Forma

Descripción generada automáticamente con confianza media

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**DOBLE GRADO EN INGENIERÍA INFORMÁTICA E INGENIERÍA DEL SOFTWARE**

**Curso Académico 2023/2024**

**Procesadores de Lenguajes**

**8 de abril de 2024**

**Práctica Obligatoria PL**

**Equipo 6**:

Ángel Marqués García

Tarek Elshami Ahmed

Álvaro Serrano

# Índice

[Índice I](#_Toc165836963)

[1. Introducción 2](#_Toc165836964)

[2. Parte Aprobado 3](#_Toc165836965)

[2.1 Reducción de la gramática a LL(1) 3](#_Toc165836966)

[2.1.1 Eliminar recursividad por la izquierda 3](#_Toc165836967)

[2.1.2 Factorizar 3](#_Toc165836968)

[2.1.3 Generar los conjuntos de CAB, SIG, DIR 4](#_Toc165836969)

[2.2 Generar la gramática de ANTLR 6](#_Toc165836970)

[3. Parte Notable 7](#_Toc165836971)

[3.1 Sentencias de control de flujo 7](#_Toc165836972)

[3.2 Reconocimiento y notificación de errores 7](#_Toc165836973)

[4. Parte sobresaliente 8](#_Toc165836974)

[4.1 Traducción 8](#_Toc165836975)

[4.1.1 Utilización de interfaces 9](#_Toc165836976)

[4.2 Distinguir entre librerías y programas 9](#_Toc165836977)

[4.3 Mejora de la generación de declaraciones 9](#_Toc165836978)

[4.4 Recuperación de errores 10](#_Toc165836979)

[5. Probar el correcto funcionamiento del reconocedor 10](#_Toc165836980)

[5.1 Correctos 10](#_Toc165836981)

[5.1.1 Caso 1 10](#_Toc165836982)

[5.1.2 Caso 2 11](#_Toc165836983)

[5.1.3 Caso 3 11](#_Toc165836984)

[5.1.4 Caso 4 11](#_Toc165836985)

[5.2 Erróneos 12](#_Toc165836986)

[5.2.1 Caso 1 12](#_Toc165836987)

[5.2.2 Caso 2 13](#_Toc165836988)

[5.2.3 Caso 3 13](#_Toc165836989)

[5.2.4 Caso 4 13](#_Toc165836990)

# Introducción

Esta memoria redacta el proceso seguido por el **grupo 6** en la realización de la práctica obligatoria de Procesadores de Lenguajes. El objetivo de esta práctica es diseñar e implementar un programa capaz de traducir código de un lenguaje fuente a un lenguaje final. El lenguaje fuente será similar a C, mientras que el lenguaje final será similar a PASCAL.

La práctica consta de tres sets de requisitos diferenciados, completar cada uno permite optar a una mejor calificación:

1. Requisitos Aprobado (parte obligatoria)
2. Requisitos Notable (sentencias de control de flujo y notificación de errores)
3. Requisitos Sobresaliente (recuperación de errores y toda la parte opcional)

A la misma vez, la práctica consta de dos entregas. La primera consta de los analizadores léxico y sintáctico, y la segunda de la práctica completa.

Por tanto, el primer paso que tomamos en el desarrollo de la práctica ha sido desarrollar las reglas léxicas y sintácticas de la gramática, y portar la gramática a ANTLR. Este proceso consta de 2 fases principales:

1. Reducir la gramática a LL (1)
2. Trasladar la gramática a ANTLR

Y de nuevo este proceso se ha dividido entre los distintos sets de requisitos, de forma que primero ha sido aplicado a la parte de aprobado, y luego a las partes de notable y sobresaliente.

Una vez tenemos la gramática en ANTLR, el siguiente paso es diseñar e implementar el sistema que se va a utilizar para la traducción. En este proceso, creamos una serie de clases externas con las que interactuamos desde la gramática, y que darán lugar a la traducción de la entrada.

