Dartam

Se ha dividido en 6 partes, análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio, optimizaciones y generación de código máquina.

Para el análisis léxico se ha utilizado JFlex. Para el sintáctico CUP. El semántico ha sido hecho a mano.

Debido a constantes trabajos de otras asignaturas y motivos personales de cada integrante del equipo, no hemos podido acabar a tiempo aun después de haberle invertido semanas de esfuerzo. Entregamos este trabajo para recibir retroalimentación del trabajo hecho hasta el momento, para así poder encaminarlo mejor para la siguiente entrega.

Integrantes: Juan Arturo Abaurrea Calafell, Daniel Salanova Dmitriyev, Marta González Juan
Asignatura: 21780 - Compiladores

Índice

Análisis Léxico

Tokens con sus patrones:

```
// reutilizables
sub_digit   = [0-9]
sub_letra   = [A-Za-z] // no confundir con caracter
sub_car     = {sub_digit}|{sub_letra}

// simbolos
sym_parenIzq  = \(
sym_parenDer  = \)
sym_llaveIzq  = \{
sym_llaveDer  = \}
sym_bracketIzq  = \[
sym_bracketDer  = \]
sym_endInstr    = \;
sym_coma     = \,
sym_comillaS = \'
sym_comillaD = \"
sym_punto    = \.

// operadores
op_eq       = \=
op_diferent = \/\=
op_mayor    = \>
op_menor    = \<
op_mayorEq  = (\>\=)|(\=\>)
op_menorEq  = (\<\=)|(\=\<)
op_inc      = \+\+
op_dec      = \-\-
op_sum      = \+
op_res      = \-
op_mul      = \*
op_div      = \/
op_pot      = \*\*
op_porcent  = \%
op_swap     = \<\>
op_or       = \|
op_and      = &
op_mod      = \\
```

```
op_neg      = \¬
op_asig     = \:
op_and_asig = \&\:
op_or_asig  = \|\:
op_sum_asig = \+\:
op_res_asig = \-\:
op_mul_asig = \*\:
op_div_asig = \/\:
op_pot_asig = \*\*\:
op_mod_asig = \\\:
op_cond     = \?
op_arrow    = \-\>

// tipos (void es tipo de retorno pero no de variable)
type_char      = "car"
type_string    = "string"
type_int       = "ent"
type_double    = "real"
type_bool      = "prop"
type_void      = "vacio"

// palabras reservadas (const esta entre type y kw)
kw_const     = "inmut"  // inmutable
kw_main      = "inicio"
kw_args      = "argumentos"
kw_method    = "f"
kw_if        = "si"
kw_elif      = "sino"
kw_else      = "no"
kw_switch    = "select"
kw_case      = "caso"
kw_default   = "_"
kw_while     = "loop"
kw_doLoop    = "do"
kw_return    = "pop"
kw_true      = "cierto"
kw_false     = "falso"
kw_in        = "enter"
kw_out       = "show"
kw_read      = "from"
kw_write     = "into"
kw_tuple     = "tupla"
```

```
// valores
val_decimal = {sub_digit}+
val_binario = 0b[01]+
val_octal   = 0o[0-7]+
val_hex     = 0x[A-Fa-f0-9]+
val_real    = {val_decimal}?\.{val_decimal}?([Ee]{val_decimal})?
val_prop    = {kw_true}|{kw_false}
val_char    = {sym_comillaS}{sub_car}{sym_comillaS}
val_cadena  = {sym_comillaD}{sub_car}*{sym_comillaD}
id          = ({sub_letra}|_)({sub_car}|_)*

// casos especiales
espacioBlanco = [ \t]+
finLinea = [\r\n]+
comentarLinea  = "\/\/"
comentarBloque = "#" // tanto para el inicio como para el final
comentario = {comentarLinea}.*|{comentarBloque}[^]*{comentarBloque}
```

