UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Ingeniería en computación

Compiladores

GRUPO: 03

PROFESORA: M.I. ELBA KAREN SAENZ GARCIA

Semestres 2024-2

Tercer Examen Parcial

Integrantes:

Esparza Rodríguez Diego

Castañeda Aranda Alma Adriana

Zuckerman Cisneros Stephan

Fecha de entrega: 06/06/2024

Actividad 1

Utilizando la gramática que define a las cadenas de la forma $a^n b^m c^m$ donde n ≥ 0, m > 0:

1: S→aS

2: S→bAc

3: A→bAc

4: A→b

Colocar a la gramática modificada los símbolos de acción necesarios para que obtenga como traducción cadenas de la forma $b^{2m} a^n$.

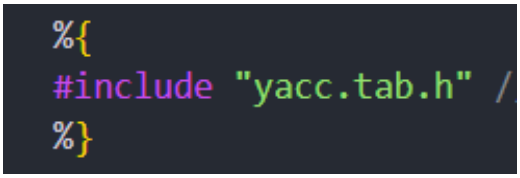
Elaborar un analizador léxico y sintáctico traductor utilizando flex y yacc/bison. El analizador debe recibir un archivo de entrada con la cadena a analizar y mostrar la traducción correspondiente.

Desarrollo

lex.l

#include "yacc.tab.h

Incluye el archivo de encabezado generado por bison, que contiene definiciones de tokens y otras declaraciones necesarias para la integración entre flex y bison.



Reglas de escaneo

Todo lo que esté entre estos delimitadores son reglas de escaneo que flex utilizará para analizar el texto de entrada.

Define cómo reconocer los tokens a, b, c, y los saltos de línea, e ignorar cualquier otro carácter.

```
%%

a   { return 'a'; } // Especifica que cuando se encuentre el carácter 'a' en el texto de entrada, el escáner debe devolver el token 'a'.
b   { return 'b'; } // Especifica que cuando se encuentre el carácter 'b' en el texto de entrada, el escáner debe devolver el token 'b'.
c   { return 'c'; } // Especifica que cuando se encuentre el carácter 'c' en el texto de entrada, el escáner debe devolver el token 'c'.
\n  { return '\n'; } //  especifica que cuando se encuentre un salto de línea en el texto de entrada, el escáner debe devolver el token '\n'.
.   { /* ignora cualquier otro carácter */ }

%%
```

Subprograma de usuario

int yywrap() { return 1; }

Esta función es llamada por el escáner generado por flex cuando llega al final de la entrada. Al devolver 1, le indica al escáner que no hay más datos para leer, lo que causa que yylex() retorne 0, indicando el fin de la entrada al analizador sintáctico (bison).

Maneja el final de la entrada del escáner.

```
/* Función es llamada por el escáner generado por flex cuando llega al final de la entrada.*/
int yywrap() {
    return 1; // Indica el fin de la entrada al analizador sintáctico.
}
```

Yacc.y

Incluye bibliotecas y declara funciones y variables necesarias.

```
%{
// Bibliotecas para entrada/salida y funciones estándar.
#include <stdio.h>
#include <stdlib.h>

void yyerror(const char *s); // Declara la función yyerror para manejar los errores.
int yylex(void); // Declara la función yylex que será definida por flex para realizar el análisis léxico.
extern FILE *yyin; // Será utilizada para establecer el archivo de entrada (test.txt).

// Declara e inicializa variables para contar las ocurrencias de a, b, y c.
int num_a = 0;
int num_b = 0;
int num_c = 0;
%}
```

Definición de tokens

%token a b c

Declara los tokens a, b, y c que serán reconocidos por el analizador léxico (flex).

```
%token a b c
```

Reglas de producción

Define la gramática y cómo deben ser analizadas las entradas.

- ***input:*** La regla input puede estar vacía o puede consistir en una o más line.
- ***línea:*** La regla line puede ser una producción que consista en S seguido de un salto de línea, o simplemente un salto de línea.
- ***S:*** La regla S puede ser un a seguido de S (incrementa num_a), o un b seguido de A y c (incrementa num_b y num_c).
- ***A:*** La regla A puede ser un b seguido de A y c (incrementa num_b y num_c), o simplemente un b (incrementa num_b).

```
%%

input:
    | input linea
    ;

linea:
    S '\n' { printf("Linea valida\n"); }
  | '\n'
  ;

S : 'a' S { num_a++; }
  | 'b' A 'c' { num_b++; num_c++; }
  ;

A : 'b' A 'c' { num_b++; num_c++; }
  | 'b' { num_b++; }
  ;

%%
```

Funciones de usuario

Define las funciones de manejo de errores y la función principal que inicia el análisis y maneja los resultados.

