Project H.U.L.K

Jossué Arteche Muñoz

1 de noviembre de 2023

Resumen

Segundo proyecto de programación de la carrera de Ciencias de la Computación, facultad de Matemática y computación, Universidad de la Habana.

1. Introducción

H.U.L.K. es un lenguaje de programación didáctico, creado por la facultad de matemática y computación de la Universidad de La Habana, diseñado para introducir conceptos de compilación a estudiantes de ciencias de la computación. Project_HULK es un interprete que ejecuta un subconjunto dicho lenguaje: la mayoría de expresiones de una línea. Este informe pretende dar una idea del funcionamiento general del interprete.

2. Manejando la cadena de entrada

Antes de ahondar en como interpretar semánticamente una entrada dada por el usuario, es necesario resolver un problema. Hay cadenas de entrada que aunque sean diferentes ejecutan una misma instrucción. Por ejemplo las cadenas *"2+3"*, *"2 + 3"*, *"2+ 3"* y *"2 + 3"* todas tienen el mismo significado, tanto para una persona como para H.U.L.K representan una adición entre 2 y 3. Es necesaria una forma de reducirlas todas solo a los caracteres o "tokens"necesarios para interpretar la entrada. A este proceso le se le llamara "análisis léxico". Los ejemplos anteriores claramente se reducen a los tokens "2", "+"y "3"en ese orden. Para resolver este problema se usa una expresión regular que coincide con cada uno de estos tokens y devuelve una colección de estos, la cual se convierte en un arreglo de string para una mayor comodidad. Los ejemplos anteriores quedan reducidos al arreglo {"2", "+", "3"} sobre el que es posible iterar fácilmente, lo cual será muy útil para el siguiente paso.

3. Interpretando entradas

Supóngase que se tiene la entrada *"2 + x * 2"*, es necesario deducir su significado semántico. Se definirá "expresión "a lo que expresa esa cadena. Por ejemplo, la cadena *"2 + x * 2"* expresa una suma entre el numero *2* y una multiplicación de una variable *x* con otro número *2*, sin embargo en concreto no es mas que caracteres concatenados. Para el intérprete una expresión es toda instancia de una clase HulkExpression que siempre tendrá un método virtual GetValue()

Las expresiones "suma"y "multiplicación"devuelven un resultado en dependencia del resultado de las expresiones que están sumando o multiplicando respectivamente. Una expresión que involucre operaciones puede ser interpretada como una estructura arbórea donde cada nodo es una expresión, que a su vez puede o no tener ramas hacia otras expresiones.

El problema radica en como es formado este árbol correctamente, porque, de hacerlo mal, es posible interpretar las expresiones en un orden incorrecto obteniéndose un resultado erróneo. Es necesario establecer un orden de importancia entre las expresiones a la hora de interpretar.

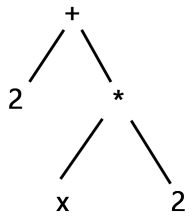

Figura 1: Expresión representada como un árbol

3.1. Operaciones

En sentido general, existen expresiones muy básicas (como la suma, la resta, o la conjunción) que se llamaran operaciones. Estas pueden tener una o dos expresiones como argumentos y tienen un nivel de precedencia, un orden operacional. Para el interprete una suma es un objeto que hace referencia a dos expresiones que son sus argumentos.

Nótese que para obtener el valor de una expresión es necesario llamar recursivamente al método GetValue() de sus argumentos, o sea, recorrer el arbol en profundidad, por tanto las expresiones que primero se evaluarán deben estar en la base del árbol. Se puede afirmar entoces que debemos "parsear"primero la ultima expresión que evaluaremos, es decir la de menor precedencia.

Por ejemplo, supóngase que se tienen solo cuatro operaciones: la suma, la resta, la multiplicación y la división. Primero se efectúan la multiplicación y división de izquierda a derecha y luego la suma y la resta de izquierda a derecha. Siguiendo la regla anterior el orden en que hay que construir expresiones sería suma y resta de derecha a izquierda y multiplicación y división de derecha a izquierda.

