Ìndice

1. Introducción

El presente documento, corresponde a la resolución del trabajo Obligatorio del curso de Diseño de Compiladotes 2012.

El objetivo principal del obligatorio era la entrega de un interprete funcional del lenguaje Ruby con las características indicadas en la letra del obligatorio publicado.

2. Diseño de la gramática

El lexico contiene las expresiones regulares que permiten detectar los tokens definidos para el lenguaje ruby en nuestra acotada realidad.

En las reglas básicamente se incluye:

- ✓ Manejo de estados para el reconocimiento de determinadas cadenas.

- %x comentario Para ignorar comentarios precedidos por '#'
- %x comment Para ignorar comentarios escritos entre las líneas '=begin' y '=end'
- %x Estring Para obtener el string delimitado por los símbolos " " (comillas dobles)
- %x EString2 Para obtener el string delimitado por los símbolos ' ' (comillas simples)
- %x EString3 Para obtener el string delimitado por ` (acento grave).

- ✓ Reconocimiento y retorno de los tokens reconocidos.

- ✓ Para los tokes no reconocidos se utiliza la ultima regla (.) que es la que captura e informa sobre algún error detectado durante el parseo, desplegando el siguiente mensaje :

  "ERROR Lexico en linea *NN*: simbolo no esperado *textoDesconocido*"

La gramatica contiene las reglas que la gramática y lo necesario para la ejecución e interpretación de un programa en ruby.

En las declaraciones de Bison se indican todos los tokens y las asociatividades.

Luego, en las reglas gramaticales se indican las reglas que van a permitir la validación de la gramática del programa.
A medida que vamos reconociendo la gramática se va armando el árbol semántico ejecutando las acciones correspondientes para el armado.

El no terminal que sirve como axioma de la gramática es "AnalisisSintactico".

(En el siguiente punto se describen las estructuras usadas para el armado del árbol).

Finalmente en la sección destinada a código C adicional, tenemos sólo la función yyerror() que captura los errores encontrados al no poder aplicar una regla a la tira leída, desplegando el siguiente mensaje:

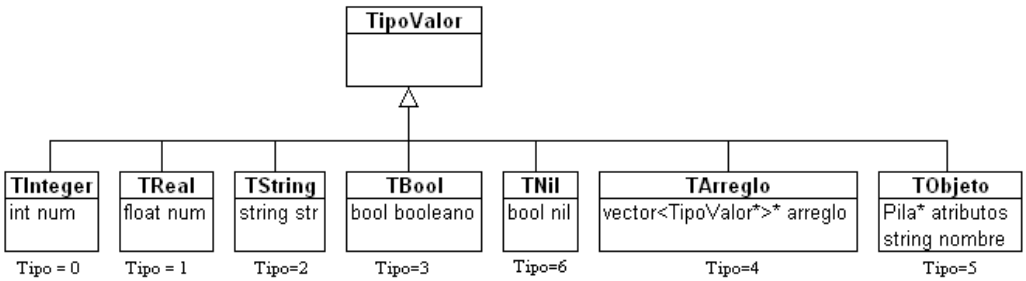
"ERROR Sintactico en linea NN: No se esperaba *textoDesconocido*"

3. Diseño de las estructuras

Aquí explicamos en base a la organización de los archivos entregados, el diseño de las estructuras utilizadas.

TipoValor

Para simplificar el manejo de los tipos de datos, decidimos hacer la siguiente clase abstracta y una codificación de los tipos de datos que tenemos:

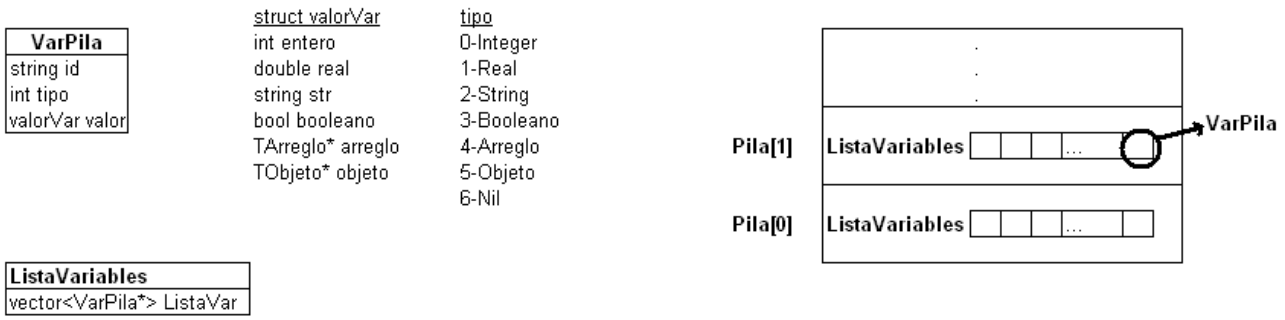


Esto nos simplifica que al momento de tratar, por ejemplo definición de variable, se llame al constructor de la clase según el tipo de dato leído seteándole el valor.
Luego para el acceso, dependiendo del código del tipo, hacemos el casteo correspondiente para obtener el valor del dato.

VarPila

Las variables en cada scope son locales al scope en que se define o redefine la misma, esto último es cuando se cambia el tipo de dato en una reasignación.

Las variables en la stack de ejecución contienen el nombre de la variable, tipo de valor y valor de la misma. El campo valor se diseñó como una estructura genérica que guarda el valor en el campo correspondiente que depende del tipo del valor de la variable.
Esto se implementa en la clase “VarPila”.



ListaVariables

Para mantener las variables locales a un scope implementamos “ListaVariables” que es un vector de variables “VarPila”.
Consideramos una variable como local, si es definida por primera vez dentro del scope actual y no estaba definida anteriormente.

ListaFunciones

Se define una clase singleton llamada “ListaFunciones” para acceder a ellas en forma global y mantener allí el cuerpo de sentencias de las mismas para que sea ejecutado cuando sea llamada.
Para su implementación se utiliza un vector de sentencias.

ListaClases

Se define una clase singleton llamada “ListaClases” para acceder a ella en forma global, y mantener alli las declaraciones de clases.
Para su implementación se utiliza un vector de sentencias tipo clase, y además contiene una lista de funciones y accesores definidos.

Pila

Con la clase “Pila”, se implementó el stack de ejecución.
En el mismo se mantiene un vector donde cada posición indica apertura de un nuevo scope y para cada scope se mantiene su Lista de variables como vimos en el item anterior.

El scope actual siempre es el tope del stack.
Al abrir un scope, se agrega al stack un lista vacía de variables para ese scope.
Al cerrar un scope, se elimina la lista que mantenía las variables locales del scope a cerrar.

ManejadorPila

Para acceder al stack en forma global se implementó otra clase singleton “ManejadorPila”.
En la misma se mantiene un vector para guardar “ListaPilas” (listas de stacks de ejecucion) y una lista “VarsGlobales” de variables globales.

La lista de pilas es para mantener multiples stacks, ya que ademas del stack de ejecución, se crea un stack propio para cada funcion en ejecución, el cual se elimina al terminar la ejecuion de la misma.

En la lista de variables globales del programa ruby que estamos ejecutando se van guardando las variables que se declaran globales ($variable) y que por ende pueden ser accedidas en forma global.

