Universidad Simón Bolívar
Dpto. de Computación y Tecnología de la Información
CI3725 - Traductores e Interpretadores
Abril-Julio 2009

# Proyecto Unico - Entrega 2
# Análisis Sintáctico

Esta segunda etapa del proyecto corresponde al módulo de análisis sintáctico del interpretador del lenguaje YISIEL que queremos construir. Específicamente, se desea que Ud. implante el módulo en cuestión combinando (y terminando si es necesario) el Analizador Lexicográfico desarrollado en la Primera Parte junto con el módulo generado por la herramienta `Racc`, además de desarrollar una librería en Ruby para manejar Tablas de Símbolos que será de utilidad en la última etapa.

El Analizador Sintáctico a ser construido en esta etapa deberá aceptar como entrada la lista de *tokens* producidos por el Analizador Lexicográfico desarrollado en la Primera Parte y procesarlos para determinar la validez sintáctica del programa YISIEL. Si la secuencia de *tokens* recibidos no corresponde con la sintaxis del lenguaje YISIEL, debe producirse un mensaje de error, acompañado de la posición dentro del archivo (línea y columna) en la cual han sido encontrados así como algún "contexto" que permita al usuario identificar el error con facilidad.

Para alcanzar estos objetivos Ud. debe:

- Diseñar una gramática libre de contexto cuyo lenguaje generado sea YISIEL y escribirla usando `Racc`. La gramática **no** puede ser ambigüa y **debe** utilizar recursión por izquierda.
- Diseñar un tipo abstracto de datos `Sym` para modelar los símbolos que ha de manipular el interpretador, desplegando una jerarquía de clases completa para modelar las variables enteras, los arreglos y los procedimientos manejados por YISIEL.
- Diseñar un tipo abstracto de datos `SymTable` para modelar las tablas de símbolos que ha de manipular el analizador y el interpretador, implantando los métodos necesarios para insertar, buscar y eliminar símbolos dentro de las tablas.
- Note que es necesario modelar un tipo adicional `AST` para los árboles abstractos, sin embargo para este proyecto solamente es necesario mencionar su existencia y usar el nombre, pero no es necesario implantarlo.

## Entrega de la Implantación

Ud. deberá entregar un archivo `.tar.gz` o `.tar.bz2` (**no** puede ser `.rar`, ni `.zip`) que al expandirse genere un directorio con el nombre del grupo (e.g. `G42`) dentro del cual encontrar:

- Los mismos archivos `Token.rb`, `Lexer.rb` y `yisiel.rb` de la primera entrega, modificados para integrarse correctamente con el resto de los módulos.
- El archivo `Parser.y` conteniendo el código fuente para `Racc` que implanta el módulo Ruby de nombre `Parser` que ofrece la funcionalidad del analizador sintáctico. La clase debe proveer:

- El constructor de instancias que recibe como argumento un objeto de la clase `Lexer`, previamente preparado, asociado al analizador lexicográfico con el cual trabajar.
- Un método público de nombre `Parser.parser` que realiza el análisis sintáctico. Este método se invocará una sola vez desde el programa principal y **debe** estar basado en el uso del método `do_parse` provisto por Racc, para lo cual Ud. debe proveer el método `Parser.next_token` de manera que retorne el Token en el formato adecuado para Racc.
- El método privado `Parser.on_error` que reporta los errores sintácticos emitiendo una excepción, y que permita indicar la línea y columna del error, así como el Token "fuera de lugar".

- El archivo `Sym.rb` conteniendo el código fuente para Ruby que implanta el módulo homónimo. La clase debe proveer un constructor y métodos abstractos. En el mismo archivo deben implantarse las clases concretas `SymVar`, `SymArray` y `SymProc` para representar tales elementos del lenguaje.
- El archivo `SymTable.rb` conteniendo el código fuente para Ruby que implanta el módulo homónimo. La clase debe proveer un constructor y los metodos:
  - `SymTable.insert( Sym )` retornando el `Sym` recién insertado o `undef`.
  - `SymTable.find( String )` retornando el `Sym` encontrado o `undef`.
  - `SymTable.delete( String )`.

