Departamento de Tecnologías de la Información

Área de Ciencias de la Computación e Inteligencia Artificial

Teoría de Autómatas y Lenguajes Formales

Práctica 4

1.- OBJETIVOS

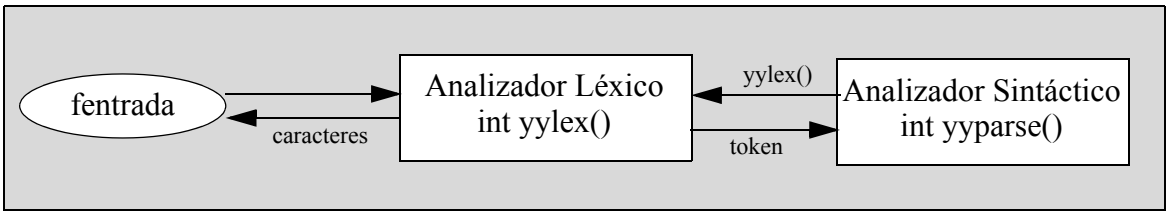
El objetivo de esta práctica es realizar un analizador sintáctico descendente recursivo en C, para el lenguaje planteado durante el curso. El analizador sintáctico a generar se apoyará en el analizador léxico desarrollado en la práctica 2.

2.- ANÁLISIS SINTÁCTICO DESCENDENTE RECURSIVO

2.1 Características generales

Un analizador sintáctico es una aplicación que recibe como entrada un flujo de componentes léxicos (tokens) y verifica que dicha secuencia es válida para una cierta gramática libre de contexto. Dada una gramática, existen diferentes formas de construir un analizador sintáctico que la reconozca. En la práctica anterior se presentó una herramienta, llamada Bison, que desarrolla de manera automática un analizador basado en un algoritmo de análisis sintáctico ascendente (Bison utiliza el algoritmo de análisis LALR). En esta práctica el analizador sintáctico se debe construir de forma manual siguiendo un algoritmo de análisis descendente.

El desarrollo del analizador sintáctico se va a traducir en un la programación de un fichero en C, que llamaremos ***Anasint.c***, que contiene un conjunto de funciones dedicadas a analizar la gramática deseada. El punto de entrada del analizador va a ser una función que vamos a llamar ***yyparse()*** (por mantener el mismo nombre que el utilizado por la herramienta Bison). La función *yyparse()* debe devolver un valor entero. Si el fichero de entrada analizado es correcto, es decir, si corresponde a la gramática deseada, la función *yyparse()* debe devolver 0. Si, por el contrario, el fichero de entrada es incorrecto, es decir, si no corresponde a la gramática deseada, la función *yyparse()* devolverá el valor 1.



Internamente, las funciones del analizador sintáctico van a llamar a la función *yylex()*, que desarrolla el análisis léxico. El analizador léxico es el único módulo que lee realmente el contenido del fichero de entrada. Cada llamada al método *yylex()* devuelve el código del último componente léxico leído. El valor de dicho componente léxico se debe almacenar en la variable *yylval*. En el desarrollo de esta práctica se hará uso del analizador léxico desarrollado en la segunda práctica de la asignatura.

2.2 Algoritmo de análisis sintáctico descendente recursivo

Los analizadores sintácticos descendentes recursivos son una forma sencilla de programación “a mano” de analizadores sintácticos. Sin embargo, esta técnica no puede emplearse

sobre cualquier gramática libre de contexto, sino sólo sobre el conjunto de gramáticas denominadas LL(1). La idea básica de este tipo de análisis consiste en asociar una función a cada símbolo no terminal de la gramática. El objetivo de esta función es reconocer sintácticamente ese símbolo. De esta manera, la ejecución de una regla formada por los símbolos *A*, *B* y *C* consiste en ejecutar las funciones asociadas a esos simbolos, es decir, " *A( ); B( ); C( );* ". Para describir el código de un analizador sintáctico descendente recursivo vamos a explicar el código asociado a la siguiente gramática:

```
Lista ::=  Elemento  SigueLista

SigueLista ::=  ‘,’  Elemento SigueLista
                |  λ

Elemento ::= ID
             |  NUMERO
```

