PRÁCTICA DE PROCESADORES DEL LENGUAJE I

Curso 2014 – 2015

Entrega de Febrero

APELLIDOS Y NOMBRE: **RAFAEL LACALLE MERINO**

IDENTIFICADOR:

DNI:

CENTRO ASOCIADO MATRICULADO: **ALZIRA-VALENCIA (VALENCIA)**

CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: **VALENCIA**

MAIL DE CONTACTO:

TELÉFONO DE CONTACTO:

GRUPO (A ó B): **B**

ÍNDICE

1.- El analizador léxico

El analizador léxico es el encargado de leer el fichero que contiene como entrada un programa escrito en el lenguaje cUNED, así como identificar todos los tokens que lo componen, detectar los posibles errores léxicos que contenga y entregar cada token válido al analizador sintáctico, según éste los vaya solicitando.

Ha sido desarrollado con la herramienta JFlex, está contenido en el archivo scanner.flex, y se encuentra ubicado en la ruta \doc\specs\ del fichero zip de entrega de la práctica.

Para la gestión de errores léxicos se utilizan las clases LexicalError (para encapsular la información relativa al error detectado) y LexicalErrorManager (para emitir el mensaje de error por la salida estándar con información sobre el lexema que produce el error, así como el número de fila y columna donde ha sido detectado).

El analizador léxico consta de 2 estados:

- YYNITIAL (Estado inicial)
  Estado de inicio y principal, realiza el reconocimiento de lexemas como tokens válidos si cumplen con la especificación del lenguaje cUNED, o como errores léxicos en caso contrario.

- COMENTARIOS (Tratamiento de comentarios)
  Nuevo estado creado para realizar el tratamiento de los comentarios del lenguaje, ignorando todas aquellas cadenas que se encuentran situadas entre los símbolos delimitadores de principio y final de comentarios. Permite el tratamiento de comentarios anidados.

Se añade una nueva directiva JLEX, %full, la cual permite reconocer caracteres pertenecientes al código ASCII extendido.

Se crean los siguientes patrones:

DIGITO	[0-9]	Reconoce dígitos del 0 al 9
LETRA	[a-z A-Z]	Reconoce cualquier letra
NUMERO	[DIGITO]+	Reconoce números formados por uno o más dígitos
IDENTIFICADOR	{(LETRA)}{(LETRA)|(DIGITO)}*	Reconoce como Identificador cualquier cadena formada por una secuencia de letras y dígitos y que empiece por una letra.
ESPACIO_BLANCO	[\ \t\b\r\n\f]+	Reconoce espacios en blanco
COMENTARIO_ABRIR	"/*"	Delimitador inicio de comentario
COMENTARIO_CERRAR	"*/"	Delimitador final de comentario
CONSTANTE_CADENA	[^\"]*	Secuencia de caracteres
fin	"fin"{ESPACIO_BLANCO}	Fin
	\"{CONSTANTE_CADENA}\"	Cadena de caracteres delimitada por comillas dobles

Se reconocen como errores léxicos los lexemas que no pertenecen a la especificación del grupo B, tales como: operador aritmético menos ("-"), operadores relacionales igual que ("==") y mayor que (">"), operador lógico or ("||"), operador de acceso a campo de registro ("."), y sentencia de flujo for ("for").

También se reconocen como errores los identificadores cuyo carácter inicial sea un dígito o un número y cualquier otro carácter no contemplado en ninguna expresión regular.

2.- El analizador sintáctico

El analizador sintáctico es el encargado de reconocer las cadenas que cumplen con la sintaxis correcta del lenguaje cUNED, definida en las producciones de la gramática, detectando los posibles errores sintácticos que puedan presentarse en el fichero de entrada.

Ha sido desarrollado con la herramienta Cup, está contenido en el archivo parser.cup, y se encuentra ubicado en la ruta \doc\specs\ del fichero zip de entrega de la práctica.

Para la gestión de errores léxicos se utilizan las clases SyntaxError (para encapsular la información relativa al error detectado) y SyntaxErrorManager (para emitir el mensaje de error por la salida estándar con información sobre el lexema que produce el error, así como el número de fila y columna donde ha sido detectado).
LexicalErrorManager.syntaxError emite un mensaje de error sintáctico.
LexicalErrorManager.syntaxFatalError emite un mensaje de error fatal y detiene el análisis.
También utilizamos LexicalErrorManager.syntaxInfo para emitir mensajes de traza del análisis sintáctico, así como los mensajes de inicio y finalización con éxito del análisis.