# Parte Aprobado

## Reducción de la gramática a LL(1)

La gramática del lenguaje fuente es la siguiente:

program ::= defines partes

defines ::= ʎ | "#define" IDENT ctes defines

ctes ::= CONSTINT | CONSTFLOAT | CONSTLIT

partes ::= part partes | part

part ::= type restpart

restpart ::= IDENT "(" listparam ")" blq

| IDENT "(" "void" ")" blq

blq ::= "{" sentlist "}"

listparam ::= listparam "," type IDENT | type IDENT

type ::= "void" | "int" | "float"

sentlist ::= sentlist sent | sent

sent ::= type lid ";" | IDENT "=" exp ";" | IDENT "(" lexp ")" ";"

| IDENT "(" ")" ";" | "return" exp ";"

lid ::= IDENT | lid "," IDENT

lexp ::= exp | lexp "," exp

exp ::= exp op exp | factor

op ::= "+" | "-" | "\*" | "/" | "%"

factor ::= IDENT "(" lexp ")" | IDENT "(" ")"

| "(" exp ")" | IDENT | ctes

Para reducirla a LL(1) hemos seguido una serie de pasos:

### Eliminar recursividad por la izquierda

Hemos identificado que las siguientes reglas tenían recursividad por la izquierda:

* listparam
* sentlist
* lid
* lexp
* exp

Y hemos eliminado la recursividad por la izquierda modificando estas reglas y añadiendo nuevas, de forma que ahora son de la siguiente manera:

listparam: type IDENT listparam2;  
listparam2: ',' type IDENT listparam2 | ʎ;

sentlist: sent sentlist2;  
sentlist2: sent sentlist2| ʎ;  
lid: IDENT lid2;  
lid2: ',' IDENT lid2 | ʎ;  
lexp: exp lexp2;  
lexp2: ',' exp lexp2 | ʎ;  
exp: factor exp2;  
exp2: op exp exp2| ʎ;

Una vez la gramática no tiene recursividad por la izquierda, el siguiente paso es:

### Factorizar

Hemos identificado que las siguientes reglas podían ser factorizadas:

* restpart
* sent
* factor
* partes

Y las hemos factorizado modificándolas y añadiendo nuevas reglas, de forma que ahora son de la siguiente manera:

restpart: IDENT '(' restpart2;  
restpart2: listparam ')' blq | 'void' ')' blq;  
sent: type lid ';'  
 | 'return' exp ';'  
 | IDENT sent2  
sent2: '=' exp ';'| '(' sent3;  
sent3: ')'';'| lexp ')'';';

factor: '(' exp ')' | ctes | IDENT factor2;  
factor2: '(' factor3 | ;  
factor3: lexp ')' | ')';

partes: part partes2;  
partes2: partes | ;

Ahora que la gramática no tiene recursividad por la izquierda ni contiene reglas que puedan ser factorizadas a simple vista, el siguiente paso es:

### Generar los conjuntos de CAB, SIG, DIR

Para generar estos conjuntos hemos utilizado la herramienta JFLAP. Dado que JFLAP sólo permite usar reglas que consten de una letra, hemos tenido que mapear cada regla a una letra del alfabeto. Así, la representación de las reglas en JFLAP es la siguiente:

A -> program  
B -> defines  
C -> ctes

D -> partes  
Ñ -> partes2  
E -> part  
F -> blq

G -> type  
H -> op  
I -> listparam  
J -> listparam2  
K -> sentlist  
L -> sentlist2  
M -> lid  
N -> lid2  
O -> lexp

P -> lexp2  
Q -> exp  
R -> exp2  
S -> factor  
T -> factor2  
U -> factor3  
V -> restpart  
W -> restpart2  
X -> sent   
Y -> sent2  
Z -> sent3

Y los símbolos terminales han sido codificados de la siguiente manera:

a -> '#defines'  
b -> IDENT  
c -> CONSTINT  
d -> CONSTFLOAT  
e -> CONSTLIT  
f -> '{'  
g -> '}'  
h -> 'void'  
i -> 'int'  
j -> 'float'  
k -> '+'  
l -> '-'  
m -> '\*'  
n -> '/'  
o -> '%'  
p -> ','  
q -> '('  
r -> ')'  
s -> ';'

t -> 'return'

Con esta traducción, JFLAP genera los siguientes conjuntos de Cabecera y Siguiente:  
Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Junto con la siguiente tabla de directores:

Imagen que contiene mucho, cubierto, llenado, luz

Descripción generada automáticamente

El objetivo de las transformaciones realizadas hasta ahora era reducir la gramática para que fuera LL(1). Esto es porque es más fácil trabajar con gramáticas de este tipo. Generar estos conjuntos, especialmente el de los directores, nos sirve para comprobar si, tras las transformaciones realizadas, la gramática es LL(1).