Esta gramática corresponde a un lenguaje formado por listas de identificadores o números separados por coma. A continuación se muestra el código C que desarrolla un analizador sintáctico descendente recursivo para esta gramática.

```
#include <stdio.h>
#include “analex.h”
#include “anasint.h”

extern int yylex();                          /* Esta es la función que desarrolla el análisis léxico */
extern void yyerror(const char*  s);    /* Esta es la función de error sintáctico */

int nextToken;                               /*  Token de preanálisis  */

int yyparse()
{
   nextToken = yylex();
   if( parseLista() != 0 )  { yyerror(“Fichero INCORRECTO”); return 1; }
   if( nextToken != EOF) { yyerror(“Fichero INCORRECTO”); return 1; }
   return 0;
}

int Reconocer(int tk)
{
   char msg[100];
   if( tk != nextToken )
   {
      sprintf(msg,“Encontrado %d; se esperaba %d \n”, nextToken, tk);
      yyerror(msg);
      return 1;
   }
   nextToken = yylex();
   return 0;
}
```

```
int parseLista()                    /*  Función asociada al símbolo Lista  */
{
   char msg[100];
   switch(nextToken) {
      case ID:
      case NUMERO:
         fprintf(yyout,"Ejecutar regla 1");
         if( parseElemento() != 0 ) { return 1; }
         if( parseSigueLista() != 0 ) { return 1; }
         return 0;
      default:
         sprintf(msg, "Encontrado %d; esperado ID o NUMERO", nextToken);
         yyerror(msg);
         return 1;
   }
}

int parseSigueLista()               /* Función asociada al símbolo SigueLista  */
{
   char msg[100];
   switch(nextToken) {
      case ',':
         fprintf(yyout,"Ejecutar regla 2");
         if( Reconocer(',') != 0 ) { return 1; }
         if( parseElemento() != 0 ) { return 1; }
         if( parseSigueLista() != 0 ) { return 1; }
         return 0;
      case EOF:
         fprintf(yyout,"Ejecutar regla 3");
         return 0;
      default:
         sprintf(msg, "Encontrado %d; esperado COMA o EOF", nextToken);
         yyerror(msg);
         return 1;
   }
}

int parseElemento()                 /* Función asociada al símbolo Elemento  */
{
   char msg[100];
   switch(nextToken) {
      case ID:
         fprintf(yyout,"Ejecutar regla 4");
         if( Reconocer(ID) != 0 ) { return 1; }
         return 0;
      case NUMERO:
         fprintf(yyout,"Ejecutar regla 5");
         if( Reconocer(NUMERO) != 0 ) { return 1; }
         return 0;
      default:
         sprintf(msg, "Encontrado %d; esperado ID o NUMERO", nextToken);
         yyerror(msg);
         return 1;
   }
}
```

2.3 Descripción del código

El código comienza con la importación de los ficheros de cabecera del analizador léxico (donde se define yylex(), yyin, etc.), del analizador sintáctico (donde se definen los códigos asociados a cada símbolo terminal) y de la biblioteca <stdio.h> (donde se define la función *sprintf()*).

```
#include <stdio.h>
#include “analex.h”
#include “anasint.h”
```

A continuación se declaran la función *yylex()* (esto no es necesario si esta función está declarada en “*analex.h*”) y la función *yyerror()* (que podría estar declarada por ejemplo en el fichero de cabecera “*error.h*”).

```
extern int yylex();
extern void yyerror(const char*  s);
```

La siguiente línea contiene la declaración de la variable global interna *nextToken*. Esta variable contiene el código del último token leído por el analizador léxico. Este token se conoce como ***token de preanalisis*** o ***lookahead*** y se utiliza para predecir cual de las reglas de un símbolo no terminal es la que se debe expandir en cada momento.

```
int nextToken;
```