El intérprete funciona usando esa idea. Itera sobre el arreglo de tokens, con dos punteros que representan el limite izquierdo y el derecho, *start* y *end*, en el sentido contrario al de la precedencia de las expresiones de ese nivel. Si en la posición i se encuentra el token correspondiente a una operación de dicho nivel, crea una instancia de la clase que representa a esa operación y pasa como argumentos el resultado de parsear las expresiones desde *start* hasta *i-1* y desde *i+1* hasta *end*, en el caso de las operaciones binarias, o el resultado de parsear desde *i+1* hasta *end*, si es unaria la operación. En el caso de encontrarse un paréntesis abierto o cerrado se debe "brincar"hasta el siguiente paréntesis cerrado o abierto respectivamente.

Este seria el flujo del programa hasta ahora: primero recibe la entrada *"2 + x * 2"*, la convierte en un array de string (tokens) {"2", "+", "x", "*", "2"}. Empieza a buscar desde *0* a *4* de derecha a izquierda por un token de suma o de resta, al ser *tokens[1] == "+"* la expresión es una suma que toma como argumentos el resultado de repetir del mismo proceso desde *0* a *0* y el resultado de parsear desde *2* a *4*

Por supuesto H.U.L.K. tiene muchas más operaciones y niveles de precedencia que en el ejemplo.

3.2. Expresiones Principales

El árbol que representan la expresión queda casi conformado de esta forma, pero no todas las expresiones son operaciones. En el fondo de la estructura se encuentran expresiones indivisibles, "átomos", estas serán llamadas expresiones principales. En esta categoría se encuentran las variables y constantes (números, valores de verdad y cadenas) y los llamados a función. Estas se parsean en el ultimo nivel ya que son las que se evalúan primero.

Parsear las variables y constantes es sencillo pues están representadas por un solo token. Si el interprete identifica una variable definida, un número, verdadero o falso, o un string, a partir de este, crea una clase Variable que toma como argumento su valor.

Para parsear los llamados a funciones se toma como nombre *tokens[start]*, y se toman como argumentos las expresiones que se encuentren entre *start + 2* y *end - 1* separadas por tokens *","* organizados en una lista, verificando primero que *tokens[start + 1]* y *tokens[end]* sean paréntesis abiertos y cerrados respectivamente.

A las expresiones principales también pertenecen las expresiones que se encuentran entre paréntesis como por ejemplo {....“(”, “2”, “+”, “2”, “)”...}. El intérprete en primera instancia no encontrará ninguna operación, ya que cuando itere sobre el arreglo desde *start* hasta *end* (o al revés) “brincará”hacia el paréntesis que cierra o abre. Cuando no encuentre ninguna operación verificará si *tokens[start]* y *tokens[end]* son “(”y “)”respectivamente y parsea la expresión desde *start + 1* hasta *end - 1*.

4. Interpretando expresiones mas complejas

Hasta ahora se han discutido expresiones que se expresan en un árbol bastante sencillo. El intérprete es capaz de reconocer expresiones mas complejas como condicionales y expresiones let-in.

4.1. Expresión if-else

H.U.L.K. soporta expresiones condicionales de una linea, al estilo de los operadores ternarios de C#. Para parsear estas se procede de la siguiente forma. Si *tokens[start]* es “if”se busca hacia la derecha un token “else”(es obligatorio que este esté presente). Luego se parsea la condición que se debe cumplir, o sea, desde *tokens[start + 2]* hasta *tokens[i]* donde i es el indice del paréntesis cerrado que corresponde al paréntesis abierto *tokens[start + 1]* (de no estar presentes alguno de los dos se devuelve un error). A continuación se parsea desde *i+1* hasta la posición anterior del token “else”y por ultimo se parsea desde la posición siguiente a este último token hasta *end*. De no existir alguna de estas expresiones se devuelve un error. Para evaluarla es sencillo, si se cumple la condición o no se devuelve el valor de la expresión correspondiente.

