FACULTAD DE INGENIERIA – INSTITUTO DE COMPUTACIÓN

DISEÑO DE COMPILADORES

LABORATORIO FINAL

MATIAS PÉRES 4474045 – GERMAN RUIZ 4317743
JULIO DEL 2013

1 ÍNDICE

2 INTRODUCCIÓN

En el presente trabajo se describe brevemente el proceso de resolución del laboratorio final de la materia Diseño de Compiladores del Instituto de Computación

La realización del trabajo se ha enfocado de manera tal que sirva como guía para el lector, no solo como solución inmediata de los clausulas, sino necesario, relevante, mediante el cual se pueda lograr una comprensión parcial de las cláusulas establecidas y de las soluciones.

Objetivos planteados al inicio del laboratorio: cumplir con las cláusulas, ahondar en la utilización de los conceptos aprendidos, lograr un mejor manejo del entorno de desarrollo y lograr un desarrollo coherente, explicando las decisiones tomadas en las principales secciones del laboratorio.

En el capítulo 3, se describe como se implementó el análisis sintáctico del intérprete. En la sección 3.1 se describe la implementación del lexer. En la sección 3.2 se describe la implementación del parser. Luego en la sección 3.3 se describe la construcción del Árbol Sintáctico Abstracto que es construido durante la etapa de parsing. Intentamos focalizar la mirada en los conceptos tratados más adelante.

En el capítulo 4 se describe la estructura de la solución. Se realizan las observaciones necesarias en cuánto a detalles técnicos, siguiendo las bases establecidas en la teoría e introduciendo cambios y decisiones. Es importante señalar que en ningún momento se introducen cambios ajenos a las cláusulas establecidas por el obligatorio.

El trabajo se encuentra respaldado por material bibliográfico confiable que ha sido seleccionado con atención, a efectos de que la información en la cual nos basemos para la realización del mismo sea certera, lo que brinda constantemente una seguridad en la lectura.

2.1 CLAUSULAS DEL OBLIGATORIO

El laboratorio consiste en la construcción de un intérprete del lenguaje LUA. Este deberá tomar un programa escrito en LUA y ejecutar dicho código.

Se deberá realizar un tratamiento mínimo de errores indicando número de línea, último token procesado y tipo de error. Se deberá además implementar

un ambiente de ejecución apropiado para permitir ejecutar los programas LUA que se puedan construir según este obligatorio. Entre otras cosas, este ambiente de ejecución regulara el llamado a funciones, la gestión de memoria, etc.

La ejecución de los programas se realizara en forma similar a como se realiza con el interprete estándar. Esto es, si el nombre del programa desarrollado es **NOMBRE_BINARIO_DE_INTERPRETE**, entonces la ejecución será realizada de esta forma: **NOMBRE_BINARIO_DE_INTERPRETE archivo.lua**

3 ANÁLISIS SINTÁCTICO

Durante el análisis sintáctico el programa fuente es parseado. El parser verifica que no haya errores en la estructura del programa fuente. Esta fase puede ser dividida en dos sub-fases. La primer sub-fase es el lexer. El lexer agrupa el flujo de caracteres en unidades llamadas tokens. Los tokens son luego procesados por la segunda sub-fase del análisis sintáctico: el parser. Este capítulo describe la estructura sintáctica de Lua y como fue implementado el análisis sintáctico.

3.1 EL LEXER

La tarea del lexer es leer el archivo fuente para producir un flujo de tokens. El flujo de tokens será usado por el parser para el análisis sintáctico. El lexer además remueve espacios en blanco y comentarios. De esta forma, el parser puede asumir que todos los tokens son 'válidos', lo que facilita la construcción del parser.

3.1.1 Especificando los Tokens

Para que el lexer pueda reconocer grupos de caracteres como tokens es necesario establecer algunas reglas. No cualquier grupo de caracteres conforman un token valido. Por ejemplo, en muchos lenguajes de programación los identificadores de variables deben comenzar con una letra (no con un dígito). Estas reglas se pueden expresar fácilmente utilizando expresiones regulares, una manera formal de describir conjuntos de cadenas de caracteres.