Reglas/acciones

```
// operadores
{op_and_asig}               { tokens += "OP_AND_ASSIGNMENT: "+yytext()+"\n";
return symbol(ParserSym.AS_ANDA, yytext()); }
{op_or_asig}                { tokens += "OP_OR_ASSIGNMENT: "+yytext()+"\n";
return symbol(ParserSym.AS_ORA, yytext()); }
{op_sum_asig}               { tokens += "OP_SUM_ASSIGNMENT: "+yytext()+"\n";
return symbol(ParserSym.AS_ADDA, yytext()); }
{op_res_asig}               { tokens += "OP_RES_ASSIGNMENT: "+yytext()+"\n";
return symbol(ParserSym.AS_SUBA, yytext()); }
{op_mul_asig}               { tokens += "OP_MUL_ASSIGNMENT: "+yytext()+"\n";
return symbol(ParserSym.AS_MULA, yytext()); }
{op_div_asig}               { tokens += "OP_DIV_ASSIGNMENT: "+yytext()+"\n";
return symbol(ParserSym.AS_DIVA, yytext()); }
{op_pot_asig}               { tokens += "OP_POT_ASSIGNMENT: "+yytext()+"\n";
return symbol(ParserSym.AS_POTA, yytext()); }
{op_mod_asig}               { tokens += "OP_MOD_ASSIGNMENT: "+yytext()+"\n";
return symbol(ParserSym.AS_MODA, yytext()); }
{op_inc}                    { tokens += "OP_INC: "+yytext()+"\n"; return
symbol(ParserSym.OP_INC, yytext()); }
```

```
{op_dec}                    { tokens += "OP_DEC: "+yytext()+"\n"; return
symbol(ParserSym.OP_DEC, yytext()); }
{op_sum}                    { tokens += "OP_SUM: "+yytext()+"\n"; return
symbol(ParserSym.OP_ADD, yytext()); }
{op_res}                    { tokens += "OP_RES: "+yytext()+"\n"; return
symbol(ParserSym.OP_SUB, yytext()); }
{op_mul}                    { tokens += "OP_MUL: "+yytext()+"\n"; return
symbol(ParserSym.OP_MUL, yytext()); }
{op_div}                    { tokens += "OP_DIV: "+yytext()+"\n"; return
symbol(ParserSym.OP_DIV, yytext()); }
{op_mod}                    { tokens += "OP_MOD: "+yytext()+"\n"; return
symbol(ParserSym.OP_MOD, yytext()); }
{op_eq}                     { tokens += "OP_EQ: "+yytext()+"\n"; return
symbol(ParserSym.OP_EQ, yytext()); }
{op_mayorEq}                { tokens += "OP_MAYOREQ: "+yytext()+"\n"; return
symbol(ParserSym.OP_BEQ, yytext()); }
{op_mayor}                  { tokens += "OP_MAYOR: "+yytext()+"\n"; return
symbol(ParserSym.OP_BT, yytext()); }
{op_menorEq}                { tokens += "OP_MENOREQ: "+yytext()+"\n"; return
symbol(ParserSym.OP_LEQ, yytext()); }
{op_menor}                  { tokens += "OP_MENOR: "+yytext()+"\n"; return
symbol(ParserSym.OP_LT, yytext()); }
{op_diferent}               { tokens += "OP_DIFERENT: "+yytext()+"\n"; return
symbol(ParserSym.OP_NEQ, yytext()); }
{op_pot}                    { tokens += "OP_POTENCIA: "+yytext()+"\n"; return
symbol(ParserSym.OP_POT, yytext()); }
{op_porcent}                { tokens += "OP_PORCENT: "+yytext()+"\n"; return
symbol(ParserSym.OP_PCT, yytext()); }
{op_neg}                    { tokens += "OP_NEG: "+yytext()+"\n"; return
symbol(ParserSym.OP_NOT, yytext()); }
{op_or}                     { tokens += "OP_OR: "+yytext()+"\n"; return
symbol(ParserSym.OP_OR, yytext()); }
{op_and}                    { tokens += "OP_AND: "+yytext()+"\n"; return
symbol(ParserSym.OP_AND, yytext()); }
{op_asig}                   { tokens += "OP_ASIG: "+yytext()+"\n"; return
symbol(ParserSym.AS_ASSIGN, yytext()); }
{op_swap}                   { tokens += "OP_SWAP: "+yytext()+"\n"; return
symbol(ParserSym.OP_SWAP, yytext()); }
{op_cond}                   { tokens += "OP_COND: "+yytext()+"\n"; return
symbol(ParserSym.OP_COND, yytext()); }
{op_arrow}                  { tokens += "OP_ARROW: "+yytext()+"\n"; return
symbol(ParserSym.ARROW, yytext()); }
```

```
// tipos
{type_double}           { tokens += "TYPE_DOUBLE: "+yytext()+"\n"; return
symbol(ParserSym.KW_DOUBLE, yytext()); }
{type_int}              { tokens += "TYPE_INT: "+yytext()+"\n"; return
symbol(ParserSym.KW_INT, yytext()); }
{type_char}             { tokens += "TYPE_CHAR: "+yytext()+"\n"; return
symbol(ParserSym.KW_CHAR, yytext()); }
{type_bool}             { tokens += "TYPE_BOOL: "+yytext()+"\n"; return
symbol(ParserSym.KW_BOOL, yytext()); }
{type_void}             { tokens += "TYPE_VOID: "+yytext()+"\n"; return
symbol(ParserSym.KW_VOID, yytext()); }
{type_string}           { tokens += "TYPE_STRING: "+yytext()+"\n"; return
symbol(ParserSym.KW_STRING, yytext()); }

// simbolos
{sym_parenIzq}          { tokens += "SYM_LPAREN: "+yytext()+"\n"; return
symbol(ParserSym.LPAREN, yytext()); }
{sym_parenDer}          { tokens += "SYM_RPAREN: "+yytext()+"\n"; return
symbol(ParserSym.RPAREN, yytext()); }
{sym_llaveIzq}          { tokens += "SYM_LKEY: "+yytext()+"\n"; return
