

TECNOLÓGICO NACIONAL DE MÉXICO INSTITUTO TECNOLÓGICO DE ENSENADA

PROGRAMA DE INGENIERÍA EN SISTEMAS COMPUTACIONALES

REPORTE FINAL

LENGUAJE OPMEZ

Autores:

Itsai Zempoaltecatl Mejia

No Control: 16760300

Profesora:

ME Xenia Padilla Madrid

Lenguajes y Autómatas 2 (enero-junio 2022)

Ensenada B.C. México
17 de junio de 2022

- **Objetivo General:** Aprender el uso de las expresiones regulares, como se conforma un lenguaje y cómo crear un lenguaje con el uso de ANTLR.
- **Objetivo Específico:** Crear un nuevo lenguaje haciendo uso de la tecnología ANTLR, Java y Jasmin para la parte computacional y con Java FX la parte visual.

1. CONTEXTO

Con ANTLR4 y Java se creó un traductor de lenguaje junto con una propuesta de lenguaje nuevo desarrollado con una sintaxis propia respetando ciertas reglas propias de los lenguajes, como son los operadores lógicos y la estructura de las cadenas de texto.

El uso de ANTLR nos permite identificar la gramática del lenguaje C para traducirlo al nuevo lenguaje y detectar asimismo la gramática de este.

Se hizo uso de los Visitors para analizar y mostrar el resultado del nuevo lenguaje así como la traducción a partir de C.

El lenguaje creado para este caso se llama Opmez.

Sentencia en C	Equivalente
void main(){}	#initial()-><-
int	use
printf();	systalk();
if	if
else	else
else if	elif
while	while
()	()
{}	-> <-
//	@
/**/	@{ }@

Figura 1: Tabla de palabras reservadas equivalente entre C y Opmez (sólo se muestran las palabras reservadas que fueron cambiadas o caracteres)

Los operadores lógicos como ' $<$ ' (menor que) ' $>$ ' (mayor que) entre otros son los mismos ya que estos no pueden variar porque son propios de las matemáticas

Editor

Para poder traducir y ejecutar nuestro código se creó un compilador en JavaFX con el apartado de editor de texto donde escribiremos el código en C que será traducido a Opmez y viceversa. El compilador tiene las funciones básicas de un editor típico.

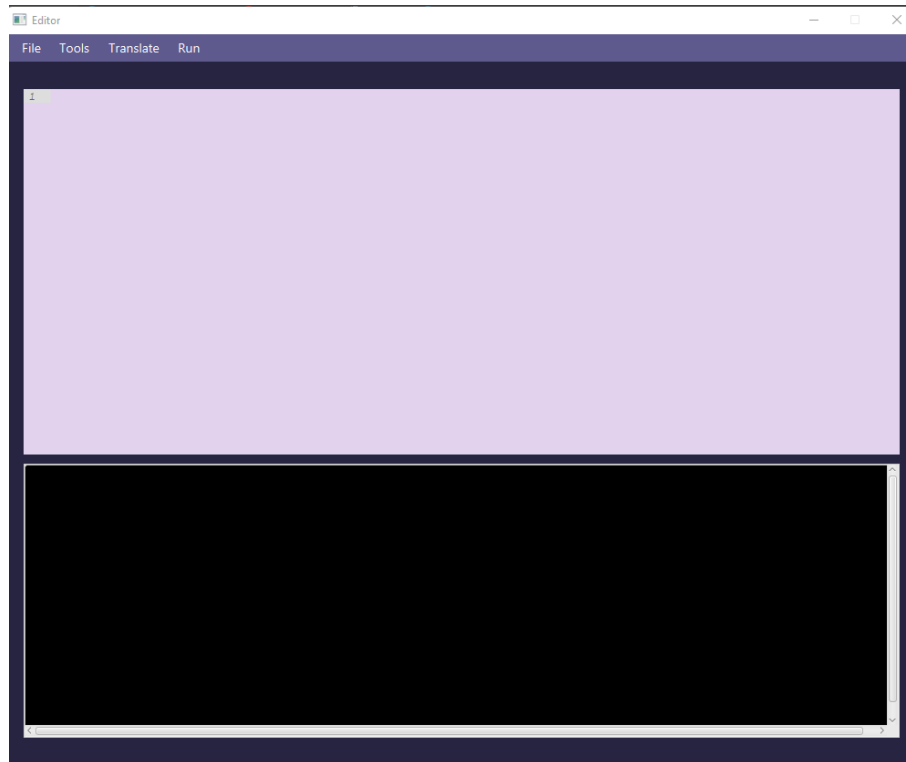


Figura 2: Editor

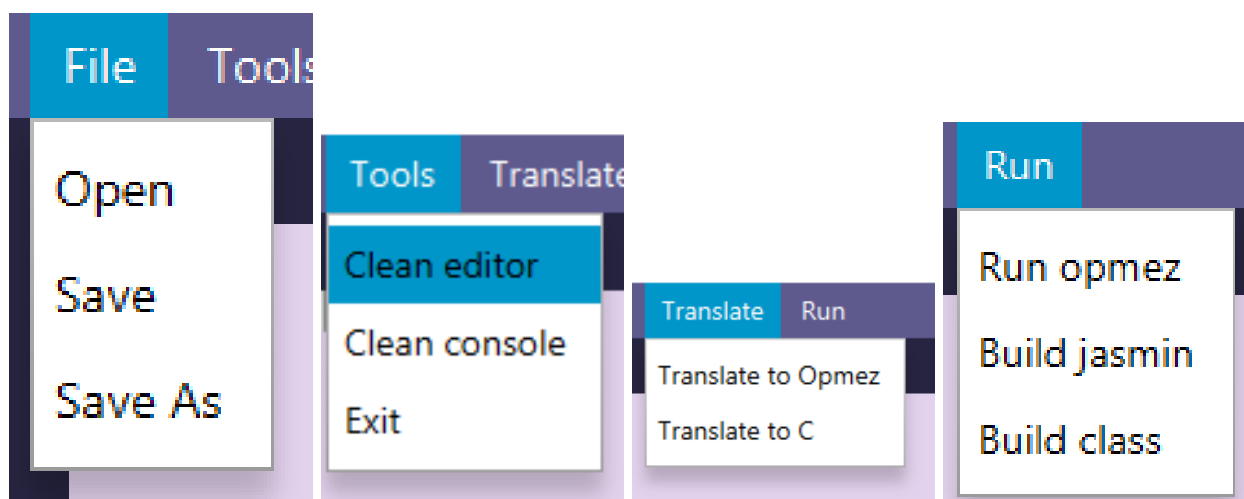


Figura 3: Funciones editor

2. DESARROLLO

2.1. Primeros pasos

Como primer ejemplo y prueba se creo una gramatica basica capaz de reconocer las operaciones matematicas basicas en el lenguaje C y desplegar un arbol sintactico de este.

```
body:
    declaracion* | asignacion;
declaracion: 'int' ID ';' ;
asignacion: ID '=' expr ';' ;
expr:
    expr op=(MULT|DIV) expr #multDiv
    |
    expr op=(SUM|SUB) expr #sumSub
    |
    SUB? NUM #num
    |
    ID #id
    |
    PO expr PC #parentesis
    ;

VOID: 'void';
MAIN: 'main';
PO: '(';
PC: ')';
MULT: '*';
DIV: '/';
SUM: '+';
SUB: '-';
NUM: '[0-9]+';
ID: '[a-zA-Z][a-zA-Z0-9]*';
WS: '[\t\r\n]+' -> skip;
```

Figura 4: Primer gramatica

La construccion de esta gramatica da inicio al reconocedor del lenguaje C, ya que expr nos sera util en la mayor parte del lenguaje considerando que es la expresion minima del lenguaje.

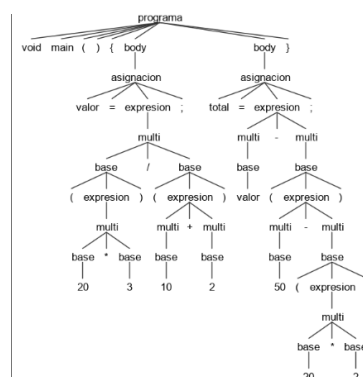


Figura 5: Arbol sintactico

3. Implementar conexión de Java con Antlr

Para implementar antlr con java es necesario seguir ciertos pasos para que todo salga correcto.