4.2. Expresiones let - in

Las expresiones let-in son por ejemplo “let x = 2 in 3 + x”. El parseo de estas expresiones es análoga al de las expresiones if-else, solo que con los tokens “let”e “in”en vez de “if”y “else”respectivamente. Las expresiones let-in toman como argumentos una secuencia de declaraciones de variable, entre “let”e “in”que pueden ser utilizadas en la expresión luego del token “in”. El parseo de la expresión luego de “in”se realiza siguiendo el proceso explicado para parsear una expresión cualquiera. El parseo de las declaraciones de variable y la evaluación de las expresiones let-in se verá en una sección aparte.

Las expresiones let-in tienen se parsean antes de las expresiones if-else y estas a su vez antes de todas las vistas anteriormente.

4.3. Declaraciones de funciones

El interprete de H.U.L.K también es capaz de recibir y reconocer declaraciones de funciones inline, las mismas son por ejemplo “function succesor(n) => x+1”. Inicialmente estas se parsean de forma parecida a las expresiones if-else y let-in, solo que con los tokens “function”y “=¿”. A la derecha de este último token se parsea una expresión cualquiera de forma usual. Entre “function”y “=>”se toma como nombre de la función el token siguiente a “function”y como nombres de los argumentos, los que se encuentren separados por comas y entre paréntesis a continuación del nombre de la función.

5. Evaluando expresiones que llaman variables

5.1. Evaluando expresiones let-in

Como se había dicho anteriormente estas expresiones usan en su cuerpo variables que declaran al principio. Estas declaraciones pueden contener explícitamente el tipo (H.U.L.K. posee tres tipos básicos, “number”, “boolean”y “string”, el cual tiene que coincidir con el valor que devuelva la expresión después de “=”) o no, en cuyo caso este se infiere a partir del valor que se asigne. Las declaraciones crean una instancia de una clase Variable, en la cual se guarda el nombre, el tipo, y el valor que almacenan o la expresión que asigna su valor

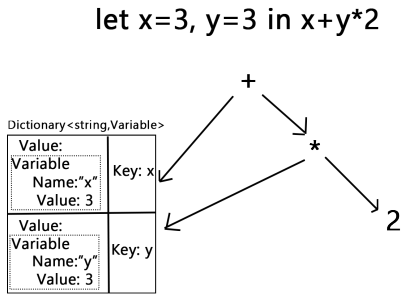

Figura 2: Ejemplo de la estructura de una expresión let-in

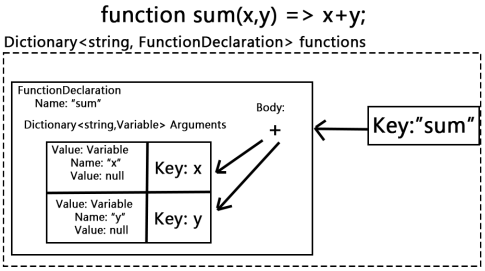

Figura 3: Representación del guardado de una función

(sobre esto se hablara mas adelante). Los argumentos del la expresión let-se guardaran como valores en un diccionario que toma como llave su nombre. Luego a la hora de construir el árbol de la expresión del cuerpo del let-in se hace referencia a esos objetos Variable de dicho diccionario, para que a la hora de evaluar sea posible recorriendo el árbol en profundidad.

5.2. Guardando y evaluando funciones

El intérprete guarda en un diccionario las funciones declaradas, accediendo a ellas con su nombre. A la hora de parsear expresiones principales, si se identifica un llamado a una función declarada entonces se procede a evaluarlas.

Las funciones al igual que las expresiones let-in tienen un diccionario que guarda los argumentos y una expresión cuerpo. En el cuerpo de las funciones al igual que en las expresiones let-in se puede hacer referencia a los argumentos de esta, con la diferencia que en una expresión let-in las variables tienen un valor preestablecido, en las funciones el valor puede cambiar en distintos llamados.

