Informe sobre Intérprete *HULK : Havana University Language for Kompilers*

Sheila Roque Alemán

Introducción

HULK es un lenguaje de programación creado y diseñado por la Facultad de Matemática y Computación de la Universidad de La Habana, *MatCom*. Es un lenguaje de programación imperativo, funcional, estática y fuertemente tipado. Casi todas las instrucciones en este lenguaje son expresiones. Este programa es un intérprete sencillo que implementa ciertas características de este lenguaje lenguaje.

HULK

El intérprete de *HULK* es una aplicación de consola, donde el usuario puede introducir una expresión de *HULK* , presionar [ENTER], e immediatamente se verá el resultado de evaluar expresión (si lo hubiere). Cada línea que comienza con '>' representa una entrada del usuario, e immediatamente después se imprime el resultado de evaluar esa expresión, si lo hubiere. Si el usuario presiona [ENTER] sin haber escrito una instrucción se tomará como una señal de que desea cerrar el programa

En este intérprete solo se aceptan expresiones de una línea. Para ello se han implementado diversas expresiones *inline* , como lo son la declaración de funciones de este tipo y ciertas expresiones como las *let-in* e *if-else* .

A pesar de esta restricción, el intérprete posee la capacidad y optimización necesarias para poder crear un programa completamente funcional y veloz habiéndose creado las funciones necesarias línea por línea.

Expresiones básicas

- Todas las instrucciones en *HULK* terminan en "*;*".
- La instrucción más simple en HULK que hace algo es:

  > print("Hello, World!");
  Hello, World!
- *HULK* además tiene expresiones y funciones aritméticas básicas:

  *> print((((1 + 2) ^ 3) * 4) / 5);*
  21.6
  *> print(sin(2 * PI) ^ 2 + cos(3 * PI / log(4, 64)));*
  -1

- *HULK* tiene tres tipos básicos: *String*, *Number*, y *Boolean*.

Funciones

- En *HULK* hay dos tipos de funciones, las funciones *inline* y las funciones regulares. En este intérprete solo se implementan las funciones *inline* . Tienen la siguiente forma:

  > function tan(x) => sin(x) / cos(x);

- Una vez definida una función, puede usarse en una expresión cualquiera:

  > print(tan(PI/2));
  ∞

- El cuerpo de una función *inline* es una expresión cualquiera, que por supuesto puede incluir otras funciones y expresiones básicas, o cualquier combinación.

- Todas las funciones declaradas anteriormente son visibles en cualquier expresión subsiguiente.

- Las funciones no pueden redefinirse.

Variables

- En HULK es posible declarar variables usando la expresión *let-in* , que funciona de la siguiente forma:

  > let x = PI/2 in print(tan(x));
  ∞

- Una expresión *let-in* consta de una o más declaraciones de variables, y un cuerpo, que puede ser cualquier expresión donde además se pueden utilizar las variables declaradas en el *"let"*.

- Fuera de una expresión *let-in* las variables dejan de existir.

  > let number = 42, text = "The meaning of life is" in print(text @ number);
  The meaning of life is 42
  > let number = 42 in (let text = "The meaning of life is" in (print(text @ number)));
  The meaning of life is 42

- El valor de retorno de una expresión *let-in* es el valor de retorno del cuerpo, por lo que es posible hacer:

  *> print(7 + (let x = 2 in x * x));*
  11

Condicionales

- Las condiciones en HULK se implementan con la expresión *if-else* , que recibe una expresión booleana entre paréntesis, y dos expresiones para el cuerpo del *if* y el *else* respectivamente.

  * *Siempre deben incluirse ambas partes*

- Como *if-else* es una expresión, se puede usar dentro de otra expresión

  > let a = 42 in if (a % 2 == 0) print ("even") else print ("odd");
  even
  > let a = 42 in print (if (a % 2 == 0) "even" else "odd");
  even

Funciones recursivas

- Dado que *HULK* tiene funciones compuestas, por definición tiene también soporte para recursión. Un ejemplo de una función recursiva en *HULK* es la siguiente:

  > function fib(n) => if (n > 2) fib(n-1) + fib(n-2) else 1;
  > fib(5);
  5
  > let x = 3 in fib(x+1);
  3
  > print(fib(6));
  8

Errores

- En *HULK* hay 3 tipos de errores que pueden ser detectados: *Léxico*, *Sintáctico* y *Semántico*.
- En caso de detectarse un error, el intérprete imprime una línea lo más informativa posible indicando el error.
- Al detectar un error, el intérprete detiene la compilación el código y lanza un error, sin causar que el programa caiga, dándole la oportunidad al usuario de corregir su error y continuando con su ejecución.
- En caso de ocurrir más de un error, solo se lanza el primero que aparece.

