HULK

Glenda Rios Rodriguez

Octubre, 2023

Resumen

A continuación y de forma breve haremos un repaso por todas las clases creadas en nuestra librería HULK y explicaremos la lógica de todo nuestro intérprete.

1. LEXER

Para construir nuestro analizador léxico hemos creado 2 clases fundamentales

1.1. Class Token

Definimos los tipos de token que usará nuestro lenguaje y además cada token contará con un TokenType que será el tipo de token , un string Grupo y un string Valor. Estos dos últimos parámetros pueden tener importancia o no en dependencia del tipo de token con que estemos trabajando .Por ejemplo , un token de tipo string contará con el valor que el usuario le asigne en la entrada del código.

A continuación los tipos de tokens válidos que podemos encontar en nuestro código:

```
public enum TokenType
```

//Tipos de Variables
String,
Number,
True,
False,
Identificador,
//Operadores
Suma,
Resta,
Concatenar,
Multiplicacion,
Division,
Pow,
Modulo,
//Comparadores
IgualIgual,
NoIgual,
MayorIgual,

MenorIgual,
Mayor,
Menor,
//Asignacion
Igual,
Flecha,
//Operador Booleano
Negacion,
And,
Or,
//Palabras Reservadas
If,
Else,
function,
Let,
In,
//Constantes
PI,
EULER,
//Separadores
Coma,
PuntoYComa,
ParentesisAbierto,
ParentesisCerrado,
//Final de el codigo
Final,

1.2. Class Tokenizer

Después de insertada la entrada , esta pasará al constructor de la clase Tokenizer con el objetivo de ir recorriendo la cadena y creando nuestros tokens , los cuales serán guardados por orden en una lista List ¡Token¿Tokens.

Para hacer esta operacion hemos creado el método GetTokens(string input) que recibirá como entrada lo que el usuario haya insertado como línea de código y procederá a recorrer char por char con un ciclo y comprobando caso por caso qué tipo de token estamos analizando.

Al finalizar el ciclo guardaremos como último token de la lista uno de TokenType.Final para así cuando estemos trabajando con los tokens poder identificar cuando hemos llegado al final de la lista.

2. PARSER

Antes de explicar la lógica en que se basa nuestro analizador sintáctico es importante hablar de la Class Expresion , que es precisamente la que hemos utilizado para crear nuestro árbol de sintaxis.

2.1. Class Expresion

Esta clase como podrán observar es abstracta debido a que no se pueden construir objectos de este tipo específicamente pero dentro de ella hemos creado una jerarquía de clases ,donde la Class Expresion es la clase padre, por decirlo de esta manera.

2.1.1. Class ExprLiteral

Expresiones de tipo bool, number o string.

2.1.2. Class ExprUnaria

Son las expresiones de tipo !(algo) para realizar un no lógico o -(algo) para negar un número.

2.1.3. Class ExprBinaria

Expresiones de tipo expresión + operador + expresión, donde el operador puede ser tanto lógico como aritmético.

2.1.4. Class ExprVariable

Expresiones de tipo identificador , por ejemplo una variable x .

2.1.5. Class ExprAsignar

Son las expresiones donde a una una variable x le asignamos a través de un operador '=' un valor específico, que puede ser a su vez una expresión.

2.1.6. Class ExprIf

Una declaración de if cuenta con una expresion ifcondicion (este debe ser un valor booleano) , una expresion ifCuerpo que se llevará a cabo en caso de ser verdadera la ifCondicion y un elseCuerpo, para el caso en que esta sea falso.

2.1.7. Class ExprLetIn

Las expresiones de tipo let in estarán conformadas por un letCuerpo que será una lista de expresiones de asignación, y el InCuerpo que será una expresión de cualquier tipo .

2.1.8. Class Funcion

Una expresión función estará compuesta por un identificador que será el nombre de la funcion , los parámetros que se le pasarán y además el cuerpo de ésta que puede estar compuesto por cualquier tipo de expresión.

2.1.9. Class ExprLlamadaFuncion

Como bien sabemos una vez construida una Expresion de tipo Función podremos llamar a la función en diferentes parámetros con el objetivo de obtener el resultado de esa función para los argumentos dados , es por ello que hemos creado el tipo de Expresiones de Llamada Función , las cuales cuentan con un identificador que será el nombre de la funcion , una lista de expresiones que serán los argumentos y una Función que será aquella con el nombre del identificador.

2.2. Class Parser

En la clase Parser es donde crearemos todo nuestro árbol sintáctico y además comprobaremos que el orden de las tokens sea el correcto , por ejemplo sabemos que una declaración de if está estructurada de la siguiente forma:

TokenType.If - TokenType.ParentesisAbierto - Expresion IfCondicion - TokenType.ParentesisCerrado - Expresion IfCuerpo - Tokentype.Else - Expresion ElseCuerpo

Análogamente ocurre con las declaraciones de tipo Let-In las cuales están estructuradas:

Tokentype.Let - Lista de expresiones de asignación separadas por TokenType.Coma - TokenType.In - Expresion InCuerpo.