3.1.2 Construyendo el Lexer

El lexer fue generado automáticamente utilizando el programa Flex. Flex toma un archivo de entrada en un formato específico y genera código C que puede ser linkeado al programa principal. Cuando el lexer reconoce un token, dependiendo del tipo de token realiza una de dos cosas. Si se reconoció una palabra clave simplemente se retorna el tipo de token. Por ejemplo, si se reconoció la palabra clave *while* se retorna un código que representa esa palabra clave. Por otro lado, si el lexer reconoció un identificador, primero guarda el valor actual del identificador en una variable a la que el parser pueda acceder, y luego retorna un código indicando que reconoció un identificador. De la misma for ocurre con los números.

Puede consultar más información de Flex en [1].

3.2 EL PARSER

El parser es responsable de encontrar la estructura sintáctica del programa fuente. Cada sentencia del programa fuente consiste de una serie de tokens que son entregados por el lexer. Mientras se reconoce la estructura sintáctica del programa, se construye una representación interna que es procesada luego por fases posteriores del intérprete.

La estructura sintáctica de un lenguaje de programación puede ser especificada por una gramática libre de contexto.

3.2.1 Construyendo el Parser

El parser fue generado con la herramienta Bison, un reemplazo GNU para YACC. El archivo procesador por Bison es una especificación de la gramática en un lenguaje particular. Para que un compilador sea útil, el parser debe hacer algo más que tan solo parsear el programa fuente. En Bison, una regla gramatical puede tener una acción semántica asociada. Cuando se reduce un handle, la acción semántica asociada a la regla es ejecutada. En el laboratorio las acciones semánticas construyen el Árbol Sintáctico Abstracto o Abstract Syntax Tree.

Para construir el parser, las reglas de producción son ingresadas en un formato que Bison acepta. El resultado es una salida en código C que representa el parser. El parser puede ser luego invocado llamando a la función *yyparse()*.

Puede consultar más información de Flex en [2].

3.3 ABSTRACT SYNTAX TREE

El compilador de Lua construye un Abstract Syntax Tree durante la fase de parsing. La estructura del AST corresponde a la estructura sintáctica de Lua. Por ejemplo, una sentencia de asignación se representa con un nodo de tipo asignación con dos hijos: un primer hijo con el lado izquierdo de la asignación, es decir con una lista de variables, y un segundo hijo con el lado derecho de la asignación, con una lista de expresiones. Tenemos por tanto, para cada tipo de sentencia y expresión un tipo especial de nodo.

3.3.1 Tree Compiler Compiler

En una primera instancia se estudió la posibilidad de utilizar la herramienta Tree Compiler Compiler (TreeCC) diseñada para asistir en el desarrollo de

compiladores y otras herramientas para lenguajes. Más específicamente, TreeCC produce el código necesario para crear el AST y detectar errores en una etapa temprana del desarrollo del compilador. Sin embargo, su uso ha sido descartado por la escasa información disponible en relación a su funcionamiento. Es además una herramienta muy poco difundida por lo que carece de soporte comunitario.

Puede consultar más información de Flex en [3].

3.3.2 Construyendo el AST

Anteriormente se mencionaron las acciones semánticas asociadas a producciones. Veamos un ejemplo de cómo se crean los nodos dentro del parser:

```
stmt    : var_list '=' exp_list
        {
                $$ = new ast(_asign, yylloc.first_line, $1, $3);
        }
        | function_call
        {
                ....
        }
```

El símbolo *$$* representa el valor de *stmt* que es especificado en Bison como un puntero a la estructura *ast*. *$1* y *$3* representan los valores de *var_list* y *exp_list* ambos punteros a *ast*. El nodo es creado especificando el tipo *_asign* indicando que se trata de una asignación. El nodo se crea además con la información para el majo de errores (número de línea en la que se aplica la acción semántica) y los hijos correspondientes.

