Práctico de Máquina Nro. 1
Análisis Lexicográfico

Fecha de Entrega del Práctico: **11 de Mayo de 2011**.
Debe ser realizado por grupos de no más de dos personas.

a. Analizar las gramáticas simplificada de *JAVA*^{TM} y *HTML* que se adjunta al final de este práctico y obtener para cada una, todos los tokens que dichas gramáticas pueden generar. Recordar que generalmente los token corresponden con los terminales de la gramática asociada. Las gramáticas adjuntas se encuentra escrita en BNF extendida, por lo cual:

- <field declaration> denota el no terminal "<field declaration>".
- $<$ ... $>$::=$<$ ... $>$ denota la producción $<$ ... $>\rightarrow<$ ... $>$
- $[<$ ... $>]$ denota la aparición o no (0 o 1 vez) del no terminal "<...>"
- $\{<$ ... $>\}$ denota 0 o más veces la aparición del no terminal "<...>"
- En el caso de la gramática de HTML, a los **no terminales** se les ha quitado el símbolo $<$ y $>$, para evitar inconvenientes, ya que son parte de los símbolos terminales. Por otro lado, los símbolos **terminales** están escritos en negrita para identificarlos de los no terminales.

b. Se debe diseñar e implementar un analizador lexicográfico para la gramática JAVA^{TM} y otro para la gramática HTML.

Para ello, se deben tomar en consideración los siguientes puntos:

- El reconocimiento de tokens:
  - Para ambas gramáticas:
    - Los blancos son significativos.
    - Las palabras claves son reservadas.
    - Una práctica habitual es almacenar las palabras reservadas en una **tabla de palabras reservadas** junto con el correspondiente código. De este modo, en su reconocimiento, primeramente se consideran como identificadores. Luego, el analizador determina si es una palabra reservada o es un identificador, recorriendo la tabla de palabras reservadas. Si es encontrada dentro de la tabla, se conforma el correspondiente token para la palabra reservada utilizando el código almacenado en la tabla. Si no se encuentra dentro de la tabla significa que es un identificador y se debe devolver el código de identificador.

- Las secuencias de símbolos entre comillas dobles (“ ... ”) no debe contener un fin de línea. Si no se encuentra la comilla que cierra antes del fin de línea, el analizador debe marcar el error y retornar el código de lo que haya reconocido, pero diferente al que hubiese retornado si encontrara la comilla doble que cierra la secuencia. Y continuará el análisis normalmente en la próxima línea.

- Los comentarios pueden incluir varias líneas del código fuente y deben terminar antes del fin de archivo:
  - Para la gramática JAVA^{TM} se denota con $/*\cdots*/$.
  - Para la gramática HTML se denota con $<!--\ldots-->$.

  El analizador debe reconocerlos, pero su contenido debe ser descartado, sin generar ni retornar el par (código,valor).
- En la gramática JAVA^{TM}, los identificadores pueden tener a lo sumo 16 caracteres. Por lo tanto, si se encuentra un identificador de mayor longitud deben considerase los primeros 16 caracteres y descartarse los restantes. Finalmente, se conforma el token con los 16 caracteres y se muestra un mensaje de aviso.
- El analizador debe detectar la mayor cantidad posible de errores, y siempre que se produzca un error, éste debe ser marcado y luego el analizador debe recuperarse, es decir, avanzar hasta una posición en la que pueda continuar analizando.

- La implementación del analizador
  1. Debe ser realizada en el lenguaje de programación JAVA^{TM}
  2. Debe incluir el diseño e implementación de todas las estructuras utilizadas. No se permite el uso de paquetes o librerías (ya existentes) que automaticen la construcción del analizador ni de los autómatas.
  3. Recordar que el análisis del código fuente es caracter a caracter, por lo cual no se pueden utilizar funcionalidades del lenguaje que permiten la lectura con formato.
  4. El analizador:
     - Debe ser invocado para reconocer un token.
     - Cuando se ha reconocido un token, se debe conformar el par $(codigo, valor)$ donde $valor$ será el lexema o la cadena vacía (“”) según corresponda.
     - Finalmente, se debe retornar al invocante con el par conformado.
     - El invocante debe almacenar en el archivo “salida.tok” cada uno de los pares $(codigo, valor)$.

c. La entrega del práctico debe cumplir con los siguientes puntos:

- Los archivos de los códigos fuentes (“.java”) debidamente comentados, necesarios para realizar la compilación y ejecución de para cada uno de los analizadores.

- Un informe escrito explicando el diseño e implementación de los analizadores, así como el listado de códigos para cada uno de los tokens reconocidos y todas aquellas consideraciones realizadas y que no se encuentren detalladas en el presente práctico.
- Los archivos (aplicación e informe) deben ser enviados a través del aula virtual.

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 >::= < primitive type > | < reference type >

< primitive type >::= < integral type > | < floating-point type >

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

< floating-point type >::= ***float*** | ***double***

< reference type >::= < class type > | < array type >

< class type >::= < type name >

< array type >::= < type > **[ ]**

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 > | < postincrement expression > |
< method invocation > | < class instance expression >

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

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

< return statement >::= ***return*** [< expression >] ***;***

Expressions

< expression >::= < assignment expression >

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

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

< left hand side >::= < expression name > | < array access >

< 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 >

< postincrement expression >::= < postfix expression > **++**

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

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

< primary >::= < primary no new array > | < array creation expression >

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

< class instance expression >::= ***new*** < class type > **(** [< argument list >] **)**

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

< array creation expression >::= ***new*** < primitive type > [< dims >]

< dims >::= **[** < expression > **]**

< array access >::= < expression name > **[** < expression > **]**

Tokens

< type name >::= < identifier >

< expression name >::= < identifier >

< method name >::= < identifier >

< 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***

Gramática simplificada de HTML en BNFE

ELEMENT_HTML ::= **<HTML>** HTML_CONTENT **</HTML>**

HTML_CONTENT ::= [ELEMENT_HEAD] [ELEMENT_BODY]

ELEMENT_HEAD ::= **<HEAD** [HEAD_ATTLIST]> HEAD_CONTENT **</HEAD>**

HEAD_ATTLIST ::= **profile =** URI

HEAD_CONTENT ::= [ELEMENT_TITLE]

ELEMENT_TITLE ::= **<TITLE>** TITLE_CONTENT **</TITLE>**

TITLE_CONTENT ::={ASCII}

ELEMENT_BODY ::= **<BODY** [BODY_ATTLIST]> BODY_CONTENT **</BODY>**

BODY_ATTLIST ::= {(CLASS | TITLE)}

BODY_CONTENT ::= {(ELEMENT_P | ELEMENT_A)}

ELEMENT_P ::= **<P>** P_CONTENT **</P>**

P_CONTENT ::= {ASCII}

ELEMENT_A ::= **<A** A_ATTLIST> A_CONTENT **</A>**

A_CONTENT ::={ASCII}

A_ATTLIST ::= **href =** URI

CLASS ::= **class = "**CDATA**"**

TITLE ::= **title = "**{ASCII}**"**

URI ::= **"**CDATA**"**

CDATA ::= LETRA{(LETRA|DIGITOS| - | _ | ***:*** | **.**)}

LETRA ::= ***a*** | **...** | ***z*** | ***A*** | **...** | ***Z***

DIGITOS ::= ***0*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** | ***7*** | ***8*** | ***9***

ASCII ::= LETRA | DIGITOS | ******* | ***?*** | ***/*** | **...** | ^ | = | **...** etc.