Y las funciones:

TokenType.funcion - TokenType.Identificador que será el nombre de la función - Tokentype.ParentesisAbierto - Lista de TokenType.Identificador separados por TokenType.Coma que serán los parámetros de la función - TokenType.ParentesisCerrado - TokenType.Flecha - Expresion funcionCuerpo

Y los llamados de funcion:

Tokentype.Identificador que es el nombre de la función a la que estamos llamando (asumamos que está definida) - TokenType.ParentesisAbierto - lista de expresiones que serán los argumentos de la funcion separados por TokenType.Coma - TokenType.ParentesisCerrado.

Recordar que al final de cada línea en HULK siempre hay un TokenType,PuntoYComa.

Ahora bien , utilizaremos para desarrollar nuestra lógica un método llamado `Análisis de descenso recursivo`. El descenso recursivo se considera un analizador de arriba hacia abajo porque comienza desde la regla gramatical superior o más externa (aquí Parse) y avanza hacia las subexpresiones anidadas antes de llegar finalmente a las hojas del árbol de sintaxis. Se llama " descenso recursivo " porque recorre la gramática. En un analizador de arriba hacia abajo, se llega primero a las expresiones de menor prioridad porque, a su vez, pueden contener subexpresiones de mayor precedencia.

El analizador consume una secuencia de entrada plana, pero está vez leemos tokens .Almacenamos la lista de tokens y utilizamos current para señalar el siguiente token que espera ansiosamente ser analizado.

La primera regla Parse() simplemente se expande a la expression() y esta se expande a la logical(), que a su ves lo hace con la equality() quedando más o menos en el siguiente orden:

2.2.1. método Parse()

Lo primero que comprobamos es que el primer token de la lista sea uno de Tokentype.funcion , de ser así estaríamos en precencia de una declaración de función que tendría como ya sabemos una estructura específica . De no ser un token de ese tipo entonces expandiríamos nuestro análisis a la expression() regla.

2.2.2. método expression()

Esta simplemete se expande a la logical().

2.2.3. método logical()

Primero se expande a la equality() regla y luego de obtener un resultado de su llamada comprueba si estamos en presencia de una ExprBinaria donde el operador es de tipo (&&) o (or).

2.2.4. método equality()

Primero se expande a la compararison() y luego de obtener un resultado después de realizada su llamada comprueba si estamos en precensia de una ExprBinaria donde el operador es (==) o (!=).

2.2.5. método comparison()

Primero se expande a la concat() y luego de obtener un resultado a su llamada comprueba si estamos en precencis de una ExprBinaria donde el operador es de tipo (mayor) o (mayor=) o (menor) o (menor=).

2.2.6. método concat()

Primero se expande a la Term() y luego de obtener un resultado a su llamada comprueba si estamos en precencis de una ExprBinaria donde el operador es de tipo (@).

2.2.7. método Term()

Primero se expande a la factor() y luego de obtener un resultado a su llamada comprueba si estamos en precencis de una ExprBinaria donde el operador es de tipo (+) o (-).

2.2.8. método factor()

Primero se expande a la pow() y luego de obtener un resultado a su llamada comprueba si estamos en precencis de una ExprBinaria donde el operador es de tipo (*) o (/).

2.2.9. método pow()

Primero se expande a la mod() y luego de obtener un resultado a su llamada comprueba si estamos en precencis de una ExprBinaria donde el operador es de tipo ($\hat{}$)

2.2.10. método mod()

Primero se expande a la unary() y luego de obtener un resultado a su llamada comprueba si estamos en precencis de una ExprBinaria donde el operador es de tipo (%).

2.2.11. método unary()

Comprueba si estamos en precencia de un TokenType.Resta o TokenType.Negacion , de ser cierto estaríamos en precencia de una ExprUnaria y llamaríamos luego a el método IFORLET() para seguir analizando los tokens de la lista.

2.2.12. método IFORLET()

Comprueba si estamos en precencia de un TokenType.If de ser así estaríamos en precencia de una declaracion de tipo if y entonces se procedería a realizar todo el análisis estructural que tienen este tipo de expresiones .De no ser así comprobaría si estamos en precencia de un TokenType.Let y análogamente como en el If procedería a hacer su respectivo análisis. Nuevamente de no ser así comprobaríamos si estamos en precencia de un TokenType.Identificador y si este coincide con el nombre de alguna función que ya tenemos declarada.Finalmente de no resultar nada de lo anterior procedería a llamar a el método primary().

2.2.13. método primary()

Comprueba si estamos en precencia de algún token que sea un string , bool (true o false) , number , ParentesisAbierto y constantes como PI o E. De no resultar nada de esto estaríamos en precencia de un error Sintáctico , el cuál lanzaríamos con un mensaje a nuestra consola.