symbol(ParserSym.LKEY, yytext()); }
{sym_llaveDer}          { tokens += "SYM_RKEY: "+yytext()+"\n"; return
symbol(ParserSym.RKEY, yytext()); }
{sym_bracketIzq}        { tokens += "SYM_LBRACKET: "+yytext()+"\n"; return
symbol(ParserSym.LBRACKET, yytext()); }
{sym_bracketDer}        { tokens += "SYM_RBRACKET: "+yytext()+"\n"; return
symbol(ParserSym.RBRACKET, yytext()); }
{sym_endInstr}          { tokens += "SYM_ENDINSTR: "+yytext()+"\n"; return
symbol(ParserSym.ENDINSTR, yytext()); }
{sym_coma}              { tokens += "SYM_COMMA: "+yytext()+"\n"; return
symbol(ParserSym.COMMA, yytext()); }
{sym_comillaS}          {}//{ tokens += "SYM_SQUOTE: "+yytext()+"\n"; return
symbol(ParserSym.SQUOTE, yytext()); }
{sym_comillaD}          {}//{ tokens += "SYM_DQUOTE: "+yytext()+"\n"; return
symbol(ParserSym.DQUOTE, yytext()); }
{sym_punto}             { tokens += "SYM_PUNTO: "+yytext()+"\n"; return
symbol(ParserSym.OP_MEMBER, yytext()); }

// keywords
{kw_main}               { tokens += "KW_MAIN: "+yytext()+"\n"; return
symbol(ParserSym.KW_MAIN, yytext()); }
```

```
{kw_args}               { tokens += "KW_ARGS: "+yytext()+"\n"; return
symbol(ParserSym.KW_ARGS, yytext()); }
{kw_method}             { tokens += "KW_METHOD: "+yytext()+"\n"; return
symbol(ParserSym.KW_METHOD, yytext()); }
{kw_const}              { tokens += "KW_CONST: "+yytext()+"\n"; return
symbol(ParserSym.KW_CONST, yytext()); }
{kw_if}                 { tokens += "KW_IF: "+yytext()+"\n"; return
symbol(ParserSym.KW_IF, yytext()); }
{kw_elif}               { tokens += "KW_ELIF: "+yytext()+"\n"; return
symbol(ParserSym.KW_ELIF, yytext()); }
{kw_else}               { tokens += "KW_ELSE: "+yytext()+"\n"; return
symbol(ParserSym.KW_ELSE, yytext()); }
{kw_while}              { tokens += "KW_WHILE: "+yytext()+"\n"; return
symbol(ParserSym.KW_LOOP, yytext()); }
{kw_doLoop}             { tokens += "KW_DO: "+yytext()+"\n"; return
symbol(ParserSym.KW_DO, yytext()); }
{kw_switch}             { tokens += "KW_SWITCH: "+yytext()+"\n"; return
symbol(ParserSym.KW_SWITCH, yytext()); }
{kw_case}               { tokens += "KW_CASE: "+yytext()+"\n"; return
symbol(ParserSym.KW_CASE, yytext()); }
{kw_default}            { tokens += "KW_DEFAULT: "+yytext()+"\n"; return
symbol(ParserSym.KW_DEFAULT, yytext()); }
{kw_return}             { tokens += "KW_RETURN: "+yytext()+"\n"; return
symbol(ParserSym.KW_RETURN, yytext()); }
{kw_in}                 { tokens += "ENTER: "+yytext()+"\n"; return
symbol(ParserSym.ENTER, yytext()); }
{kw_out}                { tokens += "SHOW: "+yytext()+"\n"; return
symbol(ParserSym.SHOW, yytext()); }
{kw_read}               { tokens += "FROM: "+yytext()+"\n"; return
symbol(ParserSym.FROM, yytext()); }
{kw_write}              { tokens += "INTO: "+yytext()+"\n"; return
symbol(ParserSym.INTO, yytext()); }
{kw_tuple}              { tokens += "KW_TUPLE: "+yytext()+"\n"; return
symbol(ParserSym.KW_TUPLE, yytext()); }

// valores
{val_binario}       { tokens += "VAL_BINARIO: "+yytext()+"\n"; return
symbol(ParserSym.ENT, Integer.parseInt(yytext().substring(2,
yytext().length()),2)); }
{val_hex}           { tokens += "VAL_HEX: "+yytext()+"\n"; return
symbol(ParserSym.ENT, Integer.parseInt(yytext().substring(2,
yytext().length()),16)); }
```

```
{val_octal}         { tokens += "VAL_OCTAL: "+yytext()+"\n"; return
symbol(ParserSym.ENT, Integer.parseInt(yytext().substring(2,
yytext().length()),8)); }
{val_decimal}       { tokens += "VAL_DECIMAL: "+yytext()+"\n"; return
symbol(ParserSym.ENT, Integer.parseInt(yytext())); }
{val_real}          { tokens += "VAL_REAL: "+yytext()+"\n"; return
symbol(ParserSym.REAL, Double.parseDouble(yytext())); }
{val_char}          { tokens += "VAL_CHAR: "+yytext()+"\n"; return
symbol(ParserSym.CAR, yytext().charAt(0)); }
{val_prop}          { tokens += "VAL_PROP: "+yytext()+"\n"; return
symbol(ParserSym.PROP, "cierto".equals(yytext())); }
{val_cadena}        { tokens += "VAL_CADENA: "+yytext()+"\n"; return
symbol(ParserSym.STRING, yytext()); }
{id}                { tokens += "ID: "+yytext()+"\n"; return
symbol(ParserSym.ID, yytext()); }

// casos especiales
{espacioBlanco}        { /* No fer res amb els espais */  }
{comentario}           { /* No fer res amb els comentaris */  }
{finLinea}             {}//{ tokens += "FIN_LINEA: \n"; return
symbol(ParserSym.ENDLINE); }
[^]                    { errores += errorToString();
System.err.println(errorToString()); return symbol(ParserSym.error); }
```