Como se puede observar en los conjuntos de directores generados, existe una notable superposición entre los conjuntos de algunas reglas. Esto indica que hay ambigüedades, por lo que la gramática no es LL(1). Esta superposición se observa especialmente en las siguientes reglas:

restpart (V) con IDENT (b)

Restpart2 (W) con ‘void’ (h)

Viendo que la gramática no es LL(1), nos veríamos inclinados a seguir transformándola hasta que lo sea. Sin embargo, las transformaciones necesarias para seguir reduciendo la gramática tienen una complejidad mucho mayor que las ya realizadas. Además, la herramienta que vamos a utilizar, ANTLR, es capaz de trabajar con gramáticas LL(k), por lo que no es necesario continuar reduciendo la gramática. Dada la complejidad que conlleva seguir reduciendo la gramática, y el hecho de que no es estrictamente necesario, hemos dejado la gramática en este estado.

## Generar la gramática de ANTLR

Para generar la gramática en ANTLR hemos copiado las reglas de la gramática final a ANTLR, ajustando el formato para que sea el adecuado, y creando además los fragmentos léxicos necesarios para el correcto funcionamiento del reconocedor. Se han creado los siguientes fragmentos léxicos:

IDENT: (LETTER | '$') (LETTER |DIGIT|'\_'| '$')\*;  
  
CONSTINT: DECIMAL\_INT | OCTAL\_INT | HEXA\_INT;  
CONSTFLOAT: DECIMAL\_REAL | OCTAL\_REAL | HEXA\_REAL;  
  
DECIMAL\_INT: ('+' | '-')? DIGIT+;  
DECIMAL\_REAL: ('+' | '-')? DIGIT+ '.' DIGIT+;  
  
OCTAL\_INT: '0' ('+' | '-')? OCT\_DIGIT+;  
OCTAL\_REAL: '0' ('+' | '-')? OCT\_DIGIT+ '.' OCT\_DIGIT+;

HEXA\_INT: '0x' ('+' | '-')? HEX\_DIGIT+;  
HEXA\_REAL: '0x' ('+' | '-')? HEX\_DIGIT+ '.' HEX\_DIGIT+;  
  
CONSTLIT: '\'' (ESC\_SEQ | ~['])\* '\'' ;  
  
GPC: '//' ~[\r\n]\* -> skip ;  
GPC2: '/\*' .\*? '\*/' -> skip;  
  
JUMP: ('\r'? '\n' | ' ')+ -> skip;  
  
fragment ESC\_SEQ: '\\' '\'';  
fragment LETTER: [a-zA-Z];  
fragment HEX\_DIGIT: [0-9A-F];

fragment OCT\_DIGIT: [0-7];  
fragment DIGIT: [0-9];

Si bien hemos definido una galería de reglas léxicas, las realmente importantes son IDENT, CONSTINT, CONSTFLOAT, y CONSTLIT. Todas las demás existen únicamente para simplificar estas y que resulte más claro y fácil de entender. IDENT representa un identificador de una variable o función, y esto es cualquier cadena de caracteres que empiece por una letra o el símbolo $, y vaya concatenado con cualquier número de letras, dígitos, o los símbolos \_ y $. Por otro lado, CONSTINT y CONSTFLOAT son muy similares entre sí, ya que ambas representan números. CONSTINT representa números enteros, y CONSTFLOAT números decimales. A su vez, un número puede venir representado en tres bases distintas, decimal, octal, y hexadecimal. Para facilitar su representación, se han creado reglas adicionales para cada posible representación de uno de estos números. Por último, CONSTLIT reconoce constantes literales, que son cadenas de caracteres encerradas entre comillas. Pueden contener comillas, pero estas deben ir precedidas del símbolo de escape \.

