

LENGUAJES Y AUTÓMATAS II

ME XENIA PADILLA MADRID



ACTIVIDAD DE LABORATORIO	10
NOMBRE DE LA ACTIVIDAD	Sentencia IF-ELSE
OBJETIVO DE APRENDIZAJE	
INSTRUCCIONES	
NOMBRE ESTUDIANTE	Itsai Zempoaltecatl
GRUPO	6SA
FECHA	22/marzo/2022

TRADUCIR:

Gramática:

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

condition alberga las diferentes condiciones que contiene un if, se manda a llamar a si misma ya que una condición puede compararse con otra condición

```
if_sentence: IF PO condition PC KO body* KC #sentenceIf;
else_sentence: ELSE KO body* KC #sentenciaElse;
```

Ahora tenemos los tokens if_sentence y else_sentence, estos verifican que el if contenga todas las partes que lo conforman, así mismo con else. Son tokens separados porque hay que acceder al body de cada uno de ellos ya que son independientes

```
body:
  |
  if_sentence else_sentence? #ifElse
  |
```

Por último, agregamos los dos tokens creados en el body, pero ya else puede o no estar después del if, le agregamos el operador '?'

Nota: en cada lexema de cada token encontramos body* y dentro de body tenemos if-else, esto es para poder poner un if dentro de otro ya que es valido

LENGUAJES Y AUTÓMATAS II

ME XENIA PADILLA MADRID

Visitor:

```
@Override
public String visitIfElse(LenguajeParser.IfElseContext ctx) {
    String lineIf = visit(ctx.if_sentence());
    String lineElse="";
    if(ctx.else_sentence()!=null){
        lineElse=visit(ctx.else_sentence());
    }
    return lineIf + lineElse ;
}
```

En nuestra clase visitor primero visitamos a IfElse, que se conforma de “if_sentence else_sentence?”, primero visitamos a if_sentence, ya que esta línea es necesaria que exista, puede que no exista un else, para eso verificamos que else_sentence no venga nulo, si contiene algo lo visitamos y regresamos el valor de las 2 visitas

```
@Override
public String visitSentenceIf(LenguajeParser.SentenceIfContext ctx) {
    StringBuilder sb = new StringBuilder();
    for(int i=0; i<ctx.body().size(); i++){
        sb.append(visit(ctx.body(i))+"\n");
    }
    return "if("+visit(ctx.condition())+"->\n"+sb + "<- ";
}

@Override
public String visitSentenciaElse(LenguajeParser.SentenciaElseContext ctx) {
    StringBuilder sb = new StringBuilder();
    for(int i=0; i<ctx.body().size(); i++){
        sb.append(visit(ctx.body(i))+"\n");
    }
    return "else->\n"+sb+"<- ";
}
```

Las visitas a cada sentencia (if-else): en el caso de SentenciaIf tenemos que visitar 2 cosas, condition y body, body es una lista así que visitamos cada uno de sus hijos. Al final devuelve el valor de las 2 visitas con el formato de mi lenguaje (Opmez)

LENGUAJES Y AUTÓMATAS II

ME XENIA PADILLA MADRID

```
@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()); }
```

La visita a cada opción de condition: en este caso solo devuelvo el texto del contexto ya que mi gramática determina si la entrada esta bien escrita o no y como los operadores lógicos en si no se modifican, se puede quedar así como viene la entrada. Solo Expression visita a su hijo, porque if puede contener alguna de las variantes de expr (ID, NUM, SUMA, etc) (if(1){} por ejemplo)

Nota: Todo devuelve un string, así puedo devolver el contexto como viene sin necesidad de visitar a cada hijo y desde mi visitIf es donde voy agregándolos al nuevo “body”

Extra:

```
expr:
    ...
    |
    SUB? NUM #num
    .

@Override
public String visitNum(LenguajeParser.NumContext ctx) { return ctx.getText(); }
```

Ya que los enteros cuentan con números positivos y negativos, modifique expr, para que NUM pueda o no estar acompañado de ‘-’ y en el visitor devuelvo el todo el texto del contexto, asi si hay un ‘-’ igual vendría acompañado del numero

LENGUAJES Y AUTÓMATAS II

ME XENIA PADILLA MADRID



EJECUTAR:

Gramática:

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

Para la gramática de mi lenguaje hice unas modificaciones a condition, con esto controlo si esta comparando condiciones o expresiones, por ejemplo, para la igualdad puede comparar $(1==1) == (0==0)$ o $(2==2)$ y son diferentes casos, en uno compara booleanos y en otro solo números, también para el caso de (mayor/menor)-igual no se pueden comparar resultados booleanos, solo valores numéricos, por esto use expr en lugar de condition.

```
if_sentence: IF PO condition PC KO body* KC #sentenciaIf;
else_sentence: ELSE KO body* KC #sentenciaElse;

body:
  |
  if_sentence else_sentence? #ifElse
```

Por otra parte, if_sentence y else_sentence quedan iguales (solo que los tokens como KO/KC son de mi lenguaje).