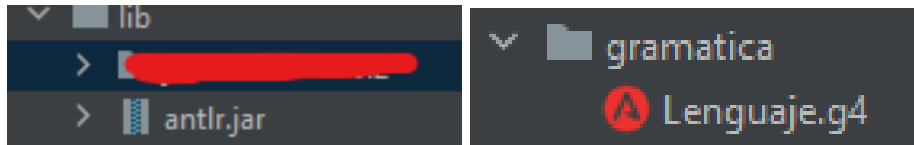


Figura 6: Carpetas y archivos

Creamos 2 carpetas “lib” y “gramática”, en lib agregamos nuestro archivo antlr.jar y en gramática creamos nuestro archivo.g4 para la gramática

```
antlr -package com.lenguaje.parser -o ../src/com/lenguaje/parser -no-listener -visitor ./Lenguaje.g4
```

Figura 7: Comando para generar archivos

Para crear nuestros archivos .java para conectar antlr con java usaremos el siguiente comando donde -package define el paquete donde se generaran los archivos, -o la carpeta donde se guardaran, -no-listener evita que se generen archivos listener y -visitor genera los archivos visitors los cuales son muy importantes

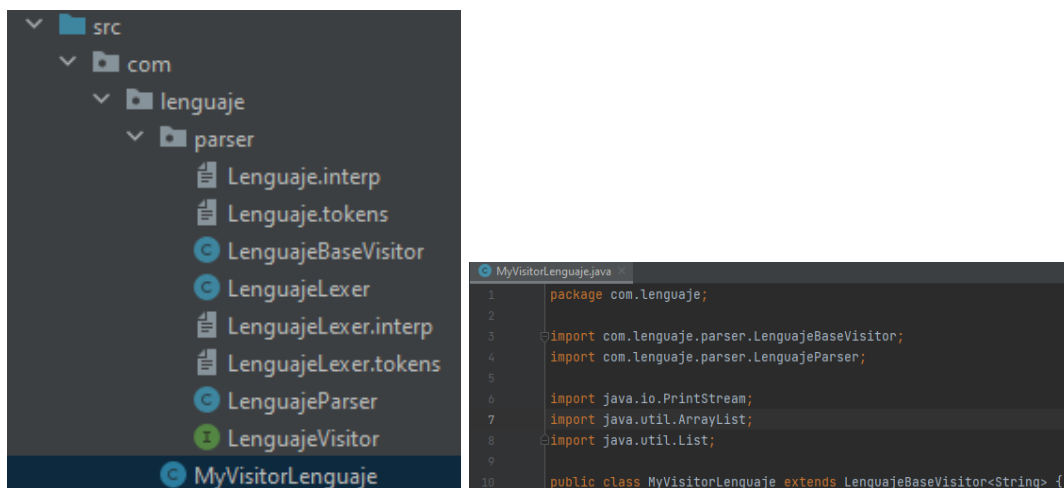


Figura 8: Archivos generados y Visitor

Una vez creados los archivos que se encuentran dentro del package “parser” creamos un archivo java con el nombre MyVisitorLenguaje (para este caso), este archivo es necesario para usar los archivos previamente creados y poder hacer uso de la gramática.

4. Creación de una gramatica en Antlr para reconocer el lenguaje C

La gramatica "Lenguaje" es la encargada de analizar que la entrada sea un codigo C. Para este caso contamos con algunas sentencias basicas para analizar, las cuales son:

```
grammar Lenguaje;  
  
program: file;  
file: VOID MAIN PO PC KO body* KC #archivo;
```

Figura 9: program-file

Es el inicio de la gramatica y el punto mas alto del arbol sitactico, en el se engloba a todos los tokens de la gramatica

```
body:  
    print  
    |  
    if_else_elif  
    |  
    assigment_declaration  
    |  
    declaration  
    |  
    assigment  
    |  
    cicle  
    ;
```

Figura 10: body

body contiene las diferentes sentencias basicas de C, se agrupan ya que al poder usarse en cualquier momento es necesario definir un solo token que contenga a todos estos tokens.

Cada uno de estos token tienen lexemas diferentes y la funcionalidad final de cada uno de ellos, es que al usar el Visitor de cada uno sera devolver un equivalente del lenguaje Opmez.

A continuacion veremos como es que se constituyen cada uno de ellos.

```

print: PRINTF PO expr PC SCOL #impresionExpr |PRINTF PO STRING PC SCOL #impresionString;
assignment: ID ASSIGN expr SCOL #asignacion;
assignment_declaration: INT ID ASSIGN expr SCOL #asigDeclar;
declaration: INT ID SCOL #declaracion;
cicle: WHILE PO condition PC KO body* KC #ciclo;
if_else_elif: if_sentence (elif_sentence | else_sentence)? #ifElse;
elif_sentence: ELSE if_sentence (elif_sentence | else_sentence)? #sentenciaElif;
if_sentence: IF PO condition PC KO body* KC #sentenciaIf;
else_sentence: ELSE KO body* KC #sentenciaElse;

```

Figura 11: Tokens con lexemas

print reconoce la entrada en C de la sentencia de impresion "printf()", contiene los 2 tipos de impresion, de numeros o expresiones (num/id) o de cadenas de texto

assignment: este reconoce la sentencia de asignacion, aunque en este caso no varia entre lenguajes es necesaria para identificarla en el codigo (ej. a=10;)

declaration: previo a una asignacion es necesario declarar la variable, este se encarga de reconocer esa sentencia

cicle: como todo lenguajes es necesario contar con ciclos, para este caso se creo un lexema que reconociera a la sentencia while

if-else-elif: por ultimo la sentencia condicional, ya que un lenguaje no tendria sentido sin if-else-elif. Como podemos notar, if-else-elif cuenta con 3 tokens ya que esta sentencia es algo mas extensa que las demas al conformarse de diferentes partes, if es una, else otra y else if otra diferente.

Los tokens if-else-else if y while contienen condiciones para su funcionamiento, para ellos es necesario crear un token que contenga diferentes lexemas que reconozcan las diferentes condiciones basicas que maneja cualquier codigo.

```

condition:
    NOT condition #condicionNegacion
    |
    condition (EQ|NEQ) condition #condicionesIgualdad
    |
    condition (GT|LT|GEQ|LEQ) condition #condicionesRelacional
    |
    condition AND condition #condicionY
    |
    condition OR condition #condicionO
    |
    TRUE #verdadero
    |
    FALSE #falso
    |
    PO condition PC #condicionParentesis
    |
    expr #expresion
    ;

```

Figura 12: condition

Las condiciones tienen la finalidad de decidir en base a su resultado si se ejecuta o no cierto fragmento de codigo. Los operadores logicos que son los que le dan vida a estas condiciones no tienen gran diferencia entre lenguajes.

condicionNegacion: reconoce la negacion de una condicion (ej. !a==b)

condicionesIgualdad: reconoce si la sentencia es una igualdad o diferencia (ej. a==b, a!=b)

condicionesRelacional: hay 4 variantes de los operadores relacionales (ej. a<b, a>b, a<=b, a>=b)

condicion Y/O: reconoce la condicion con los operadores basicos and y or (ej. ab, a——b)

true/false: true y false se pueden considerar como expresiones minimas, pero en este caso su valor es su nombre literalmente.

Por ultimo estan los parentesis que ayudan a separa por jerarquia de resolucion las condiciones y expr ya que como vemos en algunos ejemplos, se comparan expresiones minimas entre si o con otras expresiones.

```

expr:
    expr op=(MULT|DIV) expr #multDiv
    |
    expr op=(SUM|SUB) expr #sumSub
    |
    SUB? NUM #num
    |
    ID #id
    |
    PO expr PC #parenthesis
    ;

```

Figura 13: expr

En el punto 2.1 en la imagen 5 tenemos un árbol sintáctico que nos indica cual es la expresión mínima o en otras palabras, donde el árbol ya no despliega ramas. Para este caso se llama `expr` el token que contiene los lexemas más simples. Aquí se contienen las expresiones de operaciones aritméticas más básicas al igual que los números y nombres de variables.

multDiv: reconoce una multiplicación o división, según indique el operador (ej. $1*2$)

sumSub: reconoce una suma o resta, según indique el operador (ej. $1+2$)

num: reconoce un número positivo o negativo entero

id: reconoce una variable, o sea, un nombre que se le asigna a una variable

parenthesis: como en las expresiones, los parentesis ayudan a jerarquizar la resolución de las operaciones o las expresiones.

5. Creación de una clase Visitor para la gramática que reconoce el lenguaje C y traducir a Opmez

Como antes vio en el punto 3, se creó una clase llamada `MyVisitoLenguaje`, con este podemos acceder al contexto de cada lexema de cada token con ayuda de los métodos `visit`.

