Instituto Tecnológico y de Estudios Superiores de Occidente – ITESO

ITESO

Universidad Jesuita de Guadalajara

Materia: Estructuras de datos y algoritmos

Profesor: Luis Gatica

PROYECTO INTEGRADOR

Propuesta: Analizador sintáctico de fórmulas bien formadas

Fecha: 15 de mayo del 2017

Temas: Tipos de datos compuestos

Autor(es): Chávez Medina Mariana

Madriz Almanza Omar Antonio

Introducción

En lógica matemática, una fórmula bien formada (fbf) es una cadena de caracteres generada a través de una gramática formal a partir de un alfabeto (conjunto de símbolos). En lógica proposicional, las fórmulas bien formadas se definen inductivamente como sigue:

1. Cada variable proposicional[1] es, por sí misma, una fbf.
2. Si α es una fbf, entonces ¬α es una fbf.
3. Si α y β son fbf, y • es un operador binario, entonces (α•β) es una fbf. • puede ser cualquiera de los operadores binarios de la lógica proposicional: ∨, ∧, →, o ↔.

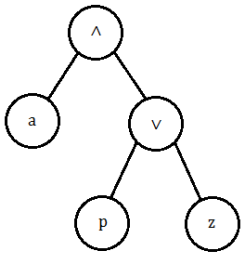
Los siguientes son ejemplos de fbf:

1. a
2. ¬a
3. ¬b
4. (a ∨ c)
5. (x ∧ y)
6. (p → q)
7. (r ↔ s)
8. (a ∧ (p ∨ z))
9. ((x → b) ∨ (a ∨ (¬x ↔ y)))
10. ¬((p ↔ p) ↔ p)

Las siguientes NO son fbf:

1. a¬
2. A
3. ∨ c
4. (p ¬∧ q)
5. ((p → q)

Podemos construir un programa que determine si una cadena es una fbf. En nuestro caso, tal programa se llamará analizador sintáctico, y para una fórmula bien formada construirá un árbol de expresión. En el caso de la fbf del ejemplo #8, este árbol puede verse como sigue:



[1] Éste se trata de un conjunto finito pero de tamaño arbitrario. Para nuestros propósitos, consideraremos que todas las letras minúsculas del alfabeto inglés son cada una la representación de una variable proposicional: a, b, c, … , x, y, z.

Note que hay una correspondencia en la jerarquía de los nodos en el árbol y la jerarquía de los paréntesis en la fórmula. Gracias a esta correspondencia es innecesario representar los paréntesis explícitamente en el árbol.

Observe también que, necesariamente, las hojas del árbol serán variables proposicionales, y los nodos que tengan descendencia serán operadores: si son operadores binarios, tendrán dos hijos; si son el operador de negación, tendrán un solo hijo.

Existe una manera de recorrer el árbol de expresión tal que al imprimir el recorrido veamos la fórmula bien formada pero escrita en notación postfija (notación polaca inversa). Para el caso del ejemplo 8, la fórmula en esta notación es:

apzV∧

Observe que aquí también los paréntesis están ausentes: en la notación polaca inversa no es necesario introducirlos para que no haya ambigüedad, lo cual la convierte en una notación mucho más eficiente para procesar los datos. Esto explica que algunas calculadoras antiguas funcionaran únicamente utilizando esta notación.

Descripción del proyecto

Usted deberá desarrollar un analizador sintáctico que determine si una cadena es una fórmula bien formada. En caso afirmativo, deberá construir el árbol de expresión que le corresponde, para concluir mostrando la fórmula bien formada en notación polaca inversa.

Ayúdese de **pilas** para verificar la congruencia de paréntesis y para construir el árbol de expresión, que puede representarse perfectamente como un **árbol binario**.

Las cadenas a introducir como entrada del programa deberán estar compuestas por los siguientes caracteres o secuencias de caracteres:

- Letras del alfabeto inglés: 'a', 'b', 'c', … , 'x', 'y', 'z'. Éstas representarán a las variables proposicionales.
- Los caracteres '¬', '|' y '&', que representarán a los operadores ¬, ∨ y ∧.
- Las secuencias de caracteres "->" y "<->", que representarán a los operadores → y ↔.
- Paréntesis: '(', ')'.
- Espacios: ' '.

