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**

**Interacción Persona Ordenador**

**26 de noviembre de 2023**

**Salvar al rey Calabaza**

**Práctica 3: Evaluación de Interfaces de Usuario**

**Equipo 5 – Proplayers 1º División**:

Ikár Vladislav Martínez de Lizarduy Kostornichenko

Lucía Domínguez Rodrigo

Sergio De Oro Fernández

Miguel Ángel Sánchez Miranda

Demelza Santamaría Gil

Ángel Marqués García

Marcos Jiménez Pulido

# Índice

[Índice I](#_Toc162025401)

[1. Introducción 2](#_Toc162025402)

# 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 en un lenguaje fuente similar a C a un lenguaje final 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 los analizadores léxico y sintáctico. Este proceso consta de 3 fases principales:

1. Reducir la gramática a LL(1)
2. Trasladar la gramática a ANTLR
3. Probar su corrección y completitud

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 aprobado y notable.

Dado que esta es la memoria de la primera entrega, las secciones pertinentes a la generación del lenguaje final a partir del lenguaje fuente no se han incluido.

# 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

Para generar estos conjuntos 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

Una vez completado este paso, podemos proceder a generar la gramática en ANTLR.

# 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' DECIMAL\_INT;  
OCTAL\_REAL: '0' DECIMAL\_REAL;  
  
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 DIGIT: [0-9] ;

# 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

### Caso 2

### Caso 3

### Caso 4

## Erróneos

### Caso 1

### Caso 2

### Caso 3

### Caso 4