Análisis Sintáctico

Gramática:

```
SCRIPT ::= SCRIPT_ELEMENTO:et1 SCRIPT:et2       {: RESULT = new
SymbolScript(et1, et2, et1xleft, et1xright); :}
        | MAIN:et                               {: RESULT = new
SymbolScript(et, etxleft, etxright); :}
        ;

SCRIPT_ELEMENTO ::= KW_METHOD:et1 TIPO_RETORNO:et2 ID:et3 LPAREN PARAMS:et4
RPAREN LKEY BODY:et5 RKEY   {: RESULT = new SymbolScriptElemento(et2, et3,
et4, et5, et1xleft, et1xright); :}
        | DECS:et                                                       {:
RESULT = new SymbolScriptElemento(et, etxleft, etxright); :}
        | KW_TUPLE:et1 ID:et2 LKEY MIEMBROS_TUPLA:et3 RKEY     {: RESULT = new
SymbolScriptElemento(et2, et3, et1xleft, et1xright); :}
        ;

TIPO_RETORNO ::= TIPO:et            {: RESULT = new SymbolTipoRetorno(et,
etxleft, etxright); :}
        | KW_VOID:et                    {: RESULT = new
SymbolTipoRetorno(etxleft, etxright); :}
        ;

MIEMBROS_TUPLA ::= DECS:et1 MIEMBROS_TUPLA:et2         {:  RESULT = new
SymbolMiembrosTupla(et1,et2, et1xleft, et1xright); :}
        |                                              {:  RESULT = new
SymbolMiembrosTupla(); :}
        ;

MAIN ::= KW_METHOD:pos KW_VOID KW_MAIN:nombre LPAREN KW_STRING LBRACKET:l
RBRACKET:r KW_ARGS:args RPAREN LKEY BODY:et RKEY  {: RESULT = new
SymbolMain(nombre, args, l, r, et, posxleft, posxright); :}
        | MAIN:et1 SCRIPT_ELEMENTO:et2     {: RESULT = new SymbolMain(et1,et2,
et1xleft, et1xright); :}
        ;

BODY ::= METODO_ELEMENTO:et1 BODY:et2   {: RESULT = new SymbolBody(et1, et2,
et1xleft, et1xright); :}
```

```
        |                                {: RESULT = new SymbolBody(); :}
        ;

TIPO ::= TIPO_PRIMITIVO:t                       {: RESULT = new SymbolTipo(t,
txleft, txright); :}
        | TIPO_PRIMITIVO:t DIMENSIONES:d        {: RESULT = new SymbolTipo(t,
d, txleft, txright); :}
        | KW_TUPLE:t ID:i                       {: RESULT = new SymbolTipo(i,
txleft, txright); :}
        | KW_TUPLE:t ID:i DIMENSIONES:d         {: RESULT = new SymbolTipo(i,
d, txleft, txright); :}
        ;

TIPO_PRIMITIVO ::= KW_BOOL:et              {: RESULT = new
SymbolTipoPrimitivo(et, etxleft, etxright); :}
        | KW_INT:et                        {: RESULT = new
SymbolTipoPrimitivo(et, etxleft, etxright); :}
        | KW_DOUBLE:et                     {: RESULT = new
SymbolTipoPrimitivo(et, etxleft, etxright); :}
        | KW_CHAR:et                       {: RESULT = new
SymbolTipoPrimitivo(et, etxleft, etxright); :}
        | KW_STRING:et                     {: RESULT = new
SymbolTipoPrimitivo(et, etxleft, etxright); :}
        ;

PARAMS ::= PARAMSLISTA:et                               {: RESULT = new
SymbolParams(et, etxleft, etxright); :}
        |                                               {: RESULT = new
SymbolParams(); :}
        ;

PARAMSLISTA ::= TIPO:et1 ID:id COMMA PARAMSLISTA:sig          {: RESULT = new
SymbolParamsLista(et1, id, sig, et1xleft, et1xright); :}
        | TIPO:et ID:id                          {: RESULT = new
SymbolParamsLista(et, id, etxleft, etxright); :}
        ;

DECS ::= KW_CONST:et1 TIPO:et2 DEC_ASIG_LISTA:et3 ENDINSTR     {: RESULT = new
SymbolDecs(true,et2,et3, et1xleft, et1xright); :}
        | TIPO:et1 DEC_ASIG_LISTA:et2 ENDINSTR                 {: RESULT = new
SymbolDecs(false, et1,et2, et1xleft, et1xright); :}
        ;
```

```
DIMENSIONES ::= LBRACKET:l OPERAND:et1 RBRACKET:r DIMENSIONES:et2   {: RESULT
= new SymbolDimensiones(et1, et2, l, r, et1xleft, et1xright); :}
        | LBRACKET:l OPERAND:et1 RBRACKET:r                          {: RESULT
= new SymbolDimensiones(et1, l, r, et1xleft, et1xright); :}
        ;

DEC_ASIG_LISTA ::= ID:et1 ASIG_BASICO:et2 COMMA DEC_ASIG_LISTA:et3       {:
RESULT = new SymbolDecAsigLista(et1,et2,et3, et1xleft, et1xright); :}
        | ID:et1 ASIG_BASICO:et2                                    {: RESULT =
new SymbolDecAsigLista(et1,et2, et1xleft, et1xright); :}
        ;

ASIG_BASICO ::= AS_ASSIGN OPERAND:et   {: RESULT = new SymbolAsigBasico(et,
etxleft, etxright); :}
        |                       {: RESULT = new SymbolAsigBasico(); :}
        ;

METODO_ELEMENTO ::= INSTR:et    {: RESULT = new SymbolMetodoElemento(et,
etxleft, etxright); :}
        | LOOP:et               {: RESULT = new SymbolMetodoElemento(et,
etxleft, etxright); :}
        | IF:et                 {: RESULT = new SymbolMetodoElemento(et,
etxleft, etxright); :}
        | SWITCH:et             {: RESULT = new SymbolMetodoElemento(et,
etxleft, etxright); :}
        ;

INSTR ::= FCALL:et ENDINSTR    {: RESULT = new SymbolInstr(et,etxleft,
etxright); :}
        | RETURN:et            {: RESULT = new SymbolInstr(et,etxleft,
etxright); :}
        | DECS:et              {: RESULT = new SymbolInstr(et,etxleft,
etxright); :}
        | ASIGS:et             {: RESULT = new SymbolInstr(et,etxleft,
etxright); :}
        | SWAP:et              {: RESULT = new SymbolInstr(et,etxleft,
etxright); :}
        ;

FCALL ::= METODO_NOMBRE:et1 LPAREN OPERANDS_LISTA:et2 RPAREN   {: RESULT = new
SymbolFCall(et1, et2, et1xleft, et1xright); :}
```

```