Si una cadena introducida no cumple con estas restricciones, deberá mostrarse un mensaje diciendo al usuario que introduzca una cadena válida, solicitándola de nuevo.

Para prescindir de la definición de una precedencia de operadores, el uso de un operador binario siempre requerirá de poner a los operandos entre paréntesis. Esto significa que no se introducirá la cadena "a | (b & c)", sino la cadena "(a | (b & c))". Sin embargo, cadenas como "(a)", "¬(b)" y "(¬c)" deben considerarse fórmulas bien formadas (esto significa que el uso de paréntesis no implica que la expresión contenida lleve un operador binario, mientras que el uso de un operador binario sí implica rodear su expresión con paréntesis).

Nota: el número de espacios en blanco entre otros caracteres o secuencias no debe afectar el resultado del análisis sintáctico.

Algoritmo de análisis sintáctico

Podemos leer una fórmula de izquierda a derecha para hacer el análisis, apoyado de dos pilas, con las siguientes reglas:

- Representaremos la primera pila (A) del lado izquierdo y crecerá hacia la derecha.
- Representaremos la segunda (B) del lado derecho y crecerá hacia la izquierda.
- Tendremos dos modos para trabajar:
    - El modo PUSH, que consistirá en leer la cadena e ir insertando nodos a la pila A.
    - El modo POP, que consistirá en extraer nodos de la pila A para procesarlos, ayudados de la pila B.
- Empezaremos en el modo PUSH, que consiste en leer la fórmula e ir convirtiendo sus caracteres o secuencias en nodos que se insertarán en la pila A, hasta alcanzar un cierre de paréntesis o el final de la fórmula, con lo que cambiaremos al modo POP. (Inclusive el cierre de paréntesis se inserta como nodo en la pila A.)
- En el modo POP, se hará pop de la pila A y se procesará cada nodo según su tipo y según si está completo o no. Cambiaremos al modo PUSH cuando la pila B vuelva a estar vacía.
- Un nodo puede ser de uno entre cinco tipos:
    - Proposición atómica
    - Operador binario
    - Operador unario (negación)
    - Apertura de paréntesis
    - Cierre de paréntesis
- Un nodo se considera completo si se cumple cualquiera de las condiciones:
    - Corresponde a una proposición atómica.
    - Corresponde a un operador binario y tiene ambos hijos asignados (no nulos).
    - Corresponde al operador unario y tiene un y sólo un hijo asignado (no nulo).
- En el modo POP, procesar un nodo N extraído de la pila A significa lo siguiente:
    - Si es un cierre de paréntesis o una proposición atómica, insertar N en la pila B.
    - Si es un operador unario, extraer un nodo de la pila B, que deberá ser un nodo completo, y convertirlo en el único hijo de N. Ahora que N es un nodo completo, insertar N en la pila B.
    - Si es un operador binario, extraer un nodo de la pila A, que deberá ser un nodo completo, y convertirlo en el hijo izquierdo de N; luego, extraer un nodo de la pila B, que deberá ser un nodo completo, y convertirlo en el hijo derecho de N. Ahora que N es un nodo completo, insertar N en la pila B.
    - Si es una apertura de paréntesis, sacar un nodo de la pila B, que deberá ser un nodo completo, y mantener una referencia a él en una variable temporal T; luego, sacar otro nodo (C) de la pila B, que deberá ser un cierre de paréntesis. Eliminar N (apertura), eliminar C (cierre) e insertar el nodo referido por T en la pila A.
- Si en el modo POP ocurre cualquier problema (que un nodo que debería estar completo no lo esté, o que un nodo que debería ser un cierre de paréntesis no lo sea), entonces la fórmula no es una fórmula bien formada.
- Si en el modo POP la pila A se queda vacía, entonces la pila B debe tener un solo nodo (completo), que se extraerá de ahí.

    - Si ya se terminó de leer la cadena, entonces ese nodo es la raíz del árbol sintáctico y hemos terminado.
    - Si no se ha terminado de leer la cadena, entonces ese nodo se inserta de nuevo en la pila A.
    - Si no fuera sólo un nodo el que quedara en B, entonces la fórmula no es una fórmula bien formada.
- Entonces, observemos que en ambos modos se insertarán nodos a la pila A, pero sólo en el modo POP se extraerán nodos de ella y sólo en el modo POP se insertarán nodos en la pila B.