Esta clase se utilizará para crear el lenguaje Opmez, cada método de la clase devuelve un `String`, cuando visitamos cada lexema y obtenemos su contexto, iremos cambiando su resultado en base a las etiquetas equivalentes del lenguaje Opmez (punto 1).

```

public static List<String> newSentence = new ArrayList<>();
public int errors = 0;
private PrintStream ps;
public MyVisitorLenguaje(PrintStream ps){
    this.ps=ps;
    System.setErr(ps);
    System.setOut(ps);
}

```

Figura 14: Variables y constructor

Estas variables ayudarán al funcionamiento del visitor así como para crear el lenguaje

newSentence: es una lista que almacena cada nueva sentencia en el lenguaje Opmez para mostrarla después

error: si ocurre un error en la traducción de alguna sentencia el número de errores aumenta, esta variable ayuda a validar que no existan errores a la hora de traducir a manera de bandera.

Constructor: El parámetro obtenido en el constructor es una clase que permite asignar a una área de texto custom las impresiones del sistema (`println`)


```

@Override
public String visitArchivo(LenguajeParser.ArchivoContext ctx) {
    newSentence.add("#initial()->");
    for (int i=0; i< ctx.body().size(); i++){
        String current = visit(ctx.body(i));
        if(current != null) {
            newSentence.add(visit(ctx.body(i)));
        }
        else{
            errors++;
            break;
        }
    }
    newSentence.add("<-");
    return null;
}

```

Figura 15: visitArchivo

visitArchivo es donde se va agregando a la lista previamente creada las nuevas sentencias, ctx.body() contiene todas las sentencias del lenguaje en C y al visitar cada una de ellas obtenemos su nuevo valor traducido a Opmez

También aquí se agrega el inicio y el fin de la estructura completa del lenguaje Opmez.

Analizando más a fondo el visitor podemos ver que cada método devuelve el equivalente de C a Opmez y así ir construyendo el nuevo lenguaje.

```

@Override
public String visitDeclaracion(LenguajeParser.DeclaracionContext ctx) {
    return "use " + ctx.ID() + ";";
}

@Override
public String visitAsignacion(LenguajeParser.AsignacionContext ctx) {
    return ctx.ID() + "=" + visit(ctx.expr()) + ";";
}

```

Figura 16: Declaracion y Asignacion

visitDeclaracion: es uno de las primeras sentencias que cambiara, de "int" a "use" y lo demás se conserva

visitAsignacion: realmente el valor de asignacion queda de la misma manera en Opmez que en C, aquí no notamos un cambio

```

@Override
public String visitImpresionExpr(LenguajeParser.ImpresionExprContext ctx) {
    try{
        return "systalk(" + visit(ctx.expr()) + ")";
    }catch (Exception e){
        errors++;
        System.out.println(e);
        return null;
    }
}

@Override
public String visitImpresionString(LenguajeParser.ImpresionStringContext ctx) {
    try{
        return "systalk(" + ctx.STRING() + ")";
    }catch (Exception e){
        errors++;
        System.out.println(e);
        return null;
    }
}

```

Figura 17: Impresiones

visitImpresionExpr: esta impresion cambia de printf a systalk la sentencia manteniendo lo demás muy similar, en este caso se visita la expr contenida en el contexto para saber obtener el valor como tal de lo que se imprimira

visitImpresionString: de la misma manera, cambia de printf a systalk la sentencia, para este caso no visitamos el resultado contenido entre los parentesis, pero si devolvemos el contexto de STRING tal y como esta originalmente en el contexto

```

@Override
public String visitParentesis(LenguajeParser.ParentesisContext ctx) {
    try{
        return '(' + visit(ctx.expr()) + ')';
    }catch (Exception e){
        errors++;
        return null;
    }
}

@Override
public String visitNum(LenguajeParser.NumContext ctx) {
    try{
        return Integer.valueOf(ctx.getText()).toString();
    }catch (Exception e){
        return null;
    }
}

@Override
public String visitId(LenguajeParser.IdContext ctx) {
    try{
        return ctx.ID().getText();
    }catch (Exception e){
        errors++;
        System.out.println(e);
        return null;
    }
}
}

```

Figura 18: Expresiones

Estos 3 visit no sufren cambios notables, en si devuelven el mismo valor del contexto, previamente se habia definido una variante a los parentesis, pero para mantener la sintaxis tipica de los lenguajes en las operaciones aritmeticas se dejaron los parentesis comunes

```

@Override
public String visitMultDiv(LenguajeParser.MultDivContext ctx) {
    try{
        return ctx.getText();
    }catch (Exception e){
        errors++;
        System.out.println(e);
        return null;
    }
}

@Override
public String visitSumSub(LenguajeParser.SumSubContext ctx) {
    try{
        return ctx.getText();
    }catch (Exception e){
        errors++;
        System.out.println(e);
        return null;
    }
}
}

```

Figura 19: Expresiones operaciones

Las operaciones basicas al no poder cambiar de operadores aritmeticos se mantienen tal y como se obtienen del contexto

```

@Override
public String visitIfElse(LenguajeParser.IfElseContext ctx) {
    boolean existNull;
    try {
        String lineIf = visit(ctx.if_sentence());
        existNull= (lineIf == null)? true:false;
        String lineElse = "";
        String lineElif = "";
        if (ctx.else_sentence() != null) {
            lineElse = visit(ctx.else_sentence());
            existNull= (lineElse == null)? true:false;
        } else if (ctx.elif_sentence() != null) {
            lineElif = visit(ctx.elif_sentence());
            existNull= (lineElif == null)? true:false;
        }
        if(!existNull)
            return lineIf + lineElse + lineElif;
        else return null;
    } catch (Exception e) {
        errors++;
        System.out.println(e);
        return null;
    }
}

```

Figura 20: visitIfElse

La sentencia mas elaborada es if-else-else if, al contar con un body se puede hacer una sentencia muy amplia, tambien hay que considerar que como puede contener solo un if como puede contener los 3 bloques (If-else-else if) hay que analizar cada uno de ellos.

Este visit construye toda la sentencia de If-else-else if en base a los resultados obtenidos de los visit de cada sentencia por separado, es por eso que es necesario verificar que las demas partes de la sentencia ya sea else o else if esten en el codigo en caso contrario se agrega una cadena vacia

```

@Override
public String visitSentenceIf(LenguajeParser.SentenceIfContext ctx) {
    boolean fail = false;
    try{
        StringBuilder sb = new StringBuilder();
        for(int i=0; i<ctx.body().size(); i++){
            String current = visit(ctx.body(i));
            if(current != null)
                sb.append(current+"\n");
            else{
                fail = true;
                break;
            }
        }
        if(!fail)
            return "if("+visit(ctx.condition())+"->\n"+sb + "<-";
        else return null;
    }catch (Exception e){
        errors++;
        System.out.println(e);
        return null;
    }
}

```

Figura 21: visitSentenceIf

La sentencia if en Opmez solo tiene la variante de las llaves que encierran el cuerpo de if (– >< – es lo mismo que {}), el resultado devuelve el inicio de la sentencia tipica de los lenguajes, la condicion y por ultimo el cuerpo de if

```

@Override
public String visitSentenciaElse(LenguajeParser.SentenciaElseContext ctx) {
    boolean fail = false;
    try {
        StringBuilder sb = new StringBuilder();
        for (int i=0; i<ctx.body().size(); i++) {
            String current = visit(ctx.body(i));
            if (current == null) {
                fail = true;
                break;
            }
            sb.append(current+"\n");
        }
        if (!fail)
            return "else->\n"+sb+"<-";
        else return null;
    } catch (Exception e) {
        errors++;
        System.out.println(e);
        return null;
    }
}

```

Figura 22: visitElse

Como en el caso de if, se cambian las llaves y en realidad es el unico cambio notorio en este bloque de la sentencia if-else-else if

```

@Override
public String visitSentenciaElif(LenguajeParser.SentenciaElifContext ctx) {
    boolean existNull;
    try {
        String lineIf = visit(ctx.if_sentence());
        existNull = (lineIf == null)? true:false;
        String lineElse = "";
        String lineElif = "";
        if (ctx.else_sentence() != null) {
            lineElse = visit(ctx.else_sentence());
            existNull = (lineElse == null)? true:false;
        } else if (ctx.elif_sentence() != null) {
            lineElif = visit(ctx.elif_sentence());
            existNull = (lineElif == null)? true:false;
        }

        if (!existNull)
            return "el"+lineIf + lineElse + lineElif;
        else return null;
    } catch (Exception e) {
        errors++;
        System.out.println(e);
        return null;
    }
}

```

Figura 23: visitElif

Ya que elif comparte la misma logica que if tiene una construccion muy similar, solo se agrego un "el" para que quede con la sintaxis "elif"

```

@Override
public String visitCiclo(LenguajeParser.CicloContext ctx) {
    boolean fail = false;
    try{
        StringBuilder sb = new StringBuilder();
        for(int i=0; i<ctx.body().size(); i++) {
            String current = visit(ctx.body(i));
            if (current == null){
                fail = true;
                break;
            }
            sb.append(current+"\n");
        }
        if(!fail)
            return "while("+ctx.condition().getText()+")->\n"+sb+"<-";
        else return null;
    }catch (Exception e){
        errors++;
        System.out.println(e);
        return null;
    }
}

