# PCCTS: DLG, ANTLR, SOURCERER

PCCTS (Purdue Compiler Construction Tool Set) es una herramienta para crear compiladores a partir de un único fichero en donde se especifican los componentes léxicos a reconocer, la gramática y las acciones semánticas a aplicar.

Trabaja con gramáticas LL(k) con k ≥ 1, lo cual es suficiente para procesar la mayoría de las construcciones de los lenguajes de programación. **Es un método de análisis sintáctico descendente.**

PCCTS posee varias características que lo hacen superior a otros generadores de analizadores:

- Integra en un único fichero la especificación del análisis léxico y sintáctico.
- Acepta construcciones de la gramática en la forma Backus-Naur extendida (EBNF). Por ejemplo:

```
exp → term ( TKN_MAS term | TKN_MENOS term)*
term → factor ( TKN_MULT factor | TKN_DIV factor )*
factor → TKN_NUM
```

- Proporciona facilidades para la construcción y recorrido del árbol sintáctico (prefijo, postfijo u otros recorridos definidos por el programador) y ejecutar las acciones semánticas. Especialmente útil para hacer traducciones de un lenguaje fuente a otro lenguaje fuente.

- Genera analizadores recursivos descendentes en C/C++, generando una función para cada símbolo no-terminal de la gramática, lo que facilita la depuración del código.

- Permite que a las reglas de la gramática (símbolos no-terminales) se les puedan pasar parámetros y devuelvan valores (como si fueran funciones) Pueden devolver múltiples valores.

**El PCCTS está compuesto de tres herramientas:**

**DLG**: (Direct Lexer Generator) es un generador de analizadores léxicos. Tiene como entrada una descripción de los tokens y crea el AFD correspondiente.

**ANTLR**: es un generador de analizadores sintácticos. (ANother Tool for Language Recognition). Tiene como entrada una descripción de la gramática y construye un analizador descendente asociando una función a cada una de las producciones de la gramática.

**SOURCERER**: es una herramienta para generar traductores de código fuente a código fuente de otro lenguaje. También se utiliza para generación de código

intermedio y código objeto. Se usa para construir los árboles y ejecutar acciones en su recorrido que hagan la traducción.

**genmk**: sirve para generar el makefile. Toma un fichero ".g" como entrada (fichero con la descripción léxica, sintáctica y construcción del árbol) y genera el makefile correspondiente.

## Estructura de un fichero de especificación.

```
#header
<< /* includes, declaraciones de variables externas, typedefs, estructuras de datos utilizadas, ficheros propios de PCCTS relacionados con la clase Token y la clase Analizador Sintactico, Cualquier código en C/C++.
Todo lo que se incluya en esta parte se copia en todos los ficheros generados por el PCCTS. ¡Cuidado! Nunca definir aquí variables, pues habría problemas de redefinición. */
>>
<< /*Definición de variables, función main(), código auxiliar necesario... */
>>

Descripción léxica:
#token TKN_NOMBRE "expresión_regular"
. . .
```

Lista de tokens y sus expresiones regulares. Se pueden incluir acciones que se ejecutan cuando se reconoce el token correspondiente. Las expresiones regulares van encerradas entre " "

- Los tokens siempre deben empezar por una letra mayúscula.
- El orden de colocación es importante pues si la entrada corresponde con dos expresiones regulares, se elige la 1ª.
- Se pueden usar los siguientes símbolos para especificar las expresiones regulares: * , + , | , ( ) , [ ] , ~ (indica negación) , { } (indica opcionalidad, es decir, para indicar que debe aparecer cero o una vez).
- Para especificar un token con un caracter corresponde a uno de los que se utilizan para definir las expresiones regulares, se debe poner el carácter de escape \ para quitarle el significado.

```
Descripción sintáctica:
class MiParser /* clase para el analizador sintáctico  */
{
      /* nombre_de_la_regla : descripcion_de_la_regla ; */
}
```

Los nombres de los símbolos no-terminales deben empezar por minúscula.