Igualmente, en body mantienen el mismo formato donde puede o no haber un else después de if.

LENGUAJES Y AUTÓMATAS II

ME XENIA PADILLA MADRID

Visitor:

```
@Override
public Object visitIfElse(OpmezParser.IfElseContext ctx) {
    try{
        boolean result= (boolean) visit(ctx.if_sentence());
        if(!result){
            if(ctx.else_sentence()!=null){
                visit(ctx.else_sentence());
            }
        }
    }catch (Exception e){
        ps.println("Error: "+e);
    }
    return null;
}
```

Como primer punto a considerar: cada metodo heredado de mi visitor devuelve un Object, lo que me permite devolver el tipo de dato que yo quiera.

La primer visita es a IfElse, como condition al evaluar devolverá un booleano, este mismo me permite que en la visita a if_sentence me devuelva un booleano, usando esto verifico si la condición entrada en el if, sino, verifico que exista un else, si es así visito else_sentence

```
@Override
public Object visitSentenciaIf(OpmezParser.SentenciaIfContext ctx) {
    boolean result = (boolean)visit(ctx.condition());
    if(result){
        for (int i = 0; i < ctx.body().size(); i++) {
            visit(ctx.body(i));
        }
        return true;
    }else{
        return false;
    }
}

@Override
public Object visitSentenciaElse(OpmezParser.SentenciaElseContext ctx) {
    for (int i = 0; i < ctx.body().size(); i++) {
        visit(ctx.body(i));
    }
    return null;
}
```

Retomando lo que explique en el punto anterior, en la visita a sentenciaIF visito condition que me devuelve un true o false, este lo evaluó, si entra al if visito todos sus body hijos y devuelvo el mismo valor de result, ya que es el mismo valor para el visitIfElse.

En else únicamente visito los hijos body, ya que este no debe evaluar nada ni devolver algún valor, por eso es return null.

LENGUAJES Y AUTÓMATAS II

ME XENIA PADILLA MADRID



```
@Override
public Object visitCondicionesIgualdad(OpmezParser.CondicionesIgualdadContext ctx) {
    boolean left = (boolean) visit(ctx.condition(0));
    boolean right = (boolean) visit(ctx.condition(1));
    if(ctx.op.getType() == OpmezParser.EQT){
        return (left == right);
    }else{
        return (left != right);
    }
}

@Override
public Object visitCondicionesIgualdadExpr(OpmezParser.CondicionesIgualdadExprContext ctx) {
    int left = (int) visit(ctx.expr(0));
    int right = (int) visit(ctx.expr(1));
    if(ctx.op.getType() == OpmezParser.EQT){
        return (left == right);
    }else{
        return (left != right);
    }
}
```

Existen 2 variantes de igualdad, en Igualdad tomo izquierda y derecha como tipos boolean, después en base al operador lógico los comparo.

IgualdadExpr tiene la misma lógica, solo que izquierda y derecha ahora son tomados como integer.

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

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

Las condiciones Mayor-Menor y Mayor-Menor igual cuentan con una lógica muy similar, los 2 lados son tomados como integer, la única variante en estos es su operador lógico matemático, esto sí afecta en el valor que devuelve.

LENGUAJES Y AUTÓMATAS II

ME XENIA PADILLA MADRID



```
@Override
public Object visitCondicionY(OpmezParser.CondicionYContext 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 visitCondicionNegacion(OpmezParser.CondicionNegacionContext ctx) {
    return !(boolean)visit(ctx.condition());
}
```

Siguen los operadores lógicos básicos (o no matemáticos) AND, OR, NOT. Como se puede apreciar, únicamente se visita a condition izquierda y derecha, tanto en Y como en O, devolviendo su valor después de evaluarlos con el operador correspondiente.

Para NO solo devuelve la negación del valor de la visita a condition.

```
@Override
public Object visitCondicionParentesis(OpmezParser.CondicionParentesisContext ctx) {
    return visit(ctx.condition());
}

@Override
public Object visitVerdadero(OpmezParser.VerdaderoContext ctx) { return true; }

@Override
public Object visitFalso(OpmezParser.FalsoContext ctx) { return false; }

@Override
public Object visitExpresion(OpmezParser.ExpresionContext ctx) {
    if ((int)visit(ctx.expr())==0){
        return false;
    }else if((int)visit(ctx.expr())==1){
        return true;
    }else{
        return visit(ctx.expr());
    }
}
```

Por último tenemos los paréntesis y las condition más básicas, true, false, 1, 0 y numero. Parentesis devuelve la visita a condition, que como ya vimos anteriormente es un boolean (aunque puede ser un numero)

Los visitor de Verdadero y Falso únicamente devuelven el boolean equivalente a su nombre.

