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

Proyecto Único - Entrega 3
Análisis de Contexto e Interpretación

Esta tercera y última etapa del proyecto corresponde al módulo de análisis de contexto e interpretación 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` y la librería en Ruby para manejar Tablas de Símbolos desarrollados en la Segunda Parte, completando ahora el diseño y construcción del TAD para los árboles abstractos de sintaxis de manera tal que sea posible realizar el análisis de contexto estático y el análisis de contexto dinámico que formalice la interpretación de programas del lenguaje `YISIEL`, logrando su ejecución.

El Analizador Sintáctico debe ser cambiado para que durante su ejecución sea capaz de construir el Arbol Abstracto de Sintaxis correspondiente al programa analizado y representado con una jerarquía de clases enraizadas en la clase abstracta `AST`, además de llenar la tabla de símbolos con aquellas definiciones encontradas durante el reconocimiento sintáctico. Inmediatamente, procederá a realizar las verificaciones de contexto estático recorriendo el árbol en profundidad y apoyándose en la información contenida en la tabla de símbolos. De no haber errores de contexto estático, deberá iniciar la interpretación del programa, asegurando las verificaciones de contexto dinámico en cada paso de interpretación.

Para alcanzar estos objetivos Ud. debe:

- Modificar el analizador lexicográfico desarrollado en la primera entrega, y el analizador sintáctico desarrollado en la segunda entrega, para incorporar un nuevo elemento al lenguaje `YISIEL`: debe utilizarse el símbolo '`|`' (la barra vertical o *pipe*) como separador entre guardias de las instrucciones. La definición del lenguaje ha sido revisada para acomodar este cambio. Si Ud. había tomado previsiones especiales en su reconocedor para manejar la ambigüedad que se presentaba, asegúrese de eliminarlas.

- Completar el diseño del tipo abstracto de datos `AST` para modelar los árboles abstractos asociados a la estructura de los programas `YISIEL`, desplegando una jerarquía de clases completa para modelar las instrucciones, las expresiones aritméticas y las expresiones boolenas.

- El análisis de contexto estático debe detectar y reportar al menos las siguientes situaciones:

  - Es un error utilizar una variable que no ha sido declarada previamente. Recuerde que aplican las reglas de alcance estático, de manera que dentro de un procedimiento son visibles las variables locales y las variables globales.
  - Es un error definir una misma variable más de una vez en el mismo alcance, esto es más de una variable *global* con el mismo nombre, o más de una variable *local* con el mismo nombre en un procedimiento. No es un error cuando se define una variable *local* a un procedimiento con el mismo nombre de una variable *global*.

- Es un error invocar un procedimiento que no ha sido definido. Note que en el cuerpo de un procedimiento `foo` es válido invocar un procedimiento `bar` que está definido *después* del procedimiento `foo`.
- Es un error utilizar un arreglo como si fuera una variable simple y viceversa.
- Es un error invocar un procedimiento con más o menos parámetros actuales de los que figuran en su definición.
- Si un parámetro formal es de salida (`out`) es un error pasar un parámetro actual que no sea una variable o una posición de un arreglo.
- Es un error asignar valores a los parámetros formales de entrada (`in`) dentro del cuerpo de un procedimiento. Los parámetros formales de entrada deben considerarse válidos solamente para lectura dentro del procedimiento.
- Es un error utilizar los parámetros formales de salida (`out`) dentro del cuerpo de un procedimiento. Los parámetros formales de salida deben considerarse válidos solamente para asignación dentro del procedimiento.
- No es un error utilizar el mismo nombre para una variable, un arreglo y un procedimiento. En otras palabras, es válido tener al mismo tiempo una variable simple de nombre `foo`, un arreglo de nombre `foo` y un procedimiento de nombre `foo`.

Los errores deben generarse como excepciones, que deben ser manejadas por el programa principal. De existir uno o más errores de contexto, deben reportarse **todos.** El reporte de cada error debe indicar la línea y columna en la cual se detectó el error.

- De no haber errores de contexto estático, debe iniciarse la ejecución del programa. Durante la ejecución, el análisis de contexto dinámico debe detectar y reportar al menos las siguientes situaciones:

  - Es un error utilizar una variable que no ha sido inicializada previamente.
  - Es un error dividir o calcular el resto de la división entre cero.
  - Es un error utilizar o asignar una posición fuera de los límites de un arreglo.
  - Es un error que un procedimiento retorne y alguno de sus parámetros de salida no haya sido asignado.

  Los errores deben generarse como excepciones, que deben ser manejadas por el programa principal. El programa debe terminar la ejecución al encontrar el primer error de contexto dinámico. El reporte de error debe indicar la línea y columna en la cual se detectó el error.

- Si durante la ejecución ocurren excepciones relacionadas con el entorno de ejecución (e.g. agotamiento de la memoria, sin excluir otras razonables) el interpretador debe manejarla y finalizar. Es **inaceptable** que el programa aborte.