## Ejemplo: Una pequeña **calculadora (calc.g):**

```
#header
<<
#include <iostream>
#include <string>
#include <math.h>
#include "AToken.h" /*fichero generado por PCCTS, class Token */
typedef ANTLRCommonToken ANTLRToken; /* fichero generado por PCCTS, class Token */
#include "MiParser.h" /* fichero generado por PCCTS, Anal. Sintc. */
extern int numlineas; /* Para acceder al número de líneas desde cualquier modulo, necesito una variable global */
>>

<<
#include "DLGLexer.h" /* Parte del anal. léxico */
#include "PBlackBox.h" /* Parte del anal. sintáctico */
int numlineas = 0; /* Definición de mi variable global */
int main()
{
ParserBlackBox<DLGLexer, MiParser, ANTLRToken> p(stdin);
p.parser()->entrada(); /* entrada es el axioma */
cout << "\nNúmero de líneas analizadas: " << numlineas << endl;
}
>>

#token "[\ \t]" <<skip();>> /* página 92 del manual. Función de la clase anal. léxico. Sirve para pedir otro token, salta el actual. */
#token "\n" <<skip(); newline(); numlineas = line();>> /* line() devuelve el número de línea */
#token TKN_ADD "\+"
#token TKN_MULT "\*"
#token TKN_DIV "/"
#token TKN_SUB "\-"
#token TKN_ASSIGN "="
#token TKN_SIN "sin"
#token TKN_NUM "[0-9]+"
#token TKN_PA "\("
#token TKN_PC "\)"
#token TKN_PTOCOMA ";"
#token TKN_ID "[a-zA-Z][a-zA-Z0-9]*"

class MiParser {
entrada: (ecuacion)+ ;
ecuacion: TKN_ID TKN_ASSIGN exp TKN_PTOCOMA ;
exp: term (TKN_ADD term | TKN_SUB term)* ;
term: factor (TKN_MULT factor | TKN_DIV factor)* ;
factor: TKN_PA exp TKN_PC | TKN_NUM | TKN_SIN TKN_PA exp TKN_PC;
}
```

## Acciones en la gramática

**Acciones de inicialización:** Se colocan justo delante de la parte derecha de la producción. Se suelen poner en este tipo de acciones, declaraciones de variables locales en esa producción, inicializaciones, etc. ¡Importante!: Se ejecutan siempre aunque lo que venga a continuación genere un error sintáctico.

Ejemplo:

```
paragraph: <<int count=0;>> (sentencia <<cout++;>>)+ <<cout
<<"Número de sentencias" <<count;>> ;
```

**Acciones dentro de la producción:** Acciones embebidas, se ejecutan durante el proceso de reconocimiento.

**Acciones de fallo:** Se ejecutan si la regla falla. Van detrás del ";" (En principio no las vamos a usar).

## Etiquetas

Los tokens y las reglas pueden ser referenciadas mediante etiquetas para acceder a los datos del objeto. Las etiquetas deben empezar con una letra minúscula y se colocan delante del elemento a referenciar separados por ":" Es equivalente al $1, $2, ..., $n del BISON pero aquí no hay que contar. Se le pone el nombre de la etiqueta, así el código es más legible.

Ejemplo:

```
factor : id: TKN_ID << cout << $id->getText();>> | num: TKN_NUM
<<cout << $num->getText();>>
```

## Paso de parámetros a una regla y retorno de valores

Supongamos que queremos pasarle `n argumentos` a una regla y que devuelva `m valores`:

```
regla [arg1,...,argn]>[val1,... ,valm]:descripcion_de_la_regla ;
```

Ejemplo:

```
entrada: <<int r=0;>> exp[3,4]>[r] << cout<<"Resultado"<<r; >> ;

exp[int a,int b]>[int res]: i:TKN_NUM <<$res=$a+$b+atoi($i-
>getText());>> ;
```

donde:
expresion[3,4] es la forma de llamar a la regla expresión con dos parámetros
> [r] es donde se guarda lo que devuelve la función expresion
Ponemos r y no $r a la hora de imprimir porque es una variable que se ha definido en el entorno de la regla y se usa directamente.
Ponemos $ cuando se trata de un parámetro o de una etiqueta: <<$result = $a+$b+atoi($i->getText());>>

## Construcción del árbol

- Se incluye el fichero AST.h (ver clase AST) que contiene la descripción de la clase árbol y la implementación.

