DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Teoría de lenguajes

Trabajo práctico

Parser

Resumen

Este trabajo consiste en parser estilo c++

Integrante	LU	Correo electrónico
Acosta, Javier Sebastian	338/11	acostajavier.ajs@gmail.com
Mastropasqua Nicolas Ezequiel	828/13	mastropasqua.nicolas@gmail.com
Negri, Franco	893/13	franconegri2004@hotmail.com

Palabras claves:

TP

Contents

El objetivo del presente trabajo práctico es la realización de un analizador léxico y sintáctico para un lenguaje de scripting, mediante el uso de una gramática LALR y la herramienta para generar analizadores sintácticos **PLY**.

Esperamos que al final del trabajo el analizador sea capaz de reconocer cadenas de caracteres similares a las del lenguaje C++.

Para ello comenzaremos generando una gramática que cumpla con las reglas del lenguaje. Luego procederemos a implementar esto y a desarrollar un chequeo de tipos que permita filtrar programas que si bien son sintácticamente validos, posean una semántica invalida (por ejemplo, multiplicar una cadena de caracteres con otra).

Además, dado un texto valido querremos guardarlo en un archivo nuevo indentandolo de manera correcta automática, agregando o quitando espacios, tabs, saltos de linea donde sea necesario.

1 Descripción General

1.1 Gramática inicial

Comenzamos el armado de la gramática definiendo los tipos básicos que debíamos aceptar en nuestro lenguaje.
Así definimos los tipos Bool, String, Float, Bool, Int, Vector, Registro. De allí el siguiente paso fue ver que operaciones se le podía realizar a cada uno de ellos.
Mientras realizábamos esto empezamos a notar que necesitaríamos determinar la precedencia de estas operaciones y la asociatividad. Para esto nos basamos en la precedencia de c++ [1] dado lo parecido que es al lenguaje de entrada.

Una vez que todas las expresiones del lenguaje estaban completas, lo pasamos a código Python usando la herramienta *PLY*. Ya terminado el código, nos encontramos con que la gramática tenía muchos mas conflictos que los que se podía ver a simple vista y por lo tanto no era LALR.

1.2 Gramática ambigua

Luego de resolver algunos conflictos, nos dimos cuenta de que era un proceso complicado pues cada conflicto implicaba añadir mas reglas a la gramática haciendo que sea mas compleja y difícil de entender.

Después de leer la documentación de ply[2] y observar que ply puede encargarse de los conflictos de una gramática ambigua usando reglas de asociatividad y precedencia, decidimos hacer una gramática ambigua sencilla y aprovechar esta característica de ply.

1.3 Gramática no ambigua

Si bien esto solucionó todos los conflictos de una manera sencilla, por desgracia, nos comunicaron que lo que se pedía era una gramática **no** ambigua. Por lo tanto decidimos tirar la gramática y comenzar de nuevo.

Con el tiempo contado y sin ideas mejores nos dimos cuenta de que la gramática inicial no estaba tan mal, tenía toda la precedencia definida de las expresiones y que una solución rápida era hacer la gramática menos restrictiva, es decir, que acepte mas cadenas de lo que se pida y rechazar las que no pertenecen al lenguaje mediante atributos. La tarea de los atributos (además

[1] http://en.cppreference.com/w/cpp/language/operator_precedence
[2] http://www.dabeaz.com/ply/ply.html

de armar el código indentado) es chequear los tipos de las expresiones para que, si una operación acepta unos tipos determinados y se le pasa una expresión de otro tipo, lance una excepción de error de tipos. Dicha gramática se encuentra definida en la sección **5**

2 Descripción de la gramática implementada:

2.1 Valores

Los valores son cualquier expresión que denote un valor (por ejemplo 3 * 4 + 3). La primer idea era definir valores como **cualquier** expresión (para poder usarla como valor en asignaciones, en funciones como parámetro, etc.):

$Valores \rightarrow ExpresionMatematica \mid ExpresionBooleana \mid ExpresionString \mid ExpresionVector \mid ExpresionRegistro \mid OperacionVariables$

Donde cada producción que empieza con *Expresion* denota una expresión de un tipo determinado que también puede generar los valores primitivos (bool, string, float, bool, int, vec, reg), variables (id) y además funciones que retornen valores a su tipo correspondiente (por ejemplo *capitalizar* en *Expresionstring*). $OperacionVariables$ genera las expresiones que usan operaciones de variables (asignación, incremento, decremento, etc).