Ejemplo

Utilizaremos la fórmula bien formada #10:

¬((p <-> p) <-> p)

Como el resultado es independiente de los caracteres espacio, tenemos la siguiente cadena:

¬((p<->p)<->p)

Colocaremos del lado izquierdo y en rojo la subcadena que ya ha sido procesada; del lado dercho y en negro la subcadena que falta de procesar.

1. Comenzamos en modo PUSH, leyendo la cadena e insertando los nodos uno a uno en la pila A. En el caso de la subsecuencia <->, sólo insertamos un nodo, que tendrá el valor ↔ (por ejemplo). Hacemos esto hasta alcanzar el primer cierre de paréntesis, que se inserta como nodo en la pila B.

¬((p<->p)<->p)

¬((p<->p)	<->p)
A: ¬, (, (, p, ↔, p,)	:B

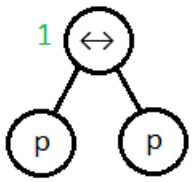
2. Cambiamos a modo POP. Extraemos el cierre de paréntesis de A y lo apilamos en B. Extraemos p y la apilamos en B.

¬((p<->p)	<->p)
A: ¬, (, (, p, ↔	p,) :B

3. Seguimos en modo POP. Extraemos ↔. Observemos que los topes de ambas pilas son proposiciones atómicas, de forma que son nodos completos.

¬((p<->p)	<->p)
A: ¬, (, (, p	p,) :B

4. Extraemos esos nodos completos y los convertimos en los hijos de ↔, que habíamos extraído antes. Ahora llamaremos 1 a este nodo padre y lo apilaremos en B.



¬((p<->p)	<->p)
A: ¬, (, (	1,) :B

5. Ahora extraemos el tope de la pila A, que es una apertura de paréntesis, y extraemos el nodo completo 1 y el cierre de paréntesis de la pila B. Eliminamos los paréntesis y devolvemos el nodo 1 a la pila A.

¬((p<->p)	<->p)
A: ¬, (, 1	:B

6. Como la pila B está vacía de nuevo, volvemos al modo PUSH, insertando nodos hasta alcanzar el cierre de paréntesis.

¬((p<->p)<->p)	
A: ¬, (, 1, ↔, p,)	:B

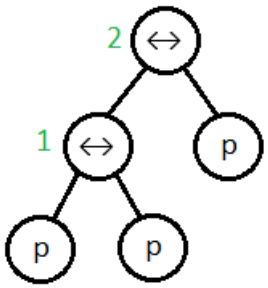
7. En modo POP, procesamos el paréntesis y la proposición atómica apilándolos en B.

¬((p<->p)<->p)	
A: ¬, (, 1, ↔	p,) :B

8. En modo POP, ahora extraemos de A el operador ↔. Observemos que, de nuevo, los topes de ambas pilas son nodos completos.

¬((p<->p)<->p)	
A: ¬, (, 1	p,) :B

9. Hacemos que 1 y p sean hijos de ↔, y a ese nodo lo llamamos 2. Lo insertamos en B.



¬((p<->p)<->p)	
A: ¬, (	2,) :B

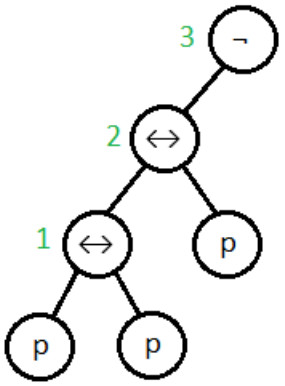
10. Extraemos de A la apertura de paréntesis. Sacamos un nodo completo de B (2) y almacenamos una referencia a él en T. Luego, sacamos de B el cierre de paréntesis y eliminamos ambos paréntesis. Insertamos el nodo referido por T en la pila A.

¬((p<->p)<->p)	
A: ¬, 2	:B

11. Como la pila B está vacía, volvemos al modo PUSH. Pero como ya hemos alcanzado el final de la cadena, volvemos al modo POP. Así, extraemos el nodo 2 de A y lo apilamos en B.