```

Figura 24: visitCiclo

El ciclo tiene una sintaxis muy similar a if, es por eso que lo unico que se cambia son las llaves y se devuelve la sentencia con while, condicion y contenido. A diferencia de if-else-else if este solo es una sentencia a la vez

```

@Override
public String visitCondicionNegacion(LenguajeParser.CondicionNegacionContext ctx) {
    return ctx.getText();
}

@Override
public String visitCondicionesIgualdad(LenguajeParser.CondicionesIgualdadContext ctx) {
    return ctx.getText();
}

@Override
public String visitCondicionesRelacional(LenguajeParser.CondicionesRelacionalContext ctx) {
    return ctx.getText();
}

@Override
public String visitCondicionY(LenguajeParser.CondicionYContext ctx) {
    return ctx.getText();
}

@Override
public String visitCondicionO(LenguajeParser.CondicionOContext ctx) {
    return ctx.getText();
}

@Override
public String visitVerdadero(LenguajeParser.VerdaderoContext ctx) {
    return ctx.getText();
}

@Override
public String visitFalso(LenguajeParser.FalsoContext ctx) {
    return ctx.getText();
}

@Override
public String visitCondicionParentesis(LenguajeParser.CondicionParentesisContext ctx) {
    return ctx.getText();
}

@Override
public String visitExpresion(LenguajeParser.ExpresionContext ctx) {
    return visit(ctx.expr());
}

```

Figura 25: Condiciones

Todas las condiciones mantienen la misma sintaxis en C y en Opmez, es por eso se que solo se devuelve el contexto tal y como viene originalmente.

5.1. Resultados Lenguaje C a Opmez

Traducir de C a Opmez

Para verificar que la traducción esta correcta corremos una prueba de traducir C a Opmez con todas las sentencias y algunas condiciones/expresiones.

```
1 void main() {  
2  
3 int a;  
4 int b;  
5  
6 int c;  
7 a=10;  
8 b=20;  
9 c=a+b;  
10  
11 printf(c);  
12 printf("hola mundo");  
13  
14 if(a>b){  
15 printf("a es mayor que b");  
16 }else if(a==b){  
17 printf("a igual a b");  
18 }else{  
19 printf("a diferente de b");  
20 }  
21 }
```

Figura 26: Ejemplo entrada C

```
1 #initial()->  
2 use a;  
3 use b;  
4 use c;  
5 a=10;  
6 b=20;  
7 c=a+b;  
8 systalk(c);  
9 systalk("hola mundo");  
10 if(a>b)->  
11 systalk("a es mayor que b");  
12 <-elif(a==b)->  
13 systalk("a igual a b");  
14 <-else->  
15 systalk("a diferente de b");  
16 <-  
17 <-  
18
```

Figura 27: Salida en Opmez

6. Creación de una gramatica en Antlr para reconocer el lenguaje Opmez

La gramatica "Opmez" se encarga de analizar la entrada en el lenguaje Opmez. A continuacion veremos como esta construida esta gramatica.

```
grammar Opmez;

program: DEFINEFUNC INITIAL PO PC KO instructions* KC #cuerpo;
```

Figura 28: program-file

Es el inicio de la gramatica y el punto mas alto del arbol sitactico, en el se engloba a todos los tokens de la gramatica

```
instructions:
    PRINT PO expr PC SCOL #impresionExpr
    |
    PRINT PO string PC SCOL #impresionString
    |
    if_sentence (elif_sentence | else_sentence)? #ifElse
    |
    DECLARE ID SCOL #declaracion
    |
    ID ASSIGN expr SCOL #asignacion
    |
    DECLARE ID ASSIGN expr SCOL #asigDeclar
    |
    WHILE PO condition PC KO body KC #cicle
    ;
```

Figura 29: instructions

instructions contiene todas las instrucciones basicas del lenguaje Opmez, como se puede apreciar estan definidos los lexemas de cada setencia, el unico que no se encuentra es if-else-elif ya que como antes se menciono, es una estructura mas compleja para quedar en solo un lexema

impresionExpr: detecta una sentencia de impresion de una expr

impresionString: detecta una sentencia para imprimir una cadena de texto (string)

ifElse: reconoce la sentencia if con sus sentencias complemente (else-elif)

declaracion: detecta una sentencia de declaracion de una expr (numero o id)

asignacion: detecta una asignacion de un ID (ej. a=20;)

cicle: reconoce una sentencia de un ciclo while

```
elif_sentence: elif_frag_condition (elif_sentence | else_sentence)? #sentenciaElif
elif_frag_condition: ELIF PO condition PC KO body KC #condicionElif;
if_sentence: IF PO condition PC KO body KC #sentenciaIf;
else_sentence: ELSE KO body KC #sentenciaElse;
body: instructions* #cuerpoScope;
```

Figura 30: Estructura if

La desicion de separa de esta manera la estructura de la sentencia if-else-elif es que al separar cada bloque en un token con lexema independiente hace mas sencilla la tarea de verificar errores de sintaxis ya que las condiciones no chocan en jerarquia o las visitas a las instrucciones.

sentenciaElif: define la estructura de elif, se parece mucho a ifElse con la diferencia que esta no pertenece a instructions por si sola, sino que debe estar siempre despues de una sentencia if

condicionElif: se usara para visitar la condicion y el cuerpo de elif unicamente, aplicando lo que se explico anteriormente.

sentenciaIf: define la estructura de la sentencia if

sentenciaElse: define la estructura de else

cuerpoScope: para visitar cada sentencia dentro de if o else o elif o while se definio un token llamado body que contiene cero una o mas instructions

```

condition:
  NOT condition #condicionNegacion
  |
  expr op=(EQT|NEQT) expr #condicionesIgualdadExpr
  |
  condition op=(EQT|NEQT) condition #condicionesIgualdad
  |
  expr op=(GT|LT) expr #condicionesMayMen
  |
  expr op=(GEQT|LEQT) expr #condicionesMayMenIgual
  |
  condition AND condition #condicionY
  |
  condition OR condition #condicionO
  |
  PO condition PC #condicionParentesis
  |
  TRUE #verdadero
  |
  FALSE #falso
  |
  expr #expresion
  ;

```

Figura 31: Condiciones Opmez

Existen 2 variantes de algunas condiciones, por ejemplo las condiciones de igualdad, una valida a las expresiones y otra valida condiciones. Esto se debe a que en un if se puede presentar un caso en el que se comparen 2 numeros o se comparen la comparacion de 2 numeros (ej. $(2==2)==(1==1)$) y el lenguaje debe permitirlo

condicionNegacion: reconoce la negacion de una condicion (ej. $!a==b$)

condicionesIgualdad: reconoce si la sentencia es una igualdad o diferencia (ej. $a==b$, $a!=b$)

condicionesRelacional(mayMen/MayMenIgual): hay 4 variantes de los operadores relacionales (ej. $a<b$, $a>b$, $a<=b$, $a>=b$)

condicion Y/O: reconoce la condicion con los operadores basicos and y or (ej. ab , $a \text{---} b$)

true/false: true y false se pueden considerar como expresiones minimas, pero en este caso su valor es su nombre literalmente.

Por ultimo estan los parentesis que ayudan a separa por jerarquia de resolucion las condiciones y expr ya que como vemos en algunos ejemplos, se comparan expresiones minimas entre si o con otras expresiones.


```
string:
    STRING #cadenaTexto;

expr:
    expr op=(MULT|DIV) expr #multDiv
    |
    expr op=(SUM|SUB) expr #sumSub
    |
    SUB? INT #numero
    |
    SUB? ID #id
    |
    PO expr PC #parentesis
    ;
```

Figura 32: string y expr

string: en esta gramática solo tiene la funcionalidad de reconocer una cadena de texto en una impresión de texto

expr: Contiene los diferentes lexemas de una expr

multDiv: reconoce una multiplicación o división, según indique el operador (ej. $1*2$)

sumSub: reconoce una suma o resta, según indique el operador (ej. $1+2$)

num: reconoce un número positivo o negativo entero

id: reconoce una variable, o sea, un nombre que se le asigna a una variable

parentesis: como en las condiciones, los parentesis ayudan a jerarquizar la resolución de las operaciones o las expresiones.