El problema con esto es que **id** se puede generar en todas las expresiones salvo en la última. Si se tiene una producción $A \rightarrow Valores$, entonces se puede llegar de 5 maneras distintas a **id** (es decir, hay mas de una posible derivación mas a la izquierda para la cadena *id*) y, por lo tanto, hay conflictos (la gramática es ambigua).

Por lo tanto decidimos sacar a **id** de cada producción. El problema es que se tiene que seguir produciendo expresiones complejas con **id** (por ejemplo *id and id*).

Para solucionar esto, decidimos sacar todos los valores primitivos de cada expresión y colocarlo en $Valores$. Además hay que observar que una función de un determinado tipo puede devolver una variable de ese tipo, por lo que decidimos sacar también a esas funciones. Lo mismo pasa con los operadores ternarios. De esta forma, cada producción *Expresion* genera expresiones con al menos un operador de su tipo:

$Valores \rightarrow ExpresionMatematica \mid ExpresionBooleana \mid ExpresionString \mid ExpresionVector \mid$
$\mid ExpresionRegistro \mid OperacionVariables \mid bool \mid string \mid float \mid bool \mid int \mid vec \mid reg$
$\mid id \mid FuncReturn \mid Ternario$

$FuncReturn \rightarrow FuncInt \mid FuncString \mid FuncBool \mid FuncVector$

$Ternario \rightarrow TernarioMat \mid TernarioBool \mid TernarioString \mid TernarioVector \mid TernarioRegistro$

2.2 Expresiones Matemáticas

Para las expresiones matemáticas consideramos la siguiente tabla de precedencia:

La producción que se encarga de realizar estas expresiones es eMat.

La primera aproximación fue hacer la gramática para poder respetar la precedencia y asociatividad de los operadores.

Este proceso se logra teniendo en cuenta que se puede fijar la precedencia (por ejemplo entre $+$ y $*$), con las reglas de la gramática. Como bien se sabe, el operador $+$ tiene menos precedencia que el $*$. Una gramática que respeta esto es la siguiente:

$$A \rightarrow A + B$$

Tipo	Operador	Asociatividad
Binario	+,-	izquierda
Binario	*,/,%	izquierda
Binario	^	izquierda
Unario	+,-	
Unario	()	

Table 1: Tabla de menor a mayor precedencia

$$B \to B * C$$
$$C \to int | float$$

Esto se debe a que antes de calcular la suma entre A y B, hay que parsear la producción B (pues es un parser ascendente).

Esto también respeta la asociatividad de ambos operados (que es a izquierda) por la misma razón que antes: La única forma de colocar mas de un operador + es usando la producción A y, por lo tanto, se tiene que parsear A antes que calcular la suma.

Este proceso se realiza con todos los operadores, de manera que si 2 operadores tienen la misma precedencia, entonces parten de un mismo no terminal.

A esta gramática le falta la posibilidad de agregar paréntesis para poder cambiar la precedencia. Una forma de hacerlos es agregarlos a C para que estos tengan mas prioridad que todas las operaciones.

Además también falta poder agregar variables. Para esto agregamos un **no-terminal** a C llamado id que las representa. La gramática resultante es esta:

$$A \to A + B$$
$$B \to B * C$$
$$C \to (A) | int | float | id$$

Para evitar problemas de ambigüedad, decidimos hacer que eMat devuelva expresiones con al menos un operador matemático. Esto es por que en la producción valores se tienen las variables (correspondientes al token ID) y estas también se generarían en las expresiones de los otros tipos. Ver explicación en **4.1**

2.3 Expresiones booleanas

Las expresiones booleanas también se soluciona de la misma manera que las expresiones matemáticas. Para mas información ver sección 5.

2.4 Expresiones de string

En principio consideramos la siguiente gramática:

$ExpresionString \to ExpresionString + ExpresionString \mid string$

Esta gramática nos planteó un problema en el parser debido a que comparte un operador con las expresiones matemáticas. El problema viene cuando se tiene una expresión $a + b$ en donde no está claro que tipo le corresponde e introduce una ambigüedad.

Por lo tanto decidimos fusionar a las expresiones de string con las expresiones matemáticas y restringir (mediante chequeo de tipos) que un string no se le puede hacer otra operación que no sea la suma.

