de ser nodos simples como los números o textos simplemente se asegura de convertir su valor al formato correcto y devuelve dicho valor. En el caso de las operaciones comprueba que los valores de cada miembro sean válidos, es decir, en las operaciones matemáticas se asegura de que sean números, en las comparaciones que sean números para menor,mayor,menor-igual y mayor-igual, y ambos del mismo tipo para igual o diferente. Para las funciones por defecto comprueba que se hayan pasado los parámetros válidos y evalúa dicho parámetro antes de pasarlo al diccionario del contexto que contiene las instrucciones de dichas funciones y devolver el resultado. Para las funciones declaradas comprueba que la función exista en el contexto, la validez y cantidad de los parámetros interesados y ejecuta ,comprueba y devuelve su cuerpo. Si se trata de una función que se está declarando se crea mediante la clase Function una nueva función que se añade al contexto para su posterior uso. En el caso de las variables evalúa su valor y/o comprueba que en el contexto actual (scope) sea válido su uso.

Program

Clase que organiza y une todas las etapas de evaluación. Crea en primer lugar los objetos lexer y ast_evaluator para su uso para cada nueva línea ingresada. Luego se ingresa a un ciclo que solo

se rompe cuando se ingresa una línea vacía (" "), en este se llama a los métodos que inician cada proceso de análisis: *Tokens*(***Lexer***),*Parse*(***Parser***) y *StartEvaluation*(***AST_Evaluator***).

Siempre antes de proseguir a la siguiente etapa se comprueba que su lista de errores correspondiente esté vacía, en caso de que no lo esté se imprime cada uno de los errores presentes y se cancela el proceso de análisis de la línea actual.

De no existir ningún error(ser una línea válida) se impreme el resultado si no es una declaración de función si lo es se añade esta al contexto.