Cuando se parsea una declaración de función, el intérprete crea objetos Variable dándoles valor null, parsea la expresión cuerpo y guarda la expresión en el diccionario donde se almacenan las funciones. A la hora de llamar una función se accede a esta expresión, se modifican los valores de las variables de la definición de la función y se evalúa la expresión cuerpo con las nuevas referencias, luego estos vuelven a recibir valor null.

Es importante aclarar que el lenguaje H.U.L.K. admite funciones recursivas y el interprete es capaz de parsearlas y ejecutarlas. Para ello ademas de buscar si la que se esta llamando en el cuerpo de la declaración se encuentra en el diccionario de funciones declaradas, se comprueba si es la función de que se esta declarando.

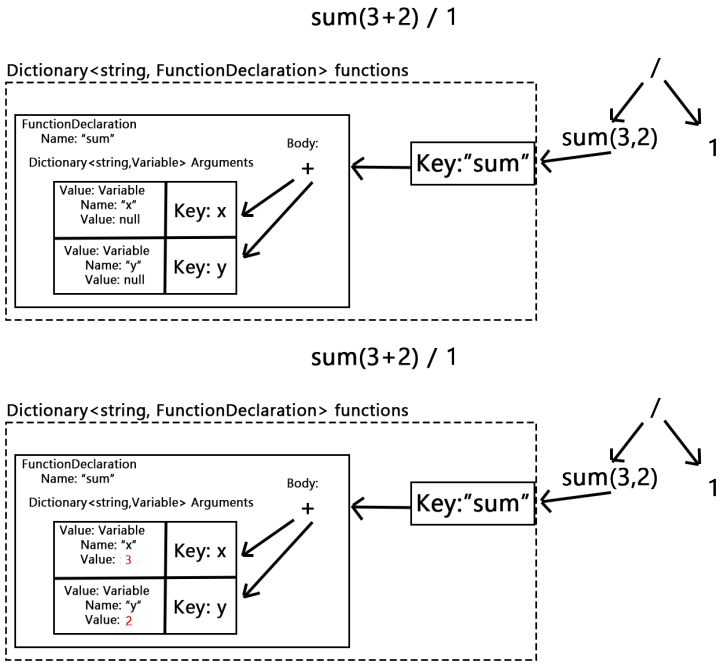

Figura 4: Parseo y evaluación de una función

6. Scope de variables

Ya se han definido todas las expresiones que reconoce el interprete de H.U.L.K., pero aun quedan interrogantes por responder. Considérese la función definida como "function f(x) => x+ (let x=2 in x)"y el llamado a esta "f(3)". Aquí suceden procesos interesantes. La expresión let-in devuelve siempre valor 2, no importa cual sea el valor que se le asigne a x en el llamado de la función y esto sucede porque la x que maneja dicha expresión existe en ese contexto y oculta el valor de la otra x. La x de afuera sí toma el valor que se le asigne, en este caso 3, siendo el resultado de f(3) igual a 5. Situaciones parecidas suceden al anidar expresiones let-in y en otros casos.

El intérprete resuelve este problema usando una pila. Cuando empieza a parsear una expresión que guarda variables (declaraciones de funciones y let-in's) antes de comenzar a parsear el cuerpo de esta, la añade la cima de una pila. Cuando construye la expresión del cuerpo y encuentra un token de nombre de variable, él hace referencia a la variable que más arriba se encuentre en la pila, sino se encuentra en ese nivel, se sigue a la expresión inferior hasta que se encuentre, de lo contrario se devuelve un error.

7. Expresiones dependientes

Cuando se declara una variable como argumento de una expresión let-in, por ejemplo "number x = 3 + 4"esta valdría 7, ya que es el resultado de la expresión a la derecha de "=". En este caso dicho resultado se copia por valor, pero no siempre se puede proceder así. En la expresión "function f(x) =¿let y = x in y"si en la declaración de la variable y copiamos por valores el resultado de la expresión "x"la variable y siempre tomaría valor null. Es necesario entonces hacer una excepción con ciertas expresiones que dependen de parámetros, estas expresiones serán llamadas expresiones dependientes y cuando se declare una variable que tome como valor el resultado de evaluar una expresión de este tipo, su valor hará referencia a la expresión como tal.