2.5 Problema del "Dangling else":

Otro problema que encontramos ya avanzados en el proceso de desarrollo de la gramática fue el problema del "Dangling else" [3] que consiste en que los else opcionales en un if hacen que la gramática sea ambigua.
Por ejemplo:

if cond1 then if cond2 then sentencia1 else sentencia2

La ambigüedad surge de no poder decidir si la sentencia2 se ejecuta como rama alternativo a cond1(es decir, cuando esta condición es falsa), o bien como la rama alternativa del *if* asociado a **cond2**. Esto último es posible ya que el *if* de **cond1** puede prescindir del bloque *else*. Para solucionar esto, construimos una solución basada en el concepto de lineas abiertas o cerradas. La primera, son todas aquellas que contienen al menos un *if* que no tiene un *else* correspondiente o bien un bucle con linea abierta. La segunda mencionada, son aquellas que resultan de tener sentencias simples, comentarios, o bien:

if (cond1) bloqueCerrado else bloqueCerrado

Donde bloqueCerrado puede ser cualquier conjunto de sentencias escritas entre llaves, sentencias simples o , razonando inductivamente, lineas cerradas. También se incluyen los bucles con lineas cerrados. Finalmente, la situación se complejizó un poco ya que hay que considerar las distintas combinaciones recién mencionadas, sumando la posibilidad de tener, o no, comentarios en cada uno de estos bloques.

3 Gramática Final

A continuación se define la gramática utilizada para construir el parser. La misma es **no ambigua** y **LALR**. La garantía de esto es que ply acepta gramáticas de este tipo, y nuestra implementación no arroja conflictos **shift/reduce** o **reduce/reduce**

G—$< V_t, V_{nt}, g, P >$

V_t es el conjunto de símbolos terminales dado por los símbolos en **mayúsculas** que aparecen en las producciones.

V_{nt} es el conjunto de símbolos no-terminales dado los literales(operadores) y los símbolos en **minúsculas** que aparecen en las producciones.

P es el conjunto de producciones dadas a continuación|

Sentencias y estructura general

g → linea g | COMMENT g | empty

[3] https://en.wikipedia.org/wiki/Dangling_else

linea → lAbierta | lCerrada

Linea Abierta: Hay por lo menos un IF que no matchea con un else

lAbierta → IF (cosaBooleana) linea |

| IF (cosasBooleana) { g } ELSE lAbierta

| IF (cosasBooleana) { g } ELSE lAbierta

| IF (cosasBooleana) { g }

| loop Labierta

Las siguientes son las variantes de tener bloques cerrados else bloques cerrados.

Un bloque cerrado puede ser una sentencia única o un bloque entre llaves.

En cada uno de estos casos puede haber, o no, comentarios. De ahí todas estas combinaciones.

lCerrada → sentencia

| IF (cosaBooleana) { g } ELSE { g }

| IF (cosaBooleana) lCerrada ELSE { g }

| IF (cosaBooleana) COMMENT com lCerrada ELSE { g }

| IF (cosaBooleana) { g } ELSE lCerrada

| IF (cosaBooleana) lCerrada ELSE lCerrada

| IF (cosaBooleana) COMMENT com lCerrada ELSE lCerrada

| IF (cosaBooleana) lCerrada ELSE COMMENT com lCerrada

| IF (cosaBooleana) COMMENT com lCerrada ELSE COMMENT com lCerrada

| loop { g }

| loop lCerrada

| loop COMMENT com lCerrada

| DO { g } WHILE (valores) ;

| DO lCerrada WHILE (valores) ;

| DO COMMENT com lCerrada WHILE (valores) ;

com → COMMENT com | λ

Sentencias básicas:

sentencia → varsOps | func ; | varAsig ; | RETURN; | ;