Se declaran como terminales los tokens definidos en el analizador léxico, tales como las palabras reservadas del lenguaje, los delimitadores, los operadores aritméticos, los operadores relacionales, los operadores lógicos, los operadores de asignación y los patrones (total: 29 símbolos) .

Se declaran como no terminales todos los símbolos situados en la parte izquierda de las producciones definidas en la gramática (total: 55 símbolos).

De acuerdo con lo estipulado en la especificación del lenguaje cUNED, se establecen las siguientes relaciones de precedencia (ordenadas de menor a mayor) y de asociatividad de terminales:

ASIGNACIÓN (“=”): Asociatividad a derechas.
AND_LOGICO (“&&”): Asociatividad a izquierdas.
DISTINTO_QUE (“!=”): Asociatividad a izquierdas.
MENOR_QUE (“<”): Asociatividad a izquierdas.
MAS (“+”): Asociatividad a izquierdas.
MATRIZ_ABRIR (“[“), MATRIZ_CERRAR (“]”): Asociatividad a derechas.
PARENT_ABRIR (“(“), PARENT_CERRAR (“)”): Asociatividad a izquierdas.
SINO (“else”): Asociatividad a izquierdas.

3.- Conclusiones

Se han realizado pruebas de análisis léxico y análisis sintáctico con los ficheros de entrada proporcionados por el equipo docente para la especificación del grupo B: ficheros con nombre testCaseXX.cuned, donde XX está comprendido entre el 01 y el 08, obteniendo los resultados esperados de análisis finalizado sin errores, en la práctica totalidad de los ficheros de prueba, presentando errores (posiblemente de transcripción) el fichero testCase07.cuned (renombrado como testCase07_error.cuned).

También se han creado dos nuevos ficheros de prueba: testCase10.cuned y testCase11.cuned, para comprobar el funcionamiento del compilador con ficheros de entrada que recojan la mayor parte de las estructuras del lenguaje cUNED definidas en la especificación del lenguaje. Las versiones finales de los nuevos ficheros de prueba, utilizados para las citadas comprobaciones, no presentan errores léxicos ni sintácticos.

La totalidad de los archivos de prueba, tanto los facilitados por el equipo docente como los dos nuevos archivos creados, se han ubicado en el directorio \doc\test\ del fichero zip de entrega de la práctica.

Como conclusiones fundamentales, podemos destacar de la realización de esta práctica el hecho que, aunque solamente se hayan tratado los análisis léxico y sintáctico, en mi caso ha sido suficiente para comprobar la gran importancia que tiene la correcta definición de los símbolos pertenecientes al lenguaje, y en mayor medida, la correcta especificación de las producciones de la gramática.

En el caso de los símbolos terminales (tokens), porque son los elementos que constituyen las piezas básicas con las que más tarde se construirán las frases, y en el caso de las producciones (reglas) de la gramática, porque constituyen el núcleo fundamental del reconocimiento de la sintaxis del lenguaje.

A lo largo de la realización de la práctica, he podido comprobar que una regla mal construida o no totalmente bien definida, puede introducir ambigüedad, suponiendo en muchas ocasiones tener que "desandar lo andado" y volver atrás en la construcción de las reglas, replanteándose de nuevo el diseño completo de la gramática.

Por último, es de reseñar la gran utilidad de los foros del curso virtual de la asignatura, donde han sido resueltas la mayor parte de las dudas surgidas durante la realización de la práctica, tanto por la participación de los compañeros exponiendo sus problemas, como por las respuestas y orientaciones del equipo docente, que en mi caso concreto me han servido para reconducir algunos aspectos, sobre todo en el diseño de la gramática.

4.- Gramática

4.1.- Consideraciones sobre el diseño de la Gramática

Antes de incluir el esquema de producciones, me gustaría realizar algunas consideraciones sobre el diseño concreto de la Gramática:

- La primera decisión surgió de la dificultad que en principio, al menos en mi caso concreto, supuso definir gramaticalmente la estructura de un programa en el lenguaje cUNED. Mi diseño original de las producciones iniciales introducía invariablemente ambigüedad en la gramática, al tener producciones vacías en el segundo nivel de las reglas.
  Dichas producciones vacías ocasionaban numerosos conflictos, tanto de reducción-desplazamiento como de reducción-reducción, que aunque el algoritmo de análisis ascendente LALR(1) trataba de resolver prefiriendo desplazar a reducir, y reducir por la primera regla, su acumulación suponía finalmente la detención del parser con errores, sin finalizar el análisis.
  Por lo tanto, la decisión de diseño adoptada ha sido la de dividir la primera regla (axiom), en tantas producciones como combinaciones posibles existen entre las cuatro "secciones" opcionales, más la función main, en que puede dividirse estructuralmente un programa en cUNED, prescindiendo con ello de las producciones vacías y evitando la ambigüedad inherente a su utilización.