7. Creacion de una clase Visitor para la gramatica Opmez para verificar la sintaxis correcta, la declaracion y asignacion de variables, los tipos de datos y a su vez la creacion de las lineas de comando para crear un archivo jasmin (.j)

Para poder ejecutar el lenguaje Opmez es necesario verificar que todo este correctamente escrito, para esto nos ayuda Antlr, pero en el caso de funcionalidad interna como el verificar que las variables no se repitan, que las condiciones o expresiones cumplan su sintaxis y logica es que se creo un Visitor encargado de esto. Igual nos sera util para crear el archivo jasmin que sera explicado mas adelante.

CheckOpmezVisitor

```
public class CheckOpmez extends OpmezBaseVisitor<Object> {
    public static HashMap<String, Integer> memory = new LinkedHashMap<>();
    public static HashMap<String, Integer> tempMemory = new LinkedHashMap<>();
    public static List<String> compilador = new ArrayList<>();
    private boolean joinIfElse = false;

    private boolean errorDeclaration = false;
    public int errors = 0;
    private PrintStream ps;
    int line = 0;
    int numLabel = 0;
    String globalPos;
    String label;
    public CheckOpmez(PrintStream ps){
        this.ps=ps;
        System.setOut(this.ps);
        System.setErr(this.ps);
    }
}
```

Figura 33: Declaracion de variables

memory: guardara las variables declaradas junto con su valor

tempMemory: guardara las variables temporales de if/else/while con su valor

compilador: es una lista de String para ir agregando las lineas para crear el archivo jasmin (.j)

joinIfElse, errorDeclaration, errors: son banderas para el control del programa

```
@Override
public Object visitCuerpo(OpmezParser.CuerpoContext ctx) {
    compilador.add(".class publicCodigo");
    compilador.add(".super java/lang/Object");
    compilador.add(".method public static main([Ljava/lang/String;)V");
    compilador.add(".limit stack 20");
    compilador.add(".limit locals 20");
    for (int i = 0; i < ctx.instructions().size(); i++) {
        visit(ctx.instructions(i));
        line++;
    }
    compilador.add(".return");
    compilador.add(".end method");
    return null;
}
```

Figura 34: visitCuerpo

Al inicio tenemos la visita al cuerpo, en este metodo visitamos a cada instruccion, aqui no se verifica nada, solo es para agregar las instrucciones para el archivo jasmin

Tambien notamos que se agregaron instrucciones que indican el inicio y el final del archivo jasmin

```

@Override
public Object visitDeclaracion(OpmezParser.DeclaracionContext ctx) {
    String id = ctx.ID().getText();

    if(!joinIfElse){
        if(memory.containsKey(id) && tempMemory.containsKey(id)){
            errorDeclaracion = true;
            errors++;
            System.out.println("Linea"+linea+": "+ctx.ID().getText()+" ya esta declarada");
        }else{
            tempMemory.put(id,null);
        }
    }else if(!memory.containsKey(id)){
        memory.put(id,0);
    }else{
        errorDeclaracion = true;
        errors++;
        System.out.println("Linea"+linea+": "+ctx.ID().getText()+" ya esta declarada");
    }
    return null;
}

```

Figura 35: visitDeclaracion

Esta verifica que la variable declarada no este previamente guardada, de no ser asi la agrega a la memoria dependiendo si esta en un if o esta en el cuerpo principal.

```

@Override
public Object visitAsignacion(OpmezParser.AsignacionContext ctx) {
    String id = ctx.ID().getText();
    if(errorDeclaracion){
        errors++;
        System.out.println("Linea"+linea+": "+ctx.ID().getText()+" no esta declarada");
        return null;
    }else{
        try{
            int value = (int)visit(ctx.expr());
            if(!joinIfElse){
                if(!tempMemory.containsKey(id)){
                    tempMemory.put(id, value);
                    return tempMemory.get(id);
                }else if(memory.containsKey(id)){
                    memory.put(id,value);
                    int pos=0;
                    for (String key: memory.keySet()) {
                        if(key.equals(id)) break;
                    }
                    pos++;
                    compilador.add("istore "+pos);
                    return memory.get(id);
                }else{
                    errors++;
                    System.out.println("Linea"+linea+": "+ctx.ID().getText()+" no esta declarada");
                    return null;
                }
            }else if(memory.containsKey(id)){
                memory.put(id, value);
                int pos=0;
                for (String key: memory.keySet()) {
                    if(key.equals(id)) break;
                }
                pos++;
                compilador.add("istore "+pos);
                return memory.get(id);
            }else{
                errors++;
                System.out.println("Linea"+linea+": "+ctx.ID().getText()+" no esta declarada");
                return null;
            }
        }catch (Exception e){
            System.out.println("Linea"+linea+": "+e.getMessage()+" un error en la asignacion de: "+id);
            return null;
        }
    }
}

```

Figura 36: visitAsignacion

Si hubo un error en la declaracion, la asignacion no se puede cumplir ni evaluar. Si todo salio bien se asigna a la variable previamente almacenada un valor.

jasmin: Para jasmin en esta visita se agrega `istore < pos >` a la lista de instrucciones

```

@Override
public Object visitId(OpmezParser.IdContext ctx) {
    try{
        String id = ctx.ID().getText();
        if(!joinIfElse){
            if(tempMemory.containsKey(id)){
                return tempMemory.get(id);
            }else if(memory.containsKey(id)){
                int pos=0;
                for (String key: memory.keySet()) {
                    if(key.equals(id)) break;
                    pos++;
                }
                compilador.add("load "+pos);
                return memory.get(id);
            }else{
                errors++;
                System.out.println("Linea"+line+": "+ctx.ID().getText()+" no esta definida");
                return null;
            }
        }else if(memory.containsKey(id)) {
            int pos=0;
            for (String key: memory.keySet()) {
                if(key.equals(id)) break;
                pos++;
            }
            compilador.add("load "+pos);
            globalPos=String.valueOf(pos);
            return memory.get(id);
        }else{
            errors++;
            System.out.println("Linea"+line+": "+ctx.ID().getText()+" no esta definida");
            return null;
        }
    }catch (Exception e){
        System.out.println(e);
        return null;
    }
}

```

Figura 37: visitId

Este visit verifica que la variable que visita este en memoria y a su vez tenga un valor **jasmin**: Para **jasmin** en esta visita se agrega `iload < pos >` a la lista de instrucciones

```

@Override
public Object visitNumero(OpmezParser.NumeroContext ctx) {
    compilador.add("bipush "+ctx.getText());
    return Integer.valueOf(ctx.getText());
}

```

Figura 38: visitNumero

Visita la expresion que coincide con un numero **jasmin**: Para **jasmin** en esta visita se agrega `bipush < value >` a la lista de instrucciones

```

@Override
public Object visitMultDiv(OpmezParser.MultDivContext ctx) {
    int left = (int) visit(ctx.expr(0));
    int right = (int) visit(ctx.expr(1));
    compilador.add(ctx.op.getType() == OpmezParser.MULT?"imul":"idiv");
    return (ctx.op.getType() == OpmezParser.MULT)? left * right: left / right;
}

@Override
public Object visitSumSub(OpmezParser.SumSubContext ctx) {
    int left = (int) visit(ctx.expr(0));
    int right = (int) visit(ctx.expr(1));
    compilador.add(ctx.op.getType() == OpmezParser.SUM?"iadd":"isub");
    return (ctx.op.getType() == OpmezParser.SUM)? left + right: left - right;
}

```

Figura 39: Operaciones

Las operaciones son validadas desde la gramatica, así que su visita solo es para agregar las instrucciones de **jasmin**, las etiquetas varían según el operador que contenga la operación.

```

@Override
public Object visitImpresionExpr(OpmezParser.ImpresionExprContext ctx) {
    compilador.add("getStatic java/lang/System/out java/io/PrintStream;");
    visit(ctx.expr());
    compilador.add("invokeVirtual java/io/PrintStream/println(I)V");
    return null;
}

@Override
public Object visitCadenaTexto(OpmezParser.CadenaTextoContext ctx) {
    compilador.add("ldc V"+ctx.STRING().getText().replace("oldChar", "newChar").trim()+"\n");
    return ctx.STRING().getText();
}

@Override
public Object visitImpresionString(OpmezParser.ImpresionStringContext ctx) {
    compilador.add("getStatic java/lang/System/out java/io/PrintStream;");
    visit(ctx.string());
    compilador.add("invokeVirtual java/io/PrintStream/print(Ljava/lang/String;)V");
    return null;
}

```

Figura 40: visit Impresiones

Se visita la expr y el string para cada caso, segun sea su sintaxis, estos visit solo sirven de control para el verificador de codigo Opmez.