La función principal del analizador sintáctico se llama *yyparse()*. Esta función devuelve el valor 0 cuando el análisis es correcto y 1 cuando el análisis es incorrecto, es decir, cuando el fichero de entrada contiene algún error sintáctico. El contenido de la función consiste en leer el primer token del fichero de entrada, reconocer el símbolo inicial de la gramática y comprobar que después de reconocer el símbolo inicial sólo queda al final de entrada. Si es así el análisis ha sido correcto y la función devuelve 0. En caso contrario la función devuelve el valor 1.

```
int yyparse()
{
   nextToken = yylex();
   if( parseLista() != 0 )  { ..... }
   if( nextToken != EOF) { ..... }
   return 0;
}
```

La función *Reconocer(int)* se utiliza para reconocer un token (un símbolo terminal) como parte de la ejecución de una regla. La variable *msg* se utiliza para generar un mensaje en caso de error. El contenido de la función consiste en verificar que el siguiente token coincide con el token que se desea reconocer. Si es así, el token se “consume” y se lee el siguiente token de la cadena de entada por medio de la función *yylex()*. Si el token a reconocer no coincide con el siguente token se genera un error. La función devuelve el valor 1 en caso de error y 0 si el token es reconocido correctamente.

```
int Reconocer(int tk)
{
   char msg[100];
   if( tk != nextToken )
   {
      sprintf(msg,.....);
      yyerror(msg);
      return 1;
   }
   nextToken = yylex();
   return 0;
}
```

El código incluye una función para cada símbolo no terminal de la gramática cuyo objetivo es reconocer ese símbolo. Por ejemplo, la función *parseLista()* es la encargada de reconocer el símbolo *Lista*. Estas funciones devuelven el valor 0 si el símbolo se reconoce correctamente y 1 si se produce algún error en el proceso de análisis.

El contenido de la función se basa en el uso de una instrucción *switch(nextToken)* que permite seleccionar la regla que se va a ejecutar en cada caso. Los valores de *nextToken* que permiten seleccionar una regla se conocen como ***conjunto de predicción de la regla***. El proceso para calcular estos conjuntos se describe en el próximo apartado.

```
int parseLista()
{
   char msg[100];
   switch(nextToken) {
      case ID:
      case NUMERO:
         fprintf(yyout,"Ejecutar...");
         if( parseElemento() != 0 ) ...
         if( parseSigueLista() != 0 ) ..
         return 0;
      default:
         sprintf(msg, .....);
         yyerror(msg);
         return 1;
   }
}
```

Dentro de la instrucción *switch*, el código de cada regla comienza con las cláusulas *case* correspondiente a los tokens del conjunto de predicción de esa regla. Por ejemplo, el conjunto de predicción de la regla

Lista ::= Elemento SigueLista

es el conjunto { ID, NUMERO }, lo que se traduce en dos cláusulas *case* con estos valores.

A continuación aparece el código de la regla. Este código consiste en ejecutar las funciones asociadas al reconocimiento de cada símbolo de la regla. Para símbolos terminales (tokens) la función es *Reconocer(token)*. Para símbolos no terminales la función es *parseSimbolo()*. Por ejemplo, el código de ejecución de la regla (*S ::= A b C*) sería *parseA(); Reconocer(b); parseC();* Para detectar los errores, se comprueba que el resultado de cada llamada es 0. En caso contrario se habrá producido un error de análisis en el símbolo correspondiente y el resultado de la función debe ser 1. Si no hay errores de ejecución en el código de la regla, el resultado final es 0.

La instrucción *switch* contiene un cláusula *default* que permite detectar errores. En tal caso se lanza la función yyerror con un mensaje de error y se devuelve el valor 1.

