Trabajo Práctico Obligatorio
Diseño de Compiladores
2013

Objetivos

El objetivo de este trabajo obligatorio, es la construcción de un intérprete del lenguaje LUA. Este deberá tomar un programa escrito en LUA y ejecutar dicho código. A continuación se presenta el lenguaje en cuestión, luego se muestran ciertos aspectos sobre la forma de ejecutar los programas y por ultimo se presentan aspectos más formales sobre este obligatorio.

Un ejemplo…

```
function get_all_factors(number)
  --[[--
    Gets all of the factors of a given number
    @Parameter: number
      The number to find the factors of
    @Returns: A table of factors of the number
  --]]--

  local factors = {}
  for possible_factor=1, math.sqrt(number), 1 do
    local remainder = number%possible_factor

    if remainder == 0 then
      local factor, factor_pair =
                  possible_factor, number/possible_factor
      table.insert(factors, factor)
      if factor ~= factor_pair then
        table.insert(factors, factor_pair)
      end
    end
  end

  table.sort(factors)

  return factors
end

-- The Meaning of the Universe is 42.

the_universe = 42
factors_of_the_universe = get_all_factors(the_universe)
```

El resultado el ejecutar lo anterior es…

Paso a paso

Veremos paso a paso el ejemplo anterior, el cual reúne las características necesarias para este intérprete de LUA

Definición de funciones

```
function get_all_factors(number)
        ...
end
```

En LUA, una función puede recibir cualquier tipo de datos, y devolver cualquier tipo de datos. Definimos una función usando la palabra clave “function”, seguida del nombre de la función, el cual debe cumplir con las normas típicas de un identificador en un lenguaje de programación

A continuación del nombre y entre paréntesis, debemos colocar los nombres de los parámetros esperados. Podemos pasar desde 0, 1 o múltiples parámetros.

```
function someFunction(parameter, anotherParameter)
    --Valid
end
```

También debemos cerrar la función con la palabra clave “end”. En LUA, todo lo que esta entre el paréntesis de cierre de los parámetros y la palabra clave “end”, se denomina “chunk”, el cual consiste en el código propiamente dicho de la función.

Algunos ejemplos de funciones invalidas:

```
function some Function(parameter, anotherParameter)
    -- Not Valid
end

Function someFunction(parameter, anotherParameter)
    -- Not Valid
end

function someFunction(parameter anotherParameter)
    -- Not Valid
end

function someFunction(parameter, anotherParameter
    -- Not Valid (missing a ')')
End
```

Comentarios

```
--[[--
Gets all of the factors of a given number
```

```
@Parameter: number
        The number to find the factors of

@Returns: A table of factors of the number
--]]--
```

En LUA, dos guiones, uno a continuación de otro significa un comentario de una línea. Los comentarios no se parsean, quedando solo para propósitos de legibilidad en el código. Para generar comentarios de múltiples líneas, colocamos al comienzo de las líneas --[[y al final de las líneas]].

Variables

En LUA los datos podemos almacenarlos en variables, las cuales se identifican por nombre, utilizando una única palabra. Algunos ejemplos:

Una variable numérica:

- CanIHazANomNom = 1000

Una variable de texto:

- GIMMEH = "HAI WORLD"

Una variable con una tabla. Una tabla es una lista de objetos entre { }

- BunchOfText = {"I Do Not Like Them Sam I Am", "I Do Not Like Green Eggs and Ham", "I Would Not Like Them Here or There", "I Would Not Like Them Anywhere"}

Una variable booleana

- NoYouMoron = false

Copiando una variable

- AreWeGonnaDieIn2012 = NoYouMoron

Switching de valores. Si el primer valor es nulo, se asigna el Segundo

- Apocalypse = AreWeGonnaDieIn2012 or "2012's a joke."

Asignando nulo

- Nothing = nil

Asignando una function (las funciones son asignables)

- someFunction = function(parameter, anotherParameter) end

Podemos asignar variables de diferente tipo

- Nothing = GIMMEH

Asignación de múltiples variables

- A, B, C, D = "a", "b", "oops I took both c and d"

Podemos crear una variable con el prefijo “local”. Una variable con el prefijo local solo puede ser usada en el chunk/bloque en el que fue declarada

- local factors = {}

Por ejemplo:

```
function GreenEggsAndHam()
        local Sam = "I Am Sam"
end

print(Sam)
```

Este código imprime “nil”, ya que la variable Sam no existe fuera de la función indicada. Si una variable local se declara dentro de un chunk, opacando una variable global, los cambios que hagamos en la local, no afectan la global.

```
Sam = "Sam I Am"
function GreenEggsAndHam()
        local Sam = "I Am Sam"
end

print(Sam)
```

En este caso, se imprime “Sam I Am”, ya que la variable local no afecto a la global.

For Loops

```
MAXIMUM = 10
STEP = 1
for the_number = 1, MAXIMUM, STEP do
        print("The Number is now ",the_number)
end
```

```
The Number is now       1
The Number is now       2
The Number is now       3
The Number is now       4
The Number is now       5
The Number is now       6
The Number is now       7
The Number is now       8
The Number is now       9
The Number is now       10
```

La sintaxis para un loop de tipo for es la siguiente:

- for variable=start, maximum (variable < maximum), increment do

Cada vez que se ejecuta el loop, la variable es incrementada según el valor de increment. Este incremento es opcional. Si no lo especificamos, se asume el valor 1.

