Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Traductores e Interpretadores
Octubre-diciembre 2021

Etapa I: Análisis Lexicográfico

Esta entrega consiste en la implementación de un analizador lexicográfico: el primer paso en la construcción de todo interpretador o traductor. También vamos a implementar y usar el REPL de una vea para probar la funcionalidad del analizador lexicográfico del lenguaje Stókhos. El analizador lexicográfico debe aceptar cualquier secuencia de caracteres como entrada y regresar una secuencia de tokens como salida o reportar errores al encontrar caracteres o secuencias de caracteres invalidas en el lenguaje.

1. Tecnologías

1.1 Herramientas

Más adelante mostramos los lenguajes permitidos, para que escojan el que prefieren. Pero es importante que, ***desde ahora***, tomen en cuenta las herramientas disponibles, en cada lenguaje, para generar el analizador sintáctico (*reconocedor* o *parser*) y al analizador lexicográfico ("*tokenizador"* o *lexer*). La implementación del *lexer* es el objetivo principal de esta entrega; el *parser* será construido en la segunda etapa.

Algunos generadores de analizadores sintácticos (*parser generators*) ofrecen la opción de generar el analizador lexicográfico (*lexer*) también. Dada la sencillez de Stókhos, implementar el *lexer* "a mano" es una posibilidad, pero ***no*** traten de implementar el *parser* de Stókhos a mano, a menos que tengan experiencia implementando un lenguaje con operadores infijos y múltiples niveles de precedencia. Todos los generadores de reconocedores que conozco proveen una interfaz bien documentada para emparejarlo con un lexer, generado o no.

1.2 Lenguajes

JavaScript: lenguaje para la Web, soportado por Web browsers, pero también disponible para uso fuera del browser. Para este laboratorio usamos *node* (https://nodejs.org/en/) que es el porte de V8 al "*server side*" para poder usar JavaScript en la implementación de servidores y programas independientes: V8 es la VM de JavaScript en todos los browsers basados en Chromium.

Para investigar generadores de reconocedores en JavaScript, hagan una búsqueda y también vean lo siguiente:

https://stackoverflow.com/questions/6211111/javascript-parser-generator/

https://peggyjs.org/

https://pegjs.org/

TypeScript: extensión de JavaScript que permite, de manera opcional, hacer chequeos de tipos estáticos. Al ser una extensión verdadera, permite usar cualquier librería disponible en JavaScript.

Python: lenguaje popular de scripting, orientado a objetos. El más lento de todos en esta lista. Para Python la herramienta generadora de analizadores lexicográficos a utilizar es a la vez la misma que genera analizadores sintácticos. Por lo tanto, es posible que tengan que entender algunas nociones de gramáticas libres de contexto

antes de verlas en clase, usando la documentación. La herramienta se llama PLY y puede ser encontrada aquí: http://www.dabeaz.com/ply/

Java: lenguaje imperativo, orientado a objetos. Estáticamente tipeado, a diferencia de los anteriores, pero menos ágil para desarrollo. El generador de reconocedores más popular es ANTLR: http://www.antlr.org/

Haskell: lenguaje funcional estáticamente tipeado. Soporta *pattern matching*, algo ideal para implementar lenguajes. Permite definiciones de reconocedores al estilo yacc (Happy) y varias librerías para hacer *parsing* con combinadores. Haskell no es recomendable si no lo han usado anteriormente.

https://www.haskell.org/happy/

https://two-wrongs.com/parser-combinators-parsing-for-haskell-beginners.html

https://crypto.stanford.edu/~blynn/haskell/parse.html

https://serokell.io/blog/parser-combinators-in-haskell

OCaml: funcional, con *pattern matching*. Las mismas consideraciones hechas para Haskell aplican para OCaml: excelente lenguaje para implementar lenguajes, pero no es recomendable a menos que le puedan dedicar tiempo para aprenderlo.