Algunas de las reglas adicionales que hemos creado no representan elementos completos, sino que son fragmentos, por ejemplo, las letras o dígitos, que pueden ser parte de varias reglas, pero no deberían ser reconocidas independientemente. Estas reglas las hemos declarado como fragmentos de ANTLR, y recogen los rangos de valores que pueden tener los distintos ítems.

# Parte Notable

Los requisitos de esta parte incluyen reconocer sentencias de control de flujo, y reconocer y notificar errores. Lo primero que hemos hecho ha sido modificar la gramática para añadir reconocimiento de las sentencias de control de flujo. Para ello hemos partido de una gramática de estas sentencias, que hemos reducido a LL(1), e incorporado al resto de la gramática.

## Sentencias de control de flujo

Partimos de las siguientes reglas:

lcond ::= lcond opl lcond | cond | "!" cond

opl ::= "||" | "&&"

cond ::= exp opr exp

opr "==" | "<" | ">" | ">=" | "<="

sent ::= …

| "if" "(" lcond ")" blq "else" blq

| "while" "(" lcond ")" blq

| "do" blq "until" "(" lcond ")"

| "for" "(" IDENT "=" exp ";" lcond ";" IDENT "=" exp ")" blq

Identificamos que la regla lcond tiene recursividad por la izquierda, y la eliminamos, obteniendo:

lcond ::= cond lcond2 | "!" cond lcond2;  
lcond2 ::= opl lcond lcond2 | ʎ;

Identificamos que ninguna regla puede ser factorizada a simple vista, y procedemos a generar los conjuntos de cabecera, siguiente, y directores. Cabe destacar que, al estar modificando la regla sent, será necesario recalcular los conjuntos de todas las reglas, pues al modificar la gramática podrían haber cambiado.

Sin embargo, dado que el objetivo de generar estos conjuntos es determinar si la gramática es o no LL(1), no es necesario volverlos a generar, ya que hemos comprobado en la sección anterior que la gramática NO es LL(1), y los cambios realizados aquí, dado que no modifican ninguna regla (añaden posibilidades a sent, pero no modifican las que ya tenía), no van a cambiar ese hecho.

## Reconocimiento y notificación de errores

Para poder notificar los errores que surjan de una manera más concisa, hemos sustituido el ErrorListener que proporciona ANTLR por defecto por un ErrorListener personalizado. Para ello, el primer paso ha sido crear un CustomErrorListener. Para que sea reconocido como un ErrorListener y para que no sea necesario implementar cada método de cero, deberá extender la clase BaseErrorListener. Luego, se sobreescribe el método syntaxError, implementando en su código lo que debe ocurrir cuando se encuentre un error. Para no tener que crear una clase para errores sintácticos y otra para léxicos, reutilizaremos la misma clase, ayudándonos de un atributo type, que recibe su valor en el constructor, y valdrá 0 para el analizador léxico, y cualquier otro valor para el sintáctico. Luego, el mensaje de error indicará, en función de este atributo, si el error era léxico o sintáctico, y en caso de ser sintáctico, añadirá un mensaje mostrando la línea en la que haya ocurrido el error con el token que lo ha causado subrayado.

Una vez creada esta clase, sólo queda utilizarlo como ErrorListener en los analizadores léxico y sintáctico. Para ello lo primero es borrar los *ErrorListeners* por defecto de ambos reconocedores.



Una vez eliminado el listener por defecto, le añadimos el listener personalizado:



Una vez hecho esto, cuando haya un error, se mostrarán mensajes de error como este:



Cabe destacar que el mensaje de error de esta imagen es el que aparecerá cuando el error sea sintáctico. Si ocurre un error léxico, sólo se verá la primera línea, anunciando la presencia de un error léxico junto con su posición y posible causa.