2.4 Cálculo de los conjuntos de predicción

Los analizadores sintácticos descendentes recursivos son analizadores predictivos, es decir, utilizan la información del siguiente token (*nextToken*) para predecir que regla es la que debe ejecutarse en cada momento. Si un símbolo no terminal de la gramática está descrito por medio de varias reglas, por ejemplo, ($S ::= \alpha_1 \mid \alpha_2 \mid \alpha_3$), el valor del siguiente token se utiliza para decidir si se debe ejecutar la regla α_1, la regla α_2 o la regla α_3. Para poder decidir entre estas reglas es necesario que sus conjuntos de predicción sean disjuntos, es decir, que no haya ningún token que aparezca en los conjuntos de predicción de varias reglas de un mismo símbolo. Esto se conoce como condición LL(1) y garantiza que una gramática pueda ser analizada por medio de un análisis descendente predictivo con un único token de preanálisis.

Para calcular los conjuntos de predicción de las reglas de una gramática es necesario calcular en primer lugar otros dos conjuntos: el conjunto Primeros y el conjunto Siguientes.

El conjunto Primeros se calcula sobre una secuencia de símbolos (terminales o no terminales) y pretende expresar los tokens por los que puede comenzar esa secuencia. Para calcular este conjunto se utilizan las siguientes reglas:

1. $Prim(\lambda) = \{\lambda\}$, es decir, el conjunto primeros de una secuencia vacía está formado por el símbolo λ.
2. $Prim(a) = \{a\}$, es decir, el conjunto primeros de un símbolo terminal (un token) está formado por dicho símbolo terminal.

3. $Prim(A|A ::= \alpha_1|\alpha_2|\alpha_3) = Prim(\alpha_1) \cup Prim(\alpha_2) \cup Prim(\alpha_3)$, es decir, el conjunto primeros de un símbolo no terminal es la unión de los conjuntos primeros de todas sus reglas.
4. Si $\lambda \notin Prim(A)$, $Prim(A\beta) = Prim(A)$, es decir, si el conjunto Prim(A) no contiene λ, el conjunto primeros de la secuencia Aβ es Prim(A).
5. Si $\lambda \in Prim(A)$, $Prim(A\beta) = (Prim(A) - \{\lambda\}) \cup Prim(\beta)$, es decir, si el conjunto Prim(A) contiene λ, el conjunto primeros de la secuencia Aβ se calcula quitando λ de Prim(A) y añadiendo Prim(β).

Por su parte, el conjunto Siguientes se aplica a los símbolos no terminales de la gramática y contiene el conjunto de símbolos terminales (tokens) que pueden aparecer detrás de un símbolo no terminal. Las instrucciones para calcular este conjunto son las siguientes:

1. Si *S* es el símbolo inicial de la gramática, entonces el final de entrada (que se denota EOF o $) pertenece a *Sig(S)*.
2. Si la gramática contiene una regla *A::=* α*B*, entonces todos los elementos de *Sig(A)* se encuentran también en *Sig(B)*.
3. Si la gramática contiene una regla *A::=* α*B*β, y $\lambda \notin Prim(\beta)$, entonces todos los elementos de *Prim(*β*)* se encuentran también en *Sig(B)*.
4. Si la gramática contiene una regla *A::=* α*B*β, y $\lambda \in Prim(\beta)$, entonces los elementos de *Prim(*β*)* salvo λ se encuentran en *Sig(B)* y los elementos de *Sig(A)* se encuentran también en *Sig(B)*.

Estas reglas permiten definir un algoritmo para calcular los conjuntos siguientes de todos los símbolos no terminales de la gramática:

1. Inicializar todos los conjuntos siguientes al conjunto vacío. $\forall A \in \Sigma_N, Sig(A) = \{""\}$
2. Incluir el final de entrada en el conjunto siguientes del símbolo inicial. $Sig(S) = \{\$\}$
3. Para toda regla *A::=* α*B*, actualizar: $Sig(B) = Sig(B) \cup Sig(A)$
4. Para toda regla *A::=*α*B*β
   Si $\lambda \notin Prim(\beta)$, actualizar: $Sig(B) = Sig(B) \cup Prim(\beta)$
   Si $\lambda \in Prim(\beta)$, actualizar: $Sig(B) = Sig(B) \cup (Prim(\beta) - \{\lambda\}) \cup Sig(A)$
5. Repetir 3 y 4 hasta que no haya más cambios en los conjuntos.