jasmin: Para jasmin en estas visita se agrega las lineas de comando para imprimir, para el caso de impresionExpr se imprime un Entero y para impresionString una cadena de texto

```

@Override
public Object visitIfElse(OpmezParser.IfElseContext ctx) {
    joinIfElse = true;
    try{
        visit(ctx.if_sentence());
        if(ctx.else_sentence()!=null){
            visit(ctx.else_sentence());
        }else if(ctx.elif_sentence()!=null){
            visit(ctx.elif_sentence());
        }
    }catch (Exception e){
    }
    compilador.add("END:");
    return null;
}

@Override
public Object visitSentenciaIf(OpmezParser.SentenciaIfContext ctx) {
    line++;
    Object result = null;
    label = "IFBODY";
    try{
        visit(ctx.condition());
        compilador.add("goto ELSEBODY");
        compilador.add("IFBODY:");
        result=visit(ctx.body());
        compilador.add("goto END");
    }catch(Exception e){
        errors++;
        ps.println("Algo fallo en: if("+ctx.condition().getText()+")");
    }
    tempMemory.clear();
    joinIfElse=false;
    return result;
}

```

Figura 41: if

Nuevamente las visitas son solo de control y verificar que todo este como deberia, ya que quien controla el error de sintaxis es la gramatica.

jasmin: Para jasmin en estas visitas se agregan las instrucciones con sus etiquetas que conforman un if en jasmin, se hace en estas visitas porque aqui se visita al body de if

```

@Override
public Object visitSentenciaElse(OpmezParser.SentenciaElseContext ctx) {
    line++;
    Object result = null;
    joinIfElse = true;
    try{
        compilador.add("ELSEBODY:");
        result=visit(ctx.body());
        compilador.add("goto END");
    }catch(Exception e){
        errors++;
        ps.println("Algo fallo en: else");
    }
    joinIfElse=false;
    tempMemory.clear();
    return result;
}

```

Figura 42: else

jasmin: Para jasmin en estas visitas se agregan las instrucciones con sus etiquetas que conforman un else en jasmin, se hace en estas visitas porque aqui se visita al body de else

```

@Override
public Object visitCicle(OpmezParser.CicleContext ctx) {
    line++;
    joinIfElse=true;
    Object result = null;
    label="WHILEBODY"+numLabel;

    try{
        visit(ctx.condition());
        compilador.add("goto DONE");
        compilador.add("WHILEBODY"+numLabel+":");
        result=visit(ctx.body());
        visit(ctx.condition());

        compilador.add("DONE:");
        numLabel++;
    }catch(Exception e){
        errors++;
        ps.println("Algo fallo en: while");
    }
    tempMemory.clear();
    joinIfElse=false;
    return result;
}

```

Figura 43: visitCicle

El ciclo while debe agregar etiquetas para jasmin con cierto numero de etiqueta (numLabel), esto se debe a que al ser un unico bloque de codigo jasmin interpreta de manera independiente a cada while (si es que hay 2 o mas), tambien agrega etiquetas que indican el cuerpo y el final del ciclo

Como podemos notar, la clase **CheckOpmez** mayormente hace una verificacion de sintaxis a cada instruccion, cuando ocurre un error al visitar que esto quiere decir que no esta bien escrito o en el caso de las declaraciones, hay algun error en estas o en las asignaciones.

8. Creacion de una clase Visitor para la gramatica Opmez para ejecutar las instrucciones de entrada y mostrar el resultado obtenido

Una vez que la validacion del codigo haya pasado y todo este correctamente procedemos a ejecutar el codigo con ayuda de un visitor que ahora si considere todos los resultados de cada visit.

```
@Override
public Object visitDeclaracion(OpmezParser.DeclaracionContext ctx) {
    String id = ctx.ID().getText();

    if(joinIfElseWhile || bodyInScope){
        if(!memory.containsKey(id) && tempMemory.containsKey(id)){
            tempMemory.put(id,null);
        }
    }else if(!memory.containsKey(id)){
        memory.put(id,null);
    }
    return null;
}

@Override
public Object visitAsignacion(OpmezParser.AsignacionContext ctx) {
    try{
        String id = ctx.ID().getText();
        Object value = visit(ctx.expr());
        if(joinIfElseWhile || bodyInScope){
            if(!memory.containsKey(id)){
                tempMemory.put(id, value);
                return tempMemory.get(id);
            }else{
                memory.put(id,value);
                return memory.get(id);
            }
        }else if(memory.containsKey(id)) {
            memory.put(id, value);
            return memory.get(id);
        }
    }catch (Exception e){
    }
    return null;
}
```

Figura 44: visit Declaracion/Asignacion

visitDeclaracion: Al igual que el validador, agrega en memoria las variables declaradas
visitAsignacion: le asigna un valor a esas variables declaradas y devuelve su valor para ser usado

```

@Override
public Object visitNumero(OpmezParser.NumeroContext ctx) {
    return Integer.valueOf(ctx.getText());
}

@Override
public Object visitId(OpmezParser.IdContext ctx) {
    try{
        String id = ctx.ID().getText();
        if(joinIfElseWhile || bodyInScope){
            if(!memory.containsKey(id)){
                return tempMemory.get(id);
            }else{
                return memory.get(id);
            }
        }else if(memory.containsKey(id)) {
            return memory.get(id);
        }else{
            return null;
        }
    }catch (Exception e){
        System.out.println(e);
        return null;
    }
}

```

Figura 45: visit Numero/Id

visitNumero: devuelve el valor de su contexto como numero entero

visitId: obtiene el valor del Id (variable) declarado y asignado

```

public Object visitImpresionExpr(OpmezParser.ImpresionExprContext ctx) {
    Object result = visit(ctx.expr());
    System.out.println(result);
    return null;
}

@Override
public Object visitCadenaTexto(OpmezParser.CadenaTextoContext ctx) {
    return ctx.STRING().getText().replace( oldChar '"', newChar ' ').trim();
}

@Override
public Object visitImpresionString(OpmezParser.ImpresionStringContext ctx) {
    System.out.println(visit(ctx.string()));
    return null;
}

```

Figura 46: visit Impresiones

visitImpresionExpr: se imprime directamente el valor de la visita a expr

visitImpresionString: imprime la cadena pero en la visita a string se remueven las comillas para que solo se imprima el texto

```

@Override
public Object visitMultDiv(OpmezParser.MultDivContext ctx) {
    int left = (int) visit(ctx.expr(0));
    int right = (int) visit(ctx.expr(1));
    return (ctx.op.getType() == OpmezParser.MULT)? left * right: left / right;
}

@Override
public Object visitSumSub(OpmezParser.SumSubContext ctx) {
    int left = (int) visit(ctx.expr(0));
    int right = (int) visit(ctx.expr(1));
    return (ctx.op.getType() == OpmezParser.SUM)? left + right: left - right;
}

@Override
public Object visitParentesis(OpmezParser.ParentesisContext ctx) {
    return visit(ctx.expr());
}

```

Figura 47: visit Operaciones

visitMultDiv: devuele el resultado de las operaciones multiplicacion o division entre 2 expr

visitSumSub:devuele el resultado de las operaciones suma o resta entre 2 expr

visitParentesis: devuelve el valor de la visita a la expr que contiene en el contexto


```

@Override
public Object visitIfElse(OpmezParser.IfElseContext ctx) {
    joinIfElseWhile = true;
    try{
        boolean result= (boolean) visit(ctx.if_sentence());
        if(!result){
            if(ctx.else_sentence()!=null){
                visit(ctx.else_sentence());
            }else if(ctx.elif_sentence()!=null){
                visit(ctx.elif_sentence());
            }
        }
    }catch (Exception e){
    }
    return null;
}

@Override
public Object visitSentenciaIf(OpmezParser.SentenciaIfContext ctx) {
    joinIfElseWhile = true;
    try{
        boolean result = (boolean)visit(ctx.condition());
        if(result){
            visit(ctx.body());
        }
        tempMemory.clear();
        joinIfElseWhile =false;
        bodyInScope=false;
        return result;
    }catch(Exception e){
        System.out.println("Algo fallo en: if("+ctx.condition().getText()+")");
        return null;
    }
}

@Override
public Object visitSentenciaElse(OpmezParser.SentenciaElseContext ctx) {
    joinIfElseWhile = true;
    visit(ctx.body());
    tempMemory.clear();
    joinIfElseWhile =false;
    bodyInScope=false;
    return null;
}

```

Figura 48: If-else

visitIfElse: esta visita realiza 3 visitas, pero solo 1 a la vez es la que ejecuta, dependiendo de el valor booleano de la visita a *if_sentence*

visitSentenciaIf: devuelve el valor booleano del resultado de la condicion que a su vez servira para que visitIfElse haga las visitas correspondientes.

