Diseño y Paradigmas de Lenguajes - Año 2014
Entrega Obligatoria N 3: Análisis sintáctico LR utilizando la herramienta Cup

Modalidad y fecha de entrega: se recepcionará la resolución de la entrega hasta el día **24 de octubre de 2014 a las 10hs.** por medio del Aula Virtual (link destinado a tal fin). De no haber recibido el mismo hasta ese horario, se considerará como práctico NO ENTREGADO. Recordar que las entregas son obligatorias para REGULARIZAR la materia. La resolución del proyecto consiste en entregar: códigos fuentes pedidos, la versión de la máquina virtual de java utilizada y un documento con lo solicitado en la consigna del Proyecto.
Modalidad de corrección: el código entregado será probado en la máquina virtual de Java y se cargará una lista con los resultados en el aula virtual. Si la cátedra considera necesario, se realizará un coloquio relacionado al desarrollo del proyecto. Si el práctico no estuviera aprobado, será devuelto al alumno para su corrección.

Proyecto Cup

Para realizar la práctica se deben realizar los siguientes items en secuencia:

a. Utilice el analizador lexicográfico realizado en la materia Fundamentos de la Computación, modifique el archivo *.flex* incorporando las siguientes sentencias:

```
import java_cup.runtime.*;
%%
%cup
```

b. Compilar el archivo *.flex* utilizando la siguiente sentencia para obtener el archivo *Ylex.java*:

```
java -jar JFlex.jar AnalizadorLexicografico.flex
```

c. Descargue del sitio ($http://www2.cs.tum.edu/projects/cup/$) la herramienta Cup (Constructor of Useful Parsers), esta permite generar automáticamente analizadores sintácticos para gramáticas LALR en el lenguaje Java, utiliza una especificación basada en una gramática e interactúa con el analizador lexicográfico.

d. Una especificación Cup consiste de las siguientes secciones:

- Especificaciones package/import (opcional).
- Componentes (código del usuario)(opcional).
- Lista de Símbolos (terminales y no terminales)
- Declaración de Precedencia (opcional).
- Gramática

En la sección *terminal* se colocan los códigos, que le correponden a cada terminal, en su analizador lexicográfico. En la *non terminal* se especifican los no terminales de la gramática. Por último se deben colocar cada una de las producciones de la gramática. En el siguiente ejemplo se muestra como define cada producción de la gramática. Los símbolos en la parte derecha de la producción pueden tener una etiqueta luego de (:), esta permite acceder al valor del no terminal, las acciones asociadas a una producción son códigos Java delimitados entre {: y :}. Para el caso de una parte izquierda con más de una producción, se coloca el símbolo | para separar las partes derechas. El símbolo inicial se considera que es el no terminal de la parte izquierda de la primera producción, pero esto se puede cambiar con la construcción *start with*.
Sean las siguientes producciones de una gramática en BNFE de un lenguaje de programación:

```
<sif> ::= if <exp> then <exp>
<exp> ::= <opdo> <op> <opdo>
<op> ::= < | >
<opdo> ::= <id> | <nro>
```

El siguiente es un ejemplo de un archivo con las especificaciones Cup de la gramática dada:

```
Ejemplo.cup

import java_cup.runtime.*;

terminal String COD_IF, COD_THEN, COD_OP_MEN,
                        COD_OP_MAY, COD_ID, COD_NUM;

non terminal String  sif, exp, op, opdo;


sif ::= COD_IF exp:e1 COD_THEN exp:e2
      {:
         System.out.println("sif -> IF"+e1+"THEN"+e2);
      :};


exp ::= opdo:e1 op:e2 opdo:e3
      {:
         System.out.println("exp -> "+e1+" "+e2+" "+e3);

     :};

op ::= COD_OP_MEN:e1
    {:
     System.out.println("op -> "+e1);
    :}
  |
           COD_OP_MAY:e2
```

```
{:
  System.out.println("op -> "+e2);
:};
```

Utilice la grámatica simplificada de Java que se adjunta con el Proyecto, para crear un archivo *AnalizadorSintactico.cup* con las secciones especificadas en el ejemplo anterior. Al reconocer una parte derecha de una producción, se debe mostrar la misma como lo realiza el ejemplo.

e. Compile el archivo obtenido del punto anterior con la siguiente sentencia:

```
java -jar java-cup-11a.jar AnalizadorSintactico.cup
```

f. Para realizar el análisis sintáctico de un archivo se debe ejecutar la siguiente sentencia, donde el argumento *TextoPrueba.txt* es el nombre del archivo a analizar:

```
java -classpath .:java-cup-11a.jar AnalizadorSintactico
                                                TextoPrueba.txt
```

g. Cuando se produce un error en el análisis sintáctico, CUP invoca a los siguientes métodos:

```
public void syntax_error(Symbol s);
public void unrecovered_syntax_error(Symbol s)
                                  throws java.lang.Exception;
```