Nodo

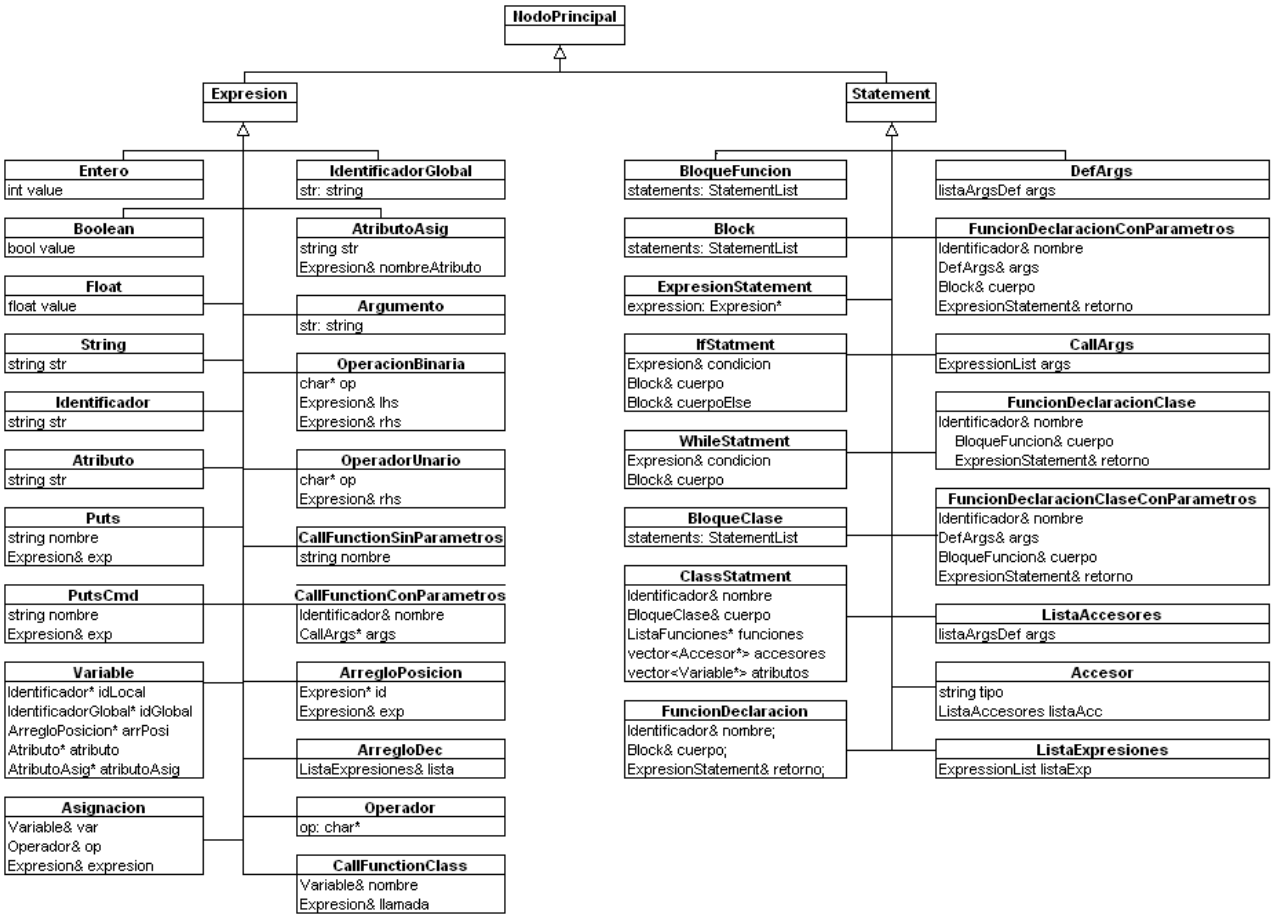
Los nodos del árbol van a tener una estructura en parte simple porque es sólo una jerarquía de clases, pero a su vez compleja por la cantidad de clases que implica y las relaciones entre las mismas.

Los nodos del árbol van a ser expresiones o sentencias.
Para el caso de las expresiones se utiliza una función evaluar, que realiza para cada tipo de expresión, la acción correspondiente a la evaluación de la expresión reconocida, retornando el valor evaluado.

Para el caso de las sentencias se utiliza la funcion ejecutar, que realiza la ejecución de la sentencia correspondiente de la sentencia reconocida.

Todos los nodos van a ir generándose dependiendo de la regla que se aplico.
En los casos que hay errores como se explicó anteriormente desde el código C++ se lanzan excepciones que interrumpen la ejecución del programa.

En la siguiente figura se representa la jerarquía de clases utilizadas para los nodos



4. Chequeos

Además de los errores léxicos y sintácticos mencionado anteriormente, tenemos el chequeo de tipos, con la captura de sus respectivos errores.

El chequeo de tipos optamos por hacerlo en tiempo de ejecución.
A medida que va generándose el árbol semántico, antes de generar los nodos se hace el control de tipos cuando corresponde.