- En el mismo sentido que lo expuesto en el párrafo anterior, he utilizado la misma decisión de diseño en la definición de la estructura de un bloque de sentencias (aunque en este caso sí es posible tener una producción vacía dentro de los delimitadores de un bloque, pero en el primer nivel de la estructura, sin introducir ambigüedad).
- Aunque en las reglas de definición de los tipos y de las variables sería posible reutilizar las mismas producciones (más genéricas que las utilizadas en mi caso), he preferido mantenerlas con distintas producciones para poder distinguir los tipos y variables globales de las locales y diferenciarlas en la traza de salida del analizador sintáctico.

- En el caso de la sentencia Return en funciones, la decisión adoptada ha sido la de reconocer la sentencia cuando se presente, sin comprobar su existencia en cada función o en cada salida de función, tarea que debe ser resuelta por el análisis semántico.
- En cuanto al paso de parámetros en la invocación de una función, me surgió una duda de última hora, al apreciar diferencias entre lo especificado en el enunciado de la práctica (página 21) y lo especificado en el documento de preguntas frecuentes sobre el Análisis Sintáctico.
  Dado que en el grupo B las variables se pasan por referencia, he optado por permitir únicamente como parámetros válidos los identificadores de variables, tal como se especifica en el documento de FAQ del Análisis Sintáctico:
  SIN.08. ¿Los parámetros por referencias pueden ser expresiones?
  R. Debido a que los parámetros por referencia son de salida o entrada / salida estos deben ser identificadores. Por tanto, no pueden ser expresiones.

- Finalmente, para resolver el problema del "else ambiguo", se ha diseñado forzando una máxima precedencia al "else" en las declaraciones de precedencia y asociatividad de la gramática.

4.2.- Esquema de producciones de la Gramática

Las producciones de la gramática son reglas, y por lo tanto, presentan un antecedente y un consecuente separados por el símbolo "::=", tal como se han utilizado en la herramienta Cup.
Antecedente de la regla: Un elemento no terminal.
Consecuente de la regla: Una forma de frase formada por elementos terminales, elementos no terminales, o por elementos terminales y no terminales.
Los elementos terminales se escriben con letras mayúsculas y los no terminales empiezan con minúsculas.

