Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Traductores e Interpretadores (CI-3725)
Abril-Julio, 2011

Proyecto Único - Etapa 2
Analizador Sintáctico
(Versión 0.1)

Esta segunda etapa del proyecto corresponde al módulo de análisis sintáctico del interpretador del lenguaje Vectorinox que queremos construir. Específicamente, se desea que Ud. implemente el módulo en cuestión combinando el Analizador Lexicográfico desarrollado en la primera etapa junto con el módulo generado por la herramienta Racc (para los que eligieron trabajar con Ruby) o Happy (para los que eligieron trabajar con Haskell), para generar una representación intermedia del programa reconocido.

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 etapa y procesarlos para construir el resultado deseado. Si la secuencia de tokens recibidos no corresponden con la sintaxis del lenguaje Vectorinox, 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 programador Vectorinox identificar el error con facilidad.

Para alcanzar estos objetivos Ud. debe:

- Diseñar una gramática libre de contexto cuyo lenguaje generado sea el mismo que el descrito por la definición de Vectorinox. La gramática no puede ser ambigua, y debe utilizar recursión por izquierda.
- Implementar el Analizador Sintáctico usando su gramática y la herramienta Racc (para los que eligieron trabajar con Ruby) o Happy (para los que eligieron trabajar con Haskell).
- Diseñar suficientes tipos de datos recursivos para representar los elementos sintácticos del lenguaje, de manera que puedan ser utilizados para completar el interpretador en la tercera etapa.
- Construir una librería de manejo de tablas de símbolos. Dicha librería debe estar basado en el uso de *hashes* (en el caso de Ruby) o usando la librería `Data.Map` (en el caso de Haskell). Se deben manejar las dos posibles tablas de símbolos que presenta el lenguaje: la tabla de funciones declaradas, y las tablas de símbolos asociadas a cada bloque.

El programa `vecti` funcionará usando las mismas opciones indicadas en la definición de lenguaje. Dichas opciones se ingresan **en la línea de comandos**. El resultado de ejecutar el analizador sintáctico sobre un programa será una representación en consola del árbol producido por el programa (en conjunción con sus tablas de símbolos, si es un bloque). Por ejemplo, el siguiente programa:

```
define prueba(x: num, y: num) of type num as
  return x + y + 1

begin
  vars
    a : num
  a := 2;
  print 'El numero es', prueba(a, .2)
end
```

puede tener la siguiente salida:

```
Funciones:
  prueba de tipo: num
  Tabla de simbolos:
     x: num
     y: num
  Instruccion:
  Return (Suma (Suma (Variable x) (Variable y)) (Numero 1))
```

```
Bloque
  Tabla de simbolos:
     a: num

  Instrucciones:
  Asignacion (Variable a) (Numero 2)
  Imprimir (Variable a) [String 'El numero es', Llamada prueba [Variable a, Numero .2]]
```

En el caso que el programa tenga un error sintáctico, su analizador mostrará solamente el primer error encontrado. Se debe mantener la detección de errores de la etapa lexicográfica.

Obligaciones en Ruby

En adición a los archivos que ud. implementó en la primera entrega (que pueden ser modificados para que funcione con esta entrega), ud. debe de implementar:

- El archivo `SymTable.rb` conteniendo el codigo fuente para Ruby que implanta el módulo homonimo. La clase `SymTable` debe proveer un constructor y los metodos:
  - `SymTable.insert( Sym )` retornando el `Sym` recien insertado o undef.
  - `SymTable.find( String )` retornando el `Sym` encontrado o undef.
  - `SymTable.delete( String )`.
  - Su librería de tabla de símbolos debe manejar los dos tipos de tablas que tiene el lenguaje usando solamente la clase `SymTable`. Para esto, defina una clase `Sym` abstracta y clases concretas `SymFunction` y `SymVariable` que representen funciones y variables.
- El archivo `AST.rb` conteniendo el codigo fuente que implanta el modulo Ruby homónimo. La clase `AST` debe proveer un constructor y metodos abstractos:
  - `AST.new()` para crear un nuevo `AST`.
  - `AST.check()` para realizar la vericacion de contexto estatico del `AST`. Para esta entrega, este método debe estar vacío.
  - `AST.run()` para realizar la vericacion dinamica e interpretacion del `AST`. Para esta entrega, este método debe estar vacío.
  - En conjunción, ud. debe crear tres sub-clases de `AST` que representen los conceptos generales del lenguaje: `ASTStatement`, `ASTExpression` y `ASTBool`. Considere crear subclases de cada una de estas clases por cada instrucción interesante del lenguaje.
- El archivo `Parser.y` conteniendo el codigo fuente para Racc que implanta el modulo Ruby de nombre `Parser` que ofrece la funcionalidad del analizador sintactico. La clase `Parser` debe proveer:
  - El constructor de instancias que recibe como argumento un objeto de la clase Lexer, previamente preparado, asociado al analizador lexicograco con el cual trabajar.
  - Un metodo publico de nombre `Parser.parser` que realiza el analisis sintactico. Este metodo se invocara una sola vez desde el programa principal y debe estar basado en el uso del metodo `do_parse` provisto por Racc, para lo cual Ud. debe proveer el metodo `Parser.next_token` de manera que retorne el Token en el formato adecuado para Racc.
  - El metodo privado `Parser.on_error` que reporta los errores sintacticos emitiendo una excepcion, y que permita indicar la lnea y columna del error, así como el token "fuera de lugar" y los siguientes tokens que lo suceden.

Obligaciones en Haskell

Ud. debe de implementar, en adición a los requisitos del primer proyecto, lo siguiente:

- El archivo `SymTable.hs` conteniendo el código fuente para Haskell que implanta el módulo Haskell de nombre SymTable. Este módulo debe exportar:
  - El TAD `SymTable` sin modificación.
  - El TAD `Symbol` modificado para contemplar las posibles formas de manejar símbolos en la tabla según sea necesario para la especificación del lenguaje.
  - Las funciones de manipulación de tabla de símbolos `isMember`, `find`, `insert` y `replace` con la misma firma con la que aparecen en el módulo de ejemplo.
- El archivo `AST.hs`, que implementa dicho módulo de Haskell, que define los TADs diseñados por Ud. para la representación intermedia. Este módulo debe exportar dichos TADs.
- El archivo `Parser.y` conteniendo el código fuente para Happy que implanta el módulo Haskell de nombre Parser de tal forma que únicamente exporta la función `parser` que debe tener la firma:

```
parser :: [Token] -> ( SymTable, AST )
```

Entrega de Implementación

Ud. debe de entregar un archivo .tar.gz (no puede ser ni .zip ni .rar) la cuál, al descomprimirse, debe generar una carpeta con su número de grupo (por ejemplo, GR16) cuyo contenido está estructurado de la siguiente forma:

- Una carpeta `src` que contenga todos sus archivos fuente.
- Una carpeta `bin` (vacía al momento de entrega) que, al completar el proceso de compilación, debe de tener el archivo `vecti` con todos los archivos necesarios para su ejecución. Dicha carpeta debe de funcionar en cualquier sitio y sin depender del contenido de este paquete.
- Una carpeta `doc` (vacía al momento de entrega) que, en el caso que se le solicite a las herramientas de compilación, contenga la documentación generada de su proyecto.
- Un `Makefile` (que reside en la raíz del proyecto) que contenga las siguientes opciones:
  - Una opción de compilación que realice los pasos de compilación necesarios y genere los resultados en la carpeta `bin`. Esta es la opción por defecto (es decir, ejecutando make solamente). Aunque Ruby es un lenguaje interpretado, recuerde que debe respetar las restricciones indicadas.
  - Una opción que permite generar la documentación. Esta opción debe ser invocada mediante `make docs`.
  - Una opción que permita borrar los archivos creados en el paso por defecto. Esta opción se invoca mediante `make clean`.
  - Una opción que borre todos los archivos generados. Esta opción se invoca mediante `make cleanall`.
- Un README con información de interés para el proyecto. Debe contener los integrantes del equipo y la funcionalidad no implementada.

Detalles de la Entrega

- **Fecha de Entrega:** Domingo 19 de junio de 2011 (Semana 8) a las 11.59 am. Todo proyecto entregado después de esta hora tendrá ponderación de cero (0) puntos.
- **Valor de Evaluación:** Diez (10) puntos.