Errores Léxicos

Errores que se producen por la presencia de tokens inválidos:

> let 14a = 5 in print(14a);
! LEXICAL ERROR: '14a' is not valid token.

Errores Sintácticos

Errores que se producen por expresiones mal formadas como paréntesis no balanceados o expresiones incompletas:

> let a = 5 in print(a;
! SYNTAX ERROR: Missing closing parenthesis after 'a'.
> let a = 5 inn print(a);
! SYNTAX ERROR: Missing 'in' keyword in 'let-in' expression.
> let a = in print(a);
! SYNTAX ERROR: Invalid expression after variable 'a' in 'let-in' expression.

Errores Semánticos

Errores que se producen por el uso incorrecto de los tipos y argumentos:
> let a = "Hello, World!ïn print(a + 5);
! SEMANTIC ERROR: Operator '+' cannot be used between 'String' and 'Number'.
> print(fib("Hello, World!"));
! SEMANTIC ERROR: Function 'fib' receives 'Number', not 'String'.
> print(fib(4, 3));
! SEMANTIC ERROR: Function 'fib' receives 1 argument(s), but 2 were given.

¿Cómo funciona?

Al recibir una instrucción por parte del usuario comienza el proceso de compilación. Primero se convierte la instrucción en una lista de tokens en orden de aparición, para facilitar el proceso de *parsing* que le sigue a continuación. Luego del proceso de *parsing* se obtiene como resultado un árbol de sintaxis abstracta compuesto por las distintas expresiones que componen a la instrucción recibida. Luego se realiza la revisión semántica de las expresiones en el árbol de derivación, y luego se evalúan para obtener el valor a devolver (en caso de que lo hubiere).

Tokenizador

- El lexer creado tendrá una instancia de *Error Léxico*, la línea de código dada que se va a estar utilizando, y un indizador que nos ayude a recorrerla
- La línea de código la provee la llamada al método *Tokenizer(string line)*
- En caso de que ocurra un error léxico al tokenizar, se lanza dicho error notificando al usuario que el token encontrado no es válido, y se devuelve el token como nulo, para que luego, de existir algún token nulo, se devuelva una lista nula y se detenga la ejecución para empezar desde el principio
- El recorrido del token es sencillo:
  Se recorre toda la línea de código, caracter por caracter, y se va analizando, según el tipo de caracter que sea, qué tipo de token puede ser
  - Si se encuentra un espacio no se debe guardar nada
  - Si se encuentra con unas comillas dobles entonces lo siguiente debe ser una expressión de string.
    Se guarda el string además de las comillas dobles que lo acompañan
  - Si se encuentra algún otro caracter válido hay que analizar cúal es y qué tipo de operador devuelve (separadores, operadores, palabraas reservadas, identificadores o literales numéricos), y se devuelve dicho token

Parser

- El parser posee una instancia de *Error Sintáctico*, una lista de tokens dada y un índice que nos ayuda a recorrer todos los tokens de dicha lista

- El proceso de *parsing* evalúa la lista de tokens dada y devuelve la expressión resultante según la sintaxis
- Se realiza el *parsing* por medio del algoritmo de *parsing recursivo descendiente predictivo*. Consiste en evaluar los tokens en orden de izquierda a derecha, formando las expresiones de forma descendiente, y prediciendo cuál puede ser la posible expresión teniendo en cuenta el primer token a evaluar de esta.

Un pequeño ejemplo pudiera ser la instrucción siguiente:

> print((1+2)ˆ 3);

La lista de tokens dados sería de esta forma:

print , (, (, 1 , + , 2 ,) , ˆ , 3 ,) , ;

Al empezar a 'parsear', evaluamos qué espresión es la expresión más externa de la instrucción evaluando al primer token. En este caso tenemos a la palabra reservada '*print*', entonces intuímos que la expresión es una llamada al método *print('Expr')*.

