DISEÑO DE COMPILADORES

2013

Integrantes

Alvaro Acuña	3.826.062-8
Luciana Canales	3.476.853-5
Gabriel Centurión	2.793.486-8

Índice

1. Introducción

Este documento corresponde a la entrega de Diseño de Compiladores del año 2013.

El objetivo es la implementación de un intérprete de un subconjunto especificado del lenguaje LUA.

2. Análisis léxico

Para realizar el análisis léxico utilizamos la herramienta flex.

3. Análisis sintáctico

Para realizar el análisis sintáctico utilizamos la herramienta bison.

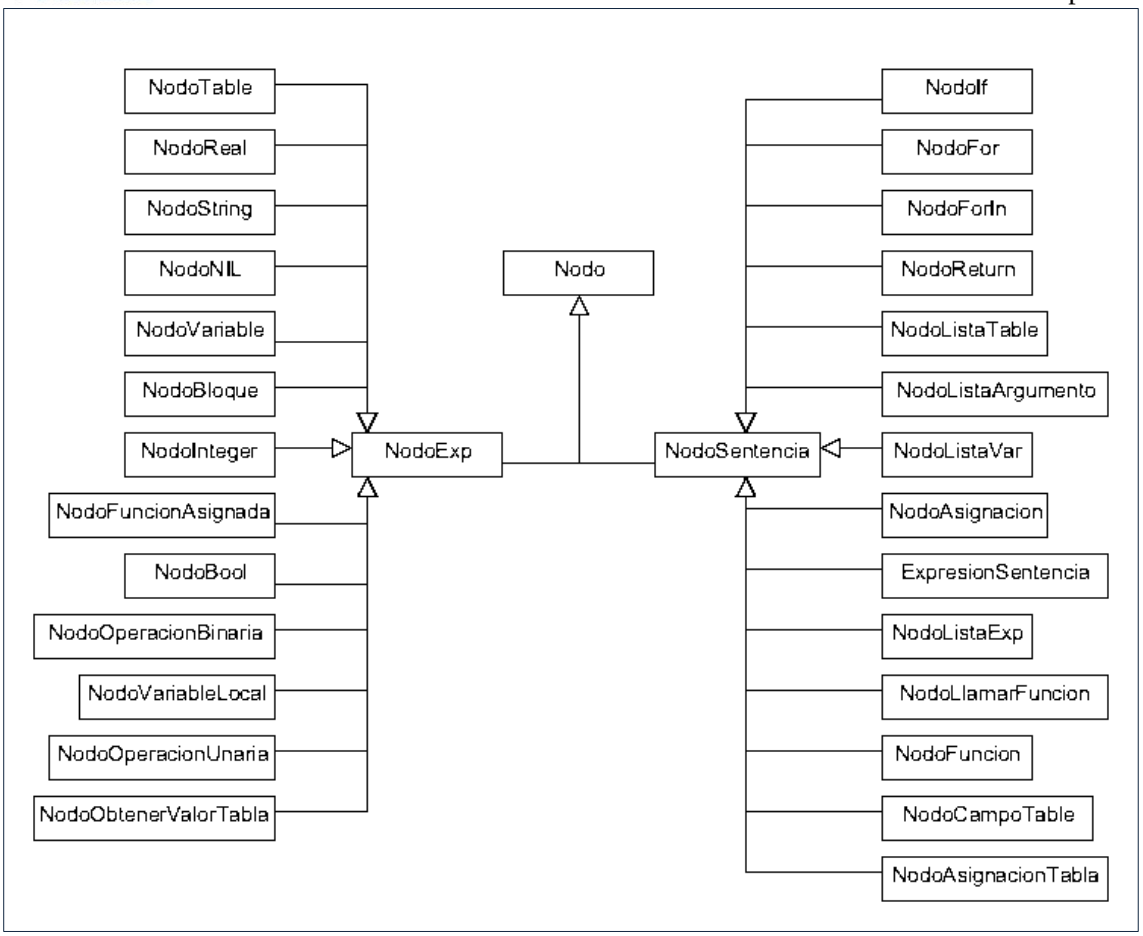
La gramática utilizada es una versión reducida de la gramática que se muestra en la referencia [9] y que cubre todos los casos expuestos en la letra del obligatorio.