- ***void yyerror(const char *s):*** Esta función maneja los errores de análisis, imprimiendo un mensaje de error.
- ***if (argc > 1):*** Verifica si se ha proporcionado un archivo de entrada como argumento
- ***else { fprintf(stderr, "El archivo de entrada debe ser: %s <input_file>\n", argv[0]); return 1; }:*** Si no se proporciona un archivo de entrada, imprime un mensaje de uso y termina el programa.
- ***printf("Analisis comenzando...\n");:*** Imprime un mensaje indicando que el análisis está comenzando.
- ***int result = yyparse();:*** Llama a la función yyparse generada por bison para realizar el análisis sintáctico.
- ***printf("Analisis finalizado: %d\n", result);:*** Imprime el resultado del análisis.
- ***printf("num_a: %d, num_b: %d, num_c: %d\n", num_a, num_b, num_c);:*** Imprime el número de a, b, y c encontrados.
- ***for (int i = 0; i < 2 * num_b; i++) { printf("b"); }:*** Imprime 2 * num_b caracteres b.
- ***for (int i = 0; i < num_a; i++) { printf("a"); }:*** Imprime num_a caracteres a.
- ***printf("\n");:*** Imprime un salto de línea.
- ***fclose(yyin);:*** Cierra el archivo de entrada.

```c
void yyerror(const char *s) { // Maneja los errores de análisis, imprimiendo un mensaje de error.
    fprintf(stderr, "Error: %s\n", s);
}

int main(int argc, char **argv) {
    if (argc > 1) { // Verifica si se ha proporcionado un archivo de entrada como argumento.
        yyin = fopen(argv[1], "r"); //Abre el archivo de entrada.
        if (!yyin) { // Si el archivo no se puede abrir, imprime un mensaje de error y termina el programa.
            perror(argv[1]);
            return 1;
        }
```

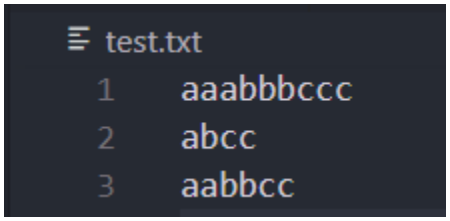
```
    } else { // Si no se proporciona un archivo de entrada, imprime un mensaje de uso y termina el programa.
        fprintf(stderr, "El archivo de entrada debe ser: %s <input_file>\n", argv[0]);
        return 1;
    }

    printf("Analisis comenzando...\n"); // Imprime un mensaje indicando que el análisis está comenzando.
    int result = yyparse(); // Realiza el analisis sintactico
    printf("Analisis finalizado: %d\n", result); // Imprime el resultado del análisis.
    printf("num_a: %d, num_b: %d, num_c: %d\n", num_a, num_b, num_c); // Imprime el número de a, b, y c encontrados.

    for (int i = 0; i < 2 * num_b; i++) { // Imprime '2 * num_b' caracteres 'b'.
        printf("b");
    }
    for (int i = 0; i < num_a; i++) { // Imprime 'num_a' caracteres 'a'.
        printf("a");
    }
    printf("\n"); // Imprime un salto de linea-
    fclose(yyin); // Cierra el archivo de entrada.
    return 0; // Termina el programa.
}
```

test.txt

La salida esperada para el programa es la traducción de las cadenas en el archivo de entrada $a^n b^m c^m$ a $b^{2m} a^n$. El archivo 'txt' debe contener la cadena que será analizada:



Para la cadena 'aaabbbccc':

- n = 3
- m = 3
- Traducción: $b^{2m} a^n = 'bbbbbb' \text{ seguido de } 'aaa' = 'bbbbbb' \ 'aaa'$

```
Analisis finalizado: 1
num_a: 3, num_b: 3, num_c: 2
bbbbbbaaa
```

Para la cadena ‘aabbcc’:

- n = 2
- m = 2
- Traducción: $b^{2m}a^{n} = '\ bbbb' seguido\ de\ '\ aa' = 'bbbb'\ 'aa'$

```
Analisis finalizado: 1
num_a: 2, num_b: 2, num_c: 1
bbbbaa
```

Funcionamiento del programa

Comandos desde la terminal:

flex lex.l

bison -d yacc.y

gcc lex.yy.c yacc.tab.c -o traductor

traductor test.txt

```
C:\Users\diego\Desktop\act1>flex lex.l

C:\Users\diego\Desktop\act1>bison -d yacc.y

C:\Users\diego\Desktop\act1>gcc lex.yy.c yacc.tab.c -o traductor

C:\Users\diego\Desktop\act1>traductor test.txt
Analisis comenzando...
Error: syntax error
Analisis finalizado: 1
num_a: 3, num_b: 3, num_c: 2
bbbbbbaaa
```

Actividad 2

De la gramática que define las sentencias declarativas del lenguaje del analizador que se realizó como en el proyecto 2:

1) Sentencia declarativa:

Ejemplos:

```
big  $Grande=2, $num;
real $numReal;
symbol  $cad="cadena#1", $cadVacia;
```

Gramática:

D → <Tipo>K;	Q → =NC
<Tipo>→ b	Q → ,K
<Tipo>→ g	N → n
<Tipo>→ #	N → r
<Tipo>→ y	N → s
<Tipo>→ x	C → ξ
K → iQ	C → ,K
Q → ξ	

A) **Incluir los símbolos de acción y los atributos (con sus reglas de asignación) para que el analizador semántico realice la actualización de la columna tipo en las variables que se declaren.**

B) **Elaborar un analizador sintáctico traductor en yacc/bison de la gramática de traducción con atributos definida para que escriba que tipo de variables se esta declarando. Dada una o varias sentencias declarativas de entrada, escriba "Las variables son de tipo (indicar tipo)".**

Desarrollo

flexer.l

#include "grammar.tab.h"

Incluye el archivo de cabecera generado por Bison que contiene las definiciones de los tokens y otras declaraciones necesarias.

#include <stdlib.h> y #include <string.h>

Incluyen las bibliotecas estándar de C para funciones de memoria y manipulación de cadenas.

void yyerror(const char *s);

Declara la función yyerror que se utilizará para manejar los errores.

```
#include "grammar.tab.h"
#include <stdlib.h>
#include <string.h>

void yyerror(const char *s);
%}
```

Reglas léxicas

En este paso se examina el texto de entrada y lo divide en tokens según patrones predefinidos. Estos tokens se envían al analizador sintáctico (parser) para su procesamiento.

Para cada patrón reconocido, se asigna un valor léxico (usando 'yylval.str = strdup(yytext')) y se devuelve un token correspondiente al analizador sintáctico.

Este proceso permite al analizador léxico transformar el texto de entrada en una secuencia de tokens que el analizador sintáctico puede utilizar para construir la estructura sintáctica del programa.

```
%%

"big"           { yylval.str = strdup("big"); return BIG; } // Reconoce la palabra 'big' , d
"real"          { yylval.str = strdup("real"); return REAL; } // Reconoce la palabra 'real',
"symbol"        { yylval.str = strdup("symbol"); return SYMBOL; } // Reconoce la palabra 'sy
\"[^\"]*\"      { yylval.str = strdup(yytext); return CADENA; } // Reconoce las cadenas de c
\$[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return IDENTIFICADOR; } // Reconoc
[0-9]+          { yylval.str = strdup(yytext); return NUMERO; } // Reconoce secuencias de un
[ \t\n\r]+      { /* Ignora espacios en blanco y saltos de línea */ }
[,;=]           { return yytext[0]; }  // Devuelve los caracteres , ; =
.               { yyerror("Carácter no válido"); } // Reconoce cualquier otro caracter que n

%%
```

int yywrap(void) { return 1; }

Define la función yywrap que se llama cuando se alcanza el final de la entrada. Devolver 1 indica que no hay más datos por procesar.

```c
int yywrap(void) {
    return 1;
}
```

grammar.y

#include <stdio.h>

Biblioteca para funciones de entrada y salida.

#include <stdlib.h>

Biblioteca para funciones de utilidades generales.

#include <string.h>

Biblioteca para manipulación de cadenas.

void yyerror(const char *s);

Declara la función yyerror que se utiliza para manejar errores sintácticos.

int yylex(void);

Declara la función yylex que se utiliza para invocar el analizador léxico.

extern FILE *yyin;

Declara una variable externa yyin que se utiliza para establecer el archivo de entrada del analizador léxico.

```c
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void yyerror(const char *s);
int yylex(void);
extern FILE *yyin;
%}
```

Definiciones de la unión y tokens

%union: Define una unión de tipos de datos que se utilizará para almacenar valores de los tokens. En este caso, se utiliza un puntero a char (char* str) para cadenas.

%token <str> : Declara los tokens que el analizador léxico puede devolver, cada uno con un tipo asociado. Aquí, todos los tokens tienen el tipo char* (cadena).

%type <str> : Declara los tipos no terminales (reglas gramaticales) utilizados en las reglas de producción, también con el tipo char*.

```
%union {
    char* str;
}

%token <str> BIG REAL SYMBOL IDENTIFICADOR CADENA NUMERO

%type <str> tipo lista_vars var valor
```

Reglas de producción

Para este proceso el analizador sintáctico utiliza las reglas de producción para construir la estructura sintáctica del programa a partir de los tokens generados por el analizador léxico. El flujo del proceso es el siguiente:

- Se invoca yyparse() para iniciar el análisis.
- yylex() genera tokens desde el archivo de entrada.
- Define la estructura de una declaración de variables,
- Define los tipos posibles de variables (big, real, symbol).
- Se ejecutan acciones para cada regla reconocida.
- Se manejan errores sintácticos si se encuentran.
- El análisis se completa y el archivo de entrada se cierra.