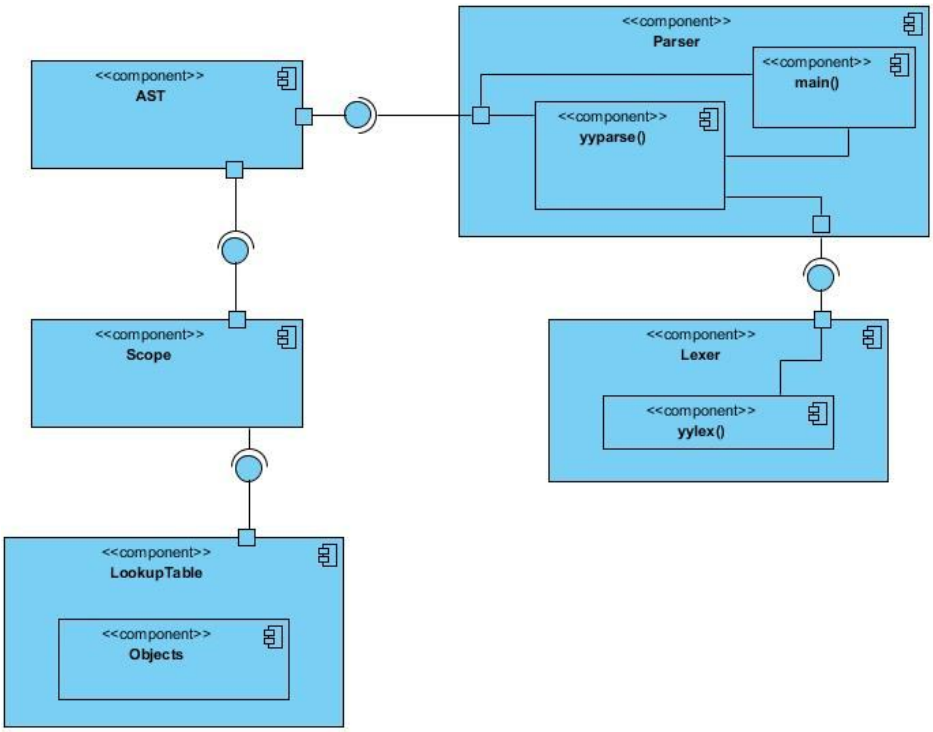
3.4 ERRORES DE SINTAXIS

Reportar errores es una cuestión importante en el diseño de un compilador. Esto es porque el compilador no solo compilará programas correctos. Sino que la mayoría de los que procese serán incorrectos. El parser generado por Bison automáticamente tiene una función de error que es llamada cuando se descubre un error de sintaxis. Durante el parsing se actualizan variable globales con información de la línea actual y posición de token en la línea. De esta forma cuando se descubre un error de sintaxis se imprime un mensaje de error indicando línea del error y en que parte de la línea se descubrió el error.

4 ESTRUCTURA DE LA SOLUCIÓN

Para la presentación de la estructura se ha optado por un diagrama de componentes y conectores. Cada elemento del diagrama, como es usual en un diagrama de componentes, se manifiesta durante la ejecución del programa (como ser objetos o librerías), consume recursos y contribuye con el comportamiento del sistema.

En la siguiente imagen se muestra el diagrama realizado. El diagrama muestra instancias, no tipos. Su realización contribuyo al desarrollo del obligatorio, ayudando a visualizar el camino de la implementación, permitiendo tomar decisiones tempranas. Se debe tener en cuenta que se trata de un diagrama primitivo, cuya semántica puede no estar perfectamente definida. En particular, esta vista ayuda a responder la pregunta de cuáles son las entidades más importantes durante la ejecución del compilador y cómo interactúan.



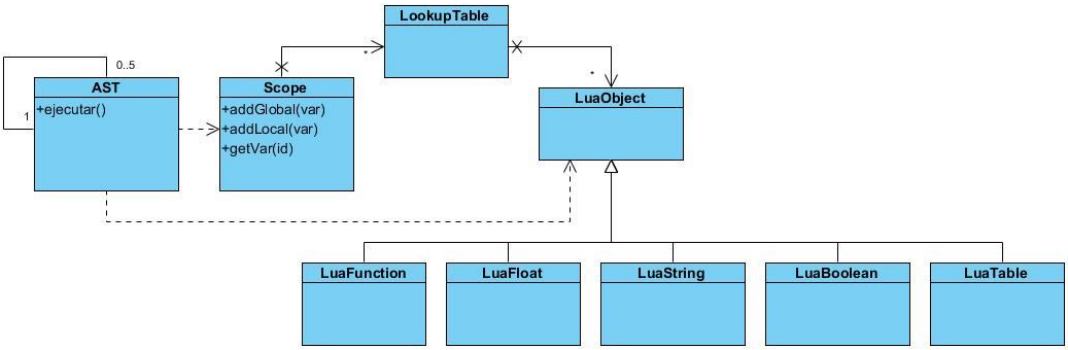
A simple vista podemos notar la presencia de dos componentes importantes, Parser y Lexer. Parser es el componente principal del intérprete, pues contiene la función *main()* que ejecuta la mencionada *yyparse()* obteniendo como resultado el Abstract Syntax Tree.