Reglas de producción de la gramática:

```
// ----------------------------
// INICIO PROGRAMA
// ----------------------------

axiom ::=       funcionPrincipal;
axiom ::=       constantes  funcionPrincipal;
axiom ::=       declaracionTipos  funcionPrincipal;
axiom ::=       declaracionVariables  funcionPrincipal;
axiom ::=       declaracionFunciones  funcionPrincipal;
axiom ::=       constantes  declaracionTipos funcionPrincipal;
axiom ::=       constantes  declaracionVariables  funcionPrincipal;
axiom ::=       constantes  declaracionFunciones  funcionPrincipal;
axiom ::=       declaracionTipos  declaracionVariables  funcionPrincipal;
axiom ::=       declaracionTipos  declaracionFunciones  funcionPrincipal;
axiom ::=       declaracionVariables  declaracionFunciones  funcionPrincipal;
axiom ::=       constantes  declaracionTipos  declaracionVariables  funcionPrincipal;
axiom ::=       constantes  declaracionTipos  declaracionFunciones  funcionPrincipal;
axiom ::=       constantes  declaracionVariables  declaracionFunciones  funcionPrincipal;
axiom ::=       declaracionTipos  declaracionVariables  declaracionFunciones  funcionPrincipal;
axiom ::=       constantes  declaracionTipos  declaracionVariables  declaracionFunciones  funcionPrincipal;

epsilon ::= ;       // producción vacía

// ---------------------------------------------------------------------
// Sección declaración de constantes simbólicas (opcional)
// ---------------------------------------------------------------------
constantes ::= constantes  declaracion_Const_Simb  |  declaracion_Const_Simb;
declaracion_Const_Simb ::= BLOQUE_CONSTANTES  IDENTIFICADOR  NUMERO  FIN_SENTENCIA ;
```

```
// ---------------------------------------------------------
// Sección declaración de tipos globales (opcional)
// ---------------------------------------------------------
declaracionTipos ::= declaracionTipos  declaracionTiposValidos  |  declaracionTiposValidos;
declaracionTiposValidos ::=  ENTERO  IDENTIFICADOR  tiposValidos;
tiposValidos ::= MATRIZ_ABRIR rango MATRIZ_CERRAR  MATRIZ_ABRIR rango MATRIZ_CERRAR
FIN_SENTENCIA;

// ---------------------------------------------------------------
// Sección declaración de variables globales (opcional)
// ---------------------------------------------------------------
declaracionVariables ::= declaracionVariables  declaracionVariablesValidas  |  declaracionVariablesValidas
declaracionVariablesValidas ::=  ENTERO  tipo_Entero  |  IDENTIFICADOR  tipo_Matriz;
tipo_Entero ::= IDENTIFICADOR lista_Enteros FIN_SENTENCIA;
tipo_Matriz ::= IDENTIFICADOR lista_Matrices FIN_SENTENCIA;
lista_Enteros ::= COMA  IDENTIFICADOR lista_Enteros  | ASIGNACION  NUMERO lista_Enteros  |  epsilon;
lista_Matrices ::= COMA  IDENTIFICADOR  lista_Matrices  |  epsilon;

// ---------------------------------------------------
// Sección declaración funciones (opcional)
// ---------------------------------------------------
declaracionFunciones ::=  declaracionFunciones  declaracionFuncionesValidas  |  declaracionFuncionesValidas;
declaracionFuncionesValidas ::=  funcionEntero  |  funcionTipoVacio;
funcionEntero ::= ENTERO  cabeceraFuncion;
funcionTipoVacio ::= TIPO_VACIO  cabeceraFuncion;
cabeceraFuncion ::= IDENTIFICADOR PARENT_ABRIR parametros PARENT_CERRAR  bloqueSentencias;
parametros ::=     sin_parametros  |  con_parametros;
sin_parametros ::=   epsilon;
con_parametros ::=  ENTERO  PASO_REFERENCIA  IDENTIFICADOR;
con_parametros ::=  IDENTIFICADOR  PASO_REFERENCIA  IDENTIFICADOR;
con_parametros ::=  ENTERO  PASO_REFERENCIA  IDENTIFICADOR COMA con_parametros;
con_parametros ::=  IDENTIFICADOR  PASO_REFERENCIA  IDENTIFICADOR COMA con_parametros;

// ---------------------------------------------------
// Sección función principal main (obligatorio)
// ---------------------------------------------------
funcionPrincipal ::= TIPO_VACIO  cabeceraPrincipal;
cabeceraPrincipal ::= PRINCIPAL  PARENT_ABRIR  PARENT_CERRAR  bloqueSentencias;


//-----------------------------------------------
// COMUNES A VARIAS SECCIONES
//-----------------------------------------------
// ------------------------------------------------------------------------
// Bloque de sentencias (obligatorio como cuerpo de una función)
// ------------------------------------------------------------------------
bloqueSentencias ::=  BLOQUE_CODIGO_ABRIR  bloque  BLOQUE_CODIGO_CERRAR;

bloque ::=          epsilon;
bloque ::=          listaTiposLocales;