Los errores devueltos describen el problema encontrado y según los controles pueden ser:

"Error: La funcion evaluada no tiene valor de retorno."
"Error: Tipos de datos no esperados para la operacion Suma."
"Error: Tipos de datos no esperados para la operacion Resta."
"Error: Tipos de datos no esperados para la operacion Producto."
"Error: Tipos de datos no esperados para la operacion Division."
"Error: Tipos de datos no esperados para la operacion Modulo."
"Error: Tipos de datos no esperados para la operacion Exponenciacion."
"Error: Tipos de datos no esperados para la operacion Comparacion."
"Error: Tipos de datos no esperados para la operacion Mayor."
"Error: Tipos de datos no esperados para la operacion Mayor Igual."
"Error: Tipos de datos no esperados para la operacion Menor."
"Error: Tipos de datos no esperados para la operacion Menor Igual."
"Error: Tipos de datos no esperados para la operacion Igual."
"Error: Tipos de datos no esperados para la operacion Distinto."
"Error: Tipos de datos no esperados para la operacion AND."
"Error: Tipos de datos no esperados para la operacion OR."
"Error: Tipo de dato no esperado para la operacion + Unario."
"Error: Tipo de dato no esperado para la operacion Unaria Negativo."
"Error: Tipo de dato no esperado para la operacion Unaria NOT."
"Error: Tipo de dato no esperado para la operacion Unaria COMPLEMENTO."
"Error: Clase “nomClase” ya definida."
"Error: Funcion “nomFuncion” ya definida."
"Error: La funcion tiene cantidad incorrecta de parametros."
"Error: La funcion “nomFuncion” no esta definida."
"Error: Variable “nomVariable” no inicializada."
"Error: Fuera de rango"
"Error: los indices del arreglo deben ser enteros."
"Error: Arreglo "nomArreglo" no definido."

5. Intérprete

En el punto 3, se explicaron algunas justificaciones y usos de las estructuras.

Queda para resaltar que :

- **Funciones**: Cuando se definen las funciones las mismas se guardan en la lista de funciones del manejador para cuando queremos invocarlas se ejecute el bloque accediendo al "arbol" de sentencias que se accede desde esa lista, por tal motivo es que usa una pila nueva para cada funcion.
- **Clases**: La definición de las clases tiene comportamiento similar al de las funciones respaldando su información en la Lista de clases.
- **Variables Globales**: Las variables globales están en una estructura diferente de las variables locales ya que tienen un trato distinto.
  Al definir una variable global o utilizarla, la búsqueda se hace directo a la lista de variables globales para verificar su existencia.
  Con las variables locales, el cambio es que si es una definición se busca primero en el scope actual, si existe en el scope actual siempre se sobrescribe (si es del mismo tipo, quedara del mismo tipo y es de tipo distinto se mantiene esta ultima definición).
  Si no está en el scope actual se empieza a buscar en los scopes anteriores:
  Si no esta en ningún scope, se define como variable local, agregándola al scope actual.
  Si existe en otro scope:
  - si es del mismo tipo ➔ se actualiza el valor.
  - si es de distinto tipo ➔ se hace una definición local de esa variable en el stack actual de ejecución redefiniendo así la variable.
- **Recursión**: La Recursión de las funciones esta implementado y fue testeada verificando su comportamiento. En los casos de prueba se muestra el ejemplo del calculo de factorial.

6. Aclaraciones

Al ejecutar Bison con el archivo de la gramática, éste detecta 19 shift/reduce conflictos.
Estos se deben a recursion en la regla EXP, pero el comportamiento esta controlado y verificado respondiendo a lo esperado.

Respecto a los casos a contemplar informados en el documento de la letra del obligatorio, estuvimos trabajando hasta ultimo momento con las clases.
Las mismas están las estructuras implementadas y las funciones que reconocen la definición de las mismas y control de redefinición de clases.
Por falta de tiempo en resolución de problemas, lo que no pudimos dar por valido la ejecución de las mismas.
Pero en los archivos quedó el código del análisis que habíamos realizado para su resolucion, como por ejemplo TObjeto dicho anteriormente cuyo objeto era mantener los atributos y el estado de cada uno de ellos de cada clase.

7. Referencias

Google:
http://www.google.com

Flex:
http://flex.sourceforge.net/manual/

Sitio oficial:
http://www.ruby-lang.org/es/

Tutorial en 20 minutos:
http://www.ruby-lang.org/en/documentation/quickstart/

Tutorial de C:
http://www.physics.drexel.edu/courses/Comp_Phys/General/C_basics/

Ejemplos resueltos:
http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/
http://www.alanmacek.com/software/interpreter/