Obligatorio de Compiladores 2012

Gustavo Evovlockas CI: 4.312.777-2 mail: gevovlockas@gmail.com
Andreas Fast CI: 3.890.167-8 mail: andis.machine@gmail.com
Martin Varela CI: 3.598.224-7 mail: martinvarela85@gmail.com

Introducción

El obligatorio planteado pretende implementar un intérprete de Ruby con características básicas. Dichas características se refieren a

1. Manejo de datos de tipo entero, flotante, String, Array y booleano.
2. Manejo de variables
3. Manejo de scope
4. Manejo de bibliotecas
5. Manejo de Clases y objetos.
6. Manejo de procedimientos y funciones.
7. Uso de bloques, condicionales e iteraciones.

Para el comportamiento de todos los puntos nos basamos en lo indicado por la letra del obligatorio y ante la duda utilizamos el mismo criterio usado por Ruby.

Implementación

Para su implementación se trabajó sobre Ubuntu 12.04, se trabajó con Flex 2.5.35 y Bison 2.4.1. Utilizamos gcc 4.6.3 para compilar. Cabe destacar que utilizamos una estructura considerada experimental llamada `unordered_map`, que funciona como un hash. Para poder utilizarla se debe indicar al compilador con la flag `-std=c++0x`. Hicimos uso de los contenedores que brinda c++ para manejar listas, stack y hash.

Compilar

Para compilar el proyecto basta con ejecutar el comando make en la carpeta raíz. El nombre del ejecutable generado es `myruby`.

Ejecutar

```
./myruby archivo.rb arg1 arg2 arg3
```

Luego de esta breve introducción se detalla el diseño, generación de representación intermedia, casos de pruebas y errores encontrados.

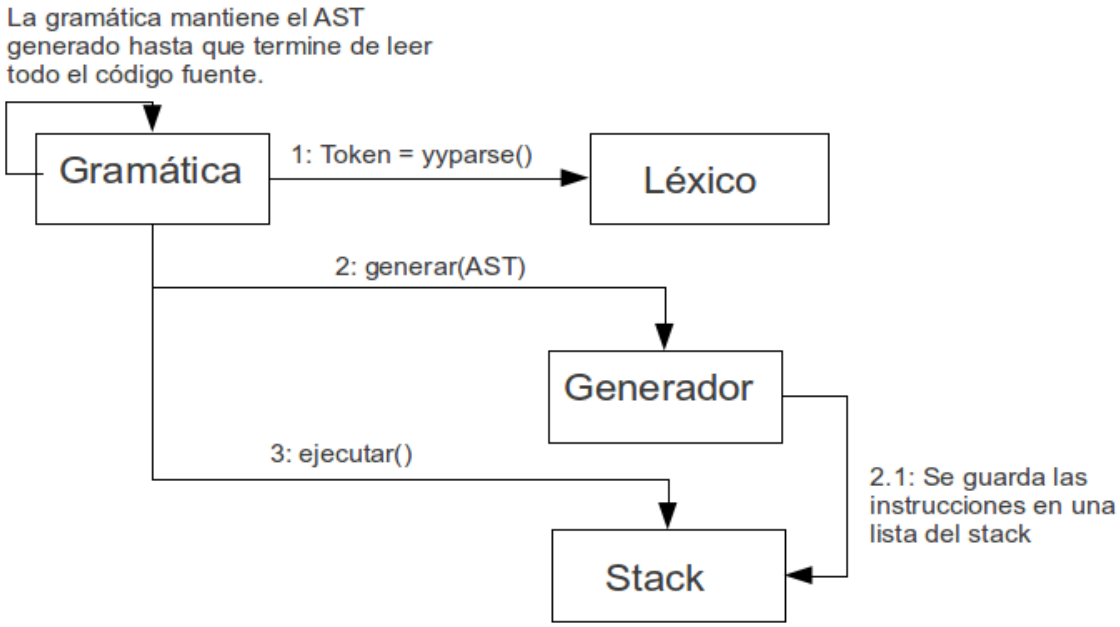
Diseño

Nuestro diseño utiliza una política de generación de código dirigido por la sintaxis. Esto quiere decir que generamos nuestro código intermedio a partir de la gramática escrita.