El componente AST provee todas las operaciones necesarias para la construcción del árbol sintáctico y su ejecución. Una vez obtenido el árbol, este

se puede ejecutar llamando a la función *ejecutar(ast*)* provista por el componente AST.

Durante la ejecución se deben utilizar estructuras que permitan realizar controles de ámbito o *scope*. Para ello se dispone del componente Scope encargado de manejar la declaración de variables globales y locales. Notar como este componente requiere a su vez de las funciones provistas por el componente LookupTable el cual representa 'a grandes rasgos' un scope en particular. Cada LookupTable contiene los objetos declarados en ese scope. No olvidemos que Lua no requiere de la declaración de variables para su uso.

Un primer modelo de dominio de la solución implico la tarea de discernir el mapeo entre los componentes presentados a entidades del sistema. Sin embargo, este mapeo no es directo, y un componente puede ser implementado por una o más clases / objetos del sistema, así como varios componentes pueden ser implementados por una clase / objeto. Una vez discutidas las decisiones analíticas de la solución se procedió a diseñar un esquema que soportara la estructura presentada. La siguiente imagen muestra un diagrama de clases de la aplicación:



La clase Scope utiliza el patrón *Singleton* y el mapeo componente - clase es directo. La clase Scope mantiene un *stack* de *LookupTables* que representan la anidación de ámbitos. Cuando se ingresa a un bloque, se hace push en el vector de LookupTables. Al final del bloque, se hace pop del stack. La búsqueda de una variable comienza en el tope del stack y va descendiendo hasta la base. Si no se encuentra una variable con el identificador señalado entonces se devuelve un *LuaObject* que representa el valor nulo.

AST provee la operación *ejecutar(ast*)* como mencionamos anteriormente, entre otras operaciones para el manejo del árbol sintáctico.

Un aspecto a señalar es que al momento de manejar expresiones, estas son 'atomizadas' por una función provista por Scope. La utilización de clases y herencia en el caso de las clases que representan los objetos de Lua, facilita notablemente la ejecución de operaciones entre distintos tipos de objetos, permitiendo que los objetos sean tratados sin distinción de tipo, tal como ocurre en el lenguaje Lua. El resultado es una consecuencia directa del polimorfismo. Las operaciones no sobrecargadas resultan en una excepción de tipos que es atrapada para imprimir el error correspondiente.