Bucles:

loop → WHILE (valores) | FOR (primerParam ; valores ; tercerParam)

primerParam → varAsig | λ

tercarParam → varsOps | varAsig | func | λ

cosaBooleana → expBool | valoresBool

Funciones:

func → FuncReturn | FuncVoid

funcReturn → FuncInt | FuncString | FuncBool

funcInt → MULTIESCALAR(valores, valores param)

funcInt → LENGTH(valores)

funcString → CAPITALIZAR(valores)

funcBool → colineales(valores,valores)

funcVoid → print(Valores)

param → valores | λ

Vectores y variables:

vec → [elem]

elem → valores,elem | valores

vecval → id [expresion] | vec [expresion] | vecVal [expresion] | ID[INT]

expresion → eMat | expBool | funcReturn | reg | FLOAT | |STRING | | RES
| BOOL | varYVals | varsOps | vec | atributos | ternario

valores → varYVals | varsOps | eMat | expBool | funcReturn | reg | INT | FLOAT
| STRING | BOOL | ternario | atributos | vec | RES

atributos → ID.valoresCampos | reg.valoresCampos

valoresCampos → varYVals | END | BEGIN

Operadores ternarios:

ternario → ternarioMat | ternarioBool | (ternarioBool) | (ternarioMat)
| ternarioVars | (ternarioVars)

ternarioVars → valoresBool ? valoresTernarioVars : valoresTernarioVars
| valoresBool ? valoresTernarioVars : valoresTernarioMat
| valoresBool ? valoresTernarioMat : valoresTernarioVars
| valoresBool ? valoresTernarioVars : valoresTernarioBool
| valoresBool ? valoresTernarioBool : valoresTernarioVars
| expBool ? valoresTernarioVars : valoresTernarioVars
| expBool ? valoresTernarioVars : valoresTernarioMat
| expBool ? valoresTernarioMat : valoresTernarioVars
| expBool ? valoresTernarioVars : valoresTernarioBool
| expBool ? valoresTernarioBool : valoresTernarioVars

valoresTernarioVars → reg | vec | ternarioVars | (ternarioVars) | atributos
| varsOps | varYVals | RES

TernarioMat → valoresBool ? valoresTernarioMat : valoresTernarioMat
| expBool ? valoresTernarioMat : valoresTernarioMat

valoresTernarioMat → INT | FLOAT | funcInt | STRING | eMAt
| ternarioMat | (ternarioMat)

ternarioBool → valoresBool ? valoresTernarioBool : valoresTernarioBool
| expBool ? valoresTernarioBool : valoresTernarioBool

valoresTernarioBool → BOOL | funcBool | ternarioBool | (ternarioBool)| expBool

varYVals:

varYVals → ID | vecVal | vecVal.varYVals

Registros:

reg → { campos }

campos → ID:valores, campos | ID:valores

Operadores de variables:

varsOps→ MENOSMENOS varYVals | MASMAS varYVals
| varYVals MASMAS | varYVals MENOSMENOS

Asignaciones:

varAsig → variable MULEQ valores | variable DIVEQ valores | variable MASEQ valores
| variable MENOSEQ valores | variable = valores| ID . ID = valores

variable → ID | vecVal | vecVal.varYVals

Operaciones binarias enteras:

valoresMat → INT | FLOAT | funcInt | atributos | funcString
| STRING | varYVals | varsOps | (ternarioMat)

eMat → eMat + p | valoresMat + p | eMat + valoresMat | valoresMat + valoresMat
| eMat - p | valoresMat - p | eMat - valoresMat | valoresMat - valoresMat | p

p → p * exp | p / exp | p | valoresMat * exp | valoresMat / exp | valoresMat
| p * valoresMat | p / valoresMat | p % valoresMat | valoresMat * valoresMat
| valoresMat valoresMat | valoresMat % valoresMat | exp

exp → exp îSing | valoresMat îSing | exp ŵvaloresMat
| valoresMat ŵvaloresMat | iSing

iSing → - paren | + paren | - valoresMat | + valoresMat | paren

paren → (eMat) | (valoresMat)

Expresiones booleanas:

valoresBool → BOOL | funcBoo l| varYVals | varsOps | (ternarioBool)

expBool → expBool OR and | valoresBool OR and |

| expBool OR valoresBool | valoresBool OR valoresBool | and

and → and AND eq | valoresBool AND eq | and AND valoresBool

| valoresBool AND valoresBool | eq

eq → eq EQEQ mayor | eq DISTINTO mayor | tCompareEQ EQEQ mayor

| tCompareEQ DISTINTO mayor | eq EQEQ tCompareEQ

| eq DISTINTO tCompareEQ tCompareEQ EQEQ tCompareEQ

| tCompareEQ DISTINTO tCompareEQ | mayor

tCompareEQ → BOOL | funcBool | varYVals | varsOps | INT

| FLOAT | funcInt | eMat | (ternarioBool) | (ternarioMat)

tCompare → eMat | varsOps | varYVals | INT | funcInt | FLOAT | (ternarioMat)

mayor → tCompare > tCompare | menor

menor → tCompare < tCompare | not

not → NOT not | NOT valoresBool | parenBool

parenBool → (expBool)