        | METODO_NOMBRE:et1 LPAREN RPAREN   {: RESULT = new SymbolFCall(et1,
et1xleft, et1xright); :}
        ;

METODO_NOMBRE ::= ID:et                {: RESULT = new
SymbolMetodoNombre(null, et, etxleft, etxright); :}
        | ENTER:et                     {: RESULT = new
SymbolMetodoNombre(ParserSym.ENTER, et, etxleft, etxright); :}
        | SHOW:et                     {: RESULT = new
SymbolMetodoNombre(ParserSym.SHOW, et, etxleft, etxright); :}
        | INTO:et                   {: RESULT = new
SymbolMetodoNombre(ParserSym.INTO, et, etxleft, etxright); :}
        | FROM:et                    {: RESULT = new
SymbolMetodoNombre(ParserSym.FROM, et, etxleft, etxright); :}
        ;

OPERANDS_LISTA ::= OPERAND:et COMMA OPERANDS_LISTA:ol     {: RESULT = new
SymbolOperandsLista(et, ol, etxleft, etxright); :}
        | OPERAND:et                                    {: RESULT = new
SymbolOperandsLista(et, etxleft, etxright); :}
        ;

RETURN ::= KW_RETURN ENDINSTR                             {: :} //Con este
que hacemos?
        | KW_RETURN OPERAND:et ENDINSTR                   {: RESULT = new
SymbolReturn(et, etxleft, etxright); :}
        ;

SWAP ::= ID:et1 OP_SWAP ID:et2 ENDINSTR           {: RESULT = new
SymbolSwap(et1, et2, et1xleft, et1xright); :}
        ;

ASIGS ::= ASIG:et1 COMMA ASIGS:et2                            {: RESULT =
new SymbolAsigs(et1, et2, et1xleft, et1xright); :}
        | ASIG:et ENDINSTR                                    {: RESULT =
new SymbolAsigs(et, etxleft, etxright); :}
        ;

ASIG ::= ID:et ASIG_OP:aop OPERAND:val                                {:
RESULT = new SymbolAsig(et, aop, val, etxleft, etxright); :}
        | ID:et1 LBRACKET OPERAND:et2 RBRACKET ASIG_OP:aop OPERAND:val  {:
RESULT = new SymbolAsig(et1, et2, aop, val, et1xleft, et1xright); :}
```

```
        | ID:et1 OP_MEMBER ID:et2 ASIG_OP:aop OPERAND:val                {:
RESULT = new SymbolAsig(et1, et2, aop, val, et1xleft, et1xright); :}
        | ID:et1 OP_INC:et2             {: RESULT = new SymbolAsig(true,
ParserSym.OP_INC, et1, et2, et1xleft, et1xright); :}  %prec PREC_R_U_EXP
        | ID:et1 OP_DEC:et2             {: RESULT = new SymbolAsig(true,
ParserSym.OP_DEC, et1, et2, et1xleft, et1xright); :}  %prec PREC_R_U_EXP
        | OP_INC:et1 ID:et2             {: RESULT = new SymbolAsig(false,
ParserSym.OP_INC, et2, et1, et1xleft, et1xright); :} %prec PREC_L_U_EXP
        | OP_DEC:et1 ID:et2             {: RESULT = new SymbolAsig(false,
ParserSym.OP_DEC, et2, et1, et1xleft, et1xright); :} %prec PREC_L_U_EXP
        ;

ASIG_OP ::= AS_ASSIGN:et                {: RESULT = new
SymbolAsigOp(ParserSym.AS_ASSIGN, et, etxleft, etxright); :}
        | AS_ADDA:et                    {: RESULT = new
SymbolAsigOp(ParserSym.AS_ADDA, et, etxleft, etxright); :}
        | AS_SUBA:et                    {: RESULT = new
SymbolAsigOp(ParserSym.AS_SUBA, et, etxleft, etxright); :}
        | AS_MULA:et                    {: RESULT = new
SymbolAsigOp(ParserSym.AS_MULA, et, etxleft, etxright); :}
        | AS_DIVA:et                    {: RESULT = new
SymbolAsigOp(ParserSym.AS_DIVA, et, etxleft, etxright); :}
        | AS_POTA:et                    {: RESULT = new
SymbolAsigOp(ParserSym.AS_POTA, et, etxleft, etxright); :}
        | AS_MODA:et                    {: RESULT = new
SymbolAsigOp(ParserSym.AS_MODA, et, etxleft, etxright); :}
        | AS_ANDA:et                    {: RESULT = new
SymbolAsigOp(ParserSym.AS_ANDA, et, etxleft, etxright); :}
        | AS_ORA:et                     {: RESULT = new
SymbolAsigOp(ParserSym.AS_ORA, et, etxleft, etxright); :}
        ;

OPERAND ::= ATOMIC_EXPRESSION:et        {: RESULT = new SymbolOperand(et,
etxleft, etxright); :}
        | FCALL:et                      {: RESULT = new SymbolOperand(et,
etxleft, etxright); :}
        | LPAREN:pos OPERAND:et RPAREN  {: RESULT = new SymbolOperand(et,
posxleft, posxright); :}
        | UNARY_EXPRESSION:et           {: RESULT = new SymbolOperand(et,
etxleft, etxright); :}
        | BINARY_EXPRESSION:et          {: RESULT = new SymbolOperand(et,
etxleft, etxright); :}
```

```
        | CONDITIONAL_EXPRESSION:et     {: RESULT = new SymbolOperand(et,
etxleft, etxright); :}
        | OPERAND:et1 LBRACKET:l OPERAND:et2 RBRACKET:r {: RESULT = new
SymbolOperand(et1, l, et2, r, et1xleft, et1xright); :}
        | OPERAND:et1 OP_MEMBER ID:et2                  {: RESULT = new
SymbolOperand(et1, et2, et1xleft, et1xright); :}
        | LPAREN:et TIPO_PRIMITIVO:t RPAREN:et2 OPERAND:op  {: RESULT = new
SymbolOperand(t, op, et, et2, etxleft, etxright); :} %prec CASTING
        ;

UNARY_EXPRESSION ::= L_UNARY_OPERATOR:et1 OPERAND:et2                       {:
RESULT = new SymbolUnaryExpression(et1, et2, et1xleft, et1xright); :} %prec
PREC_L_U_EXP
        | OPERAND:et1 R_UNARY_OPERATOR:et2                                  {:
RESULT = new SymbolUnaryExpression(et1, et2, et1xleft, et1xright); :} %prec
PREC_R_U_EXP
        ;

L_UNARY_OPERATOR ::= OP_NOT:et  {: RESULT = new
SymbolLUnaryOperator(ParserSym.OP_NOT, et, etxleft, etxright); :}
        | OP_INC:et             {: RESULT = new
SymbolLUnaryOperator(ParserSym.OP_INC, et, etxleft, etxright); :}
