Práctica 2: Introducción al análisis sintáctico

Entrega 18 Septiembre 2018

1 Introducción a los analizadores sintácticos

El análisis sintáctico consiste en determinar si una sucesión de símbolos (una cadena) de un lenjuaje puede ser formada con las reglas de su gramática; es decir, pertenece al lenguaje.

Los lenguajes formales cuentan con modelos teóricos reconocedores, pero sin perder de vista que esos modelos serán implementados como parte de un compilador, necesitan tener buen desempeño.

Podemos garantizar un reconocimiento lineal para lenguajes regulares, pero éstos no cuentan con la expresividad suficiente que demanda un lenguaje de programación, es por ello que los lenguajes libres del contexto son una buena opción para la implementación de los lenguajes de programación. Como recordaremos, en general los lenguajes libres del contexto tiene reconocimiento de $O(n^3)$ y algunos de ellos son inherentemete ambiguos. Pero si nos limitamos a los lenguajes libres del contexto que nos son ambiguos, se puede garantizar un reconocimiento lineal.

Tradicionalmente existen dos tipos de analizadores sintácticos:

1. **Descendentes** (*Top-Down*). Los lenguajes libres del contexto que pueden ser reconocidos en tiempo lineal por uno de estos analizadores son identificados como LL (*Left-to-right Left-most-derivation*). Como su nombre lo indica buscan la derivación empezando por el símbolo inicial.

2. **Ascendentes** (*Bottom-UP*). Los lenguajes libres del contexto que pueden ser reconocidos en tiempo lineal por estos analizadores se les conoce como LR(*Left-to-right Right-most-derivation*) y contienen propiamente a todos los lenguajes LL. Dentro de esta clasificación existen otras, por ejemplo $LALR$ es importante mencionar esta clasificación ya que los lenguajes que son reconocidos por analizadores generados por herramientas como *bison*, *yacc* ó *byaccj* caén en esta clasificación. Entender el funcionamiento de estos analizadores es menos intuitivo,

ya que empieza con la cadena completa y por medio de reducciones llegar la símbolo inicial, si es que la cadena pertenece al lenguaje.

2 Byacc/J

Es un generador de analizadores sintácticos, la J es para denotar que los analizadores que genera están implementados en *Java* a diferencia de sus predecesores (*yacc*, *bison*). Hay muy poca documentación pero copia casi identicamente el funcionamiento y la sintáxis de *yacc*. En general este tipo de herramientas recibe los elementos de la gramática (símbolo inicial, símbolos terminales, símbolos no terminales y producciones) y construye el analizador sintáctico. En la construcción de un compilador, el único componente que trata directamente con el código fuente es el analizador léxico, el analizador sintáctico recibe los átomos reconocidos por el primero.

A continuación se describirá brevemente su instalación, sintaxis, funcionamiento e interacción con el analizador léxico.

2.1 Instalación

1. Descargar el comprimido que contienen el binario[1].
2. Instalar bibliotecas de compatibilidad con software para arquitectura de 32 bits[2]. En algunos sistemas operativos ya están instaladas.
3. Extraer el binario, `yacc.linux`.
4. Se sugiere que se mueva el binario a alguna ruta en el PATH para que se pueda ejecutar desde cualquier ruta.

```
# cp yacc.linux /usr/local/bin/byaccj
$ byaccj
  usage:
   byaccj [-dlrtvj] [-b file_prefix] [-Joption] filename
    where -Joption is one or more of:
     -J
     -Jclass=className
     -Jvalue=valueClassName (avoids automatic value class creation)
     -Jpackage=packageName
```

[1] `http://byaccj.sourceforge.net/#download`
[2] `http://askubuntu.com/questions/454253/how-to-run-32-bit-app-in-ubuntu-64-bit`

```
-Jextends=extendName
-Jimplements=implementsName
-Jsemantic=semanticType
-Jnorun
-Jnoconstruct
-Jstack=SIZE   (default 500)
-Jnodebug (omits debugging code for better performance)
-Jfinal (makes generated class final)
-Jthrows (declares thrown exceptions for yyparse() method)
```

2.2 Sintáxis

El archivo que *byaccj* recibe tienen las siguientes secciones:

- **Declaraciones**

  Entre `%{` y `%}`, si es necesario, se importan bibliotecas, sino se pueden omitir los delimitadores. Los símbolos terminales y no terminales también son descritos en esta sección y de la siguiente manera:

  ```
  %token "nombre" [ "nombre" ... ] /* Terminales */
  %type "nombre" [ "nombre" ... ]  /* No Terminales */
  ```

  Como se muestra en el ejemplo, los comentarios están permitidos. Internamente a cada uno de esos símbolos se les asigna un objeto de tipo `ParserVal` en el que pueden guardar su valor semántico. Si importar si durante el reconocimiento se utiliza el valor semántico de los símbolos, siempre es necesario declarar los símbolos terminales, ya que son el contrato con el analizador léxico.

- **Acciones**

  Las producciones de la grámatica son descritas de la siguiente manera:

  ```
  símbolo_no_terminal: <forma_sentencial>       { }
                     | <otra_forma_sentencial>  { }
                     ;
  ```

  Con forma sentencial debe entenderse una secuencia de símbolos terminales y/o símbolos no terminales. Se pueden escribir tantas formas sentenciales como sea necesario, sólo deben estar separadas por `|` y la lista debe terminar con `;`. Si se requiere hacer algo más que sólo el

reconocimiento sintáctico, puede ponerse código de *java* dentro de `{` y `}` que se encuentran a la derecha de las reglas y con la directiva `$n` se puede tener acceso al objeto que aloja el valor semántico del símbolo `n` de la forma sentencial.