bloque ::=          listaVariablesLocales;
bloque ::=          listaSentencias;
bloque ::=          listaTiposLocales  listaVariablesLocales;
bloque ::=          listaTiposLocales  listaSentencias;
bloque ::=          listaVariablesLocales  listaSentencias;
bloque ::=          listaTiposLocales  listaVariablesLocales  listaSentencias;
```

```
// ------------------------------------------------
// declaración de tipos locales  (opcional)
// ------------------------------------------------
listaTiposLocales ::=	listaTiposLocales  tipoLocal  |  tipoLocal;
tipoLocal ::= ENTERO  IDENTIFICADOR  MATRIZ_ABRIR rango MATRIZ_CERRAR  MATRIZ_ABRIR rango
MATRIZ_CERRAR  FIN_SENTENCIA;

// ------------------------------------------------
// declaración de variables locales  (opcional)
// ------------------------------------------------
listaVariablesLocales ::=  listaVariablesLocales  variableLocal  |  variableLocal;
variableLocal ::=  ENTERO tipo_Entero_Local  |  IDENTIFICADOR  tipo_Matriz_Local;
tipo_Entero_Local ::=	IDENTIFICADOR  enteros_Locales  FIN_SENTENCIA;
tipo_Matriz_Local ::=  IDENTIFICADOR  matrices_Locales  FIN_SENTENCIA;
enteros_Locales ::= COMA IDENTIFICADOR enteros_Locales  |  ASIGNACION NUMERO enteros_Locales  | epsilon;
matrices_Locales ::= COMA IDENTIFICADOR matrices_Locales  |  epsilon;
rango ::= NUMERO  |  IDENTIFICADOR;

// --------------
// Sentencias
// --------------

listaSentencias ::=  listaSentencias  bloqueSentencias  |  listaSentencias  sentencia;
listaSentencias ::=  bloqueSentencias  |  sentencia;
sentencia ::=	sentencia_If;
sentencia ::=	sentencia_while;
sentencia ::=	sentenciaFuncion;
sentencia ::=	asignacion;
sentencia ::=	imprimeCadena;
sentencia ::=	imprimeEntero;
sentencia ::=	sentenciaReturn;

// sentencia condicional if-else
sentencia_If ::=  SI  PARENT_ABRIR expresion PARENT_CERRAR listaSentencias;
sentencia_If ::=  SI  PARENT_ABRIR expresion PARENT_CERRAR listaSentencias SINO listaSentencias;

// sentencia condicional while
sentencia_while ::=  BUCLE_WHILE PARENT_ABRIR expresion PARENT_CERRAR listaSentencias;

// sentencias de asignación
asignacion ::=	IDENTIFICADOR tipo_asig;
tipo_asig ::=	ASIGNACION  expresion  FIN_SENTENCIA;
tipo_asig ::=	MATRIZ_ABRIR rango MATRIZ_CERRAR  MATRIZ_ABRIR rango MATRIZ_CERRAR
ASIGNACION  expresion  FIN_SENTENCIA;

// sentencia de invocación de procedimientos (funciones que se comportan como procedimientos)
sentenciaFuncion ::=  expInvocacionFunciones  FIN_SENTENCIA;

// sentencia imprimir cadena
imprimeCadena ::= IMPRIME_CADENA  PARENT_ABRIR  CONSTANTE_CADENA
PARENT_CERRAR  FIN_SENTENCIA;

// sentencia imprimir entero
imprimeEntero ::= IMPRIME_ENTERO  PARENT_ABRIR  expresion  PARENT_CERRAR
FIN_SENTENCIA;
```

```
// sentencia return
sentenciaReturn ::= RETORNO_FUNCION expresion FIN_SENTENCIA;
sentenciaReturn ::= RETORNO_FUNCION FIN_SENTENCIA;

// ----------------
//  Expresiones
// ----------------

expresion ::= expSuma;
expresion ::= expLogica;
expresion ::= expAccesoMatriz;
expresion ::= expInvocacionFunciones;
expresion ::= expParentizada;
expresion ::= IDENTIFICADOR | NUMERO;

// Expresiones aritméticas tipo suma
expSuma ::=     expresion MAS expresion;

// Expresiones lógicas y de comparación de expresiones
expLogica ::=   expresion comparador expresion;
comparador ::= AND_LOGICO | MENOR_QUE | DISTINTO_QUE;

// Expresiones de acceso a matrices
expAccesoMatriz ::=     IDENTIFICADOR MATRIZ_ABRIR expresion MATRIZ_CERRAR
MATRIZ_ABRIR expresion MATRIZ_CERRAR;

// Expresiones de invocación de funciones
expInvocacionFunciones ::=IDENTIFICADOR PARENT_ABRIR parametros_funcion PARENT_CERRAR;
parametros_funcion ::=  sin_parametros | con_parametros_funcion;
con_parametros_funcion ::= IDENTIFICADOR | IDENTIFICADOR COMA con_parametros_funcion;

// Expresiones parentizadas (para modificar precedencia de operadores)
expParentizada ::= PARENT_ABRIR expresion PARENT_CERRAR;
```

Asociatividad:

- Left: asociatividad a izquierdas.
- Right: asociatividad a derechas.
- Nonassoc: el operador no es asociativo.

Declaración de relaciones de Precedencia: ordenada de menor a mayor precedencia:

```
precedence right        ASIGNACION;
precedence left         AND_LOGICO;
precedence left         DISTINTO_QUE;
precedence left         MENOR_QUE;
precedence left         MAS;
precedence right        MATRIZ_ABRIR, MATRIZ_CERRAR;
precedence left         PARENT_ABRIR, PARENT_CERRAR;
precedence left         SINO;
```

Valencia, 16 de febrero de 2015
Fdo.: RAFAEL LACALLE MERINO