Universidad Simón Bolívar
Departamento de Computación y tecnología de la Información
Traductores e Interpretadores
Febrero-abril 2022

Proyecto: Interpretador de Stókhos

0. Sinopsis del Proyecto

Stókhos (del griego στόχος) es un lenguaje interactivo orientado a simulaciones estocásticas. El lenguaje está fuertemente simplificado para facilitar su implementación, por medio de un interpretador, en un trimestre. En particular, no se permiten definiciones de funciones, lo cual es un tópico adecuado para la cadena de lenguajes, aunque si se soportan llamadas a funciones predefinidas.

El laboratorio requiere la implementación de un simple REPL [1], es decir una interfaz básica de línea de comandos. Esto se debe a varias razones, principalmente: 1) el lenguaje enfatiza el uso interactivo – si Uds. han trabajado con Python o Node.js desde un "Shell" por medio de comandos, Uds. ya han usado un REPL; 2) el REPL incluye comandos especiales para cargar programas y realizar pruebas – esto ayuda a los estudiantes y preparadores a detectar problemas en la implementación rápidamente.

1. Definición Informal de Stókhos

Esta definición no provee una gramática formal del lenguaje: la intención es que los estudiantes desarrollen la capacidad de abstraer y definir gramáticas a partir de ejemplos y definiciones informales, incluso incompletas, lo cual, para bien o para mal, es el caso común en entornos no académicos.

La gramática de Stókhos es muy simple, como podrán apreciar. Las expresiones son parecidas a las de la mayoría de los lenguajes de programación que Uds. conocen. Nótese que, por lo general, resaltamos en negrillas las palabras reservadas del lenguaje.

1.1 Tipos y Constantes

Stókhos posee dos tipos básicos, que se denotan con las palabras reservadas ***num*** (números) y ***bool*** (valores booleanos). Además, si *B* es un tipo básico, una expresión de la forma [*B*]**,** denota el tipo "arreglo de elementos de tipo B". Estos son todos los tipos del lenguaje.

1.1.1 Constantes Booleanas

Los posibles valores booleanos se representan con ***false*** (falso) y ***true*** (verdadero).

1.1.2 Números

Algunos ejemplos de constantes numéricas son : 0, 3.141592654, 42, 67.67, 997, 987654321.123.

En Stókhos no distinguimos entre números enteros y aproximaciones a números reales. La precisión de los números en *su* implementación de Stókhos va a reflejar la del lenguaje que usen en el laboratorio;

dicho lenguaje casi seguramente soporta una o más representaciones del estándar IEEE 754. De ser posible, usen la representación IEEE 754 *binary64*, conocida como doble precisión. [2]

Al implementar el reconocedor (parser) de un lenguaje, hay situaciones en las que tenemos un número grande, incluso infinito, de unidades lexicográficas que necesitamos agrupar por razones prácticas. Ejemplos clásicos son identificadores y números: todo identificador o constante numérica es una unidad indivisible (símbolo terminal) para propósitos de análisis sintáctico, pero sería imposible listarlos a todos en una gramática. Por esta razón, definimos pseudo-terminales como *<id>* y *<number>* para representar a todo símbolo terminal correspondiente a un identificador o un número. Nótese que al reconocer un pseudo-terminal, necesitamos guardar la secuencia de caracteres que apareció en la entrada: dicha secuencia se denomina "lexema". Sin esa información es imposible implementar un lenguaje: desde el punto de vista sintáctico, las expresiones siguientes tienen la misma secuencia de símbolos terminales, específicamente *<id> + <number>*, es decir un identificador seguido del operador de suma seguido por un número, pero desde el punto de vista semántico son diferentes:

```
i + 1
gamma + 42.67
```

para reiterar, eviten confundir al *código* de una unidad lexicográfica (token) con el *valor* de ésta: toda constante numérica tiene el mismo código de token, pero el valor del token corresponde al número.