visitSentenciaElse: cuando el resultado de condition es positiva, visita a if, si es negativo se visita else

```

@Override
public Object visitSentenciaElif(OpmezParser.SentenciaElifContext ctx) {
    joinIfElseWhile = true;
    try{
        boolean result= (boolean) visit(ctx.elif_frag_condition());
        if(!result){
            if(ctx.else_sentence()!=null){
                visit(ctx.else_sentence());
            }else if(ctx.elif_sentence()!=null){
                visit(ctx.elif_sentence());
            }
        }
        joinIfElseWhile = false;
        bodyInScope=false;
    }catch (Exception e){
    }
    return null;
}

@Override
public Object visitCondicionElif(OpmezParser.CondicionElifContext ctx) {
    try{
        joinIfElseWhile = true;
        boolean result = (boolean)visit(ctx.condition());
        if(result){
            visit(ctx.body());
        }
        joinIfElseWhile = false;
        tempMemory.clear();
        return result;
    }catch(Exception e){
        fails++;
        System.out.println("Algo fallo en: elif("+ctx.condition().getText()+")");
        return null;
    }
}

```

Figura 49: elif

En realidad la logica detras de un elif es la misma que un if normal, solo que en la gramatica no se permite estar a esta instruccion sola, sino que siempre debe estar despues de un if, y repitiendo, la logica es la misma.

```

public Object visitCicle(OpmezParser.CicleContext ctx) {
    joinIfElseWhile = true;
    try{
        while(true){
            if(!((boolean)visit(ctx.condition()))==false){
                break;
            }
            visit(ctx.body());
        }
        tempMemory.clear();
        joinIfElseWhile = false;
        bodyInScope=false;
        return null;
    }catch(Exception e){
        return null;
    }
}

```

Figura 50: Ciclo while

visitCicle: el funcionamiento de este while internamente es en realidad un while infinito que se detendra solo cuando la visita a condition sea verdadera, mientras ejecutara N veces lo que este en body

```

@Override
public Object visitCondicionesIgualdad(OpmezParser.CondicionesIgualdadContext ctx) {
    boolean left = (boolean) visit(ctx.condition(0));
    boolean right = (boolean) visit(ctx.condition(1));
    return (ctx.op.getType() == OpmezParser.EQ) ? (left == right) : (left != right);
}

@Override
public Object visitCondicionesIgualdadExpr(OpmezParser.CondicionesIgualdadExprContext ctx) {
    int left = (int) visit(ctx.expr(0));
    int right = (int) visit(ctx.expr(1));
    return (ctx.op.getType() == OpmezParser.EQ) ? (left == right) : (left != right);
}

@Override
public Object visitCondicionesMayMen(OpmezParser.CondicionesMayMenContext ctx) {
    int left = (int) visit(ctx.expr(0));
    int right = (int) visit(ctx.expr(1));
    return (ctx.op.getType() == OpmezParser.GT) ? (left > right) : (left < right);
}

@Override
public Object visitCondicionesMayMenIgual(OpmezParser.CondicionesMayMenIgualContext ctx) {
    int left = (int) visit(ctx.expr(0));
    int right = (int) visit(ctx.expr(1));
    return (ctx.op.getType() == OpmezParser.GEQ) ? (left >= right) : (left <= right);
}

@Override
public Object visitConditionNegacion(OpmezParser.ConditionNegacionContext ctx) {
    return !((boolean)visit(ctx.condition()));
}

```

Figura 51: Condiciones pt 1

Complementando la descripción de la figura 31 donde se explica cada condición, internamente se decide que operador lógico es el que evaluara proviene de la propiedad `ctx.op` que almacena las opciones de lexemas y en base a su equivalente del Parser se decide el operador lógico.

```

@Override
public Object visitCondicionV(OpmezParser.CondicionVContext ctx) {
    boolean left = (boolean) visit(ctx.condition(0));
    boolean right = (boolean) visit(ctx.condition(1));
    return (left && right);
}

@Override
public Object visitCondicionO(OpmezParser.CondicionOContext ctx) {
    boolean left = (boolean) visit(ctx.condition(0));
    boolean right = (boolean) visit(ctx.condition(1));
    return (left || right);
}

@Override
public Object visitVerdadero(OpmezParser.VerdaderoContext ctx) {return true;}
@Override
public Object visitFalso(OpmezParser.FalsoContext ctx) {return false;}
@Override
public Object visitCondicionParentesis(OpmezParser.CondicionParentesisContext ctx) {
    return (boolean) visit(ctx.condition());
}

@Override
public Object visitExpresion(OpmezParser.ExpresionContext ctx) {
    if(ctx.expr().getText().equals("1") || ctx.expr().getText().equals("0")){
        return ctx.expr().getText().equals("1")? true : false;
    }else{
        System.out.println("Se esperaba {1, 0, true, false, condition}");
        return null;
    }
}
}

```

Figura 52: Condiciones 2

Para los casos donde el operador logico es directo como and (&&), or (||), true o false no es necesario hacer uso de esta propiedad porque en si no la contiene el lexema y el resultado se evalua con los operadores logicos tipicos.

8.1. Resultados Opmez

Ejecutar codigo Opmez, crear archivo jasmin.

```

1  #initial()->
2  use a;
3  use b;
4  use c;
5  a=10;
6  b=20;
7  c=a+b;
8  systalk(c);
9  if (a==b)->
10 systalk("verdadero");
11 systalk(a);
12 <-else->
13 systalk("falso");
14 systalk(b);
15 <-
16 use contador;
17 contador=0;
18 while (contador<10)->
19 systalk(contador);
20 contador=contador+1;
21 <-
22 <-
23

```

Figura 53: Entrada Opmez

```
1  #initial()->
2  use a;
3  use b;
4  use c;
5  a=10;
6  b=20;
7  c=a+b;
8  systalk(c);
9  if (a==b)->
10 systalk("verdadero");
11 systalk(a);
12 <-else->
13 systalk("falso");
14 systalk(b);
15 <-
16 use contador;
17 contador=0;
18 while(contador<10)->
19 systalk(contador);
20 contador=contador+1;
21 <-
22 <-
23
```

```
30
falso
20
0
1
2
3
4
5
6
7
8
9
```

Figura 54: Resultado ejecucion Opmez

Crear archivo jasmin

Para crear el archivo jasmin es necesario correr el programa Opmez para que agreguen a la lista las instrucciones en jasmin, despues de esto lo podemos generar y por ultimo tenemos que crear el archivo .class que se genera a partir del archivo .j

```
MenuItem item_jasmin = new MenuItem(s: "Build jasmin");
item_jasmin.setOnAction(actionEvent -> {
    if(opmezCreado){
        text_Output.appendText(s: "Generate jasmin file\n");
        String file_in = new File ( pathname: "code.j").getAbsolutePath();
        File file = new File(file_in);
        if(!file.exists()){
            try {
                file.createNewFile();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        FileWriter fw = null;
        try {
            fw = new FileWriter(file_in, append: false);
        } catch (IOException e) {
            e.printStackTrace();
        }

        for (String line: compilador) {
            try {
                fw.write(str: line+"\n");
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        compilador.clear();
        try {
            fw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        compilador.clear();
        jasminCreado=true;
    }
}
```

Figura 55: Como se genera el .jasmin

```
MenuItem item_class = new MenuItem(s: "Build class");
item_class.setOnAction(actionEvent -> {
    if(jasminCreado){
        ProcessBuilder builder = new ProcessBuilder(
            ...command: "cmd.exe", "/c", "cd "+new File( pathname: "").getAbsolutePath()+
            "%& java -jar jasmin.jar code.j" +
            "%& java Codigo");
        builder.redirectErrorStream(true);
        Process p = null;
        try {
            p = builder.start();
        } catch (IOException e) {
            e.printStackTrace();
        }
        BufferedReader r = new BufferedReader(new InputStreamReader(p.getInputStream()));
        String line;
        while (true) {
            try {
                line = r.readLine();
                if (line == null) { break; }
                System.out.println(line);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Figura 56: Como se crear el .class

```
1  #initial()->
2  use a;
3  use b;
4  use c;
5  a=10;
6  b=20;
7  c=a+b;
8  systalk(c);
9  if (a==b)->
10 systalk("verdadero");
11 systalk(a);
12 <-else->
13 systalk("falso");
14 systalk(b);
15 <-
16 use contador;
17 contador=0;
18 while(contador<10)->
19 systalk(contador);
20 contador=contador+1;
21 <-
22 <-
23
```

