# Patrones Léxicos

**// \*\*\*\*\*\*\*\*\*\*\*\* Patrones (macros) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Rubbish = [ \t\n\r]

CommentV1 = #.\*

CommentV2 = \"""~\"""

Letter = [a-zA-Z]

Digit = [0-9]

Ident = [\_a-zA-Z][a-zA-Z\_0-9]\*

IntConstant = [0-9]\*

RealType = [0-9]+[.][0-9]\* | [.][0-9]+

RealConstant = {RealType}|{RealType}E[+|-][0-9]+|[0-9]+e[+|-][0-9]+

Character = \'.\'

CharacterASCII = [']\\[0-9]\*[']

%%

**// \*\*\*\*\*\*\*\*\*\*\*\* Acciones \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**// \* THINGS TO IGNORE**

{Rubbish}

{CommentV1}

{CommentV2}

**// \* RESERVED WORDS**

Input

print

def

while

if

else

int

double

char

struct

return

void

main

**// \* OPERATORS**

"+"

"-"

"\*"

"/"

"%"

">"

"<"

">="

"<="

"="

==

"!="

"!"

"||"

"&&"

**// \* CONSTANTS**

{IntConstant}

{RealConstant}

{Ident}

'\\n'

'\\t'

{CharacterASCII}

{Character}

**// \* DELIMITERS**

"{"

"}"

"("

")"

"["

"]"

";"

","

":"

"."

**// \* Other**

.

# GLC

**// \* Declaraciones Yacc**

%token INT\_CONSTANT

%token INPUT

%token PRINT

%token DEF

%token WHILE

%token IF

%token ELSE

%token INT

%token DOUBLE

%token CHAR

%token STRUCT

%token RETURN

%token VOID

%token ID

%token REAL\_CONSTANT

%token CHAR\_CONSTANT

%token GREATER

%token SMALLER

%token EQUALS

%token NEGATION

%token MAIN

%token OR

%token AND

%right '='

%left OR AND

%left EQUALS NEGATION SMALLER '<' GREATER '>'

%left '-' '+'

%left '\*' '/' '%'

%nonassoc CAST

%right UNARIO

%nonassoc '!'

%left '.'

%nonassoc '[' ']'

%nonassoc '(' ')'

%nonassoc ':'

%nonassoc ELSE

**%%**

**// \* Gramática y acciones Yacc**

programa : definiciones DEF MAIN '(' ')'':'VOID '{' body '}';

definiciones: definiciones definicion

| /\* empty \*/

;

definicion: def ';'

| funcion

;

**// \*\*\*\*\*\*\*\*\* FUNCIONES \*\*\*\*\*\*\*\*\***

funcion: DEF ID '(' params ')' ':' retorno '{' body '}';

retorno: tipo

| VOID

;

body: defs

| sentencias

| defs sentencias

|

;

params: /\* empty \*/

| param

;

param: par

| param ',' par

par: ID ':' tipo;

**// \*\*\*\*\*\*\*\*\* DEFINICIONES \*\*\*\*\*\*\*\*\***

defs: def ';'

| defs def ';'

;

def: ids ':' tipo

ids: ID

| ids ',' ID

;

tipo: INT

| DOUBLE

| CHAR

|'['INT\_CONSTANT']' tipo

| STRUCT '{' campos '}'

;

campos: campo

|campos campo

;

campo: ids ':' tipo ';';

**// \*\*\*\*\*\*\*\*\* SENTENCIAS \*\*\*\*\*\*\*\*\***

sentencias: sentencia

| sentencias sentencia

;

sentencia: PRINT list ';'

| INPUT list ';'

| RETURN expresion ';'

| condicionalSimple

| condicionalComplejo

| while

| asignacion ';'

| invocacion ';'