        | OP_DEC:et             {: RESULT = new
SymbolLUnaryOperator(ParserSym.OP_DEC, et, etxleft, etxright); :}
        | OP_ADD:et             {: RESULT = new
SymbolLUnaryOperator(ParserSym.OP_ADD, et, etxleft, etxright); :}
        | OP_SUB:et             {: RESULT = new
SymbolLUnaryOperator(ParserSym.OP_SUB, et, etxleft, etxright); :}
        ;

R_UNARY_OPERATOR ::= OP_PCT:et  {: RESULT = new
SymbolRUnaryOperator(ParserSym.OP_PCT, et, etxleft, etxright); :}
        | OP_INC:et             {: RESULT = new
SymbolRUnaryOperator(ParserSym.OP_INC, et, etxleft, etxright); :}
        | OP_DEC:et             {: RESULT = new
SymbolRUnaryOperator(ParserSym.OP_DEC, et, etxleft, etxright); :}
        ;

BINARY_EXPRESSION ::= OPERAND:et1 BINARY_OPERATOR:et2 OPERAND:et3        {:
RESULT = new SymbolBinaryExpression(et1, et2, et3, et1xleft, et1xright); :}
%prec PREC_B_EXP
        ;
```

```
CONDITIONAL_EXPRESSION ::= OPERAND:et1 OP_COND OPERAND:et2 ARROW OPERAND:et3
{: RESULT = new SymbolConditionalExpression(et1, et2, et3, et1xleft,
et1xright);; :} %prec PREC_C_EXP
        ;

ATOMIC_EXPRESSION ::= ID:et     {: RESULT = new SymbolAtomicExpression(true,
et, etxleft, etxright); :}
        | STRING:et             {: RESULT = new SymbolAtomicExpression(false,
et, etxleft, etxright); :}
        | PROP:et               {: RESULT = new SymbolAtomicExpression(et,
etxleft, etxright); :}
        | ENT:et                {: RESULT = new SymbolAtomicExpression(et,
etxleft, etxright); :}
        | REAL:et               {: RESULT = new SymbolAtomicExpression(et,
etxleft, etxright); :}
        | CAR:et                {: RESULT = new SymbolAtomicExpression(et,
etxleft, etxright); :}
        ;

BINARY_OPERATOR ::= OP_ADD:et   {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_ADD, et, etxleft, etxright); :}
        | OP_SUB:et             {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_SUB, et, etxleft, etxright); :}
        | OP_MUL:et             {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_MUL, et, etxleft, etxright); :}
        | OP_DIV:et             {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_DIV, et, etxleft, etxright); :}
        | OP_MOD:et             {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_MOD, et, etxleft, etxright); :}
        | OP_POT:et             {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_POT, et, etxleft, etxright); :}
        | OP_EQ:et              {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_EQ, et, etxleft, etxright); :}
        | OP_BEQ:et             {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_BEQ, et, etxleft, etxright); :}
        | OP_BT:et              {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_BT, et, etxleft, etxright); :}
        | OP_LEQ:et             {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_LEQ, et, etxleft, etxright); :}
        | OP_LT:et              {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_LT, et, etxleft, etxright); :}
```

```
        | OP_NEQ:et             {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_NEQ, et, etxleft, etxright); :}
        | OP_AND:et             {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_AND, et, etxleft, etxright); :}
        | OP_OR:et              {: RESULT = new
SymbolBinaryOperator(ParserSym.OP_OR, et, etxleft, etxright); :}
        ;

LOOP ::= KW_LOOP:et1 LOOP_COND:et2 LKEY BODY:et3 RKEY                   {:
RESULT = new SymbolLoop(et2, et3, et1xleft, et1xright); :}
        | KW_DO:et1 LKEY BODY:et2 RKEY LOOP_COND:et3 ENDINSTR           {:
RESULT = new SymbolLoop(et2, et3, et1xleft, et1xright); :}
        ;

LOOP_COND ::= OPERAND:et                                                {:
RESULT = new SymbolLoopCond(et, etxleft, etxright); :}
        | DECS:et1 OPERAND:et2 ENDINSTR ASIGS:et3            {: RESULT = new
SymbolLoopCond(et1, et2, et3, et1xleft, et1xright); :}
        ;

IF ::= KW_IF OPERAND:et1 LKEY BODY:et2 RKEY ELIFS:et3 ELSE:et4          {:
RESULT = new SymbolIf(et1, et2, et3, et4, et1xleft, et1xright); :}
        ;

ELIFS ::= ELIF:et1 ELIFS:et2                        {: RESULT = new
SymbolElifs(et1, et2, et1xleft, et1xright); :}
        |                                           {: RESULT = new
SymbolElifs(); :}
        ;

ELIF ::= KW_ELIF OPERAND:et1 LKEY BODY:et2 RKEY     {: RESULT = new
SymbolElif(et1, et2, et1xleft, et1xright); :}
        ;

ELSE ::= KW_ELSE:pos LKEY BODY:et RKEY                  {: RESULT = new
SymbolElse(et, posxleft, posxright); :}
        |                                           {: RESULT = new
SymbolElse(); :}
        ;

SWITCH ::= KW_SWITCH OPERAND:et1 RKEY CASO:et2 PRED:et3 LKEY {: RESULT = new
SymbolSwitch(et1, et2, et3, et1xleft, et1xright); :}
```

```
        ;

CASO ::= CASO:et1 KW_CASE OPERAND:et2 ARROW BODY:et3 {: RESULT = new
SymbolCaso(et1, et2, et3, et1xleft, et1xright); :}
        |                                            {: RESULT = new
SymbolCaso(); :}
        ;

PRED ::= KW_CASE KW_DEFAULT ARROW BODY:et            {: RESULT = new
SymbolPred(et, etxleft, etxright); :}
        ;
```