¬((p<->p)<->p)	
A: ¬	2 :B

12. En modo POP, extraemos el tope de A: ¬. Como es un operador unario, extraemos el tope de B (2), que debe ser un nodo completo, y lo convertimos en el hijo de ¬. Ahora llamaremos 3 a este nodo padre, y lo insertaremos en B.



¬((p<->p)<->p)	
A:	3 :B

13. En modo POP, resulta que la pila A se quedó vacía, así que extraemos el único nodo que debe haber en B (3). Como ya hemos leído completa la cadena de entrada, hemos terminado y 3 es el nodo raíz del árbol sintáctico.

¬((p<->p)<->p)	
A:	:B

Solución

Main Principal de programa (.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include "functAn.h"
#include <string.h>
#include "theStackSet.h"

#define MAX 5000


int main(void) {
	Stack stackA, stackB;
	Set tree;
	stackA = stack_create();
	stackB = stack_create();
	tree = set_create(printChar);

	char lexA[MAX];
	char * p1 = lexA;
	int i;
	bool sortie = false;

	setbuf(stdout, NULL);


	printf("Bienvenidx a su analizador de fórmulas favorito \n");

	//Analizador Lexico

	do{
	printf("Introduzca la fórmula que desea analizar:\n");
	gets(lexA);
	cleanArr(&p1);
	}while(!validChars(lexA));

	//Analizador Sintactico

	for(i=0; lexA[i]!='\0' && !sortie ; i++){
		if(lexA[i] != ')'){
			PUSH(stackA, lexA[i]);
		}
		else{
			PUSH(stackA, lexA[i]);
			if(POP(stackA, stackB, tree) == false){
				sortie = true;
			}
		}
		if(lexA[i+1] == '\0')
			if(POP(stackA, stackB, tree) == false)
				sortie = true;
	}
```

```
	if(sortie == false){
		printf("¡Yeih! lo haz logrado, es una fórmula bien formada, tienes
la lógica ;) \n");
		printf("Notación polaca: ");
		set_print(tree);
	}
	else
		printf("No es una fórmula bien formada, suerte para la proxima
\n");

	set_destroy(tree);
	stack_destroy(stackA);
	stack_destroy(stackB);

	printf("¡Hasta luego!");

	return 0;
}
```

Definicion de contenedores (.h)

```
#ifndef THESTACKSET_H_
#define THESTACKSET_H_
typedef void* Type;
typedef struct node* Node;
typedef struct stack* Stack;
typedef struct treenode* TreeNode;

typedef enum { false, true } bool;
typedef enum {Atomic, Aperture, Closure, Unary, Binary, Nodes} Types;

Stack stack_create();
int   stack_size(Stack);
bool  stack_isEmpty(Stack);
TreeNode  stack_top(Stack);
void  stack_push(Stack, TreeNode);
TreeNode  stack_pop(Stack);
void  stack_destroy(Stack);

typedef struct set* Set;
typedef void (*PrintFunc) (char); //cambio de type a char

Set set_create(PrintFunc);
int set_size(Set);
TreeNode set_add(Set, TreeNode, TreeNode, TreeNode);
void set_destroy(Set);
void set_print(Set);
TreeNode node_create(char f);
char node_value(TreeNode v);
bool nodeIsComplete(TreeNode);
void set_root(Set, TreeNode);
```

```
#endif /* THESTACKSET_H_ */
```

Implementación de contenedores (.c)

```
#include <stdlib.h>
#include <stdio.h>
#include "theStackSet.h"

struct treenode{
	TreeNode left, right;
	char data;
	bool isComplete;
};

struct node{
	Node prior;
	TreeNode data;
};

struct stack{
	Node top;
	unsigned int size;
};


struct set{
	TreeNode root;
	unsigned int size;
	PrintFunc printFunc;
};

Stack stack_create(){
	Stack s = calloc(1,sizeof(struct stack));
	return s;
}

int   stack_size(Stack s){
	return s->size;
}

bool  stack_isEmpty(Stack s){
	return s->size == 0;
}

TreeNode stack_top(Stack s){
	if(s->top == NULL)
		return NULL;

	return s->top->data;
}
```

```
void stack_push(Stack s, TreeNode d){
	Node new = calloc(1,sizeof(struct node));
	new->data= d;

	if(s->size==0)
		s->top=new;
	else{
		new->prior = s->top;
		s->top = new;
	}
	s->size++;
}