;

expresion: ID

| INT\_CONSTANT

| CHAR\_CONSTANT

| REAL\_CONSTANT

| '(' expresion ')'

| expresion '[' expresion ']'

| expresion '.' ID

| '(' tipo ')' expresion %prec CAST

| '-' expresion %prec UNARIO

| '!' expresion

| expresion '\*' expresion

| expresion '/' expresion

| expresion '%' expresion

| expresion '+' expresion

| expresion '-' expresion

| expresion '>' expresion

| expresion GREATER expresion

| expresion '<' expresion

| expresion SMALLER expresion

| expresion NEGATION expresion

| expresion EQUALS expresion

| expresion AND expresion

| expresion OR expresion

| ID '(' args ')'

;

list: expresion

| list ',' expresion

;

asignacion: expresion '=' expresion ;

invocacion: ID '(' args ')'

**// \*\*\*\*\*\*\*\*\* WHILE \*\*\*\*\*\*\*\*\***

while: WHILE expresion ':' '{' sentencias '}' ;

**// \*\*\*\*\*\*\*\*\* IF-ELSE \*\*\*\*\*\*\*\*\***

condicionalSimple: IF expresion ':' cuerpo;

condicionalComplejo: IF expresion ':' cuerpo else;

else: ELSE cuerpo ;

cuerpo: sentencia

| '{' sentencias '}'

;

**// \*\*\*\*\*\*\*\*\* INVOCACIÓN DE FUNCIONES \*\*\*\*\*\*\*\*\***

args: /\* empty \*/

| arg

;

arg: expresion

| arg ',' expresion

# Gramática Abstracta

Program: Program -> Definition\*

VarDefinition: Definition -> Type ID

FunDefinition: Definition -> Type Statement\*

Write: Statement -> Exp

Read: Statement -> Exp

Assigment: Statement -> Exp1 Exp2

IfStatement: Statement -> Exp if:Statement\* else:Statement\*

WhileStatement: Statement -> Exp Statement\*

Invocation: Statement -> Variable Exp\*

Return: Statement -> Exp

IntLiteral: Exp -> IntConstant

ChaLiteral: Exp -> CharConstant

RealLiteral: Exp -> RealConstant

Variable: Exp -> ID

Arithmetic: Exp -> left:Exp right:Exp

Comparison: Exp -> left:Exp right:Exp

Cast: Exp -> CastType valor:Exp

Logical: Exp -> left:Exp right:Exp

UnaryNot: Exp -> valor:Exp

FieldAcces: Exp -> valor:Exp ID

Indexing: Exp -> left:Exp right:Exp

Invocation: Exp -> Variable Exp\*

# Plantillas de Código

**EXECUTE[[Program: Program -> Definition\*]]()**

for(Definition d:Definition)

if(d instanceof VarDefinition)

EXECUTE[[d]]()

for(Definition d:Definition)

if(d instanceof FunDefinition)

EXECUTE[[d]]()

<CALL MAIN>

<HALT>

**EXECUTE[[FunDefinition: Definition -> Type Statement\*]]()**

Definition.Name <:>

<ENTER> Definition.LocalBytes

for(Statement s:Statement\*)

if(!s instanceof VarDefinition)

EXECUTE[[s]]()

if(Type.ReturnType instanceof VoidType)

<RET> 0 <,> Definition.LocalBytes <,> Definition.ParamBytes

**EXECUTE[[Write: Statement -> Exp]]()**

VALUE[[Exp]]()

<OUT> Exp.Type.Suffix()

**EXECUTE[[Read: Statement -> Exp]]()**

VALUE[[Exp]]()

<IN> Exp.Type.Suffix()

<STORE> Exp.Type.Suffix()

**EXECUTE[[Assigment: Statement -> Exp1 Exp2]()**

ADDRESS[[Exp1]]()

VALUE[[Exp2]]()

cg.convert(Exp2.Type,Exp1.Type)

<STORE> Exp1.Type.Suffix()

**EXECUTE[[IfStatement: Statement -> Exp if:Statement\* else:Statement\*]]()**

int label = cg.getLabels(2);

VALUE[[Exp]]()

<JZ><LABEL> label

for(Statement s:if)

EXECUTE[[s]]()