La descripciones de las gramáticas por lo general se hacen en notación *EBNF*[3] la cuál incluye en su sintáxis cerraduras de Kleene, cerraduras positivas y `[e]` para denotar que la expresión `e` puede estar presente una o cero veces, en este sentido la sintáxia de *byaccj* no es compatible ya que no acepta ese tipo de azúcar sintáctica, sólo acepta notación *BNF*. Descomponer las reglas usando recursión será la opción para *byaccj*.

- **Código**

  En la última sección se puede escribir código en *java* que puede ser usado en las acciones de las reglas o para modificar alguna funcionalidad del analizador.

Las secciones son separadas por `%%` .

2.3 Funcionamiento e interacción con Jflex

El analizador sintáctico solicitará átomos al analizador léxico cada que requiera hacer una operación *shift*. Es decir solicita átomos sobre demanda. Por lo tanto sólo es requerida una pasada del código fuente para realizar ambos reconocimientos. El analizador léxico y el analizador sintáctico deben interactuar. Para ello deben estar conscientes de su existencia y establecer los términos de la comunicación.

- Para que el analizador léxico sepa de la existencia del analizador sintáctico, se le agregará un atributo y un nuevo constructor que permita la inicialización de ese nuevo atributo.

```
//Flexer.flex Segunda sección
private Parser yyparser;

/* Constructor original */
public Flexer(java.io.Reader in) {
  this.zzReader = in;
}
```

[3] `https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form`

```
/* Nuevo constructor */
public Flexer(java.io.Reader r, Parser parser){
  this(r);
  this.yyparser = parser;
}
```

- Para que el analizador sintáctico esté enterado de la existencia del analizador léxico se hace básicamente lo mismo que en el caso anterior

```
//Parser.y Tercera sección
/* Atributo nuevo */
private Flexer lexer;

/* lexer is created in the constructor */
public Parser(Reader r) {
    lexer = new Flexer(r, this);
}
```

  Es importante de recalcar que el código puede cambiar conforme se nombren las clases de los analizadores (En el ejemplo `Flexer` y `Parser`).

- Para acordar los términos bajo lo cuáles se hará la comunicación se necesitan agregar las siguientes líneas en `Parser.y`

```
//Primera sección
//Definición del vocabulario .
%token IDENTIFICADOR ENTERO REAL
%%
...
%%
//Tercera sección
/* Definición del método mediante el cual se le solicita
   al analizador léxico un átomo y el analizador léxico
   trabaja para devolver el siguiente átomo del código. */
private int yylex () {
  int yyl_return = -1;
  try {
    yyl_return = lexer.yylex();
```

```
  }
  catch (IOException e) {
    System.err.println("IO error :"+e);
  }
  return yyl_return;
}
```

Los átomos declarados en la primera sección son puestos como constantes estáticas dentro de la clase `Parser`

`lexer.yylex()` es la función del analizador léxico que hace el trabajo de análisis. Si se deseara leer los átomos de otra fuente (consola, por ejemplo) del método `Parser.yylex()` es el que debe ser modificado.

```
public class Parser {
 ...
  public final static short IDENTIFICADOR=259;
  public final static short ENTERO=260;
  public final static short REAL=261;
...
}
```

Estas constantes definen la interfaz del analizador sintáctico, es decir, cualquier analizador léxico que desee interactuar con él, debe hacerlo en términos de esos átomos. En nuestro caso:

```
//Flexer.flex Tercera sección
{REAL}                  { return Parser.REAL;}
{ENTERO}                { return Parser.ENTERO;}
{IDENTIFICADOR}         { return Parser.IDENTIFICADOR;}
```

Finalmente para poner a trabajar al analizador sintáctico sobre un archivo determinado:

```
//Parser.y  - Tercera sección
public static void main(String args[]) throws IOException {
  Parser yyparser = new Parser(new FileReader(args[0]));
  yyparser.yyparse();
}
```

3 Ejercicios

1. Instalar *byaccj*
2. Leer `http://matt.might.net/articles/grammars-bnf-ebnf/`.
3. Implementa un parser para cada una de las siguientes dos gramáticas descritas en formato *EBNF*. Utiliza: *byaccj* y *jflex*. Recuerda que *byaccj* acepta solo la notación *BNF*.

   - Gramática 1

     ```
     <expr> : {<expr> ("+"|"-"} <term>
     <term> : {<term> ("*"|"/")} <factor>
     <factor> : ["-"] NUMBER
     ```

   - Gramática 2

     ```
     <expr> : <term> {("+"|"-") <expr>}
     <term> : <factor> {("*"|"/") <term>}
     <factor> : ["-"] NUMBER
     ```

   Las gramáticas implementadas en sintaxis *EBNF* deben conservar el sentido de su recursión; hacia la derecha o hacia la izquierda.

   Los parsers debe tener el siguiente comportamiento ante una expresión aritmética bien construida:

   ```
   // entrada.txt
   1 + 2 + 4

   $ java Parser entrada.txt
   $ [ok]
   ```

   Comportamiento ante una expresión aritmética mal construida:

   ```
   // entrada.txt
   1 + 2 texto

   $ java Parser
   $ [ERROR] La expresión aritmética no esta bien formada.
   ```

4. Encuentra una manera de imprimir la pila de reconocimiento cada que se hace una reducción. Pista, revisa el código generado por *byaccj*

3.1 Extra (1 punto):

Modifica las gramáticas para que acepten un número arbitrario de expresiones aritméticas separadas por un salto de línea.