El programa comienza a ejecutar en el main que se encuentra en el archivo que describe la gramática. Aquí es donde se solicita al analizador léxico que vaya detectando tokens que serán usados por la gramática para generar un AST que culmina al leer fin de archivo.

Luego de tener el AST completo se pasa a la etapa de generación de código intermedio el cual a partir del AST genera una lista de instrucciones que serán ejecutadas por nuestro ambiente.

Una vez que tenemos la lista de instrucciones generadas solo debemos proceder a ejecutarlas. El siguiente dibujo pretende aclarar mejor el diseño.



Análisis léxico

El analizador léxico emite tokens cuando detecta determinados patrones. Sin embargo hay algunos casos particulares para los cuales no fue necesario emitir tokens. Estos casos son

1. Espacios en blanco y tabulación ya que no tienen valor alguno.
2. Los comentarios ya que esto no genera ninguna instrucción para ejecutar.
3. Las instrucciones load y require no generan tokens sino que realizan carga de nuevos archivos en caso que corresponda.

Además en todo momento el analizador puede generar un error que será captado por el analizador sintáctico y así emitir un mensaje pertinente.

Análisis sintáctico y generador de instrucciones

El analisis sintactico le solicita al léxico que le reporte tokens, este procesa los token y va generando el AST en base a las estructuras que tiene definidas. Esta generación se va haciendo recursivamente a medida que vamos detectando las diferentes estructuras y así armando el arbol.

Una vez que leemos todo el archivo de entrada ya tenemos el árbol completo y por lo tanto es necesario pasarlo al generador para que retorne la lista de instrucciones en código intermedio a ser ejecutadas por nuestro ambiente.

Para nosotros una lista de instrucciones es una lista de elementos de tipo Instruccion. El tipo Instruccion se define como una tupla que posee cinco elementos

- op es un enumerado que indica el operador de la instrucción
- arg1 es el argumento en donde se guarda el resultado de la operación
- arg2 y arg3 son operandos de la instrucción.
- linea es usado para indicar a que linea de nuestro código fuente corresponde esta instrucción. Esto se usa para saber en que linea estamos parados al momento de reportar errores de ejecución.

La siguiente tabla muestra una descripción breve de los operadores posibles de nuestro código intermedio.

Instrucciones	Breve descripción
FIN	Indica el fin de la lista de instrucciones a ejecutar
PUTS, GETS	Comunicación con la entrada y salida estándar
CALL	Usado para invocar metodos globales
ADD, MULT, SUB, DIV,POW,MOD	Operaciones para enteros, flotantes y strings(solo el ADD y MULT en este caso)
ASSIGN_TMP, ASGN	Asignación de variables declarada por el usuario y temporales (usadas por el compilador)
IF, ELSIF, ELSIFCOND, CASE, CASEREC CASERECCOND, ELSE	Usado para bloques condicionales
WHILEEND, WHILE	Usado para iteraciones
DO, END	Definicion de bloques
AND, OR, NOT, G, GE, L, LE, EQ, NEQ TOBOOL	Operadores booleanos y operadores de comparacion.
GETV, PUTV	Usado para guardar o consultar variables del stack
GETV_ARR	Usado para colocar el valor de una posicion del arreglo en una variable. Uso tipico para a=ARGV[0]
PUSH_ARG, POP_ARG	Esto es usado en la invocacion y retorno de metodos y funciones, para el pasaje y retorno de parametros
ENDFUNC	Esta instruccion determina el fin de una funcion, se busca el puntero a la siguiente instruccion en el stack y se continua ejecutando
RETURN	Se utiliza para devolver variables de una funcion
CLASS_INST_CALL	Llamado a un metodo de una clase
PUT_INST_V, GET_INST_V	Consultar o guardar variables de instancia de clases
NEW	Usado para instanciar una clase y asignarla a una variables
WRITE_ATTR	Instruccion de asignacion attr_writer de una clase
SET_ARR_POS	Cargar el valor de una variable en cierta posicion del array
PUTS_COMMAND	Imprime el resultado de un comando enviado al

	sistema
SIZE	Imprime el tamaño de un array o string
NEW_SCOPE	Crea un nuevo scope, utilizado en iteraciones
DROP_SCOPE	Elimina el scope actual, usado al salir de una iteracion
TO_S	Convertir cualquier objeto a string
OBJECT_ID	Retornar el id de un objeto
INSTANCE_OF	Realiza la operacion de Instance of
RESPOND_TO	Realiza la operacion de respond_to

En algunos casos y antes de generar las instrucciones es necesario guardar datos en el stack para ser usados posteriormente.