<JMP><LABEL> label+1

<LABEL> label <:>

for(Statement s:else)

EXECUTE[[s]]()

<LABEL> label+1 <:>

**EXECUTE[[WhileStatement: Statement -> Exp Statement\*]]()**

int label = cg.getLabels(2);

<LABEL> label <:>

VALUE[[Exp]]

<JZ><LABEL> label+1

for(Statement s:Statement\*)

EXECUTE[[s]]()

<JMP><LABEL> label

<LABEL> label+1 <:>

**EXECUTE[[ Invocation: Statement -> Variable Exp\*]]()**

VALUE[[ (Expression) Statement]]()

if(Variable.Type.ReturnType != IO.VoidType)

<POP> Variable.Type.ReturnType.Suffix();

**EXECUTE[[Return: Statement -> Exp]](FunDefinition)**

VALUE[[Exp]]()

cg.convert(Exp.Type,FunDefinition.Type.ReturnType);

<RET> FunDefinition.ReturnType.NumberBytes

<,> FunDefinition.LocalBytes

<,> FunDefinition.ParamBytes

**VALUE[[IntLiteral: Exp -> IntConstant]]()**

<PUSHI> Exp.VALUE

**VALUE[[ChaLiteral: Exp -> CharConstant]]()**

<PUSHB> Exp.VALUE

**VALUE[[RealLiteral: Exp -> RealConstant]]()**

<PUSHF> Exp.VALUE

**VALUE[[Variable: Exp -> ID]]()**

ADDRESS[[EXP]]()

<LOAD> Exp.Type.Suffix()

**VALUE[[Arithmetic: Exp1 -> Exp2 Exp3 ]]()**

VALUE[[Exp2]]()

cg.convert(Exp2.Type,Exp1.Type)

VALUE[[Exp3]]()

cg.convert(Exp3.Type,Exp1.Type)

cg.arithmetic(Exp1.operator,Exp1.Type)

**VALUE[[Comparison: Exp1 -> Exp2 Exp3 ]]()**

supertype = Exp2.Type.SuperType(Exp3.Type)

VALUE[[Exp2]]()

cg.convert(Exp2.Type,supertype)

VALUE[[Exp3]]()

cg.convert(Exp3.Type,supertype)

cg.comparison(Exp1.operator,supertype)

**VALUE[[Cast: Exp1 -> CastType Exp2]]()**

VALUE[[Exp2]]()

cg.cast(Exp2.Type, CastType)

**VALUE[[Logical: Exp1 -> Exp2 Exp3 ]]()**

VALUE[[Exp2]]()

VALUE[[Exp3]]()

cg.logig(Exp1.operator)

**VALUE[[UnaryNot: Exp1 -> Exp2]]()**

VALUE[[Exp2]]()

<NOT>

**VALUE[[FieldAcces: Exp1 -> Exp2 ID]]()**

ADDRESS[[Exp1]]()

<LOAD>Exp1.Type.Suffix()

**VALUE[[Indexing: Exp1 -> Exp2 Exp3 ]]()**

ADDRESS[[EXP1]]()

<LOAD>Exp1.Type.Suffix()

**VALUE[[Invocation: Exp -> Variable Exp\*]]()**

for(Expression e:Exp\*)

VALUE[[e]]()

<CALL> Variable.Name

**ADDRESS[[Variable: Exp -> ID]]()**

if(Exp.Definition.scope == 0)

<PUSHA> Exp.Definition.Offset

else

<PUSH BP>

<PUSHI> Exp.Definition.Offset

<ADDI>

**ADDRESS[[ Indexing: Exp1 -> Exp2 Exp3 ]]()**

ADDRESS[[Exp2]]()

VALUE[[Exp3]]()

<PUSH> Exp1.Type.NumberBytes

<MUL>

<ADD>

**ADDRESS[[FieldAcces: Exp1 -> Exp2 ID]]()**

ADDRESS[[Exp2]]

<PUSH>Exp2.Type.get(ID).Offset

<ADD>

# Ampliaciones