Medialuna

Interpretador de Lua

Diseño de Compiladores

Curso 2013

Javier Rey
C.I. 4.549.396-3
javirey@gmail.com

Índice

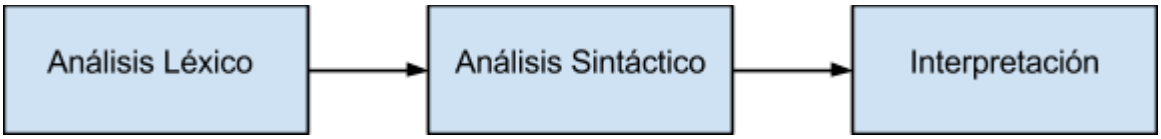
Diseño

Análisis

Para el analisis lexico y sintactico se utilizaron las herramientas FLEX y BISON respectivamente.

Se utilizó el lenguaje C++ para aprovechar las ventajas de OOP y la C++ Standard Library (especialmente std::string y std::vector).

La estructura del intérprete es la siguiente:



Se decidió saltar el paso de código intermedio y interpretar las estructuras de datos que se generan en el análisis sintáctico directamente. Aunque de esta manera se pierde flexibilidad de extender las implementación al abstraer la interpretación de la generación de código, y las posibilidad de optimizar dicho código intermedio, se ganó en velocidad de desarrollo y en las posibilidad de cambiar “en la marcha” cuestiones de diseño que fueron surgiendo.

El análisis sintáctico genera un árbol donde cada nodo es una clase. La raíz es de tipo NBlock y contiene una lista de declaraciones. Cada nodo tiene sus atributos que pueden ser del tipo declaración o del tipo expresión.

Ejecución de código

Para la ejecución de código la idea es tener un stack de bloques en el contexto en los cuales se guardan las variables y funciones definidas.

Si una ejecución quiere utilizar una variable esta se empieza a buscar en el stack de bloques desde el último al primero hasta encontrarse. De forma análoga con las funciones.

El contexto también implementa las primitivas de “math” y “table” además de la función “print”.

Implementación

Generación del AST

El AST se genera durante la etapa del análisis sintáctico.

Todos los nodos del árbol heredan de `NStatement`. Los posibles nodos son de tipo:

- `NBlock`
- `NMultiAssignment`
- `NLastStatement`
- `NIfCond`
- `NIf`
- `NExpressionStatement`
- `NForLoopIn`
- `NForLoopAssign`
- `NFunctionDeclaration`

Estos pueden tener atributos de tipos que heredan de `NStatement` como de `NExpression`, l a clase base para las expresiones.

Todos (excepto `NIfCond` que se utiliza directamente por `NIf`) implementan el método `runCode` que ejecuta, dentro de un contexto el código la declaración y devolviendo una expresión.

Por ejemplo, en el caso de `NBlock::runCode`, se ejecutan las declaraciones secuencialmente y finalmente se retorna la ejecución del `lastStatement` (si es que se tiene).

Ademas de los `NStatement` se generan los `NExpression`. Los posibles `NExpression` son:

- `NNil`
- `NBreak`
- `NBoolean`
- `NInteger`
- `NDouble`
- `NString`
- `NIdentifier`
- `NExpressionList`
- `NFunctionCall`
- `NBinaryOperator`
- `NUnaryOperator`
- `NTableFieldSingleExpression`

- `NTableFieldIdentifier`
- `NTableFieldExpression`
- `NTableFieldList`
- `NTableFieldKey`
- `NTable`
- `NTableExpr`
- `NAnonFunctionDeclaration`

Hay varios de estas clases que solo se utilizan de forma auxiliar al generar el árbol y que luego en la ejecución no se toman en cuenta ya que son reemplazados.

Por ejemplo, en el caso de las "`tables`" aunque se tienen varias clases varias se utilizan todo en el constructor de una "`table`" y son reemplazadas.

Todos estas `NExpression` implementan el método "evaluate" que toma como parámetro el contexto de evaluación. Varios de estos son expresiones que se consideran "finales" ya que la evaluación devuelve el mismo objeto (`NNil`, `NBoolean`, `NInteger`, `NDouble`, `NString`).

Otros valores como `NBreak` no tienen el fin de ser evaluados pero se utilizan de forma intercambiable con el resto. Por ejemplo, una función puede devolver `NExpression` de tipo `NBreak` como puede devolver `NBinaryOperator`, pero obviamente no es la idea evaluar `NBreak`.

Las `NExpression` también implementan el método `type` que devuelve el valor del enum `ExpressionType` que los identifica (a su vez que `type_str` que se utiliza para fácilmente imprimir el tipo en modo debug).

Ejecución del código

Cada vez que se ejecuta un bloque, primero se agrega al stack de bloques del contexto. Luego se corre el bloque (utilizando el método `runCode` que recibe el contexto) y finalmente se hace pop del stack de bloques borrando todas las variables asociadas únicamente al bloque.

Para insertar/sobreescribir funciones o variables al contexto, primero hay que saber si se desea insertar de forma local, es decir, insertar en el último bloque del stack, o si se desea buscar si existe la variable o función y sobreescribirla en su respectiva posición del stack.

En el caso de las funciones o variables definidas explícitamente con la keyword `local`, siempre se inserta como local.