1.2 Expresiones y Variables

Las expresiones en Stókhos se parecen mucho a las de los lenguajes que ya conocen, por ejemplo:

```
A[i] <= A[i + 1]
log(Alpha + 2 * Beta^(Gamma / 2))
[ i + 1, 'x + y', 'A[i + 1] – A[i]' ]
```

La semántica de algunos operadores, en particular la acotación de expresiones y la construcción de arreglos, se explicará con ejemplos más adelante, y con mayor formalidad, antes de las etapas 3 y 4.

1.2.1 Precedencia de Operadores en Stókhos

De mayor a menor precedencia:

Operador(es)	Descripción	Asociatividad
(), [ ], ' '	Paréntesis y acotación (comillas simples)	n.a.
^	Elevación a potencia	derecha a izquierda
+, - , !	Más y menos unarios, negación lógica	n.a.
*, %	Multiplicación y módulo	izquierda a derecha
+, -	Suma y resta	izquierda a derecha
<, <=, >=, >	Comparación	n.a.
=, <>	Igualdad y desigualdad	n.a.
&&	Y (And) – Conjunción lógica	izquierda a derecha
||	O (Or) - Disyunción lógica	izquierda a derecha
:=	Asignación	n.a.

1.2 Variables

1.2.1 Identificación de Variables

Las variables se identifican con símbolos alfanuméricos, también conocidos como identificadores. Un identificador es una cadena de caracteres de longitud arbitraria que consiste únicamente de letras mayúsculas y minúsculas del alfabeto ASCII ('A'-'Z' y 'a'-'z'), los dígitos del '0' al '9', y el carácter '_' (underscore). Los identificadores no pueden comenzar con un digito y son sensibles a mayúsculas y minúsculas: rap, Rap, y RAP son nombres distintos. No toda variable tiene un identificador: los elementos de un arreglo son variables independientes y no tienen nombre propio, pero pueden denotarse con una expresión utilizando el operador [ ], como veremos más adelante.

Las palabras reservadas no pueden ser usadas como identificadores: ¡son *reservadas* precisamente por esta razón! Como explicado anteriormente, debemos agregar un pseudo-terminal a la gramática del lenguaje para referirnos a un identificador arbitrario.

1.2.2 Introducción a la Semántica de las Variables

Las variables en Stókhos almacenan estado y son mutables. El estado de una variable es una *expresión*. El sistema de tipos garantiza que el tipo de la expresión almacenada en una variable es consistente con el tipo declarado en la definición de la variable [3]. Dadas las definiciones siguientes:

```
num x := 6;
num y := 7;
num p := x * y;
```

los estados de las variables *x*, *y*, y *p* son 6, 7, y 42 respectivamente.

Cuando decimos que el estado de una variable es una *expresión*, enfatizamos que dicha expresión no está necesariamente en forma reducida, es decir evaluada. En el ejemplo anterior, podemos apreciar que la expresión *x * y*, a la derecha del símbolo de asignación, es evaluada *antes* de pasar a ser el estado de la variable *p*. Entonces, ¿cómo podemos lograr que el estado de una variable sea una expresión no reducida? La acotación de expresiones permite hacer esto: el resultado de evaluar una expresión acotada es la expresión contenida entre las comillas simples. Si definimos *z* de esta manera:

```
num z := 'x * y';
```

el estado de z es la expresión *x * y*. Noten que la expresión al lado derecho de la asignación *siempre* se evalúa. Nuevamente, *el resultado de evaluar una expresión de la forma 'ψ' es ψ*. Por brevedad, de ahora en adelante usaremos la notación ==> para denotar la transformación efectuada por la evaluación:

```
'x * y' ==> x * y
```

