PCCTS: DLG, ANTLR, SOURCERER

PCCTS es una herramienta para crear compiladores a partir de un único fichero en dónde se especifican los tokens a reconocer, la gramática y las acciones a aplicar.

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

PCCTS es más potente que las herramientas existentes hasta ahora para generar analizadores sintácticos (como YACC/BISON) porque trabaja con un tipo especial de gramáticas: las gramáticas de predicados: pred-LL(k).

Ej:

Supongamos dos producciones iguales desde el punto de vista sintáctico pero diferentes desde el punto de vista semántico (su significado depende del contexto):

elemento → ID "(" expresion_lista ")" // referencia a un array: a(i)

elemento → ID "(" expresion_lista ")" // llamada a procedimiento

¿Qué producción deberíamos aplicar?

Depende de si ID corresponde a un identificador de un vector o de un procedimiento ⇒ Por tanto, deberíamos tener unas funciones del tipo "es_array(texto_actual)" y "es_procedimeinto(texto_actual)" que devolverían el valor VERDADERO o FALSO según correspondiera ⇒ La gramática quedaría de la forma:

elemento → << es_array(texto_actual) >>? ID "(" expresion_lista ")"

elemento → << es_procedimiento(texto_actual) >>? ID "(" expresion_lista ")"

donde este tipo de funciones << ... >>? se llaman predicados semánticos.

Nota: << >>, las acciones van encerradas entre esos símbolos

usamos la variables texto_actual para hacer referencia al nombre del ID

A este tipo de predicados se les llama semánticos porque están relacionados con el significado de la producción. Pero también se introducen otro tipo de predicados: los predicados sintácticos que están asociados a fragmentos de la gramática que únicamente predicen la producción a la que están asociados.

Por ejemplo:

Supongamos la gramática:

E → (a)* b

E → (a)* c

Esta gramática no es LL(k). No importa lo grande que hagamos k (el número de elementos de pre-análisis que consideremos) porque podría venir una secuencia de k+1 a's y no sabríamos cual de las dos producciones elegir.

Para resolver este problema, se introducen los predicados sintácticos y, entonces, la gramática se modifica de la siguiente forma:

E → ((a)* b)? (a)* b

$E \rightarrow (a)^* c$

Es decir, para predecir la 1ª producción, cero o más a's deben ir seguidos de una b. Si este predicado sintáctico falla, la 2ª producción se utilizará por defecto.

El hecho de utilizar predicados hace que este tipo de gramáticas sean más poderosas que las LL(k) e incluso que las LR(k), porque:

- Los predicados semánticos se pueden utilizar para analizar construcciones sensibles al contexto.
- Se tiene acceso también a un número arbitrario de símbolos de pre-análisis.

pred-LL(k)

|

LR(k)

|

LL(k)

A parte de esta ventaja, PCCTS posee otras características que lo hacen superior a otros generadores de analizadores:

- Integra en un único fichero la especificación del análisis léxico y sintáctico. Las expresiones regulares correspondientes a los tokens se pueden colocar encerradas entre " " en la descripción de la gramática.
- Acepta construcciones de la gramática en la forma Backus-Naur extendida (EBNF)

Ej:

expr $\rightarrow$ factor

factor $\rightarrow$ term ("+" term)*

- Proporciona facilidades para la construcción y recorrido del árbol sintáctico y ejecutar acciones. 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 le hace fácil de depurar.
- Permite que, a las reglas de la gramática, se les pueda pasar parámetros y devuelvan valores. Los parámetros son en realidad los parámetros que se les pasa a la función que se ha generado correspondiente a ese no-terminal. Pueden devolver múltiples valores.

El PCCTS está compuesto de tres herramientas:

- DLG $\rightarrow$ es un generador de analizadores léxicos. Tiene como entrada una descripción de los tokens y crea el AFD correspondiente.
- ANTLR $\rightarrow$ es un generador de analizadores sintácticos. (ANother Tool for Language Recognition). Tiene como entrada una descripción de la gramática usando la notación EBNF (Extended Backus-Naur Form) y construye un analizador descendente asociando una función a cada una de las producciones de la gramática.
- SOURCERER $\rightarrow$ 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.

Por otra parte está:

- 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 (.g)

```
#header
<<
    /* includes, declaraciones de variables externas, typedefs, descripción de las
    estructutras...
    Todo lo que se incluya en esta parte se copia en todos los ficheros generados por el
    PCCTS. ⇒ ¡Cuidado! Nunca definir ahí variables, pues habría problemas de
    redefinición y desperdicio de memoria.
    */

/* Ejemplo de una calculadora que suma, multiplica y hace la función seno */
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "AToken.h"          /* del PCCTS ⇒ token */
typedef ANTLRCommonToken ANTLRToken ;    /* del PCCTS ⇒ token */

#include "MiParser.h"        /* para incluir mi clase */
extern int numlineas;          /* Para controlar el número de líneas desde fuera, necesito
                               una variable global */
>>
<<
    /* Definición de variables, función main(), código auxiliar necesario... */
#include "DLGLexer.h"        /* Parte del léxico */
#include "PBlackBox.h"       /* Parte del sintáctico */

int numlineas = 0;      /* Definición de mi variable global */
int main()
{
```

```
    ParserBlackBox<DLGLexer, MiParser, ANTLRToken> p(stdin);
    p.parser()->entrada();          /* entrada es la producción por la que empezamos el
                                    parser */
    cout << "\nNúmero de líneas analizadas: " << numlineas << endl;
}
>>
```