Una vez calculados los conjuntos Primeros y Siguientes es fácil calcular los conjuntos de predicción de cada regla. Estos conjuntos de predicción están formados por los tokens que "disparan" la ejecución de esa regla, es decir, por los tokens que hay que utilizar como cláusulas *case* en la función asociada a cada símbolo. Para calcular el conjunto de predicción de una regla se utilizan estas instrucciones:

1. $Pred(A ::= \alpha) = Prim(\alpha)$, si $\lambda \notin Prim(\alpha)$
2. $Pred(A ::= \alpha) = (Prim(\alpha) - \{\lambda\}) \cup Sig(A)$, si $\lambda \in Prim(\alpha)$

3.- GRAMÁTICA DEL LENGUAJE A ANALIZAR

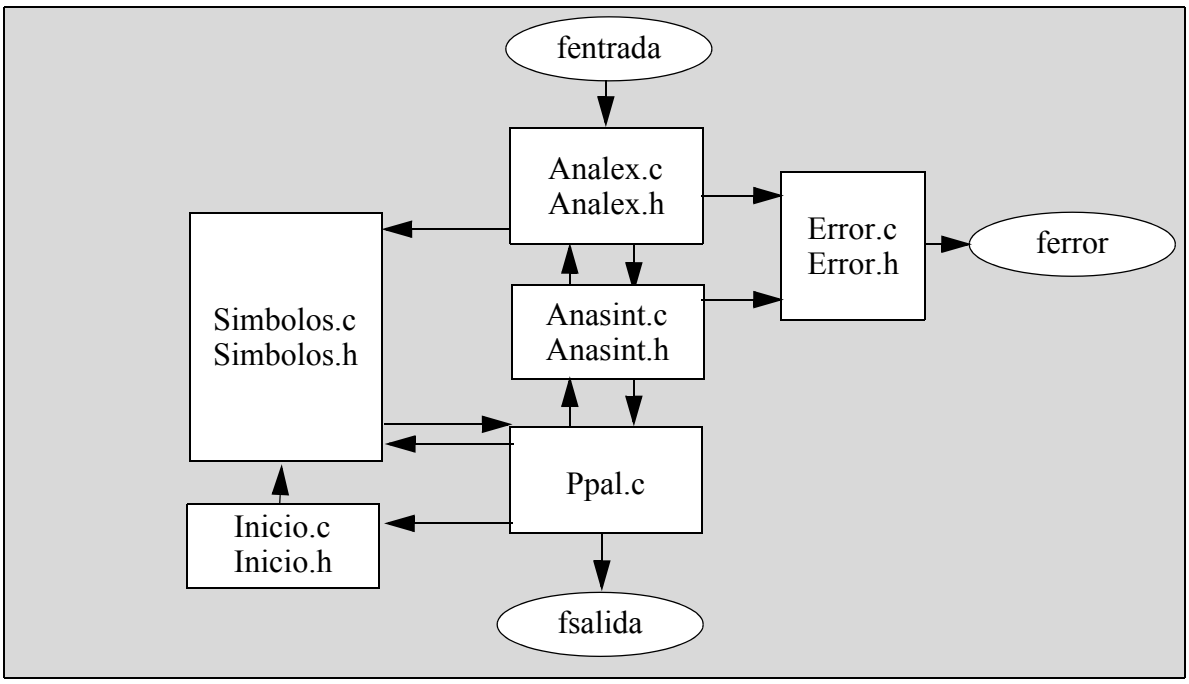
A continuación se muestra una gramática que describe el lenguaje objetivo de las prácticas de este curso. Esta gramática cumple la condición LL(1) y, por tanto, puede ser procesada por