TreeNode  stack_pop(Stack s){
	Node temp;
	TreeNode d;
	if(s->size==0)
		return NULL;

	else{
		temp = s->top;
		d = s->top->data;
		s->top = s->top->prior;
		free(temp);
		s->size--;
		return d;
	}
}

void  stack_destroy(Stack s){
	Node current = s->top;
	Node prior;

	while(current!=NULL){
		prior = current->prior;
		free(current);
		current = prior;
	}
	free(s);
}

Set set_create(PrintFunc pf){
	Set s = calloc(1,sizeof(struct set));
	s->printFunc = pf;
	return s;
}

int set_size(Set s){
	return s->size;
}

TreeNode set_add(Set s, TreeNode parent, TreeNode leftChild, TreeNode
rightChild){
	parent->left = leftChild;
```

```
	parent->right = rightChild;
	parent->isComplete = true;
	s->size++;
	return parent;
}

void print(TreeNode tn, PrintFunc pf){
	if(tn != NULL){
		print(tn->left, pf);
		print(tn->right, pf);
		pf(tn->data);
	}
}

void set_print(Set s){
	print(s->root, s->printFunc);
	printf("\n");
}

void destroy(TreeNode tn){
	if(tn != NULL){
		destroy(tn->left);
		destroy(tn->right);
		free(tn);
	}
}

void set_destroy(Set s){
	destroy(s->root);
	free(s);
}

TreeNode node_create(char f){
	TreeNode new = calloc(1, sizeof(struct treenode));
	new->data = f;
	return new;
}

char node_value(TreeNode v){
	return v->data;
}

bool nodeIsComplete(TreeNode v){
	return v->isComplete;
}

void set_root(Set s, TreeNode t){
	s->root = t;
}
```

Implementación de funciones utilizadas en Main

```
#include "theStackSet.h"


void cleanArr(char ** Arr){
	int i, j;

	//Quita espacios
	for(i=0, j=0;(*Arr)[i] !='\0';i++){
		if((*Arr)[i] != ' '){
			(*Arr)[j] = (*Arr)[i] ;
			j++;
		}
	}
	(*Arr)[j] = '\0';

	//Quita <-> y ->
	for(i=0, j=0; (*Arr)[i] !='\0'; i++, j++){
		if((*Arr)[i]  == '<' && (*Arr)[i+1] == '-' && (*Arr)[i+2] == '>'){
			(*Arr)[j] = '=';
			i+=2;
		}
		else if ((*Arr)[i]  == '-' && (*Arr)[i+1] == '>'){
			(*Arr)[j] = '>';
			i+=1;
		}
		else
			(*Arr)[j] = (*Arr)[i] ;
	}
	(*Arr)[j] = '\0';
}

bool validChars(char arr[]){
	int i=0, An;//categorizará los caracteres

	while(arr[i] != '\0'){

		if(arr[i] >= 'a' && arr[i] <= 'z')
			An = 0;
		else if (arr[i] == '(' || arr[i] == ')')
			An = 1;
		else if (arr[i] == '¬' || arr[i] == '|' || arr[i] == '&')
			An = 2;
		else if(arr[i] == '=' || arr[i] == '>')
			An = 3;
		else
			An = 4;

		switch(An){//logra recorrer  el arreglo sin necesidad de un for
		case 0:
			i++;
			break;
		case 1:
```

```
                        i++;
                        break;
                case 2:
                        i++;
                        break;
                case 3:
                        i++;
                        break;
                case 4:
                        printf("Introduzca una cadena válida :,v\n");
                        return false;
                }
        }
        return true;
}

void PUSH(Stack  s, char f){
        TreeNode new = node_create(f);
        stack_push(s, new);
}

int kind(TreeNode d){
        if(node_value(d) >= 'a' && node_value(d)  <= 'z')
                return Atomic;
        else if (node_value(d) == '(')
                return Aperture;
        else if (node_value(d) == ')')
                return Closure;
        else if (node_value(d) == '¬' )
                return Unary;
        else if(node_value(d) == '=' || node_value(d) == '>' || node_value(d) ==
'|' || node_value(d)  == '&')
                return Binary;
        return -1;
}


bool POP(Stack A, Stack B, Set Tree){
        TreeNode d, t, t2; //Temporal para almacenar el nodo que hace pop

        if(stack_size(A) == 1){

                if(stack_top(A) == NULL)
                        return false;

                if(kind(stack_top(A)) == Atomic){
                t = set_add(Tree, stack_pop(A), NULL, NULL);
                set_root(Tree, t);
                return true;
                }
                else if(nodeIsComplete(stack_top(A))){
                        set_root(Tree, stack_pop(A));
                        return true;
                }
                return false;
```