Por lo tanto en el stack se tiene estructuras para guardar

- Funciones que se vayan declarando.
- Variables tanto globales como locales, así como temporales utilizadas por la RI.
- Clases declaradas.
- Métodos globales.
- Manejo de Scope

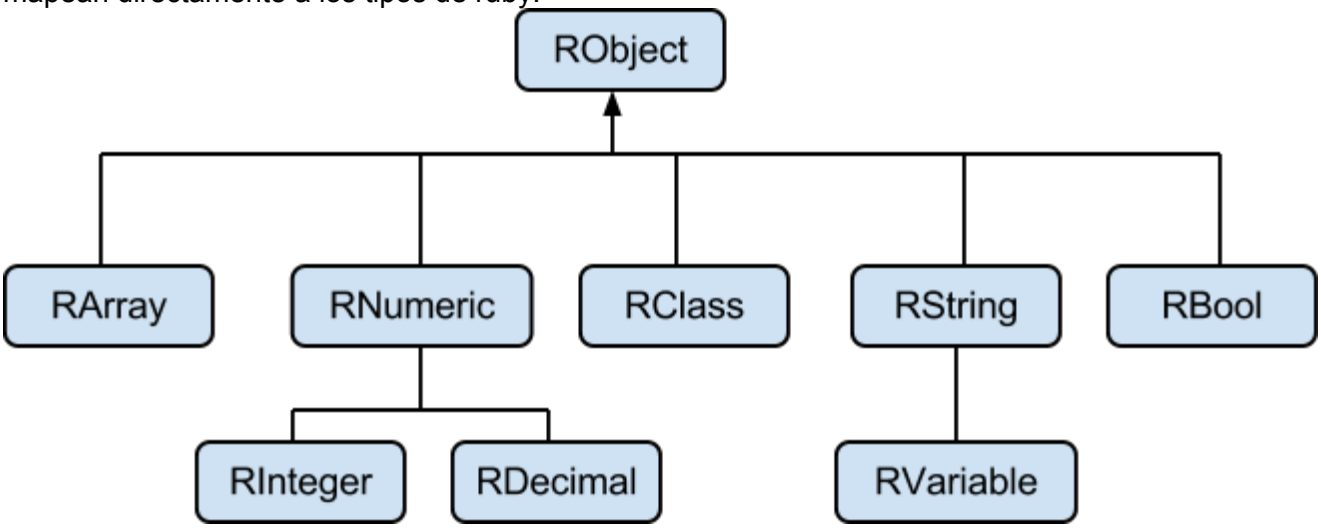
Además en el stack es donde mantenemos una lista con las instrucciones que se van generando. Esta lista será la que recorreremos al momento de la ejecución.

Manejo de Scope

El scope actual es una estructura hash del tipo unordered_map donde se guardan referencias a los objetos en el scope actual. Además se van guardando en un stack para cuando se crean y eliminan. Cuando se busca por variables se empieza por el stack más reciente hasta finalmente buscar en la estructura global.
Para aprovechar el uso del stack, las variables temporales usadas en la RI se guardan en el scope global. Para impedir colisiones se agrega un simbolo de ‘+’ al comienzo y se maneja un contador que se incrementa. Una vez se llega a MAX_INT, se hace append de MAX_INT al prefijo de variable temporal y se comienza de nuevo. De esta manera tenemos una cantidad prácticamentee ilimitada de variables temporales.

Estructuras de soporte

Para facilitar la implementación y el pasaje de parámetros en la RI, creamos clases que mapean directamente a los tipos de ruby.



Cabe mencionar que RClass se utiliza para representar una clase definida por el programador. Contiene estructuras para mantener variables de instancia y métodos definidos para la clase. RVariable es una especialización de RString porque como el anterior mantiene un string pero es utilizada internamente para indicar que se debe ir a buscar la variable al scope. De esta manera entonces los argumentos para la RI son de tipo RObject y facilitan mucho las cosas al evaluar cada operación.