5 ESTRUCTURA SINTÁCTICA DE LUA

```
chunk: block

block: stmt_list stmt_end

stmt_list:      /* vacío */
                | stmt_list stmt opt_semicolon

stmt_end:       /* vacío */
                | RETURN opt_exp_list opt_semicolon
                | BREAK opt_semicolon

opt_semicolon:      /* vacío */
                    | ';'

stmt:           var_list '=' exp_list
                | function_call
                | DO block END
                | WHILE exp DO block END
                | REPEAT block UNTIL exp
                | IF exp THEN block opt_elseif_list opt_else END
                | FOR for_stmt
                | FUNCTION function_name function_body
                | LOCAL FUNCTION IDENTIFICADOR function_body
                | LOCAL name_list initializer

name_list:      IDENTIFICADOR
                | name_list ',' IDENTIFICADOR

initializer:    /* vacío */
                | '=' exp_list

var_list:       var
                | var_list ',' var

exp_list:       exp
                | exp_list ',' exp
```

```
opt_exp_list: /* vacío */
            | exp_list

opt_elseif_list:    /* vacío */
                    | opt_elseif_list ELSEIF exp THEN block

opt_else:   /* vacío */
            | ELSE block

for_stmt:   IDENTIFICADOR '=' exp ',' exp opt_comma_expr DO block END
            | name_list IN exp_list DO block END

opt_comma_expr:  /* vacío */
                 | ',' exp

var:  IDENTIFICADOR
      | prefixexp index

index: '[' exp ']'
      | '.' IDENTIFICADOR

exp:  NIL
      | FALSE
      | TRUE
      | NUMERO
      | STRING
      | PUNTOS
      | FUNCTION function_body
      | prefixexp
      | table_constructor
      | binop
      | unop

prefixexp:  var
            | function_call
            | '(' exp ')'

function_name: dotted_name_list opt_colon_name

dotted_name_list:  IDENTIFICADOR
```

```
                    | dotted_name_list '.' IDENTIFICADOR

opt_colon_name:     /* vacío */
                    | ':' IDENTIFICADOR

function_body: '(' param_list ')' block END

function_call:      prefixexp args
                    | prefixexp ':' IDENTIFICADOR args

args:   '(' opt_exp_list ')'
        | table_constructor
        | STRING

param_list:     /* vacío */
                | PUNTOS
                | name_list
                | name_list ',' PUNTOS

table_constructor: '{' table_contents '}'

table_contents:     /* vacío */
                    | field_list opt_field_sep

field_list:     field
                | field_list fieldsep field

field:  '[' exp ']' '=' exp
        | IDENTIFICADOR '=' exp
        | exp

opt_field_sep:      /* vacío */
                    | fieldsep

fieldsep:       ','
                | ';'

binop: exp OR exp
        | exp AND exp
        | exp '<' exp
```

```
        | exp '>' exp
        | exp MENORIGUAL exp
        | exp MAYORIGUAL exp
        | exp DISTINTO exp
        | exp IGUALIGUAL exp
        | exp PUNTOPUNTO exp
        | exp '+' exp
        | exp '-' exp
        | exp '*' exp
        | exp '/' exp
        | exp '%' exp
        | exp '^' exp

unop: NOT exp
        | '#' exp
        | '-' exp
```

6 CONCLUSIÓN

Para finalizar el informe sobre la realización del laboratorio final, es importante mencionar los objetivos cumplidos.

El proceso hacia la formalización, ha tenido como propósito, según lo mencionado en ocasiones anteriores, ser coherentes en el desarrollo del informe, a diferencia de las ocasionales contradicciones que pueden encontrarse en piezas de información varias. Se pretendió fijar la atención en aspectos importantes y esenciales para la comprensión del desarrollo.

Durante la preparación del trabajo los cambios y variantes, tanto de detalles como de perspectiva se hicieron numerosos y de múltiples estratos, sin embargo, en busca de una lectura confiable y coherente, se basaron las notas, observaciones y conclusiones en las decisiones finales. Se expusieron imágenes a modo de esclarecer la situación y agilizar la lectura.

Debemos mencionar que las decisiones tomadas a lo largo de la implementación surgieron como respuesta a los resultados obtenidos hasta el momento.

No resulta fácil exponer una síntesis final que valga como resumen de un laboratorio de tan largo proceso y aspectos tan diversos. Reconocemos los riesgos que se derivan de toda investigación cuyo objeto suponga abarcar más de las pautas planteadas. Sin embargo, debemos admitir que ha sido, una actividad integradora, enriquecedora desde el punto de vista técnico y didáctico. El ejercicio o adiestramiento en la utilización de las herramientas estudiadas es fundamental para comprender mejor los postulados teóricos y creemos son el objetivo de las cláusulas en el laboratorio.

7 BIBLIOGRAFÍA

[1] http://flex.sourceforge.net/

[2] http://www.gnu.org/software/bison/

[3] http://www.gnu.org/software/dotgnu/treecc/treecc.html

[4] http://www.lua.org/manual/5.1/es/manual.html

[5] http://lua-users.org/wiki/ScopeTutorial

[6] http://bytes.com/topic/c/answers/166621-compile-flex-bison

[7] http://www.oracle.com/technetwork/articles/servers-storage-dev/mixingcandcpluspluscode-305840.html#cpp_from_c

[8] http://staff.lero.ie/stol/files/2011/12/stol2003.pdf

[9] https://github.com/jhays200/C---Lua-Interpreter

[10] http://lua.lickert.net/syntax/index_en.html