# Parte sobresaliente

La parte sobresaliente tiene tres requisitos principales: haber completado la parte notable, haber completado toda la parte opcional, y haber proporcionado recuperación de errores léxica y sintáctica. La parte notable ya se ha explicado en su propia sección, así que esta la dedicaremos a explicar cómo se han implementado los elementos restantes de la parte opcional y la recuperación de errores. Quedaban dos partes por implementar de la sección opcional: distinguir entre librerías y programas, y mejorar la generación de declaraciones. El método que hemos seguido para realizar estas tareas está estrechamente enlazado al procedimiento que hemos seguido para la traducción, por lo que también incluiremos en esta sección una explicación de este proceso.

## Traducción

El enfoque utilizado para la traducción es el siguiente:

1. Estudiar ambas gramáticas para familiarizarnos con ellas y entenderlas a fondo
2. Identificar los principales elementos del lenguaje final, ayudándose para ello de la gramática proporcionada. Estos elementos son cosas como asignaciones, declaraciones de variables, llamadas a procedimientos/funciones, tipos, expresiones, etc.
3. Para cada elemento, crear una clase. Se creará una nueva instancia de esta clase cada vez que se identifique que hay un elemento de este tipo, y esa instancia guardará la información relevante. Por ejemplo, para una asignación, al encontrar “someVar = 2;”, crearemos una instancia de asignación, que almacenará el nombre de la variable y su valor.
4. Para cada clase, hacer que implemente una interfaz Translation, que consta en un método getTranslation() que devuelve una String. Este método es el responsable de, partiendo de la información almacenada en la clase, formar la String que corresponde al elemento en la gramática final. Volviendo al ejemplo anterior, el método getTranslation de la clase asignación podría ser algo similar a “return variable + ‘:=’ + exp.getTranslation()’, asumiendo que el nombre se almacena en una propiedad tipo String llamada variable, y su valor en una variable de tipo <? extends Translation> llamada exp. En nuestro caso, la clase del valor es de tipo Expression.
5. Al haber reconocido la gramática completa, llamar al método getTranslation de la clase de más alto nivel. En nuestro caso, esa clase es Program. Al llamar a su getTranslation, este irá llamando de manera recursiva al resto de elementos para obtener su traducción, resultando al final en una string que contiene la traducción completa.

La imagen siguiente muestra un diagrama de clases ilustrando las clases creadas y cómo estas están relacionadas entre sí. En este diagrama, algunas conexiones tienen color. Este color se ha añadido para mejorar la claridad del diagrama, y no conlleva ningún tipo de información adicional.

Interfaz de usuario gráfica

Descripción generada automáticamente

Durante el proceso de diseñar la traducción, observamos que la gramática final tiene una serie de listas heterogéneas. Por ejemplo, un bloque consiste en una lista de declaraciones y una lista de sentencias. Pero una sentencia puede ser de varios tipos: puede ser una asignación, una llamada a otra función, un return, e incluso, incluyendo la parte opcional, algún tipo de condicional o bucle. Algo similar ocurre con las declaraciones. En casos como estos, para poder almacenar en una misma lista estos objetos distintos hemos creado interfaces. De forma que el objeto Blq tiene una lista de objetos que implementan Declaration y otra de objetos que implementan Sent. Esto ha sido esencial para permitir almacenar los valores de manera sencilla, y ha facilitado mucho el proceso.

## Distinguir entre librerías y programas

Para distinguir entre librerías y programas, la clase Program tiene un atributo booleano hasMain. Cada vez que se reconoce una declaración de una función, se comprueba su nombre. En caso de que fuera “main”, se le asigna true a hasMain, que por defecto vale false. Cuando se llame program.getTranslation(), si hasMain vale true, se traducirá con el formato de un programa, y en caso contrario se traducirá como una unidad.

## Mejora de la generación de declaraciones

Este punto consistía en identificar casos en los que se declaran, en dos líneas consecutivas, declaraciones de variables del mismo tipo, para, al traducir, poder declararlas todas en una misma sentencia. De igual manera, era necesario identificar casos en los que, al declarar una función, se declaraban parámetros del mismo tipo de manera consecutiva, con el mismo fin de agruparlos en la traducción.

Para conseguir esto hemos modificado los métodos getTranslation de Blq, que contiene la lista de declaraciones, y de MethodDcl, que es la clase que contiene la sentencia de declaración de un método, y por tanto, sus parámetros. Sin embargo, hemos usado métodos ligeramente distintos en cada uno.