medio de un analizador sintáctico descendente recursivo como el descrito en el apartado anterior.

```
Fichero ::= PROGRAMA  INICIO  ListaInstrucciones FIN

ListaInstrucciones ::=  Instruccion ListaInstrucciones
                      | λ

Instruccion ::= Lectura
              | Escritura
              | Asignacion
              | Condicional

Lectura ::=  LEER  '('  ListaIdentificadores  ')'  ';'

Escritura ::= ESCRIBIR  '('  ListaExpresiones  ')'  ';'

Asignacion ::=  ID  '='  Expresion  ';'

Condicional  ::=  SI  '('  Comparacion  ')'  '{'  ListaInstruciones  '}'  EnOtroCaso

EnOtroCaso ::=  SINO '{'  ListaInstruciones  '}'
              | λ

ListaIdentificadores ::=  ID  MasIdentificadores

MasIdentificadores ::=  ','  ID  MasIdentificadores
                      | λ

ListaExpresiones ::=  Expresion  MasExpresiones

MasExpresiones ::=  ','  Expresion  MasExpresiones
                      | λ

Comparacion ::= Expresion  Comparador  Expresion

Comparador ::=  '=' '='
              | '!' '='
              | '>'
              | '<'

Expresion ::=  Termino  MasTerminos

MasTerminos ::=  '+'  Termino  MasTerminos
               | '-'   Termino MasTerminos
               | λ

Termino ::=  Factor  MasFactores

MasFactores ::=  '*'  Factor  MasFactores
               | '/'   Factor MasFactores
               | λ

Factor ::=  ID
          | NUMERO
          | HEXADECIMAL
          | '('  Expresion ')'
```

4.- ESTRUCTURA DEL CÓDIGO A GENERAR

El proyecto a desarrollar debe estar formado por varios módulos que se relacionan entre sí tal y como muestra a siguiente figura:



El proyecto a desarrollar debe estar formado por los siguientes módulos:

- **Analex.c**: es el analizador léxico generado en la práctica 2.
- **Analex.h**: fichero cabecera del analizador léxico.
- **Anasint.c**: es el analizador sintáctico descendente recursivo.
- **Anasint.h**: fichero de cabecera del analizador sintáctico. Su contenido es básicamente el del fichero Tokens.h de la práctica 2.
- **Simbolos.c**: contiene las rutinas de manejo de la Tabla de Símbolos.
- **Simbolos.h**: archivo de cabecera asociado al fichero Simbolos.c.
- **Inicio.c**: llama a las rutinas de Simbolos.c para inicializar la Tabla de Símbolos.
- **Inicio.h**: archivo de cabecera asociado al fichero Inicio.c.
- **Error.c**: tratamiento de los errores (por ej. void error(int e)).
- **Error.h**: archivo de cabecera asociado al fichero Error.c.
- **Ppal.c**: contiene el programa principal

La ejecución del proyecto sobre un fichero correcto debe producir un fichero de salida en el que aparezcan los mensajes de ejecución de cada regla, indicando de esta forma la traza del proceso de análisis sintáctico. Por su parte, la ejecución sobre un fichero incorrecto debe escribir en el fichero de salida el contenido de la traza hasta el punto en el que se produce el error sintáctico, seguido de un mensaje de explicación del error detectado.

5.- DOCUMENTACIÓN A PRESENTAR

La documentación a presentar de la práctica debe contener los ficheros fuentes indicados, el ejecutable generado, un conjunto de pruebas desarrolladas y un fichero memoria.doc con la descripción de la gramática empeada, el cálculo de los conjuntos Primero, Siguientes y de Prediccion y una breve descripción de la implementación realizada.

6.- PLAZO DE ENTREGA

La fecha límite de entrega de la práctica es el día 27 de mayo. La práctica se entregará por medio de la plataforma moodle.