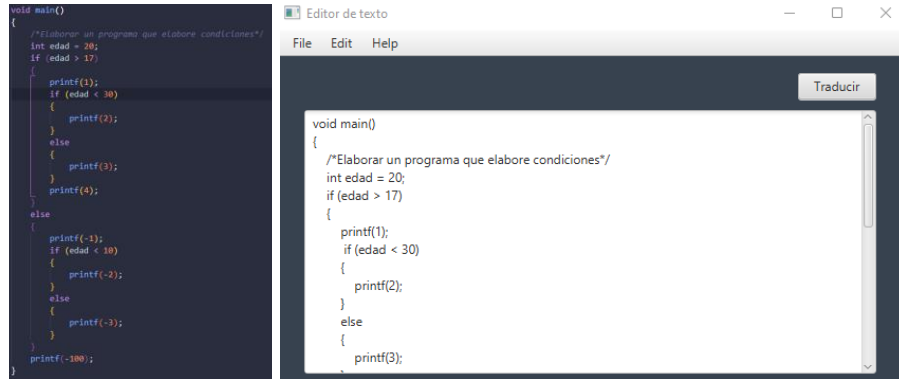
Como antes mencione, la ventaja de que mi visitor sea de tipo Object, es que puede devolver cualquier valor, por ejemplo, en Expresion verifica si es 0, 1 o cualquier otro número, si es 0 devuelve false, si es 1 true y en caso contrario devuelve la visita a expr como tal.

LENGUAJES Y AUTÓMATAS II

ME XENIA PADILLA MADRID

RESULTADOS:

Traducción:



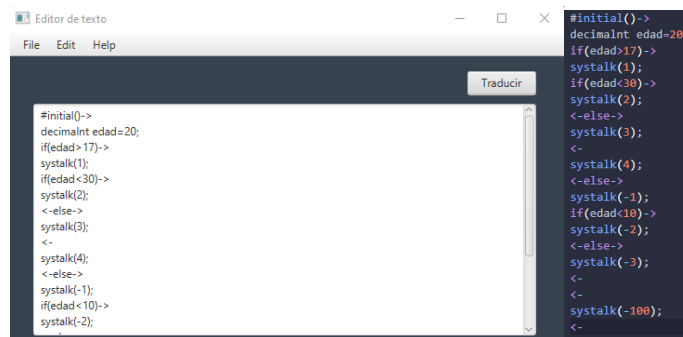
The image shows two windows side-by-side. The left window, titled 'Editor de texto', contains C code:

```
void main()
{
    /*Elaborar un programa que elabore condiciones*/
    int edad = 20;
    if (edad > 17)
    {
        printf(1);
        if (edad < 30)
        {
            printf(2);
        }
        else
        {
            printf(3);
        }
        printf(4);
    }
    else
    {
        printf(-3);
        if (edad < 10)
        {
            printf(-2);
        }
        else
        {
            printf(-3);
        }
    }
    printf(-100);
}
```

 The right window, also titled 'Editor de texto', shows the translated code:

```
void main()
{
    /*Elaborar un programa que elabore condiciones*/
    int edad = 20;
    if (edad > 17)
    {
        printf(1);
        if (edad < 30)
        {
            printf(2);
        }
    }
    else
    {
        printf(3);
    }
}
```

Entrada

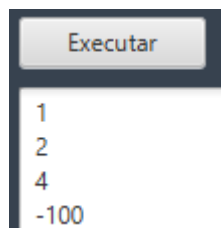


The image shows a single window titled 'Editor de texto' containing the translated code with comments:

```
#initial()->
decimalInt edad=20;
if(edad>17)->
systalk(1);
if(edad<30)->
systalk(2);
<-else->
systalk(3);
<-
systalk(4);
<-else->
systalk(-1);
if(edad<10)->
systalk(-2);
<-else->
systalk(-3);
<-
systalk(-100);
<-
```

Salida

Ejecución:



The image shows a terminal window with a button labeled 'Ejecutar' at the top. Below the button, the output of the program is displayed:

```
1
2
4
-100
```

Resultado

CONCLUSIÓN: es importante analizar los tipos de datos que puede manejar cada una de nuestras sentencias en nuestro código, esto nos podría ahorrar tiempo al momento de realizar las visitas.

El uso de Object igual ayuda demasiado a tener un código mas limpio con verificaciones sencillas, si hubiera seguido con Integer las verificaciones hubieran sido muy frustrantes.