```
Generate jasmin file
Generated:Codigo.class
30
falso
20
0
1
2
3
4
5
6
7
8
```

Figura 57: Crear archivo .j y .class y mostrar los resultados

9. Pruebas completas generales

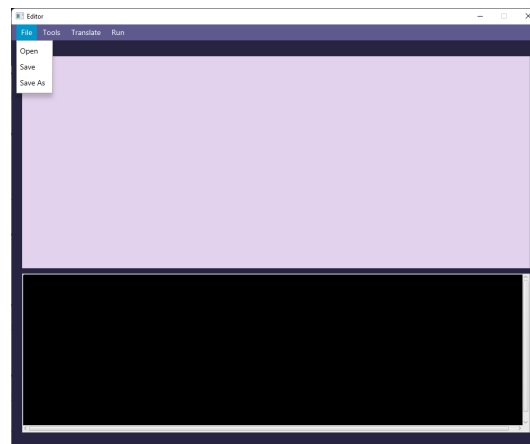


Figura 58: Abrir explorador

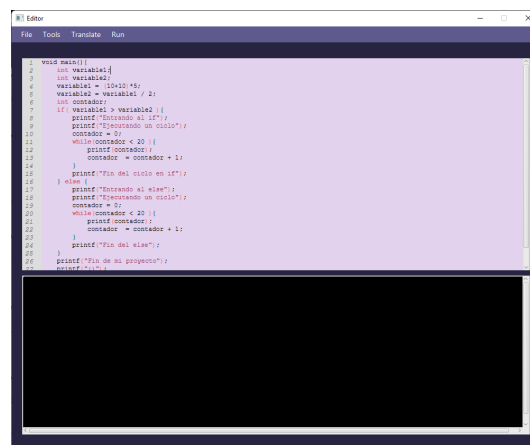


Figura 59: Seleccionar archivo C

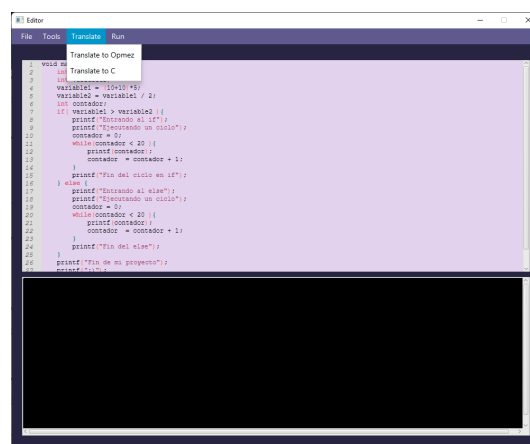
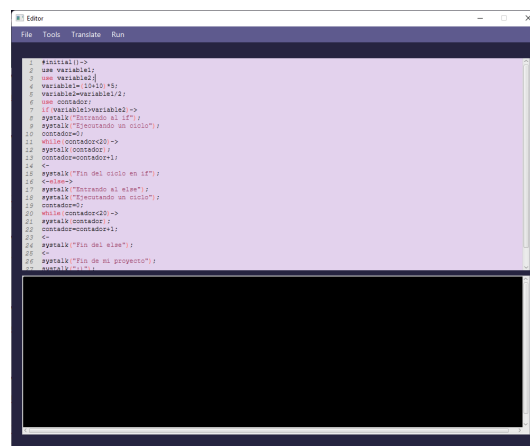


Figura 60: Traducir a Opmez

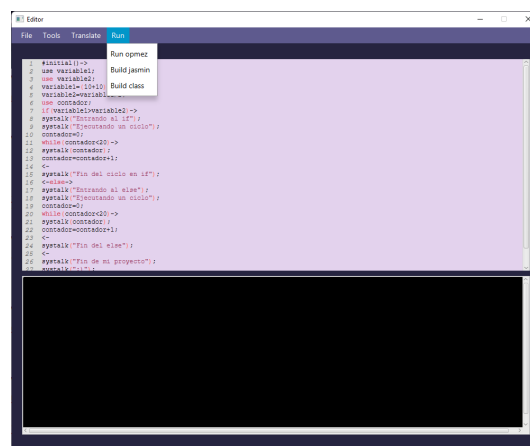


```

1 #Inicializa->
2 use variable1;
3 use variable2;
4 variable1=10;
5 variable2=variable1;
6 use contador;
7 if variable1>variable2->
8   pyrtalk("Entrando al if");
9   pyrtalk("Ejecutando un ciclo");
10  contador=0;
11  while contador<20->
12    pyrtalk(contador);
13    contador=contador+1;
14  <-fin-
15  pyrtalk("Fin del ciclo en if");
16  <-fin-
17  pyrtalk("Entrando al else");
18  pyrtalk("Ejecutando un ciclo");
19  contador=0;
20  while contador<20->
21    pyrtalk(contador);
22    contador=contador+1;
23  <-
24  pyrtalk("Fin del else");
25  <-
26  pyrtalk("Fin de mi proyecto");
27  contador=0;

```

Figura 61: Resultado de traduccion

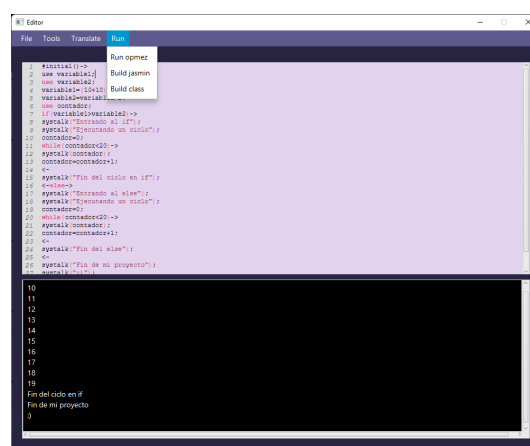


```

1 #Inicializa->
2 use variable1;
3 use variable2;
4 variable1=10;
5 variable2=variable1;
6 use contador;
7 if variable1>variable2->
8   pyrtalk("Entrando al if");
9   pyrtalk("Ejecutando un ciclo");
10  contador=0;
11  while contador<20->
12    pyrtalk(contador);
13    contador=contador+1;
14  <-
15  pyrtalk("Fin del ciclo en if");
16  <-fin-
17  pyrtalk("Entrando al else");
18  pyrtalk("Ejecutando un ciclo");
19  contador=0;
20  while contador<20->
21    pyrtalk(contador);
22    contador=contador+1;
23  <-
24  pyrtalk("Fin del else");
25  <-
26  pyrtalk("Fin de mi proyecto");
27  contador=0;

```

Figura 62: Ejecutar opmez



```

10
11
12
13
14
15
16
17
18
19
Fin del ciclo en if
Fin de mi proyecto
3

```

Figura 63: Crear archivo jasmin


```
.class public Codigo
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack 20
.limit locals 20
bipush 10
bipush 10
iadd
bipush 5
imul
istore 0
iload 0
bipush 2
idiv
istore 1
iload 0
iload 1
if_icmpgt IFBODY
goto ELSEBODY
IFBODY:
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Entrando al if
"
invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Ejecutando un ciclo
"
invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
bipush 0
istore 2
iload 2
bipush 20
if_icmplt WHILEBODY0
goto DONE
WHILEBODY0:
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 2
invokevirtual java/io/PrintStream/println(I)V
```

Figura 64: Archivo generado (.j)

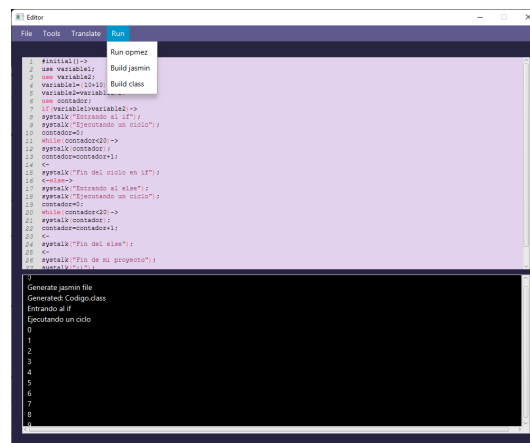


Figura 65: Crear archivo .class

```
j)
Generate jasmin file
Generated:Codigo.class
Entrando al if
Ejecutando un ciclo
0
1
2
3
4
5
6
7
8
9
```

Figura 66: Resultado del archivo .class

10. CONCLUSION

La creacion de un lenguaje otorga una percepcion del desarrollo de software y de la programacion en general mas abstracta. El hecho de saber como se conforma un lenguaje, partiendo desde una gramatica para identificar el lenguaje hasta la parte logica que se encarga de ejecutar las operaciones aritmeticas y logicas proporciona al programador una habilidad de adecuarse mas facilmente a otros lenguajes ya que conoce la estructura de los mismos desde el nucle.

Personalmente me parece un ejercicio muy util ya que complementa perfectamente el pensamiento logico en uno ya que se aprenden cosas mas complejas que muy dificilmente aprendamos en cualquier otro campo de la programacion.