Stókhos posee una función *especial*, llamada *formula*, que nos permite observar el estado de una variable. Esta función es especial porque no evalúa su argumento. En general, Stókhos emplea orden aplicativo, también conocido evaluación estricta (strict evaluation: [4]) es decir que los argumentos deben ser evaluados antes de llamar a una función: *formula* es una *excepción* a la regla – simplemente retorna *el estado de la variable*, es decir *la expresión almacenada* en la variable. Esto implica que el

argumento debe ser una variable. En Stókhos, *evaluar una variable no es lo mismo que retornar su estado*: el resultado de evaluar una variable es igual al resultado de evaluar el *estado* de la variable, es decir es igual al resultado de evaluar la expresión almacenada en la variable. Por lo tanto, tenemos

formula(z) ==> x * y
z ==> 42

En el caso de la variable p, el estado de la variable es una expresión reducida, y tenemos lo siguiente:

formula(p) ==> 42
p ==> 42

Para aclarar lo que debería ser obvio dada la explicación anterior, *no hay dos reglas diferentes* – una para variables cuyo estado es una expresión reducida y otra para variables cuyo estado no es una expresión reducida. La semántica de evaluar variables es siempre la misma. El estado de *p* y el resultado de evaluar a *p* coinciden por una razón muy simple:

42 ==> 42

La semántica descrita implica que el valor de una variable como *z*, cuyo estado es una expresión no reducida que depende de otras variables, es afectado por cambios a esas variables. Por ejemplo:

x := x + 1;
formula(z) ==> x * y
z ==> 49

La primera instrucción es una asignación (más propiamente una reasignación) de la variable *x*. Siguiendo las reglas de evaluación dadas, el lado derecho de la asignación se evalúa (esto incluye evaluar a *x*) y el resultado de dicha evaluación pasa a ser el nuevo estado de *x*. Al evaluar la formula de z, vemos que el estado de z no ha cambiado: sigue siendo la misma expresión. Sin embargo, *evaluar z* implica evaluar la expresión *x * y*, el ahora el resultado es 49.

El valor de *z* cambia en reacción al cambio de *x*. Este tipo de comportamiento se llama *reactivo*.

La semántica reactiva, en combinación con funciones estocásticas predefinidas, va a permitirnos realizar simulaciones interesantes.

Recuerden que para las etapas 1 y 2 no tienen que preocuparse por la semántica, sólo por la sintaxis. Sin embargo, *es **importante** que comiencen a asimilar la semántica del lenguaje*.

1.2.3 Arreglos

Un arreglo es, *a la vez*, una variable y un *contenedor*, posiblemente vacío, de *variables independientes*. El estado de un arreglo *A* es una expresión de tipo arreglo cuyos componentes son los estados de los componentes de *A*.

En el ejemplo siguiente, vemos cómo el operador [ ] es utilizado para inicializar un arreglo:

num x := 6;
num y := 7;
[**num**] vector := [x + 1, y + 4.2, 'x * y'];

Usado de la manera anterior, el operador [ ] es un *constructor de arreglo*. Los elementos usados en el constructor deben ser del mismo tipo, de otra forma la expresión es invalida. La evaluación de una expresión de tipo arreglo resulta ser el arreglo de sus elementos evaluados:

```
[ x + 1, y + 4.2, 'x * y' ] ==> [ 7, 11.2, x * y ]
```

Por lo tanto, usando las reglas explicadas en la sección anterior:

```
formula(vector) ==> [ 7, 11.2, x * y ]
vector ==> [ 7, 11.2, 42 ]
```

Stókhos provee varias funciones que operan sobre arreglos; el significado de las siguientes es intuitivo:

```
length(vector) ==> 3
sum(vector) ==> 60.2
```

Cada elemento de la variable vector es una variable en su propio derecho. A pesar de no tener un identificador propio, nos podemos referir a estos elementos usando el operador [ ] como selector:

```
vector[0] + vector[1] ==> 18.2
formula(vector[2]) ==> x * y
vector[2] ==> 42
```

1.3 La Máquina Virtual de Stókhos

Al arrancar la Stókhos VM (máquina virtual de Stókhos) los únicos símbolos presentes son los de las funciones predefinidas, que son inmutables. Se define el estado de la VM como el conjunto de estados de las variables presentes.

1.3.1 Acciones

Para cambiar el estado de la VM debemos ejecutar acciones, también llamadas instrucciones, que permiten agregar variables o cambiar el estado de éstas. Ya hemos visto las acciones principales de Stókhos: definiciones y reasignaciones de variables. La única diferencia sintáctica entre estos dos tipos de acciones es que la definición de una variable requiere especificar su tipo. Una variable debe ser definida antes de poder ser usada. Además, no se permite definir una variable dos veces.