Memoria

Para evitar problemas de memoria durante la ejecucion e impedir que se libere memoria que podria ser usada por el programa, se implemento listas a punteros de cada objeto que se crea de las clases antes mencionadas, se encola el puntero al crear la clase y al final del programa se recorren las listas para liberar la memoria que se reservo. Esto se encuentra en memory.cpp.

Casos de prueba.

Para ejecutar las pruebas realizadas hay que ejecutar los scripts:

- **ejecutar_pruebas.sh:** ejecuta los archivos de prueba y los compara con la salida de ruby. Los archivos de salida se guardan en las carpetas `salidas_ruby` y `salidas_nuestras` respectivamente. Como las salidas pueden dar diferente si se utiliza ruby 1.9 o 1.8, debido a diferencias en cómo se maneja la impresión de nil, hemos incluido las salidas de ruby en la entrega, pero basta con eliminar la carpeta `salidas_ruby` para que el script intente autogenerar las salidas con la versión de ruby del sistema.
- **ejecutar_pruebas_errores.sh:** ejecuta pruebas que dan error y se muestran en la consola

Casos exitosos

test1.rb

Aquí se prueban los siguientes puntos.

- Comentarios de una linea
- Comentarios de varias lineas
- Varias instrucciones en una misma linea
- Una instrucción en varias lineas
- Instrucciones que terminan tanto en “;” como en fin de linea.

test2.rb

Aquí se prueban los siguientes puntos.

- Operaciones aritméticas con números enteros y flotantes (+, -, *, /, %, **)
- Operaciones de asignación con suma y resta (+=, -=)
- Operaciones con strings (+, *)
- Strings con interpolación
- Largo de strings
- Comandos al sistema operativo (pej: puts `dir`)

test3.rb

Aquí se prueban los siguientes puntos.

- Asignación dinamica de variables
- Variables globales ($0, $:, $$)
- Argumentos del programa (ARGV)
- Entrada de datos (dato = gets)

test4.rb

Aquí se prueban los siguientes puntos.

- Metodos
- Arrays
- Bloques

- Alcance, scope de variables
- Iteración dentro de un arreglo con el método each

test5.rb

Aquí se prueban los siguientes puntos.

- Las sentencias if
- Las sentencias case
- Las sentencias while

test6.rb

Aquí se prueban los siguientes puntos.

- Clases

test7.rb y test8.rb

Se prueba require y load

testvacio.rb

Se prueba un programa vacío

Test para manejo de errores

A continuación se muestra un conjunto de test que pretenden mostrar el manejo de errores que el programa implementa.

1. El archivo **testErr1.rb** muestra un error de tipos, ya que se intenta sumar un entero con un string
2. El archivo **testErr2.rb** muestra un error de sintaxis, porque falta un paréntesis que abre (o está sobrando el que cierra)
3. El archivo **testErr3.rb** muestra un error cuando se intenta hacer new de una clase que no existe.
4. El archivo **testErr4.rb** muestra un error al intentar crear una instancia de una clase, ya que se ingresa 1 argumento cuando se esperan 2.
5. El archivo **testErr5.rb** muestra un error al intentar ejecutar un metodo con un numero de parametros erroneo.

Errores detectados y funcionalidades faltantes.

La siguiente tabla muestra funcionalidades no implementadas y errores encontrados.

Errores detectados y funcionalidades faltantes	Ejemplo
No funciona secuencias de escape	puts 'Ruby's party'
No se puede definir constantes pero si variables.	PI = 3.1416 puts PI
No se pueden crear arreglos de la forma a = Array.new(12) pero si podemos crear un arreglo usando la forma a = [] y a = [1,2,3]]	a = Array.new(12)
Las condiciones que se evaluan en condicionales e iteraciones no pueden ser de un sólo argumento, salvo true y false.	if 10; if a; if nil; while “hola” # Estos ejemplos no andan
Da error si hacemos operaciones sin asignación a una variable.	a = 2 + 2 # Anda 2 + 2 # No anda
Operador ternario de decisión no anda	b = (a==1? true: false)
El metodo .class	a= 10 puts a.class