  Note que los tipos `Sym` y `SymTable` son mutuamente recursivos en virtud de la necesidad de que los procedimientos manejen su propia tabla de símbolos local.
- El archivo `AST.rb` conteniendo una declaración de clase vacía Ruby para el módulo homónimo a ser desarrollado en la tercera etapa, pero cuya presencia es necesaria para el diseño de la tabla de símbolos en relación con los procedimientos.
- El archivo `yisiel.rb` conteniendo el código fuente Ruby para el programa principal. Este archivo será utilizado con Ruby 1.8 en Debian GNU/Linux sin modificación alguna; los profesores **no** vamos a editar archivos en modo alguno, de manera que Ud. debe verificar que el archivo que envía funciona perfectamente con la versión de Ruby incluida en Debian GNU/Linux. Haga la prueba en el LDC para estar seguro.
- Los módulos **deben** estar completa y correctamente documentados utilizando la herramienta RDoc. Ud. **no** entregará los documentos HTML generados, sino que deben poder **generarse** de manera automática incluyendo acentos y símbolos especiales. Es **inaceptable** que la documentación tenga errores ortográficos.
- Un `Makefile` que permita construir el programa y generar la documentación, utilizando de manera adecuada los utilitarios `racc` y `rdoc`.
- El programa principal será utilizado desde la línea de comandos:
  - Si se invoca el programa suministrado **exactamente** un argumento en la línea de comandos, el programa debe intentar abrir ese archivo y procesarlo. Si se invoca el programa **sin** argumentos en la línea de comandos, el programada debe ofrecer

un *prompt* en el cual escribir el nombre del archivo a procesar. Si el archivo suministrado como argumento no existe o no puede abrirse por la razón que sea, el programa debe terminar indicando el problema. Es **inaceptable** que el programa aborte inesperadamente.

- Si el programa es sintácticamente correcto mostrar en pantalla las reglas "utilizadas", haciendo algo similar a

```
expr : expr TkPlus expr
    { puts "expr-> expr + expr\n" }
expr : TkNum
    { puts "expr-> TkNum(", $1, ")\n" }
```

  de manera que aparezca en pantalla la definición de la regla particular, justo cuando ha sido reducida.

- Si el programa tiene errores lexicográficos, mostrarlos **todos** tal como en la Primera Parte.
- Si el programa no tiene errores lexicográficos, pero tiene errores de sintaxis, debe mostrar el **primer** error de sintaxis, indicando la línea y columna donde ocurre, e indicando los *tokens* del contexto.

- Un archivo PDF con el desarrollo de las preguntas de la siguiente **Revisión Teórico-Práctica**:

1. ¿De qué manera general puede utilizarse la salida de su analizador sintáctico en la construcción de derivaciones?
2. Sea la gramática $G_1 = (\{E\}, \{+, \mathbf{num}\}, P, E)$, con $P$ compuesto por

$$\begin{aligned} E &\rightarrow E{+}E \\ E &\rightarrow \mathbf{num} \end{aligned}$$

   *a*) Muestre que la frase $\mathbf{num} + \mathbf{num} + \mathbf{num}$ es ambigua.

   *b*) Dé una gramática no ambigua $G_I$ que genere el mismo lenguaje que $G$ y que asocie las expresiones aritméticas generadas hacia la izquierda. Dé una gramática no ambigua $G_D$ que genere el mismo lenguaje que $G$ y que asocie las expresiones aritméticas generadas hacia la derecha.

3. En la definición de YISIEL se presenta al punto y coma como el *separador* de instrucciones dentro de un bloque. Suponga que para el manejo de esa construcción se utiliza la gramática $G_2 = (\{Instr\}, \{;, \mathbf{IS}\}, P, Instr\}$, con $P$ compuesto por

$$\begin{aligned} Instr &\rightarrow Instr\,;Instr \\ Instr &\rightarrow \mathbf{IS} \end{aligned}$$

   Por conveniencia, momentáneamente se ignora al resto de los constructores de instrucciones compuestas del lenguaje, simplificando las instrucciones con el símbolo terminal **IS**.

*a*) ¿Presenta $G_2$ los mismos problemas de ambigüedad que la gramática $G_1$? ¿Cuáles son las únicas frases no ambigüas de $G_2$?

*b*) ¿Importa si la ambigüedad se resuelve con asociación hacia la izquierda o hacia la derecha?

4. Considere la gramática $(\{L, D, I, T\}, \{\mathbf{a}, \mathbf{c}, \mathbf{i}, \mathbf{s}, \mathbf{v}\}, P, L)$, con $P$ compuesto por

$$\begin{aligned} L &\to L\,D \\ L &\to D \\ D &\to \mathbf{v}\,I\,\mathbf{c}\,T \\ I &\to I\,,\mathbf{i} \\ I &\to \mathbf{i} \\ T &\to \mathbf{s} \\ T &\to \mathbf{a} \end{aligned}$$

Estudie el Capítulo *"Parsing: An introduction"* de [2] y aplique ambos algoritmos descritos en el texto para presentar los árboles construidos por cada reconocedor al suministrar la frase

**vi, i, icavics**

- **Fecha de Entrega.** Lunes 2009-06-08 (Semana 8) hasta las 15:30 VET.
- **Valor de Evaluación.** Nueve (9) puntos.

# Referencias


[1] *Racc: a Ruby Parser Generator*
`http://i.loveruby.net/en/projects/racc/`

[2] T. Sudkamp
*Languages and Machines*
Capítulo 18