2. REPL

Para este laboratorio los estudiantes deben implementar una simple interfaz de línea de comando para interactuar con la VM. En pocas palabras, los estudiantes deben implementar un REPL.

REPL as un acrónimo para "**R**ead-**E**val-**P**rint-**L**oop":

Read: leer el comando
Eval: evaluar el comando
Print: imprimir el resultado
Loop: seguir en este ciclo hasta que el usuario se salga

Desde el punto de vista de ingeniería de software, la implementación de un REPL es una buena práctica, considerada *de rigor* al implementar un lenguaje de programación.

2.1 El REPL y La VM

El REPL no es la VM. El REPL es un ***cliente*** de la VM: una interfaz simple que ofrece gran conveniencia tanto para realizar pruebas interactivas, ad-hoc, como para automatizar pruebas de regresión [5]. Esto beneficia a los estudiantes y a los preparadores del laboratorio. La VM de un lenguaje *L* crea la ilusión que la computadora entiende *L*. Esta ilusión es la razón por la cual se le llama "máquina virtual".

Siempre mantengan una separación sagrada entre el REPL y la VM. La VM debe tener una interfaz extremadamente limpia, en principio la interfaz consiste en una sola función de la forma:

```
process(input)
```

donde input es un string en el lenguaje usado para implementar la VM. La función *process* ejecuta la acción o evalúa la expresión indicada en el input. Si el input no es una acción o expresión sintácticamente válida, la VM lo debe rechazar retornando un error. Lo mismo aplica si el input es sintácticamente válido, pero falla algún chequeo semántico. En lo posible, el mensaje de error debe ayudar al usuario a entender qué es lo que está mal. Aquí se puede apreciar una sesión interactiva de Stókhos usando el REPL:

```
< Stókhos > num x := 6;
ACK: num x := 6;
< Stókhos > num y := 7;
ACK: num y := 7;
< Stókhos > num z := 'x * y';
ACK: num z := 'x * y';
< Stókhos > x
OK: x ==> 6
< Stókhos > y
OK: y ==> 7
< Stókhos > z
OK: z ==> 42
< Stókhos > x := x + 1;
ACK: x := x + 1;
< Stókhos > z
OK: z ==> 49
< Stókhos > .
$
```

Lo que el usuario escribe en el teclado se muestra en negrillas, y las respuestas de la VM en azul. Nótese que el REPL muestra el *prompt*, lee lo que el usuario escribe (**READ**), lo envía a la VM para ser procesado (**EVAL**), imprime la respuesta de la VM (**PRINT**), y repite el ciclo (**LOOP**), hasta leer el comando mágico al final (simplemente un punto) que permite al usuario salirse, terminando la sesión. Desde el punto de vista del REPL, un comando mágico es uno que se ejecuta localmente, es decir sin invocar la VM. Otros comandos mágicos serán agregados en cada etapa.

Como pueden ver el REPL es sumamente sencillo. También les va a ser muy útil a lo largo del proyecto. En cada etapa, dependiendo de los objetivos de esta, les diremos como extender el REPL y la VM.

Recuerden:

- El REPL es un cliente de la VM.
- Mantengan una separación sagrada entre el REPL y la VM.
- Los REPL no son los únicos clientes posible de una VM, pero tienen sus ventajas.

3. Notas

[1] REPL es el acrónimo de Read-Eval-Print-Loop: Leer-Evaluar-Imprimir-Ciclar.

[2] https://en.wikipedia.org/wiki/IEEE_754

[3] La mayor razón para tener un sistema de tipos es poder dar garantías sobre el comportamiento de un programa *antes* de ejecutarlo. Al definir una variable se le asocia un tipo que restringe los estados posibles de la misma: el estado de una variable puede cambiar, pero no su tipo.

[4] https://en.wikipedia.org/wiki/Evaluation_strategy#Strict_evaluation

[5] https://en.wikipedia.org/wiki/Regression_testing