Este proceso permite al analizador sintáctico construir la representación estructurada del programa y realizar acciones específicas (como imprimir mensajes) basadas en la gramática definida.

```
%%

/*Define una declaración que consiste en un tipo, una lista de variables y un punto y coma.*/
declaracion:
    tipo lista_vars ';' { printf("Las variables son de tipo %s\n", $1); } // Nos especifica que una declaración
;

/*Define los posibles tipos de variables: BIG, REAL, y SYMBOL.*/
tipo: // Cada acción semántica se le asigna una cadena que representa el tipo al valor de la regla ($$).
    BIG { $$ = strdup("big"); }
  | REAL { $$ = strdup("real"); }
  | SYMBOL { $$ = strdup("symbol"); }
;

/*Define una lista de variables que puede ser una sola variable o una lista de variables separadas por comas.*/
lista_vars:
    var { } // Es una acción vacía por lo tanto no se realiza ninguna operación adicional.
  | lista_vars ',' var { }
;

/*Define una variable que puede ser un identificador solo o un identificador con un valor asignado.*/
var:
    IDENTIFICADOR { } // Es una acción vacía por lo tanto no se realiza ninguna operación adicional.
  | IDENTIFICADOR '=' valor { }
;

/*Define un valor que puede ser una cadena o un número.*/
valor:
    CADENA { } // Es una acción vacía por lo tanto no se realiza ninguna operación adicional.
  | NUMERO { }
;

%%
```

Subrutinas

El propósito de esta parte del código es manera y reportar errores sintácticos, la función ‘main’ actúa como el punto de entrada del programa. Configura el archivo de entrada y ejecuta el analizador sintáctico.

Llama a ‘yyparse()’ para iniciar el análisis sintáctico, cierra el archivo de entrada una vez que el análisis se completa y finalmente termina el programa con un código de éxito o error, según sea el caso.

```c
/*Función yyerror que se llama cuando se encuentra un error sintáctico.*/
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main(int argc, char *argv[]) { // Punto de entrada del programa.
    if (argc != 2) { // Verifica que este correctamente nombre del archivo de entrada (test.txt).
        fprintf(stderr, "El archivo de entrada debe ser: %s <archivo_entrada>\n", argv[0]);
        return 1;
    }

    FILE *archivo_entrada = fopen(argv[1], "r"); // Abre el archivo de entrada (.txt).
    /*Si el archivo no se abre correctamente imprime un mensaje de error.*/
    if (!archivo_entrada) {
        perror("No se puede abrir el archivo de entrada");
        return 1;
    }

    yyin = archivo_entrada; // Asigna el archivo de entrada a la variable yyin que el analizador léxico usará.
    yyparse(); // Llama al analizador sintáctico para que procese archivo de entrada.
    fclose(archivo_entrada); // Cierra el archivo de entrada.
    return 0; // Termina el programa.
}
```

El paso de subrutinas nos proporciona las funciones necesarias para manejar errores y ejecutar el análisis sintáctico correctamente, configurando el entorno y los archivos de entrada adecuados para el proceso.

test.txt

Una vez ejecutado el comando “parser test.txt” el analizador estará listo para recibir las declaraciones de variables. Se debe proporcionar una entrada válida que cumpla con la gramática definida para ver los resultados.

test.txt
```
1   big $Grande=2, $num;
2   real $numReal;
3   symbol $cad="cadena#1", $cadVacia;
```

Para el caso de *“big $Grande=2, $num;”* se deberá de observar en la terminal el siguiente resultado:

```
C:\Users\diego\Desktop\act2>parser test.txt
Las variables son de tipo big
```

Para el segundo caso *“real $numReal;”* se deberá de observar en la terminal el siguiente resultado:

```
C:\Users\diego\Desktop\act2>parser test.txt
Las variables son de tipo real
```

Finalmente para el segundo caso *“symbol $cad="cadena#1", $cadVacia;”* se deberá de observar en la terminal el siguiente resultado:

```
C:\Users\diego\Desktop\act2>parser test.txt
Las variables son de tipo symbol
```

Funcionamiento del programa

Comandos desde la terminal:

flex lexer.l

bison -d grammar.y

gcc lex.yy.c grammar.tab.c -o parser

parser test.txt

```
C:\Users\diego\Desktop\act2>flex lexer.l

C:\Users\diego\Desktop\act2>bison -d grammar.y

C:\Users\diego\Desktop\act2>gcc lex.yy.c grammar.tab.c -o parser

C:\Users\diego\Desktop\act2>parser test.txt
Las variables son de tipo symbol
```