4 Implementacion Del Lexer

Para la construccion del lexer se definió un conjunto de literales con operadores y otros simbolos propios del lenguaje.

literals = $[+, -, *, /, \%, <, >, =, !, \{\ ,\}\ , (,), [,], ?, :, ;, ,, .]$

A su vez, utilizamos otra funcionalidad de ply para definir las palabras reservadas del lenguaje:

reserved = { 'begin' : 'BEGIN', 'end' : 'END', 'while' : 'WHILE', 'for' : 'FOR',

'if' : 'IF', 'else' : 'ELSE','do' : 'DO', 'res' : 'RES', 'return' : 'RETURN',

'AND' : 'AND', 'OR' : 'OR', 'NOT' : 'NOT', 'print' : 'PRINT',

'multiplicacionEscalar': 'MULTIESCALAR', 'capitalizar': 'CAPITALIZAR',

'colineales': 'COLINEALES', 'print': 'PRINT', 'length': 'LENGTH', }

Esto permitió evitar tener que definir demasiadas reglas simples para este tipo de operadores o palabras claves.

Para el resto de los tokens definidos, fue necesario utilizar expresiones regulares, se muestra

mas abajo como se definieron cada una de ellos: (Por claridad, se omite el resto del codigo para la regla de los tipos,ya que es análoga a la de **string**):

```
t_EQEQ = r"=="
t_DISTINTO = r"!="
t_MENOSEQ = r"-="
t_MASEQ = r"\+="
t_MULEQ = r"\*="
t_DIVEQ = r"/="
t_MASMAS = r"\+\+"
t_MENOSMENOS = r"–"
```

```
def t_STRING(token):
    r' ' ' " .*? " ' ' '
    atributos = {}
    atributos["type"] = "string"
    atributos["value"] = token.value
    token.value = atributos
    return token
```

```
def t_BOOL(token) :
    r"true | false"
def t_FLOAT(token):
    r"[-]?[0-9]
def t_INT(token) :
    r"[-]?[1-9][0-9]* — 0"
```

```
def t_ID(token):
    r"[a-zA-Z_][a-zA-Z_0-9]*"
    token.type = reserved.get(token.value,'ID')
def t_NEWLINE(token):
    r"\n+"
    token.lexer.lineno += len(token.value)
```

```
def t_error(token):
  message = "Token desconocido:"
  message += "\n type:" + token.type
  message += "\n value:" + str(token.value)
  message += "\n line:" + str(token.lineno)
  message += "\n position:"+str(token.lexpos)
  raise Exception(message)
```

```
def t_COMMENT(token):
  r'#.*'
t_ignore = ' \ t'
```

5 Implementacion Del Parser

La implementación del parser consistió en transcribir la gramatica final del apartado 2 a la sintaxis de ply.

De esta manera, dada una producción:

$$Valores \rightarrow ExpresionMatematica$$

Fue necesario reescribirla como:

$def\ p\_valores(subexpressions)$:

$'''\ valores\ :\ ExpresionMatematica\ '''$

Ademas contamos con la funcionalidad de ply que, una vez que se a utlizado una producción permite ejecutar codigo adicional. Siguiendo el ejemplo anterior, suponiendo que cada vez que el parser utiliza la producciòn antedicha deseo imprimirla, puedo escribir:

$def\ p\_valores(subexpressions):$

$'''\ valores\ :\ ExpresionMatematica\ '''$

$print\ subexpressions[1]$

Esto lo utilizamos tanto para escribir el output formateado con la salida correcta como para realizar el chequeo de tipos.

5.1 Chequeo de tipos

5.1.1 Resumen atributos

Para el chequeo de tipos utilizamos atributos sintetizados que se enumeran a continuación:

- var : Denota la variable de la expresión (si es que hay solo una variable).
- type : Denota el tipo de la expresión.
- elems : Si la expresión es de tipo vector, denota los tipos de los elementos del vector.
- varsVec : Si la expresión es de tipo vector, denota las variables de los elementos del vector (si tienen alguna).
- regs:
- campo: denota el valor de un campo de un registro.
- varAsig e indice : denota la variable y el indice en asignaciones como a[i] = 1, en donde la expresión a la derecha del "=" **no** es un valor.