Métodos para la impresión de errores:

```
    @Override
    public void unrecovered_syntax_error(Symbol cur_token) {
        String causa = "" + cur_token.value;
        if (cur_token.sym == ParserSym.EOF) {
            causa = "No se ha encontrado metodo main. Sintaxis: \n"+
                "f void inicio(string[]argumentos){ # codigo # }\n";
        }
        System.err.println("No se ha podido recuperar del ultimo error.
\nCausa: " + causa);
        done_parsing();
    }

    /**
     * Error sintactico.
    **/
    @Override
    public void syntax_error(Symbol cur_token){
        report_error("Error sintactico: ", cur_token);
    }

    @Override
    public void report_error(String message, Object info) {
        if (cur_token.sym == ParserSym.EOF) {
            return;
        }
        if (info instanceof ComplexSymbol token) {
```

```
            List expected = expected_token_ids();
            String tokens = "";
            for (Object t : expected){
                tokens += ParserSym.terminalNames[(int)t] + ", ";
            }
            if (!tokens.isEmpty()) {
                tokens = "Se esperaba algun lexema de los siguientes tipos: "
+ tokens.substring(0, tokens.length() - 2) + ".\n";
            }
            System.err.println(message + "Desde la linea " +
token.xleft.getLine() + " y columna " + token.xleft.getColumn() + " hasta la
linea " + token.xright.getLine() + " y columna " + token.xright.getColumn() +
". \n"
                    + tokens + "Se ha encontrado '" + token.value + "' de tipo
" + ParserSym.terminalNames[token.sym] + ".\n");
        } else {
            System.err.println(message + "No se esperaba este componente\n: "
+cur_token.value+".");
        }
    }

    @Override
    public void report_fatal_error(String message, Object info) throws
Exception {
        report_error("Error fatal: " + message, info);
        done_parsing();
    }
```