4. Estructuras

Para el diseño de las estructuras de datos utilizadas en la construcción del intérprete nos basamos en el ejemplo que se expone en la referencia [1].

Cada nodo del AST cuenta con una estructura que se adecua a cada producción del lenguaje LUA reducido.

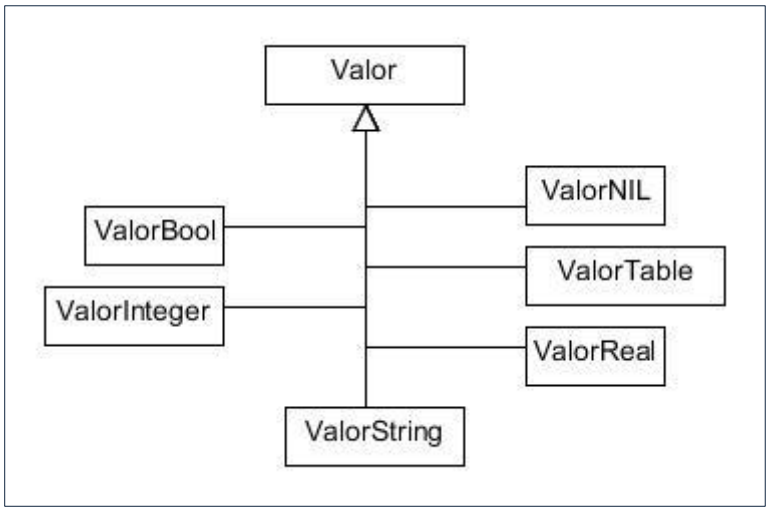
La estructura de los nodos es la siguiente:



Las definición de las clases que representan a los nodos se encuentran en el archivo parser.h.

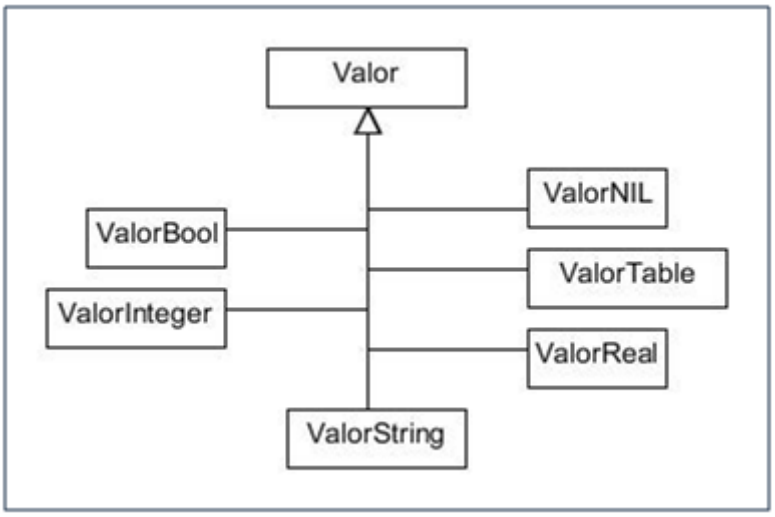
Cada nodo del AST cuenta con la función evaluar, que realiza la acción correspondiente retornando en cada caso un valor.

La estructura de valores es la siguiente:



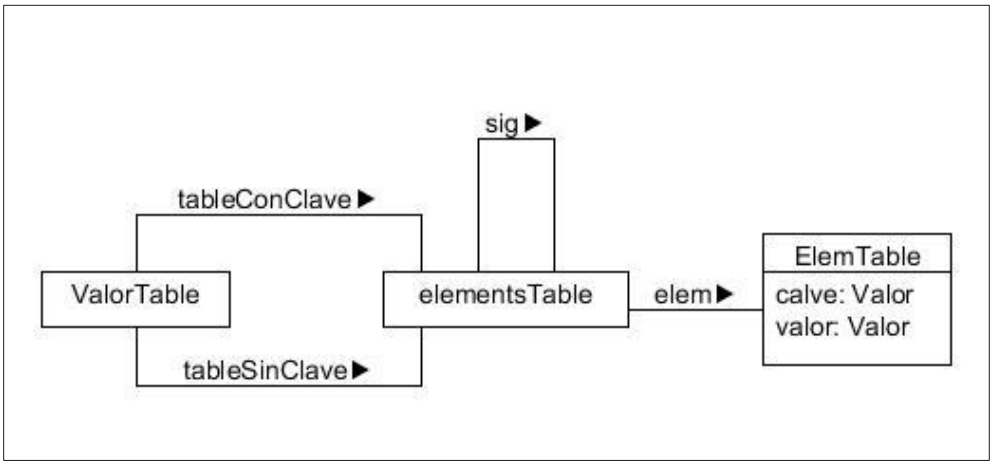
Cada valor representado tiene como atributo un elemento que se corresponde con el tipo de datos.

Para la representación de los tipos de datos se utiliza la siguiente estructura:



4.1 Tablas

En base a la complejidad de sus elementos con clave y sin clave, claves de tipo string, enteras, autogeneradas, etc, la estructura que se decidió usar para representar las tablas de LUA es la siguiente:



Cuenta con dos listas llamadas tableSinClave y tableConClave. La primera mantiene los elementos sin claves y la segunda los elementos con las claves indicadas por el usuario. Para el caso de tableSinClave, elem.clave es nulo y cada vez que ejecuta alguna operación que hace uso de dicho valor este es manejado con un contador que comienza en 1.

4.2 Contexto

La clase contexto es la encargada de definir las estructuras utilizadas para poder mantener el estado de nuestra ejecución. La misma se pasa por parámetro en cada evaluación del nodo del árbol de derivación.

Para localizar el valor de una variable se busca por nombre en el vector de scopes (indicado más abajo) desde el último scope ingresado (scope actual) hasta el inicial. Si no se encuentra en los scopes se realiza la búsqueda en las variables globales, retornando la primer coincidencia de ambos casos.

4.3 Variables globales

Mediante un vector de variables se mantienen las variables globales declaradas. Las mismas pueden ser declaradas en cualquier bloque además del bloque principal.

4.4 Scopes

Se mantienen los scopes a través de un vector de scopes. En la medida que se crea un nuevo bloque se ingresa un nuevo scope que contendrá las variables locales declaradas en dicho bloque. Se toma como referencia al scope actual el último scope ingresado al vector de scopes. Al finalizar el bloque se quita el scope correspondiente al bloque finalizado. Cada scope es un vector de variables locales.

4.5 Funciones

Se registran las funciones en el contexto mediante un vector de NodoFuncion, los mismos alojan la funciona a ser evaluada en caso de llamarse. Las mismas se identifican por su nombre.

5. Manejo de errores

5.1 Léxico

Cuando no se puede llegar a reconocer un token se muestra el mensaje:

> *Error lexico en linea %d columna %d cerca de '%s' \n*

Para saber el número de línea utilizamos la variable yylineno, para saber cerca de que texto se produjo: la variable yytext, y para la columna una variable llamada columna que actualizamos en cada espacio en blanco o carácter.

5.2 Sintáctico

En caso de error sintáctico se llamara a la función "yyerror" de Bison que muestra el siguiente mensaje:

Error Sintactico en la linea %d, cerca de %s \n

5.3 En ejecución

Se realiza al momento de evaluar cada nodo. Mediante un chequeo de los parámetros del nodo y el estado de ejecución se identifican los posibles errores. En caso de existir se aborta la ejecución indicando el error en cuestión y la línea donde se produce el mismo.

Para conocer la línea exacta, en el léxico actualizamos la variable yylloc para cada token, y luego es usada en el sintáctico para crear el nodo con la línea correspondiente.