Evaluamos el resto de la expresión teniendo en cuenta la sintaxis de una expresión de llamado al print. Revisamos el siguiente token, procurando que sea un '(', lo cual es cierto en este caso.

Luego hay que evaluar la expresión interna del print. Primero, nos encontramos que el siguiente token es un '(', lo cual nos indica que existe otra expresión más interna dentro de unos paréntesis. Se evalúa el siguiente token y encontramos que es un literal numérico, y consecuentemente inferimos que la expresión es una expresión numérica.

Una expresión numérica se evalúa con las siguientes reglas en cuanto al parsing descendiente y predictivo:

NumExpr -> T X

T -> num Y | (NumExpr) Y | ...

X -> + NumExpr | - NumExpr | e

Y -> * T | / T |% T | ˆ T | e

Entonces evaluamos el token '1' como el inicio de una expresiónn numérica, que a su vez se va a evaluar como un término, que se evalúa en este caso en la forma T -> num Y, siendo '1' el 'num'.

Hasta ahora la derivación va siendo: (num Y X

Luego evaluamos con la sintáxis de Y. Al ser el siguiente token el operador '+', entonces tenemos que Y -> e, y continuamos hacia el evaluar X. La expresión X resulta en este caso como X -> + NumExpr, y este NumExpr se deriva como NumExpr -> num Y, donde Y -> e. Luego encontramos los paréntesis de cierre de esta expresión

Ahora tenemos la derivación: (num + num)

Evaluando que la expresión de externa es otra expresión numérica (que puede ser otro tipo de expresión que pueda utilizar una numérica, pero en este caso no), podemos inferir que la siguiente evaluación sea de otra Y. En este caso, Y sería Y -> ˆ T, y este T sreía T -> num Y, del cual Y -> e.

De esta forma concluímos la derivación con (num + num) ˆ num

Ya teniendo la expresión numérica evaluada, terminamos de evaluar el print con el siguiente token que es ')'. Luego, para finalizar, revisamos que la instrucción termine con un ';', el cual en este caso existe y aparece en el final de la expresión.

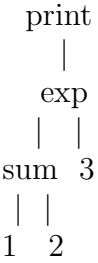
Semántica

Una vez definido el árbol de sintáxis abstracta dado por el parser, pasamos a evaluar la semántica.

Definimos una clase abstracta que Expression será la madre de todas nuestras diferentes expresiones:

- Toda expresión posee un tipo y una instancia de *Error Semántico*.
- Las expresiones tienen que poder validarse y, luego de ser válidas, evaluarse para devolver el valor que expresan.

Sigamos con el ejemplo de la vez anterior, tiendo un árbol de derivación resultante de esta forma:



Empezamos a validar desde afuera hacia adentro, aunque luego la ejecución terminará validando primero las expresiones más internas para que las externas puedan ser validadas basándose en ellas.

Comenzando por el print, solo debemos validar la expresión que lleva en su interior. Dentro tenemos la expresión numérica compuesta por una expresión binaria de exponente de: una expresión binaria de suma entre los literales numéricos '1' y '2', y el literal numérico '3'. Los literales numéricos son válidos de por sí, así que no hace falta evaluarlos (al hacerlo siempre devuelve que son válidos), así que pasamos a evaluar las expresiones binarias.

Al validar nos daremos cuenta que la instrucción es válida, no aparecen llamadas a métodos ni a variables que es de lo que mayormente se trata de verificar su validez, apoyándose en los contextos para guardar las variables y las definicionesde las funciones.

Al ser esta instrucción válida, es momento de evaluar para definir el valor que va a ser devuelto por esta instrucción, ya que en este caso devuelve un valor. Comenzamos igualmente desde la expresión más externa hasta llegar a la más interna y devolver así todo el valor calculado. Primero llegamos a evaluar la expresión binaria de la suma de '1' y '2', esto devuelve '3'. Luego esto se le devuelve al miembro izquierdo de la expresión binaria de exponente de, ahora '3', y '3', que devuelve '27'. Por último se devuelve el '27' al print que lo devuelve igualmente para que salga en pantalla para el usuario, concluyendo así tanto el proceso de evaluar el valor, como el proceso de compilación en general.