Se ha escogido el análisis sintáctico ascendente con reducciones y desplazamientos ofrecido por CUP porque era para Java y hemos pensado que tenía una buena cantidad de recursos en el aula digital.

Análisis Semántico

Se ha recorrido el árbol generado por el análisis sintáctico a partir de la raíz *SymbolScript* y a cada caso se ha comprobado que la coherencia semántica se mantuviera. Primero se procesan las tuplas, luego las declaraciones y finalmente los métodos independientemente del orden en el script, esto es por si se hace una declaración de una variable de tipo tupla y se utiliza esta variable global dentro de un método. Antes de procesar el interior de los métodos se añaden a la tabla de símbolos para que se puedan llamar entre ellos (A llama a B y B llama a A). El resto de código es una contemplación tras otra de posibles errores generados por el usuario, y si es así se le avisa con el mensaje indicado.

Tabla de Símbolos

La tabla inacabada está inspirada en la de los apuntes, con un cambio, los parámetros de las funciones y los miembros de las tuplas son instancias de *DescripcionSimbolo* en vez de tener su propia clase. Lo hemos hecho así porque no hemos visto el motivo para crear una diferente.

La clase *TablaSimbolos* contiene la propia tabla *td*, el nivel actual *n*, la tabla de ámbitos *ta* y la tabla de expansión *te*. Contiene una clase *Entrada* que representa cada entrada de la tabla de expansión. Están todos los métodos necesarios para poner, quitar y modificar elementos de las tablas, incluidos los relacionados con arrays, tuplas y funciones.

Restricciones:

1. No se pueden hacer asignaciones complejas sobre arrays (+:, -:, …).
2. Los incrementos/decrementos solo se pueden hacer sobre variables primitivas (arr[0]++ y tupl.miem++ no están permitidos).
3. No se pueden realizar varias operaciones de incremento/decremento seguidas (a++-- y ++a-- no están permitidos).
4. No pueden haber dos métodos con el mismo nombre (no se permite sobrecarga).
5. Tuplas y arrays no pueden ser constantes.
6. No se pueden asignar valores a las estructuras (arrays y tuplas), sino a sus elementos (*tupla Pila* **pila1;** *tupla Pila* **pila2: pila1;** no está permitido).
7. Las tuplas solo pueden ser declaradas fuera de los métodos y de otras tuplas (no se puede tener *tupla Arbol* { *tupla Nodo* { ⋯ } }).

Casos de prueba:

Estos son los errores resultantes de la ejecución de cada uno de los ficheros de prueba, almacenados en la carpeta ./scripts:

- En el fichero errLex se produce un error debido a que “ º ” no está dentro del alfabeto del analizador léxico.

- En errSin se produce un error debido a que se espera una definición de tupla, método o variable y se encuentra con un id (inicio). La sintaxis correcta es f vacio inicio…

- El fichero errSem produce un error debido a que se intenta asignar un valor a una variable constante que ya tenía valor asignado.

- Los siguientes archivos persona, matriz y mcd producen errores debido a la falta de depuración en el compilador.