En el caso de ejecutar el bloque de una función o un `for`, antes de correr el bloque se insertan los parámetros de la función o variables del `for` como local.

Antes de insertar una variable ésta siempre es evaluada dentro del contexto que se encuentra, por ejemplo si se llama a la función `is_odd(a)`, se va a insertar la evaluación de "`a`" en el nuevo bloque como una variable de nombre "`a`".

Dependiendo del tipo de declaración que se quiere ejecutar que lógica se realiza. Por ejemplo, en el código de NIf::runCode se itera sobre la lista de condiciones que se tiene, se realiza la evaluación de cada una pasando el resultado a un booleano, y en el caso de tener éxito se hace un nuevo push al stack de bloques y se corre el bloque de la condición verdadera.

Como se pide brevedad en el informe, no se va a describir cómo se realiza en detalle cada ejecución pero se puede ver en detalle en el archivo "node.cpp" la implementación de los métodos runCode.

Evaluación de expresiones

La evaluación de expresiones se realiza de forma análoga a la ejecución de código de las declaraciones. Dependiendo del tipo de expresión se implementa la logica en C++.

Por ejemplo, en el caso de NBinaryOperator, dependiendo de qué operador se utilice, se evalúan las expresiones de la izquierda y de la derecha hasta llegar a expresiones finales y si son expresiones válidas, se realiza la operación en C++ y se devuelve el resultado correspondiente.

En el caso de NTable, este operador solo se evalúa en la construcción de una tabla, y se construye un NTableExpr, cuyos índices y valores son expresiones finales.

Tables

Las `tables`, a partir de ahora tablas, tienen una lógica un poco diferente del resto de las expresiones. Aunque se guardan como variables, al ser variables sus elementos internos se define como `NTableExpr`, que tiene un puntero a una lista de campos del tipo `NTableFieldExpression`.

Aunque en teoría son un `Map`, como las claves pueden ser de valores distintos entre sí, se implementó con una lista para la cual se tienen primitivas de insertar, quitar y ordenar. Aunque de esta manera se pierde en performance ya que para insertar hay que chequear toda la lista si existe, transformando la operación en `O(n)`, se permitió fácilmente emular la funcionalidad sin complicaciones.

Para darle a las tablas la funcionalidad de Lua de índices ordenados, se mantiene internamente un conteo de los índices utilizados.

Librería básica de Lua

Se emula la utilización de la librería math y las funciones print, y pairs, ipairs en el caso del for. Como no está dentro del alcance del intérprete la utilización y creación de librerías, se modificó el parser para que en el caso de encontrar funciones del tipo `IDENTIFICADOR "." IDENTIFICADOR`, se devuelve como un `NIdentifier` cuyo valor tipo `String` sea "iden.iden". Luego, cuando se llama una función, se hace un chequeo si es del tipo "`math`", o "`print`" y en esos casos llamar a las funciones realizadas en C++ que simula la funcionalidad esperada.

La librería `math` no se encuentra 100% completa pero soporta bastante de las funciones posibles utilizando la librería "math.h".

El print puede imprimir un valor como una lista de valores.

Limitaciones

Aunque se implementaron todos los requerimientos según la letra del obligatorio, quedaron muchas cosas pendientes para tener un interpretador más usable de Lua.
Se listan algunas de estas limitaciones a continuación.

- No existe el while
- No existe el repeat
- No existe el terminal “...”
- No se manejan funciones anónimas
- No hay manejo de módulos
- No se implementa la librería estándar de Lua (excepto lo mencionado)
- Solo se utilizan strings de una sola línea.
- Solo se utilizan comentarios de una sola línea (“--”).

Errores conocidos

El orden de recorrida de las tablas queda distinto muchas veces opuesto de como lo hace Lua, de cualquiera manera cuando se realiza el table.sort, el resultado de la recorrida es igual.
Esto se debe a que en realidad Lua maneja los valores de las tablas como un stack, recorriendolo a la inversa.

La operación table.remove funciona de manera inconsistente con respecto al interpretador oficial. Por ejemplo en el siguiente código:

```
t = {"a", "b"}
table.remove(t, 1)
```

Al hacer print de la tabla, el intérprete oficial devuelve:

```
1	b
```

pero el intérprete creado devuelve

```
2	b
```

Esto se debe a cómo Lua maneja las tablas con respecto a cómo se decidió manejarlas.

Manejo de memoria

El manejo de memoria es terrible, hay bastantes memory leaks que habría que buscar y arreglar, pero debido a que la competencia de programación y manejo de memoria en C++ no es parte esencial del enfoque del curso se decidio no darle mucha importancia a este error que en cualquier otra circunstancia es muy grave.

Building & Testing

Se creó un Makefile que crea el binario "`medialuna`" para poder ser utilizado (`./medialuna script.lua`). Para crear el binario basta con ejecutar "`make`".
También se cuenta con la opción "`make test`" en la cual se ejecutan los tests que se encuentran en la carpeta "test" con el ejecutable binario y con el intérprete oficial de Lua y se realiza un diff de los resultados.

Bibliografia

http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/
Fue una gran guía para los primeros pasos.

http://lua-users.org/wiki/LuaGrammar
Importante recurso para la creación de la gramática.

http://www.lua.org/manual/5.1/manual.html