https://ocaml.org/manual/lexyacc.html

https://github.com/pyrocat101/opal

2. Evaluación de Proyectos

2.1 GitHub

Vamos a usar GitHub para comunicar código entre preparadores y estudiantes. Cada equipo debe tener un proyecto (*privado*, para evitar "espionaje") para este laboratorio. Además de los integrantes del equipo, los colaboradores del proyecto deben incluir al asistente de laboratorio. El asistente de laboratorio ejecutará un ***git pull*** a la hora de entrega. Es *posible* (pero sin prometer nada) que el asistente de laboratorio agregue comentarios y sugerencias a su código para ayudarles a mejorar su trabajo antes de la entrega y mejorar sus prácticas de ingeniería de software.

2.2 Fechas de Entrega

Para las primeras etapas vamos a ofrecer la opción de hacer una entrega preliminar del proyecto. El equipo deberá indicar, por correo electrónico, que desea una revisión rápida para obtener "feedback temprano". Dada la escasez de recursos (tiempo) la revisión no puede ser exhaustiva, pero solo puede beneficiarlos, dándole la posibilidad de corregir errores antes de la entrega final.

- Etapa 1: entrega preliminar 21 de febrero - entrega final 24 de febrero
- Etapa 2: entrega preliminar 7 de marzo - entrega final 10 de marzo
- Etapa 3: (tentativa) entrega final 31 de marzo
- Etapa 4: (tentativa) entrega final 21 de abril

3. Tokens

Todo token debe tener un código entero único: este código es usado en las tablas del parser para dirigir el reconocimiento. Además, para propósitos de "debugging" y corrección, deben usar un símbolo único para que les permita identificar rápidamente de que token estamos hablando. Para facilitar la corrección del proyecto, identifiquen a los tokens usando las reglas siguientes.

Palabras reservadas: los tokens correspondientes a palabras reservadas como ***num*** y ***bool***, deben llamarse Tk<Palabra Clave>, donde <Palabra Clave> es una palabra clave del lenguaje, con su primera letra en mayúscula. Por ejemplo, el token para ***bool*** es **TkBool**, y para ***false*** es **TkFalse**.

Constantes numéricas: los números se agrupan con un pseudo-terminal, <number>, representado por el token **TkNumber** – notese que **TkNum** es el token correspondiente a la palabra reservada ***num***. Al reportar el token de un pseudo-terminal se debe agregar el lexema entre paréntesis, como si fuese un argumento del token. Por ejemplo, el número 42 debe mostrarse de esta manera: **TkNumber**(42).

Identificadores: se agrupan con un pseudo-terminal, <id>, representado por el token TkId. De manera análoga a lo hecho por números, el valor del token (el lexema es el identificador reconocido) debe mostrarse, Por ejemplo, las variables nombradas *v*, *w*, y *suma*, se representan TkId("v"), TkId("w"), y TkId("suma").

Operadores: la siguiente lista contiene operadores que aún no han sido añadidos al lenguaje.

Operadores	Tokens
( )	TkOpenPar y TkClosePar
[ ]	TkOpenBracket y TkCloseBracket
{ }	TkOpenBrace y TkCloseBrace
!	TkNot
^	TkPower
*, /, %	TkMult, TkDiv, y TkMod
+, -	TkPlus y TkMinus
<, <=, >=, >	TkLT, TkLE, TkGE, y TkGT
=, <>	TkEQ, TkNE
&&	TkAnd
||	TkOr
‘	TkQuote
,	TkComma
:=	TkAssign
;	TkSemicolon
:	TkColon

Nótese que Stókhos es *case-sensitive*: se debe preservar la diferencia entre mayúsculas y minúsculas. Los espacios en blanco, tabuladores, saltos de línea y comentarios deben ser ignorados. Es inaceptable manejarlos como *tokens*,