Cada método para analizar una regla gramatical produce un árbol de sintaxis para esa regla y se lo devuelve al método que llama.

Para esto utilizaremos métodos auxiliares, los cuales son :

2.2.14. método peek()

Este método devuelve el token de la lista en la posición current es decir , el token actual

2.2.15. método previous()

devuelve el Token en la posición current-1 , es decir , el anterior.

2.2.16. método advance()

que devuelve token en la posición current -1 pero además, en caso de comprobar que no está al final de la línea aumenta en uno el valor del current

2.2.17. método check()

dado un TokenType te devuelve true si coincide con el tipo de token que estamos analizando

2.2.18. método IAE()

te comprueba si el token en que estamos es el último token de la lista

2.2.19. método match(List¡TokenType¿)

dada una lista de tokentype devuelve true si alguno de estos coincide con el que estamos analizando

2.2.20. método consume(TokenType token, string mensaje)

Este es utilizado para expresiones que tienen cierta composición fija , como habíamos puesto de ejemplo anteriormente en la declaración de if , podemos suponer que después de un TokenType.If tiene que venir , necesariamente un paréntesisabierto y como mensaje , pasamos el tipo de error en caso de que el token que debe ir no coincida con que realmente está.

2.3. Class Funciones

Esta clase ha sido creada específicamente para guardar en memoria cada una de las funciones que el usuario desee declarar para luego ser reutilizarlas a través de un ExprLlamadaFuncion.

En esta clase contamos con un Dictionary¡string , Funcion¿donde cuando llamemos a su constructor se guardará en el diccionario el nombre de la funcion con su Función correspondiente. Recordemos que en Hulk hay funciones globales como es el caso del print , sin , cos , log, sqrt , es decir, este tipo de funciones ya están definidas y es imposible redefinirlas . Explicaremos los métodos que utilizamos para hacer estas operaciones.

2.3.1. método AddFuncion

Este método es el que utilizamos para agregar una nueva función a nuestro diccionario, se le deben pasar como parámetros el nombre de una función y la Función que por referencia será null inicialmente (para el caso de sin , cos , etc) una vez llamado al método, se actualiza el valor Funcion de aquella con el nombre insertado.

2.3.2. método FuncionesEspeciales

Aquí solamente lo llamamos para guardar desde un inicio las funciones globales que habíamos mencionado anteriormente .

2.3.3. método ContainsFuncion

Este método se utiliza en el caso que el usuario quiera declarar una nueva función, si no está creada proseguirá a seguir analizando las otras partes de el programa , de ya estarlo , retornará un error pues la función no podría redefinirse.

2.3.4. método GetFuncion

Se le inserta como parámetro el nombre de una función ya creada y este te devuelve su Valor en caso de que tenga alguno.

3. Evaluador

Una vez hecho el análisis sintáctico solo nos queda evaluar nuestra expresión en los valores que el usuario haya insertado.

Para esto utilizamos específicamente un método que se encuentra en la clase Evaluador.

3.1. Class Evaluador

3.1.1. método GetValue

A este método se le pasan como parámetros inicialmente la expresión creada en el Parser y un diccionario que está inicialmente vacío. El objetivo de nuestro diccionario es que a veces tenemos alguna asignación de un valor a una variable y en este caso guardaríamos esta asignación en el diccionario para luego cambiar la variable por su respectivo valor . Inicialmente la expresión que entra en el método tiene alguna composición que la hace ser de un tipo específico, en dependencia de cual sea será evaluada esta expresión siempre comprobando que las operaciones sean válidas, como sumar string +number , que eso retornaría un error.

Finalmente imprimimos , en caso de que todo nuestro análisis estuviera correcto , un resultado en consola.

4. ERROR

4.1. Class ERROR

Al constructor de esta clase le pasamos como parámetros el tipo de error y algún mensaje específico sobre este.

4.1.1. Errores Léxicos

Estos errores son los que suelen aparecer en el analizador léxico que no son más que tokens inválidos que inserta el usuario

Ejemplo: let x=13a in x;

En este caso aparecería en consola el siguiente mensaje

LexicalError , 13a is kot a valid token.

4.1.2. Errores Sintácticos

Estos errores suelen aparecer en la segunda fase o Analizador Sintáctico que no son más que algún error en cuanto a la estructura de la expresión insertada por el usuario

Ejemplo : if(6==3 print(true) else print(false);

Se imprimiría en la consola

SyntaxError missing ')' after expresion in 5.

Donde 5 sería la posición del token.

4.1.3. Errores Semánticos

Estos suelen aparecer en el Evaluador o analizador semántico .

Ejemplo : let x=2 , y=rosaïn x+y;

En este caso aparecerá en consola:

SemanticError Operator '+' cannot be between 9 rosa

En el caso de los Errores Sintácticos y Semánticos, el programa se detendrá y no seguirá realizando su análisis.