- Si Ud. considera que es conveniente incluir verificaciones de contexto estático y dinámico adicionales, justifíquelas por correo electrónico *antes* de hacer la entrega para convalidarlas. Estas verificaciones adicionales pueden ayudarle en su evaluación solamente si **todas** las verificaciones solicitadas explícitamente en este enunciado han sido implantadas correctamente en su proyecto.

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 mismas archivos de la primera y segunda entrega, modificados para incorporar las modificaciones solicitadas e integrarse con el restro de los módulos.
- El achivo `AST.rb` conteniendo el código fuente que implanta el módulo Ruby homónimo. La clase debe proveer un constructor y métodos abstractos
  - `AST.new()` para crear un nuevo AST.
  - `AST.check()` para realizar la verificación de contexto estático del AST.
  - `AST.run()` para realizar la verificación dinámica e interpretación del AST.

  En el mismo archivo deben implantarse al menos las clases concretas `ASTMath`, `ASTBool` y `ASTStmt` para representar tales AST. Considere especializar dichas clases concretas con clases refinadas para el resto de los elementos. Cambiar la firma de los métodos antes mencionados para refinar el comportamiento solamente está permitido en las subclases.
- 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 su archivo 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, 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 suministrando **exactamente** un argumento en la línea de comandos, el programa debe intentar abrir ese archivo y procesarlo. Si se invoca el programa **sin** argumentos, el programa 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 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, junto con los *tokens* del contexto, tal como en la Segunda Parte.

- Si el programa es sintácticamente correcto, pero tiene errores de contexto estático, mostrarlos **todos**, indicando su naturaleza, el identificador o valor que causa el error, así como la línea y columna donde ocurren.
- Si el programa es sintácticamente correcto y no tiene errores de contexto estático, exhibir el comportamiento del programa (si es que el programa emplea `show`, claro está). Si se detecta un error de contexto dinámico, el programa debe finalizar indicando la naturaleza del error, junto con la línea y columna en la que se detectó.

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

  1. Sea G_{1i} la gramática recursiva-izquierda $(\{S\}, \{a\}, \{S \to Sa,\ S \to \lambda\}, S)$ y sea G_{1d} la gramática recursiva-derecha $(\{S\}, \{a\}, \{S \to aS,\ S \to \lambda\}, S)$. Ambas generan el lenguaje denotado por la expresión regular a^*.

     a) Muestre que ambas gramáticas son $LR(1)$ y construya sus analizadores sintácticos según el método de tabla $SLR(1)$.

     b) Compare la eficiencia de ambos analizadores en términos de *espacio*, i.e. los tamaños de sus tablas y la cantidad de pila utilizada para reconocer cada frase de a^*, y de *tiempo*, i.e. la cantidad de movimientos realizados por el autómata de pila para reconocer cada frase de a^*. *Nota:* un análisis serio de eficiencia debe hacerse en términos de órdenes de complejidad empleando la notación O.

  2. Sea G_2 la gramática $(\{Instr\}, \{\mathbf{i}, ;\}, \{Instr \to \mathbf{i},\ Instr \to Instr\,;Instr\}, Instr)$

     a) Muestre que G_2 **no** es una gramática $LR(1)$, intentando construir un analizador para la gramática y consiguiendo que tiene un conflicto. Especifique el tipo de conflicto encontrado identificándolo en la tabla.

     b) Considere las dos posibilidades de resolución del conflicto (i.e. en favor del *shift* o en favor del *reduce* si se tratase de un *shift-reduce*, o en favor de una y otra de las producciones si se tratase de un *reduce-reduce*). Muestre, para ambas alternativas de resolución del conflicto, la secuencia de reconocimiento para la frase

     $$\mathbf{i};\mathbf{i};\mathbf{i}$$

     dando como salida la secuencia de producciones reducidas. Si consideramos al separador de instrucciones (;) como un operador asociativo, ¿a qué sentido de asociatividad corresponde cada una de las alternativas de resolución?

     c) Compare la eficiencia de ambos analizadores en términos de *espacio* y de *tiempo*, para reconocer frases de la forma $\mathbf{i}(;\mathbf{i})^n$. *Nota:* un análisis serio de eficiencia debe hacerse en términos de órdenes de complejidad empleando la notación O.

- **Fecha de Entrega.** Lunes 6 de julio de 2009 (Semana 12). La evaluación y revisión del proyecto será **presencial** en el orden acordado con su profesor de laboratorio durante la Semana 11.

- **Valor de Evaluación.** Quince (15) puntos.

Referencias

[1] A. Aho, R. Sethi & J.D. Ullman
Compilers: Principles, Techniques and Tools
ISBN 0-201-10088-6
QA76.76.C65A37
Capítulo 6 (Type Checking)

[2] M. Scott
Programming Language Pragmatics
ISBN 0-12-633951-1