5.1.2 Asignación con variables, vectores y registros

Los tipos posibles en el atributo *type* son: string, float, int, bool, vec y reg. En algunas producciones se usan como simples atributos sintetizados, en el caso de que una expresión sea una variable se necesita saber que tipo tuvo esa variable cuando se inicializó.

Para esto se cuenta con el diccionario global *variables* que tiene de claves a las variables y de valor su tipo (con fines declarativos hay otro diccionario dentro del valor, que tiene una clave "type" y el valor es el tipo de la variable).

Algo similar ocurre con variables que referencian a vectores. Se agrega un diccionario global *vectores* en donde se guarda la variable como clave y una lista de tipos como valor, que denotan los tipos de las respectivas posiciones del vector.

Para los vectores de vectores hay un diccionario global *variablesVector* que guarda como clave a la variable correspondiente al vector de vectores y como valor una lista de variables que se corresponden con las variables de los vectores dentro del vector.

Como se pueden hacer operaciones tales como [1,2,3][0] en donde no se involucra ninguna variable, se agrego el atributo *elems* que se encarga de obtener esa lista de tipos del vector.

Observar que, como no existe referencia a este vector, no es necesario guardarlo en el diccionario *vectores* ya que no se utilizará en otro lugar.

En caso de los registros se guarda un atributo *campos* en donde se guardan los campos del registro. En el se guarda una lista de tuplas, donde la primer coordenada corresponde al nombre del campo, la segunda al tipo del campo y la tercera a la variable del vector si tuviera un campo de tipo vector.

En caso de las asignaciones se guarda en un diccionario global *registros* la variable correspondiente al registro y los campos obtenidos a través del atributos *campos*.

6 Implementacion Salida

Para la implementación de la salida cada producción de la gramatica tendrá un atributo sintetizado "value" en el que se guardará el texto de salida basandose en el valor de cada uno de los terminales y no terminales que lo compenen. Esto es relativamente sencillo para las sentencias que solo requieren escribir sus terminales y no terminales seguidos de un salto de linea, pero requerirá un cuidado especial para el caso de los condicionales y los loops, ya que estos necesitan saltos de linea en lugares intermedios y una indentación adecuada.
Para ejemplificar que debemos hacer utilizamos la produccón:

$$lAbierta \rightarrow IF\ (\ cosaBooleana\ )\ COMMENT\ com\ lCerrada\ ELSE\ \{\ g\ \}$$

como caso de estudio.
Lo que querremos guardar en el atributo value de lAbierta es primero un $IF\ (\ cosaBooleana\ )$. Notese aquí que cosaBolleana es un conjunto de simbolos que que deben reducir como ultima instancia a una Expresón Booleana. queda a cargo de cosaBooleana la responsabilidad de saber como imprimir cada uno de esos posibles terminos que se encuentren presentes.

Esto estará seguido de un salto de linea, seguido de un comentario (con un tab) y un salto de linea, luego un no terminal que contendrá cero o uno o varios comentarios (todos conteniendo su indentación apropiada), otro salto de linea con una palabra reservada else, unas llaves que abren y una o varias sentencias indentadas un corchete que cierra.
En primera instancia, la salida que nos dará este if es la deseada, sin embargo puede que ocurra el caso donde tenemos dos ifs anidados. En este caso lo esperable es que el segudo if (el interno) este indentado y que las sentencias dentro de ese if tengan dos tabs en vez de uno. Luego, no es suficiente con agregar tabs en los lugares adecuados, tambien necesitamos señalizar donde comienza una nueva linea y en el caso de que sea necesario agregar varios tabs, que se pueda recorrer la salida agregandolos donde se necesite.
Para el manejos de la indentación utilizaremos al comienzo de cada nueva linea.

7 Conclusiones

Este trabajo nos permitió generar una idea de como generar una gramática para un lenguaje de programación, cuales son los mayores desafíos, el problema de las ambigüedades en el lenguaje, como en el problema del dangling else y sirvió como disparador para pensar diferentes soluciones a estas problemáticas. Además, sirvió para discutir temas tales como que chequeos de tipo pueden

realizarse efectivamente en tiempo de compilación y cuales no queda mas alternativa que solucionarlos en tiempo de ejecución.