La traducción de Blq consta de dos partes, la traducción de la lista de declaraciones y la traducción de la lista de sentencias. Para esta sección sólo nos interesa la primera parte. Esta tiene un funcionamiento algo peculiar. Dado que en el lenguaje final las declaraciones de constantes y variables van agrupadas, creamos tres partes de la traducción: constants, vars, y other. Mientras iteramos por la lista de declaraciones, comprobamos sus tipos. Si la declaración es una VarDcl, y su isConstant devuelve true, añadimos su traducción a constants. Si es VarDcl y isConstant devuelve false, su traducción se añade a vars, y en otro caso, se añade a other. Para implementar esta funcionalidad, sólo es necesario modificar la sección en la cual la declaración es una instancia de VarDcl, y su isConstant devuelve false.

Se ha modificado siguiendo la siguiente lógica:

* Si esta variable es la primera declaración, o la primera variable de su tipo, se añade una tabulación a la traducción.
* Si esta variable no es la última declaración, y la siguiente declaración en la lista es una variable del mismo tipo que la variable actual, se añade a la traducción el nombre de la variable seguido por una coma.
* En caso contrario, se añade la traducción de la variable devuelta por getTranslation().

En lo referente a las listas de parámetros, antes de implementar esta mejora hacíamos lo siguiente: al traducir la declaración de un método, cuando lleguemos a sus parámetros simplemente iterábamos por la lista de parámetros del método llamando a getTranslation sobre cada parámetro, e incluyendo manualmente comas entre ellos. Para incluir esta mejora, hemos modificado notablemente este proceso. Se sigue el siguiente algoritmo:

* Crear una variable lastType, inicializada a “”. Esta variable guarda el tipo de la variable en la posición i-1.
* Añadir a la traducción el nombre del parámetro actual, seguido por una coma, y actualizar lastType
* Al principio del bucle, si lastType no coincide con el tipo del parámetro actual, eliminar la coma introducida en el paso anterior, y escribir “: ” seguido de lastType.
* Si el parámetro actual es el último, realizar el mismo proceso de arriba (eliminar coma, escribir “: ” + lastType)

## Recuperación de errores

No hemos considerado necesario modificar la recuperación de errores por defecto de ANTLR, ya que cubre todos los errores comunes, y dada la complejidad de las estrategias utilizadas, no creemos que podamos aportar una implementación mejor o más completa.

La recuperación de errores por defecto es capaz de solucionar situaciones en las que insertamos un token de más o uno de menos sin ninguna dificultad (si el siguiente token es el correcto), y en situaciones más complejas, es capaz de, mediante el “modo pánico” y el uso de un conjunto de siguientes, recuperarse de errores a cambio de potencialmente ignorar algunos tokens.

Sin embargo, en el caso de esta gramática, por la definición recursiva de algunas listas (en vez de usando cierres), cuando encuentra un error en algunas listas, ignora el resto de elementos de la lista. Esto ocurre por ejemplo en la definición de constantes. Si declaras una constante, a continuación pones un identificador sólo, y seguido a él otra definición, no reconocerá la segunda. Esto es, sin embargo, comportamiento esperado. Para evitar este error podríamos modificar la gramática para que use los cierres (+,\*) para hacer listas en vez de recursividad, ya que se ha dedicado mucho esfuerzo a que ANTLR sea capaz de recuperarse de errores que encuentre en listas definidas con los cierres.

# Probar el correcto funcionamiento del reconocedor

Por último, hemos realizado una serie de pruebas para comprobar que el reconocedor funciona de la forma esperada. Se ha probado su funcionamiento con 8 casos de prueba en total, cuatro de los cuales son correctos, y el resto erróneos. Los casos de prueba son los siguientes:

## Correctos

### Caso 1

Este caso prueba la traducción de una función simple en C que utiliza un bucle **for** para sumar un rango de números.

int sum(int a, int b) {  
 int result;  
 int i;  
 result = 0;  
  
 for (i = a; i < b; i = i + 1) {  
 result = result + i;  
 }  
 return result;  
}

La traducción debe conservar la semántica del bucle **for**, ajustando la sintaxis para cumplir con las convenciones de Pascal.