- Se modifica la función main() creando un puntero ASTBase *root; y se llama al método parser pasándoselo como argumento:
  p.parser()->entrada(&root);

- Se utilizan dos operadores: **^** para definir quién es el padre, **!** para especificar que no se desea que se genere un nodo para ese símbolo. También se puede aplicar sobre las reglas para especificar que no se devuelva el árbol. Si en un terminal no se le dice nada, se engancha directamente al símbolo que hayamos definido como padre. Estos operadores sólo se pueden usar sobre tokens (símbolos terminales).

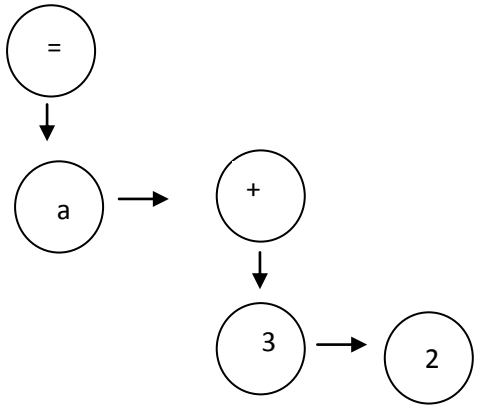
```
Forma del árbol
padre
|
hijo – hijo
```

Ejemplo:

```
ecuacion : TKN_ID TKN_ASSIGN^ exp ";"! ;

exp : term (TKN_MULT^ term)* ;

term : factor (TKN_DIV^ factor)* ;

factor : TKN_NUM^ ;
```

Por ejemplo para la entrada a = 3 + 2;

Crearía el árbol




Al recorrerlo en modo preorden, la acción por defecto es imprimir el lexema (ver clase AST):

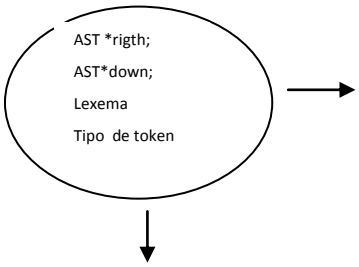
(= a (+ 3 2 ))

- Para hacer referencia a los nodos del árbol se utiliza #
- Si queremos crear un nodo a mano, tendríamos que hacerlo dentro de una acción. Por ejemplo, aquí estamos creando un nodo de tipo TKN_MULT y lexema MULT.

  << #0 = # (#[TKN_MULT, "MULT"], #hijo1 , #hijo2 );>>

- En el makefile, habrá que añadirle la opción de generar árbol, por lo que pondremos –trees cuando usemos el genmk.

Cada nodo del árbol contiene la siguiente información. Recordad que el operador ^ sólo se aplica a símbolos terminales:




Para acceder a esos datos tenéis los métodos

```
type() //devuelve el tipo de nodo (tipo de token)
getText()   //devuelve el lexema
right(); //devuelve el puntero al hermano
down(); //devuelve el puntero al primer hijo
```

# Generación del makefile

**genmk –CC –class MiParser –project miejemplo –trees miejemplo.g > makefile**

donde: CC es el interfaz de C++, MiParser es el nombre de la clase, miejemplo es el nombre del ejecutable, -trees sólo se pone cuando queremos generar el árbol.

Hay que hacer tres modificaciones en el makefile dependiendo de la instalación que tengáis:

PCCTS = /usr
ANTLR_H = $(PCCTS)/include/pccts o bien, ANTLR_H = $(PCCTS)/lib/pccts/h
CCC = g++

Si se quiere cambiar el número de elementos de pre-análisis, es decir, el valor de k, esto se hace modificando la línea del makefile ANTLR y añadiéndole el flag –k número_de_símbolos.