Condicionales

Están representados en LUA por sentencias de tipo if, como por ejemplo:

- if true then print("Hi Mom") end

La ejecución del código dentro del Then, ejecutara si la condición booleana NO EVALUA en false o nil. Esto es útil por ejemplo para testear si una variable existe.

```
function SayHiTo(name)
    if name then
        print("Hello",name)
    end
end

SayHiTo("Jack")
SayHiTo()

Hello Jack
```

Tambien podemos utilizar if-then-else, con el siguiente formato

```
function SayHiTo(name)
    if name then
        print("Hello",name)
    else
        print("Hello Stranger")
    end
end
```

Para el caso de las expresiones booleanas, tenemos el operador not, el cual niega el valor booleano de una expresión.

```
function SayHiTo(name)
    if not name then
        print("Hello Stranger")
    else
        print("Hello", name)
    end
end
```

Podemos usar los operadores == para testear igualdad, y ~= para testear desigualdad

```
function I_want_to_find(the_number)
    for a_number = 1, 100 do
```

```
        if the_number == a_number then
            print ("I found him!")
        else
            print ("Sorry, he's not here.")
        end
    end
end
I_want_to_find(50)
I_want_to_find(0)

I found him!
Sorry, he's not here.
```

Operadores

LUA soporta los operadores tradicionales que podemos encontrar en otros lenguajes. A continuación veremos algunos:

Modulo

- print(14 % 5) -- Outputs 4

Operadores tradicionales: + - * / ^

- Solo funcionan con números, por ejemplo: print(2+2)
- print("Hello"+2010) genera un error

Concatenación

- Se utiliza el operador ..
- print("I.."am".."Sam"), imprime “IamSam”

Largo

- Se utiliza el operador #
- Solo aplica a tablas o Strings
- bag_of_stuff = {"do", "re", "me", "fa", "so", "la", "si"}
- print("I haz "..#bag_of_stuff.." things")

Asignacion

- Se utiliza el operador “=”
- me = "me"

Lógicos

- Se soportan los operadores lógicos tradicionales, **and**, **or** y **not**

Relacionales

- Se soportan los operadores tradicionales: >, >=, <, <=

Más información sobre el uso de los operadores, puede encontrarse en el siguiente sitio:}

- http://lua.lickert.net/operators/index_en.html

Tablas

Son equivalente a arrays/mapas de los lenguajes tradicionales. Son creadas usando el constructor {}

```
t = {}    -- construct an empty table and assign it to variable "t"
print(t)
table: 0035AE18
```

Accedemos a la información de la tabla usando una sintaxis de la forma tabla[clave].

```
> t = {}
> t["foo"] = 123 -- assign the value 123 to the key "foo" in the table
> t[3] = "bar" -- assign the value "bar" to the key 3 in the table
> = t["foo"]
123
> = t[3]
bar
```

Si no existe un valor asociado a una clave, no se da error, sino que se devuelve nil. Cualquier valor puede ser usado como clave, siempre y cuando no sea nil.

También existen otras formas de inicializar el contenido de una tabla:

```
> t = {["foo"] = "bar", [123] = 456}
> = t.foo
bar
> = t[123]
456

> t = {foo = "bar"} -- same as ["foo"]="bar" (but not [foo]="bar" ,
that would use the variable foo)
> = t["foo"]
bar
```

Para iterar en las parejas de una tabla, usamos la función pairs(t)

```
> t = {foo = "bar", [123] = 456}
> for key,value in pairs(t) do print(key,value) end
foo bar
123 456
```

Un array es un caso particular de tabla, en la cual los elementos utilizan como clave su índice.

```
> t = {"a", "b", "c"}
> = t[1]
```

```
a
> = t[3]
c

> t = {[1]="a", [2]="b", [3]="c"}
> = t[1]
a
> = t[3]
c
```

También podemos insertar y borrar de una tabla, usando las funciones table.insert y table.remove

```
> t = {"a", "c"}
> table.insert(t, 2, "b")
> = t[1], t[2], t[3]
a b c

> t = {"a", "b", "c"}
> table.remove(t, 2)
> = t[1], t[2]
a c
```

Tratamiento de errores

El requerimiento mínimo es terminar el proceso de traducción una vez que se detecta un error indicando el numero de línea, el ultimo token procesado y de ser posible el tipo de error (sintáctico, de tipo, etc.).

Son aceptables soluciones en los que los mensajes de error son de la forma:

`Error sintáctico en línea 2 cerca de "22"`

`Tipo de datos incorrecto en sentencia IF en línea 10`

Mecanismos para el manejo de error mas elaborados, serán tenidos en cuenta, pero se sugiere comenzar con un manejo mínimo del error, y luego expandirlo si se tiene tiempo disponible.

Ambiente de ejecución

Deberá 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.