```
	}

	do{
		if(stack_isEmpty(A) && stack_size(B) > 1)
			return false;

		if(stack_isEmpty(A) && stack_size(B) == 1){
			d = stack_pop(B);
			if(kind(d) == Atomic || nodeIsComplete(d)){
				set_root(Tree, d);
				return true;
			}
			else
				return false;
		}

		d = stack_pop(A);

		if(kind(d) == Atomic || kind(d) == Closure || nodeIsComplete(d)){
			stack_push(B, d);
		}
		else if(kind(d) == Unary){

			if(stack_top(B) == NULL)
				return false;

			if(kind(stack_top(B)) == Atomic ||
nodeIsComplete(stack_top(B))){
				t = set_add(Tree, d, stack_pop(B), NULL);
				stack_push(B, t);
			}
			else
				return false;
		}
		else if(kind(d) == Binary){

			t2 = stack_pop(A);

			if(stack_top(A) == NULL)
				return false;

			if(kind(t2) == Atomic && kind(stack_top(A)) == Unary){
				t = set_add(Tree, stack_pop(A), t2, NULL);
				stack_push(A, t);
				t2 = stack_pop(A);
			}

			if(stack_top(B) == NULL)
				return false;

			if(kind(t2) == Atomic && kind(stack_top(B)) == Atomic)
				t = set_add(Tree, d, t2, stack_pop(B));
			else if(nodeIsComplete(t2) && nodeIsComplete(stack_top(B)))
				t = set_add(Tree, d, t2, stack_pop(B));
			else if(kind(t2) == Atomic && nodeIsComplete(stack_top(B)))
```

```
                        t = set_add(Tree, d, t2, stack_pop(B));
                    else if(kind(stack_top(B)) == Atomic && nodeIsComplete(t2))
                        t = set_add(Tree, d, t2, stack_pop(B));
                    else
                        return false;

                    stack_push(B, t);
                }
                else if(kind(d) == Aperture){
                    t = stack_pop(B);
                    if(!stack_isEmpty(B)){

                        if(stack_top(B) == NULL)
                            return false;

                        if(kind(stack_top(B)) == Closure)
                            stack_pop(B);//Cierre
                        else
                            return false;
                    }
                    else
                        return false;
                    stack_push(A, t);
                }
        }while(!stack_isEmpty(B));

        return true;
}

void printChar(char t) {//por cambio de type a char
        //char * pC = (char *) t;
        if(t == '=')
                printf("<->");
        else if (t == '>')
                printf("->");
        else
                printf("%c", t);
}
```

Capturas de pantalla de ejecución

```
Bienvenidx a su analizador de fórmulas favorito
Introduzca la fórmula que desea analizar:
666
Introduzca una cadena válida :,v
Introduzca la fórmula que desea analizar:
```

```
Bienvenidx a su analizador de fórmulas favorito
Introduzca la fórmula que desea analizar:
( a                -> v) |d
No es una fórmula bien formada, suerte para la proxima
¡Hasta luego!
```

```
Bienvenidx a su analizador de fórmulas favorito
Introduzca la fórmula que desea analizar:
(((((e&(f|g))&((e&g)->¬(h|i))))&((¬h|¬i)->¬(e&f)))&¬(h->i))
¡Yeih! lo haz logrado, es una fórmula bien formada, tienes la lógica ;)
Notación polaca: efg|&eg&hi|¬->&h¬i¬|ef&¬->&hi->¬&
¡Hasta luego!
```

Mayor dificultad

A mitad del camino tratando de resolver el problema presentado, nos dimos cuenta, gracias a una asesoría con nuestro profesor, que habíamos entendió mal el algoritmo que se nos había facilitado, por lo tanto, las partes fundamentales de nuestro programa eran funciónales, pero erróneas. Una vez entendido esto, re estructuramos las partes que debíamos cambiar del programa.