/* Descripción léxica: Lista de tokens y expresiones regulares asociadas. 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 de la descripción de los tokens debe ser igual que en BISON, pues aquí también, 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 cuyo símbolo corresponde a uno de los que se utilizan en las expresiones regulares, se debe poner el carácter de escape \

```
*/
#token "[\ \t]"         <<skip();>>   /* página 92 del manual. Función dentro de la clase
                                      analizador léxico. skip() 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_MENOS        "\-"
#token TKN_ASSIGN       "="
#token TKN_SIN      "sin"
#token TKN_NUM      "[0-9]+"
#token TKN_ID       "[a-zA-Z][a-zA-Z0-9]*"


/* Descripción sintáctica */
class MiParser          /* Nombre de la clase que define al analizador sintáctico (lo damos
                        nosotros) */
{
    /* Si se quiere utilizar alguna función propia de esta clase, entonces:
    << protected:
```

```
        virtual void store(char *tmp, float v) { ; }
        virtual float value (char* tmp) { ; }
    >>
    */
    /* nombre_de_la_regla   :      descripcion_de_la_regla  ; */

    entrada   :     (ecuacion)+   ;
    ecuacion  :     TKN_ID "=" expresion ";"   ;
    expresion :     termino (TKN_ADD termino)*      ;
    termino   :     factor ("\*" factor)*   ;
    factor    :     "\(" expresion "\)"   |   TKN_NUM   |   TKN_SIN   "\("
                    expresion "\)"      ;
}
/*
```

- Para especificar la gramática se puede usar la EBNF ⇒ *, +, {},...
- Se pueden incluir las especificaciones de los tokens
- Los nombres de las reglas deben empezar con minúscula

*/

Acciones en la gramática

Hay tres tipos:

- **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 no se ejecute. ¡Cuidado con eso!

  Ejemplo:

  ```
  paragraph   :   <<int count=0;>>  (sentencia  <<count++;>>)+
                  <<printf("Número de sentencias %d\n",count);>> ;
  ```

  donde:

  <<int count=0;>> ⇒ es una acción de inicialización

  <<count++;>> ⇒ es una acción dentro de la producción

- **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 ":"

etiqueta : elemento

Es equivalente al $1, $2, ..., $n del BISON pero aquí no hay que contar. Se le pone el nombre de la etiqueta ⇒ el código es más legible.

Ejemplo:

```
factor  :   id: TKN_ID <<printf("%s\n", $id->getText());>>  |
            num: TKN_NUM <<printf("%f\n", atof($num->getText());>>
```

donde:

id → es un puntero a un token

$id → es la manera de usar la variable

getText() → es una función miembro de la clase token

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 [arg_1, ... , arg_n] > [val_1, ... , val_m] : descripcion_de_la_regla ;

Ejemplo:

```
entrada  :   <<int r=0;>> expresion[3,4] > [r] <<printf("Resultado %d", r);>> ;

expresion [int a, int b] > [int result]  :   i : TKN_INT
                                             <<$result = $a+$b+atoi($i->getText());>> ;
```

donde:

expresion[3,4] → es la forma de llamar a la regla expresion con dos parámetros

> [r] → es donde se guarda lo que devuelve la función expresion

ponemos r y no $r en printf("Resultado %d", r); porque si es una variable que se ha definido en el entorno de la regla ⇒ se pone directamente.

ponemos $ cuando se trata de un parámetro: <<$result = $a+$b+atoi($i->getText());>>

Construcción del árbol

- Se incluye (donde la función main()) 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 ASTBase *root; y se llama a la función 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.

Ejemplo:

ecuacion : TKN_ID "="^ expresion ";"! ;

Si no se le dice nada, el sourcerer va enganchando los hijos a quien hayamos definido como padre usando la notación:

padre

|

hijo – hijo

Crea un nodo de tipo correspondiente al token y como nombre le pone el de la expresión regular asociada.

- Para hacer referencia a los nodos del árbol se utiliza #
- Si queremos crear un nodo a mano, tendríamos que hacer, dentro de una acción:

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

  donde:

  "MULT" es el nombre que le damos al nodo

  hijo1 e hijo2 son los hijos que queremos ponerle (se identifican por su etiqueta).

  Sirve por ejemplo, si esperamos entradas de la forma: a = 3x + 5 → y queremos multiplicar el 3 por x pero no estamos recibiendo el TKN_MULT de la entrada, por lo que lo debemos crear a parte.

  Para el caso de la definición de funciones, habrá que crearlo a mano.
- En el makefile, habrá que añadirle la opción de generar árbol, por lo que pondremos –trees cuando usemos el genmk.

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:

- Variables de entorno:

  PCCTS = /usr

  Dependiendo de dónde se ha instalado el pccts en las diferentes versiones de Linux, la variable de entorno ANTLR_H puede tener que definirse en diferentes caminos (habrá que verificar cuál es el camino que debemos usar en nuestro caso). Los más habituales son:

ANTLR_H = $(PCCTS)/include/pccts o bien,

ANTLR_H = $(PCCTS)/lib/pccts/h

- Compilador de C++:

CCC = g++ (#CCC=CC)

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

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).