### Caso 2

En este caso, hemos modificado ligeramente el caso 1 de tal forma que el **for** sea algo más complejo.

int sum(int a, int b) {  
 int result;  
 int i;  
 result = 0;  
  
 for (i = a; i <= b; i = i + 2) {  
 result = result + i;  
 }  
 return result;  
}

Debido a la complejidad del bucle **for**, se deberá convertir a un bucle **while** en Pascal para manejar correctamente los incrementos y la condición de parada.

### Caso 3

En este caso tenemos un código con una función **main**, varias constantes y múltiples tipos de funciones.

#define decimalFloatConst +9.32  
#define octalFloatConst 0-3.06  
#define hexFloatConst 0x+A9F.7D  
  
void main(int intParam, float floatParam) {  
 float floatVar1, floatVar2;  
  
 someVar=((puedesAnidarTantosComoQuieras));  
 someOtherVar=32;  
 yetAnotherVar=someFunct(3, someVar, anotherFunction());  
 lastVar= 32 + times(2,3);  
  
 int intVar1, intVar2;  
  
 someFunction();  
  
 return value(var1,var2,2) / 3;  
  
 unreachableCall();  
 float unreachableVar;  
 unreachableAssignment = 0x9.58;  
}

### Caso 4

Evalúa la capacidad del traductor para manejar múltiples estructuras de control y tipos de datos en un programa más extenso.

#define decimalFloatConst +9.32  
#define octalFloatConst 0-3.06  
#define hexFloatConst 0x+A9F.7D  
  
int functionName(void){  
 if (0+7 < var1 || !var2==f() && 0x9A>(-10\*0+5)){  
 noPermiteBloquesVacios();  
 } else {  
 noPermiteBloquesVacios();  
 }  
  
 while ((2 + 2)==$\_5) { //$\_5 is a variable  
 noPermiteBloquesVacios();  
 }  
  
 do {  
 int i;  
 int j;  
 i = 0;  
 j = i;  
 } until (f()>+3)  
  
 for (var0 = -07; var0>f(0xFF); var0 = var0 + 1) {  
 noPermiteBloquesVacios();  
 }  
  
}  
  
void main(int intParam, float floatParam) {  
 float floatVar1, floatVar2;  
  
 someVar=((puedesAnidarTantosComoQuieras));  
 someOtherVar=32;  
 yetAnotherVar=someFunct(3, someVar, anotherFunction());  
 lastVar= 32 + times(2,3);  
  
 int intVar1, intVar2;  
  
 someFunction();  
  
 return value(var1,var2,2) / 3;  
  
 unreachableCall();  
 float unreachableVar;  
 unreachableAssignment = 0x9.58;  
}

## Erróneos

### Caso 1

Como primer caso de **input** erróneo tenemos una función en la que al **if** le faltan corchetes y la variable **y** se declara y es asignada un valor en la misma línea.

void example() {  
 if (x > 0)  
 int y = 2;  
 else  
 y = 3;  
}

### Caso 2

En este segundo caso de error, se prueba la capacidad del traductor para identificar y recuperarse en casos en los que falta un identificador. Además, se prueba un posible error, como es que se nos olvide un signo en una operación.

int (int var3) {  
 int var1;  
 var1 = 3;  
 var2 = 4;  
 return var1 var2;  
}

### Caso 3

En este caso 3, se evalúa la habilidad del traductor para recuperarse de errores leves.

#define decimalIntConst +9  
#define octalIntConst 0-3  
whatever  
#define hexIntConst 0x+A9F  
  
int main(int var3) {  
 int var1;  
 var1 8 = 3;  
 var2 = 4;  
 return var1 + var2;  
}  
  
int main2(int var3) {  
 int var1;  
 var1 = 3;  
 return var1;  
}

### Caso 4

Y por último, se verifica el comportamiento del traductor cuando falta una palabra clave como “until”.

int main1(int var3) {  
 do {  
 int var1;  
 var1 = 3;  
 return var1;  
 } (var1 < 3)  
}

int main2(int var2) {  
 var1 = 3;  
 return var2;  
}