Por ejemplo, si queremos que k, en lugar de ser el valor por defecto que es k=1, sea k=2, modificaremos la correspondiente línea del makefile para que quede: ANTLR = $(BIN)/antlr -k 2

## *Información adicional importante*

Revisar las siguientes partes de la guía de referencia del PCCTS:
Funciones disponibles para la clase léxica (páginas 92-93).
Sintaxis de las expresiones regulares (páginas 93-95).
Para saber más sobre las clases léxicas (páginas 99-100).
Funciones de la clases AST (páginas 119-120).
Estructura del árbol generada por el ANTLR (páginas 124-125).
Creación de nodos de forma manual (páginas 125-127).

Ejemplo calculadora

Fichero calc.g

```
/* Ejemplo de una calculadora que hace sumas, multiplicaciones y
funcion sin sobre una entrada dada. Imprime el arbol sintactico y el
resultado semantico de la cadena de entrada */

#header <<
#include <string>
```

```
#include <iostream>
#include <math.h>
#include "AToken.h"
typedef ANTLRCommonToken ANTLRToken;
#include "MiParser.h"
extern int numlineas;
>>

<<
#include "DLGLexer.h"#include "PBlackBox.h"
#include "AST.h"
int numlineas=0;
int main(){
  ParserBlackBox<DLGLexer, MiParser, ANTLRToken> p(stdin);
  ASTBase *root =NULL;
  p.parser()->entrada(&root);
  cout << "Lineas analizadas: " << numlineas << endl;
}
>>

#token "[\ \t]" <<skip();>>
#token "\n" << skip();  newline(); numlineas=line(); >>
#token TKN_ADD "\+"
#token TKN_MULT "\*"
#token TKN_DIV "/"
#token TKN_MENOS "\-"
#token TKN_ASSIGN "="
#token TKN_SIN "sin"
#token TKN_NUM "[0-9]+"
#token TKN_ID "[a-zA-Z][a-zA-Z0-9]*"


class MiParser{
      entrada :  (ecuacion) ;
      ecuacion : << float r;>> TKN_ID "="^  exp >[r]  ";"!
            <<    cout << "Valor" << r;
                  AST *k; k= (AST *) #0;
                  k->preorder();
            >> ;

      exp > [float r] :
      << float d;>> term>[$r] (TKN_ADD^ term>[d] <<$r+=d;>> )* ;

      term>[float r] :
      << float d;>> factor>[$r] (TKN_MULT^ factor>[d] <<$r*=d;>> )* ;

      factor>[float r] :
      "\("! exp> [$r] "\)"! |
      num:TKN_NUM^ << $r=atof($num->getText());>>  |
```

```
      <<double  d;>> TKN_SIN^ "\("! exp>[d] "\)"! << $r= sin(d);>> ;
}
```

### Fichero AST.h (generado automáticamente)

```
#ifndef AST_h
#define AST_h

#include "ASTBase.h"
#include "AToken.h"
#include "ATokPtr.h"

class AST : public ASTBase {
protected:
      ANTLRTokenPtr token;
public:

      AST(ANTLRTokenType tok, char *s) { token = new ANTLRToken(tok,
s); }
      AST(ANTLRTokenPtr t) { token = t; }
      void preorder_action() {
            cout <<  token->getText()  << " " ;
      }

        void preorder_before_action() {
            cout << "(" ;
      }

        void preorder_after_action() {
            cout <<  ")" ;
      }
      virtual int type() { return token->getType(); }
      char *getText()      { return token->getText(); }

        void preorder ()
      {

        AST * tree = this;

        while (tree !=NULL)
        {
          if (tree->down()!=NULL)
            tree->preorder_before_action();

          tree->preorder_action();

          if (tree->down()!=NULL) {
            AST *d;
            d=(AST*) tree->down();
            d->preorder();
            tree->preorder_after_action();
          }
          tree= (AST *) tree->right();
        }
       }
};

#endif
```