4. REPL

4.1 Convenciones

Para simplificar la implementación del REPL, vamos a asumir que cada comando se encuentra en una sola línea. También vamos a procesar archivos una línea a la vez, como si fuesen comandos dados interactivamente por el usuario. Líneas "vacías", incluyendo las que sólo tienen caracteres "blancos", no deben considerarse un error, sino ser ignoradas. Un comando puede ser una expresión o una acción (definición o asignación) de Stókhos. Además, tenemos comandos especiales del REPL, descritos más adelante, que siempre comienzan con un punto. Los comandos especiales del REPL ***no*** son parte del lenguaje: ***no*** los incluyan en la gramática de Stókhos. El REPL en sí es *trivial*, ***no*** vale la pena usar herramientas de *parsing* adecuadas para manejar lenguajes libres de contexto para implementarlo: basta ver el primer carácter de una línea para saber si un comando es especial; en caso de serlo, deben manejarlo como se describe más adelante.

Si una línea no es vacía y no comienza con un punto (el caso más común al completar el proyecto), se supone que debe ser una expresión o una acción (definición o asignación) de Stókhos. En este caso, el REPL ha completado la R (leer la entrada) y debe despachar la línea a la VM de Stókhos, para efectuar la E (evaluación) del comando, llamando a la función *process*(input) de la VM. La función *process* reconoce el comando (etapa 2) y lo transforma en un árbol abstracto (etapa 3) para evaluar expresiones y ejecutar acciones en el contexto de las variables existentes en memoria, El resultado de *process*(input) es una cadena de caracteres, que *debe seguir exactamente el formato descrito a continuación*:

Caso 1: si el comando es una expresión válida de Stókhos:

> **OK:** <expresión> **==>** <resultado>

Donde <expresión> es el comando y <resultado> es el resultado calculado por la VM.

Caso 2: si el comando es una acción válida de Stókhos:

> **ACK:** <acción>

Donde <acción> es el comando.

Caso 3: si el comando no es sintácticamente correcto, o no se pudo ejecutar exitosamente:

> **ERROR:** <mensaje>

Donde <mensaje> es un mensaje de error descriptivo.

Ahora bien, el interprete va a estar parcialmente listo en la etapa 3, y completamente implementado en la etapa 4. Siguendo las reglas anteriores (y *deben* seguirlas) el comportamiento del REPL es el siguiente:

```
< Stókhos > 6 * 7
ERROR: interpretación no implementada

< Stókhos > num x := 42;
ERROR: interpretación no implementada
```

Es decir, el REPL honestamente reporta que hay un error, y el error no es culpa del usuario, sino debido a tener una implementación incompleta: la función *process*(input) regresa este mensaje sin importar el input. En esta etapa Uds. no están implementado la interpretación (semántica) de los comandos. Más aún, ni siquiera están implementado el reconocimiento sintáctico de los comandos. Lo único que están haciendo es convertir una secuencia de caracteres (el comando de entrada) en una secuencia de tokens, algunos de ellos con valores asociados. Simplemente están "*tokenizando*" la entrada, lo cual es relativamente simple. Tenemos un problema: ¿cómo vamos a probar el *"tokenizador"* sin tener que implementar el resto del proyecto?

Lo vamos a lograr con una pequeña "trampa" que podemos perdonar 100% por ser una buena práctica de ingeniería de software al desarrollar el proyecto: vamos a añadir una función, *lextest*, a la interfaz de la VM y a la vez agregar un *comando especial* al REPL para llamar dicha función.

4.2 Comandos Especiales

.

Una línea que sólo tiene un punto termina la sesión interactiva: es decir, nos permite salir del REPL. Este es el primer comando especial que deberían implementar: si modifican a process para que haga eco del comando, podrán probar que el REPL está mandando los comandos correctamente y terminar la sesión limpiamente.