5.4 Ejemplos

- ERROR: La función fun no existe y fue llamada desde la linea: 2
- Tipo de dato incorrecto en math.sqrt en linea 1
- Argumento #1 invalido en table.insert, se esperaba una tabla en linea 1
- Tipo de datos incorrecto en sentencia FOR cerca de línea: 1

Hay que tener en cuenta que LUA permite una serie de cosas que no son consideradas errores y que tuvimos en cuenta en nuestra implementación por ejemplo:

- Si a una función se la llama con menos parámetros, no es erróneo. Se ponen en nil el resto de los parámetros no pasados.
- Si hay 2 funciones definidas con el mismo nombre no da error, simplemente se sobrescribe.

6. Ejecución del intérprete

Funciona en ambiente Linux. Se entrega un makefile que generará el ejecutable del intérprete realizando make en la carpeta src. Las pruebas se encuentran en src/lua.

-Ir al directorio src
-Ejecutar make
-Ejecutar alguna prueba : ./main lua/testEjemplo.lua

6.1 Pruebas

Los siguientes archivos hacen un repaso de todas las pruebas encontradas en la letra, divididas en cuál es la función específica que se intenta probar. Al ejecutar cada una de ellas se imprime en pantalla los pasos de la misma así como los valores esperados.

Archivo LUA	Descripción
testEjemplo.lua	Prueba el ejemplo de la letra
testTabla1.lua	La idea principal es probar lo métodos insert, remove y sort.
testTabla2.lua	La idea principal es la inicialización de la tabla con la forma t = {"do", "re", ["dore"] = "mefa"} y las asignaciones de la forma t[1]="la", etc.
testTabla3.lua	Ejemplos de la letra acerca de las tablas.
testLocal1.lua	Se testean varios niveles de scopes anidados.
testLocal2.lua	Se testean variables globales y locales en funciones.
testLocal3.lua	Ejemplos que aparecen en la letra relacionados con variable local
testOperador.lua	Se realizan pruebas de: modulo, operadores tradicionales, lógicos y concatenación. De lo que se encuentra en la letra y otros agregados.
testAsignacion.lua	Pruebas de asignaciones: numéricas, texto, tablas, asignación múltiple, switching de variables. Ejemplos de la letra y otras agregadas.
testFor.lua	Se prueban los FOR con diferentes variables de STEP, sin variable STEP (toma 1 por default), se prueba el FOR IN en una tabla, y el uso de las variables dentro del bloque.
testIf.lua	Se prueba el “if” común, el not, el “if name then” donde si name es distinto de nil se evalúa.
testFuncion.lua	Se prueba la asignación de funciones en variables, el llamado con parámetros, con menos parámetros a la función y con más parámetros. El comportamiento es como el de LUA.
testErrores.lua	Prueba de los mensajes de error esperables. Para eso se deben descomentar algunas funciones del código para ir viendo que mensajes de error lanzan.

7. Tener en cuenta

-El comportamiento de las tablas puede que no sea igual al de LUA cuando se prueban tablas con elementos variados, es decir, elementos con clave, sin clave, claves de tipo string y enteras.

Las tablas no fueron pensadas para elementos booleanos tanto en las claves como en los enteros. Es impredecible su comportamiento si se prueban con dichos elementos.

-La división por 0 en LUA no genera errores dejando el valor de la variable como inf. En nuestra solución si lanzamos error de división por 0.

-Los strings en LUA aceptan retrobarra como \n y \t o \", en nuestra solución esto no es así, pues al intentar agregar estos cambios en el sintáctico este comportamiento funcionaba correctamente pero dejaba sin andar otras importantes funciones de la implementación. Balanceamos el impacto de esto y dejamos el reconocimiento de la retrobarra afuera.

8. Referencias

[1] http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/all/1/

[2] http://stackoverflow.com/questions/656703/how-does-flex-support-bison-location-exactly

[3] http://www.fing.edu.uy/inco/cursos/compil/teoricos/05_Flex.pdf

[4] http://www.fing.edu.uy/inco/cursos/compil/teoricos/12_Bison.pdf

[5] http://repl.it/languages/Lua

[6] http://lua-users.org/lists/lua-l/2005-12/msg00091.html

[7] http://inghobbit.blogspot.com/

[8] http://web.eecs.utk.edu/~bvz/cs461/notes/parse_tree/

[9] http://www.lua.org/manual/5.1/es/manual.html#8