Ejecución de los programas LUA

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`

Objetivos, formas y plazos de entrega

Algunos aspectos formales a tener en cuenta…

Sobre el objetivo

El objetivo principal del obligatorio es la entrega de un intérprete funcional **(TIENE QUE FUNCIONAR)** del lenguaje, descripto en esta letra. Las características del lenguaje que deben ser tenidas en cuenta para dicho intérprete son las presentadas en este documento. Toda otra característica que no este en este documento, se considera opcional (a menos que sea aclarada su obligatoriedad en la pagina web del curso)

Sobre la forma de trabajo

El trabajo deberá ser realizado en grupos de hasta tres estudiantes. Los mismos deberán ser informados antes del 9/6 al docente encargado del curso. El día 10/6 será informada la lista final de grupos del obligatorio. Se realizaran clases de consulta todos los miércoles hasta la fecha de entrega, en el horario de 8:30 a 9:30, en el salón 301.

IMPORTANTE: Una vez formado un grupo, no se aceptaran separaciones o divisiones en dicho grupo. En caso de darse esta situación, TODOS los integrantes del curso pierden el obligatorio. En caso de que parte de los integrantes abandonen, solo a estos se les considera como no aprobado el curso.

Sobre la plataforma de trabajo

El trabajo deberá ser realizado utilizando el lenguaje C o C++. Se podrán (y se recomienda) utilizar los utilitarios BISON y FLEX para realizar el análisis sintáctico y léxico respectivamente. En caso de ser necesario, se podrá utilizar código descargado de Internet (debidamente documentado), siempre y cuando este NO RESUELVA una parte fundamental del problema planteado. Por ejemplo, no seria aceptable si se entrega un intérprete desarrollado en C++ de LUA, descargado de Internet.

Se puede utilizar plataforma Windows o Linux para trabajar. Se recomienda el uso de plataforma Linux (Ubuntu) por su disponibilidad de herramientas. Sin embargo, queda a elección del grupo la plataforma de trabajo

MUY IMPORTANTE:

Se deberá entregar el código fuente de la solución, así como los scripts necesarios para compilar y obtener una versión funcional del intérprete. El grupo se deberá asegurar que el código fuente se puede compilar, ya que SOLO se ejecutara el intérprete obtenido de la compilación de los fuentes entregados.

No se aceptaran soluciones que utilicen archivos propietarios de proyectos desarrollados en IDEs como Visual Studio, DJGPP, CodeBlocks, etc. El grupo deberá asegurarse que el código se puede compilar sin necesidad de contar con dichas IDEs

Se recomienda entonces entregar el fuente, junto con un makefile que genere el binario del intérprete.

Sobre la entrega

La fecha de entrega final para este obligatorio es el lunes 15 de julio del 2013 (inclusive)

Se deberá entregar un sobre con los datos del grupo, como ser nombres y C.I. de los integrantes. El sobre deberá contener:

- Documentación del obligatorio (breve) presentando los aspectos principales del diseño de la gramática, chequeos, las tablas de símbolos, la representación intermedia, tratamiento de error y el proceso de interpretación. Se espera, muy especialmente, que fundamenten adecuadamente las decisiones de diseño tomadas a lo largo del obligatorio. No se desea un listado del código fuente del intérprete

- Un listado de las pruebas ejecutadas por el grupo sobre el intérprete. En caso de las pruebas que contienen error, indicar los errores obtenidos. En caso de que haya un problema conocido con la ejecución de los programas de prueba entregados, esto deberá ser documentado apropiadamente

- Un cd con los dos puntos anteriores, y el código fuente del interprete

Se aclarara oportunamente en la página web el lugar donde deberá ser entregado este sobre.

Sobre la evaluación

Se evaluaran los siguientes puntos:

- Correcto funcionamiento del intérprete, ejecutando los programas de prueba publicados en la página del curso, y con otros programas de prueba que se consideren necesarios. **SI EL INTERPRETE NO FUNCIONA CORRECTAMENTE, EL CURSO SE CONSIDERA PERDIDO PARA TODOS LOS INTEGRANTES**
- Calidad en la implementación. **SOLO SI EL INTERPRETE FUNCIONA CORRECTAMENTE**, se evaluaran como puntos extra, la calidad con que sea haya implementado la solución, las features extra que se le hayan agregado al intérprete, las mejoras al manejo e informe de errores, etc.
- En caso de duda por parte del docente sobre la implementación del interprete, los integrantes del grupo (en su totalidad o parcialmente) podrán ser llamados a una defensa, en la cual se les realice preguntas concretas sobre la implementación realizada. **EN CASO DE COMPROBARSE QUE UN MIEMBRO DEL EQUIPO NO TRABAJO EN EL OBLIGATORIO, EL CURSO SE CONSIDERA PERDIDO PARA DICHO INTEGRANTE**
- El puntaje máximo del obligatorio es de 100 puntos. Se aprueba con el 60% de los puntos.

Sobre las copias

Para resolver este tema, se utilizara el mecanismo tradicional aplicado en el InCo cuando se detectan copias entre los grupos**. TODOS LOS INVOLUCRADOS PIERDEN EL CURSO** y se informa apropiadamente a la Facultad sobre esta situación.