En el momento que se produce un error se invoca al método *syntax_error*, luego se intenta recuperar del error, si no se hace nada para recuperarse del error se detiene el análisis ante el primer error que se genere y luego se invoca el método *unrecovered_syntax_error*. El objeto *s* de la clase *Symbol* contiene el último Token consumido por el analizador. Estos mtodos se pueden redefinir dentro de la declaración parser code {: ... :}.
Probar el analizador con programas con errores sintácticos y muestre cuál es la salida. Se debe entregar al menos 2 códigos de prueba y lo que muestra el analizador.

h. Modifique el codigo fuente de *AnalizadorSintactico.cup* adicionando el siguiente trozo de código que redefine los métodos *syntax_error* y *unrecovered_syntaxError*:

```
parser code {:
public void syntax_error(Symbol s) {
  report_error("Error sintctico en la lnea " + s.left, null);
}

public void unrecovered_syntax_error(Symbol s) throws
  java.lang.Exception {
report_fatal_error("No se realiza recuperaci'on de errores",
                                                      null);
}
:};
```

El atributo *left* del objeto *s* de la clase Symbol, contiene el número de la línea en la que se encontraba el Token en el archivo de entrada. Vuelva a compilar (item e) y ejecute con los 2

códigos de prueba suministrados en el punto anterior y diga cuáles fueron las diferencias de las dos salida del analizador.

Gramática simplificada de JAVA^{TM} en BNFE

Programs

< compilation unit >::=[< type declarations >]

Declarations

< type declarations >::= < class declaration > {< class declaration >}

< class declaration >::= [***public***] ***class*** < identifier >< class body >

< class body >::= **{** [< class body declarations >] **}**

< class body declarations >::= < class body declaration > {< class body declaration >}

< class body declaration >::= < field declaration > | < method declaration >

< constructor body >::=**{** [< block statements >] **}**

< field declaration >::= < type >< variable declarators > ***;***

< method declaration >::= < method header >< method body >

< method header >::= < type >< method declarator >

< method declarator >::= < identifier > **(** [< formal parameter list >] **)**

< formal parameter list >::= < formal parameter > { **,** < formal parameter >}

< formal parameter >::= < type >< variable declarators >

< variable declarators >::= < identifier >

< method body >::= < block > | ***;***

Types

< type >::= ***int*** | ***char*** | < class type >

< class type >::= < type name >

Blocks and Commands

< block >::= **{** [< block statements >] **}**

< block statements >::= < block statement > {< block statement >}

< block statement >::= < local variable declaration > | < statement >

< local variable declaration >::= < type >< variable declarators >

< statement >::= < if then else statement > | < while statement >
< expression statement > | < return statement >

< expression statement >::= < statement expression > ***;***

< statement expression >::= < assignment > | < method invocation >

< if then else statement >::= ***if*** **(** < expression > **)** < statement > ***else***
< statement >

< while statement >::= ***while*** **(** < expression > **)** < statement >

Expressions

< expression >::= < assignment expression >

< assignment expression >::= < conditional expression > | < assignment >

< assignment >::= < left hand side >< assignment operator >< assignment expression >

< left hand side >::= < expression name >

< assignment operator >::= **=**

< conditional expression >::= < conditional or expression >

< conditional or expression >::= < conditional and expression >
{ **||** < conditional and expression >}

< conditional and expression >::= < equality expression >
{ **&&** < equality expression >}

< equality expression >::= < relational expression > { **==** < relational expression >}

< relational expression >::= < additive expression >
{ (**<** | **>**) < additive expression >}

< additive expression >::= < multiplicative expression >
{ (**+** | **−**) < multiplicative expression >}

< multiplicative expression >::= < unary expression >
{ (***** | **/**) < unary expression >}

< unary expression >::= < postfix expression >

< postfix expression >::= < primary > | < expression name >

< method invocation >::= < identifier > **(** [< argument list >] **)**

< primary >::= < primary no new array >

< primary no new array >::= < literal > | **(** < expression > **)**|
< method invocation >

< argument list >::= < expression > { **,** < expression >}

Tokens

< type name >::= < identifier >

< expression name >::= < identifier >

< method name >::= < identifier >

< identifier >::= < character > {< character > | < digit >}

< literal >::= < integer literal > | < floating-point literal > |
< character literal > | < string literal >

< integer literal >::=[< digits >]

< digits >::= < digit > {< digit >}

< digit >::= ***0*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** | ***7*** | ***8*** | ***9***

< floating-point literal >::= < digits > **.** [< digits >]

< character literal >::= **'** < single character > **'**

< single character >::= < input character >

< string literal >::= **"** [< string characters >] **"**

< string characters >::= < string character > {< string character >}

< string character >::= < input character >

< input character >::= < character > | < digit > | ***** | ***?*** | ***/*** | **...** | **^** | **=** | **...** etc.

< character >::= ***a*** | **...** | ***z*** | ***A*** | **...** | ***Z***