.lex <texto>

En vez de pasar la entrada a la función *process*, el REPL llama a *lextest* pasando el <texto> que sigue la palabra clave del comando especial (recuerden: las palabreas claves del REPL ***no*** forman parte del lenguaje Stókhos)
La función, *lextest* simplemente llama a al *lexer* (la función interna que reconoce tokens) y construye una secuencia de tokens de acuerdo con el formato indicado en los ejemplos.

< Stókhos > .lex 6 * 7
OK: lex("6 * 7") ==> [TkNumber(6), TkMult, TkNumber(7)]

< Stókhos > .lex num x := 42;
OK: lex("num x := 42;") ==> [TkNum, TkId("x"), TkAssign, TkNumber (42), TkSemicolon]

< Stókhos > .lex
OK: lex("") ==> [ ]

< Stókhos > .lex ! && || * / + - ;
OK: lex("") ==> [TkNot, TkAnd, TkOr, TkMult, TkDiv, TkPlus, TkMinus, TkSemicolon]

< Stókhos > .lex double malo := @9;
ERROR: caracter inválido ("@") en la entrada

Respeten los formatos! No tener un estandar para la salida del programa dificulta la evaluacion y es penalizado. Los corchetes, espacios en blanco (por ejemplo, luego del ":" y alrededor del "==>") la etiqueta de la respuesta (OK, ACK, ERROR, …) todo esto es importante y debe . Pueden cambian el prompt como les de la gana, pero no lo hagan muy largo. Si consiguen un token que no tiene un identificador sugerido, definan uno Uds. mismos, pero ***respeten los formatos*** usados en los ejemplos si fuesen parte de la sintaxis de un lenguaje.

La mayoría de los comandos especiales (por ejemplo, el de salirse del REPL) no llaman a la VM para nada. Los que lo hacen, no llaman a la función *process* o lo hacen de manera indirecta, como es el caso con **.load**: son "especiales" precisamente porque no siguen la regla general. pero implementan algo de utilidad para el usuario.

Los siguientes comandos especiales deben ser implementados antes de la etapa 2, pero recomendamos hacerlo ahora ya que la entrega siguiente (CFL parsing: reconocimiento libre de contexto) va a ser más compleja.

.load <archivo>

Donde <archivo> es el camino relativo, desde el directorio de ejecución del REPL, a un archivo. Este comando hace que el REPL lea el archivo, línea por línea, procesando cada línea como como si el usuario la hubiese escrito directamente. Este comando especial, como pueden intuir, es indispensable para automatizar pruebas, que eventualmente van a ser programas enteros en Stókhos.

Si un comando devuelve un error (es decir la respuesta comienza con "ERROR: ") el REPL debe tomar nota del siguiente triple:

(<archivo>, <línea>, <mensaje de error>)

donde, como pueden imaginarse, <archivo> es el archivo en el cual se encontró el error, <línea> es la línea donde ocurrió el error, y <mensaje de error> es … bueno, el mensaje de error ☺ - guardar estos triples permite implementar el comando especial siguiente.

.failed

Imprime la lista de errores, uno por línea, usando este formato:

```
[
        (<archivo>, <línea>, <mensaje de error>),
        (<archivo>, <línea>, <mensaje de error>)
]
```

.reset

El comando especial **.reset** va a hacer cosas adicionales en el futuro, incluyendo llamar a la VM, pero por ahora sólo debe limpiar la lista de errores. Ejecutar el comando **.failed** inmediatamente después de un **.reset** siempre retorna una lista vacía.

Otros (pocos) comandos especiales adicionales van a ser agregados en entregas futuras. Algunos pueden ser opcionales, pero en general son de *conveniencia*, para enriquecer ***su*** propio ambiente de pruebas.

Nuestro tiempo es limitado: **¡lean la definición del lenguaje y comiencen el proyecto de una vez!**

Es todo por ahora. Todos les deseamos buena suerte. Nos vemos en clase para aclarar dudas.

Enzo Alda / febrero 15, 2022