### Makefile

```
# PCCTS makefile for: calc.g
#
# Created from: genmk -class MiParser -project calc -trees calc.g
#
# PCCTS release 1.33MR10
# Project: calc
# C++ output
# DLG scanner
# ANTLR-defined token types
#
TOKENS = tokens.h
#
# The following filenames must be consistent with ANTLR/DLG flags


DLG_FILE = parser.dlg
ERR = err
HDR_FILE =
SCAN = DLGLexer
PCCTS = /usr
ANTLR_H = $(PCCTS)/include/pccts
BIN = $(PCCTS)/bin
ANTLR = $(BIN)/antlr
DLG = $(BIN)/dlg
CFLAGS = -I. -I$(ANTLR_H) $(COTHER)
AFLAGS = -CC -gt -mrhoist off $(AOTHER)
DFLAGS = -C2 -i -CC $(DOTHER)
GRM = calc.g
SRC = calc.cpp \
      MiParser.cpp \
      $(ANTLR_H)/AParser.cpp $(ANTLR_H)/DLexerBase.cpp \
      $(ANTLR_H)/ASTBase.cpp $(ANTLR_H)/PCCTSAST.cpp \
       $(ANTLR_H)/ATokenBuffer.cpp $(SCAN).cpp
OBJ = calc.o \
      MiParser.o \
      AParser.o DLexerBase.o \
      ASTBase.o PCCTSAST.o \
      ATokenBuffer.o $(SCAN).o
ANTLR_SPAWN = calc.cpp MiParser.cpp \
              MiParser.h $(DLG_FILE) $(TOKENS)
DLG_SPAWN = $(SCAN).cpp $(SCAN).h
CCC=g++
#CCC=CC

calc : $(OBJ) $(SRC)
      $(CCC) -o calc $(CFLAGS) $(OBJ)

calc.o : $(TOKENS) $(SCAN).h calc.cpp
      $(CCC) -c $(CFLAGS) -o calc.o calc.cpp

MiParser.o : $(TOKENS) $(SCAN).h MiParser.cpp MiParser.h
      $(CCC) -c $(CFLAGS) -o MiParser.o MiParser.cpp

$(SCAN).o : $(SCAN).cpp $(TOKENS)
      $(CCC) -c $(CFLAGS) -o $(SCAN).o $(SCAN).cpp

$(ANTLR_SPAWN) : $(GRM)
      $(ANTLR) $(AFLAGS) $(GRM)
```

```
$(DLG_SPAWN) : $(DLG_FILE)
      $(DLG) $(DFLAGS) $(DLG_FILE)

AParser.o : $(ANTLR_H)/AParser.cpp
      $(CCC) -c $(CFLAGS) -o AParser.o $(ANTLR_H)/AParser.cpp

ATokenBuffer.o : $(ANTLR_H)/ATokenBuffer.cpp
      $(CCC) -c $(CFLAGS) -o ATokenBuffer.o
$(ANTLR_H)/ATokenBuffer.cpp

DLexerBase.o : $(ANTLR_H)/DLexerBase.cpp
      $(CCC) -c $(CFLAGS) -o DLexerBase.o $(ANTLR_H)/DLexerBase.cpp

ASTBase.o : $(ANTLR_H)/ASTBase.cpp
      $(CCC) -c $(CFLAGS) -o ASTBase.o $(ANTLR_H)/ASTBase.cpp

PCCTSAST.o : $(ANTLR_H)/PCCTSAST.cpp
      $(CCC) -c $(CFLAGS) -o PCCTSAST.o $(ANTLR_H)/PCCTSAST.cpp

clean:
      rm -f *.o core calc

scrub:
      rm -f *.o core calc $(ANTLR_SPAWN) $(DLG_SPAWN)
```