8. Chequeo de tipos

Como fue mencionado anteriormente en H.U.L.K. hay tres tipos primitivos: number (números enteros o con coma), boolean (valores de verdad) y string(cadenas de caracteres). En H.U.L.K. todas las expresiones devuelven un valor excepto las declaraciones de funciones y variables, y siempre tienen un tipo. Al ser un lenguaje de tipado estático es necesario comprobar los tipos antes de que se evalúen las expresiones. Para esto toda HulkExpression implementa el método CheckType().

Las expresiones tienen tipos de entrada y un tipo de salida. El método CheckType() en esencia lo que hace es comprobar que los tipos de entrada sean los correctos, para lo cual llama recursivamente al método CheckType() de las expresiones a las que hace referencia. Si estos son acertados devolver el tipo de salida esperado y en caso contrario lanzar un error. Existen casos en los que no es posible determinar el tipo devuelto de una expresión, por ejemplo una expresión condicional que devuelva un tipo en caso de que se cumpla su condición y otro en caso opuesto, un parámetro de una variable. Aun así en la mayoría de los casos es posible inferirlo si se efectúa con dichas expresiones una operación que siempre devuelve un tipo. En caso de que no fuera posible inferirlo, el interprete entiende que la expresión tiene tipo dinámico, cambia en dependencia de cierta situación.

9. Errores

En H.U.L.K hay varios tipos de errores, siendo los más importantes semántico, léxico y sintáctico.

Los errores léxicos son los que se lanzan por la presencia de tokens inválidos. Estos son detectados en el proceso de parseo. De ser encontrado, por ejemplo un nombre invalido para una variable se lanza una excepción.

Los errores sintácticos se producen por expresiones mal formadas o paréntesis mal balanceados y al igual que los léxicos son detectados a la hora de parsear. Si se no se encuentra, por ejemplo, una expresión después de un token “else”en una expresión if-else, se lanza una excepción.

Los errores semánticos son los que se producen por el uso de tipos y argumentos. La mayoría de estos son capturados en la fase de chequeo de tipos y en los casos restantes a la hora de evaluar una expresión.

Para manejar estos errores se usan una clases que controlan cada uno de los tres tipos de error, que heredan de una clase HulkError, que a su vez hereda de Exception. Cada una recibe ciertos argumentos con los cuales forman un mensaje de error adecuado a la situación. Luego son lanzadas y capturadas por la interfaz del programa que se encarga de imprimirlos en pantalla.

10. Print y operador :=

El operador binario “:=”y la función print() son casos particulares de las operaciones y funciones ya que no solo devuelven un valor sino que también ejecutan una acción.

En el caso del operador “:=”cambia el valor de la variable a la izquierda de el por el valor de la expresión de la derecha.

La función print() como su nombre lo indica imprime el valor de la expresión. La clase que representa esta expresión recibe como parámetro un delegate que es la función que se encargara de imprimir en la pantalla. Esto da la posibilidad de tener varias interfaces gráficas que accedan a la misma implementación, y definir en dependencia de estas como se visualizara la salida del programa sin interferir en el funcionamiento del motor del intérprete.

11. Conclusiones

En este informe, se ha presentado una visión general del intérprete Project H.U.L.K., que ejecuta un subconjunto del lenguaje H.U.L.K. Se destacó la importancia del análisis léxico para reducir las cadenas de entrada a tokens significativos. Además, se discutió el proceso de interpretación de expresiones y la

construcción de árboles de expresión para garantizar el orden correcto de evaluación. Estos conceptos son fundamentales para comprender el funcionamiento del intérprete y sentar las bases para futuros desarrollos en el proyecto.