Al no haber anidamiento no hacen falta una pila de tablas de símbolos.

Nos servirá tener:

- Una TSG (General) que se crea al principio y siempre esta

disponible.

- Una TSL (Local) que se crea al comenzar una función y se

destruye al terminar.

El Analizador Semántico se encarga de comprobar el flujo del control (por ejemplo, si el return aparece dentro de una función con el tipo adecuado, o el break en un case, o el continue en una sentencia iterativa...) además de la comprobación de tipos.

Dentro de una función se tiene acceso a las variables locales, a sus argumentos y a las variables globales. Si en una función se declara una variable local o un argumento con el mismo nombre que una variable global, ésta última no es accesible desde dicha función.

Las Tablas de Símbolos (TS) deben estar contenidas en un único fichero. El fichero deberá generarse inmediatamente antes de destruir cada TS, de forma que tenga toda la información recopilada durante el análisis.

Terminales = { var id ; if ( ) while } { switch case break default int bool string return print

prompt function , || && == != < > -= ! = + - \* / % CAD : cteent ctebool }

NoTerminales = { P B T S \_S F H L Q A K X C E \_E R \_R U \_U V \_V W \_W Z \_Z G \_G D }

Axioma = P

Producciones = {

P -> B P {Nada}

P -> F P {Nada}

P -> lambda {Nada}

B -> var {declaración=true} T id{insertaTipoTS(id.lex, T.Tipo); insertaDespTS(id.lex, T.desp);} ; {declaración=false}

{B.tipoRet = tipo\_vacio}

B -> if ( E ) {S.switch = B.switch } S

{B.Tipo = if(E.tipo=bool AND S.tipo=tipo\_OK) then tipo\_OK; B.tipoRet = S.TipoRet;

else tipo\_error}

B -> while ( E ) {C.switch = B.switch } { C } {B.tipoRet=C.tipoRet}

B -> switch ( E ) {D.switch = B.switch} { D } {B.tipoRet=C.tipoRet}

B -> {S.switch = B.switch} S {B.Tipo= S.tipo; B.tipoRet=S.tipoRet}

**T -> int** {T.Tipo = int}

**T -> bool** {T.Tipo = bool}

**T -> string** {T.Tipo = string}

**S -> id \_S** {S.tipo = if(buscaTipo(id.pos) == \_S.tipo->t then tipo\_OK; S.tipoRet=tipo\_vacio;

else if(buscaTipo(id.pos) == \_S.tipo then tipo\_OK; S.tipoRet=tipo\_vacio;

else if(buscaTipo(id.pos) == tipo\_vacio then insertaTipoTS(id.pos, int); if(\_S.tipo==id.tipo) then tipo\_ok ;S.tipoRet=tipo\_vacio; }

else tipo\_error}

S -> return X ; {S.Tipo=tipo\_OK; S.tipoRet = X.tipo}

S -> print ( E ) ; {S.Tipo= if(E.tipo IN {int,bool,string}

then tipo\_OK; S.tipoRet=tipo\_vacio; else tipo\_error)}

(E puede ser de cualquiera de los 3 tipos )

S -> prompt ( id ) ; {S.Tipo= if(buscaTipo(id.pos) IN {int, string}

then tipo\_OK; S.tipoRet=tipo\_vacio; else tipo\_error)}

(id tiene que ser entero o cadena)

S -> break ; {if(S.switch = false then Error(“Break fuera de switch”) else S.tipo = tipo\_OK ); S.tipoRet=tipo\_vacio;} (Puede que haga falta algo para el break)

**\_S -> = E ;** {\_S.tipo = E.tipo}

**\_S -> -= E** ; {\_S.tipo = if(E.tipo==int) then int else tipo\_error }

**\_S -> ( L ) ;** {\_S.tipo=L.tipo}

F -> function {declaración=true} H id {TL=creaTS(); desp=0; declaración=true } ( A ) {declaración=false} {insertaTipoTS(id.pos, A.Tipo->H.Tipo) insertaEtiq(id.pos, nuevaEtiq()) ; declaración=false} { C } {if(C.tipo == tipo\_error ) then Error(“Sentencias incorrectas dentro de la funcion”); if(C.tipoRet != H.tipo) then Error(“ Valor de retorno con tipo incorrecto”); destruirTS(TL)}

H -> T {H.Tipo = T.Tipo}

H -> lambda {H.Tipo = tipo\_vacio}

L -> E Q {L.Tipo = E.tipo x Q.Tipo}

L -> lambda {L.tipo = tipo\_vacio}

Q -> , E Q {Q.Tipo = E.tipo x Q1.Tipo}

Q -> lambda {Q.Tipo = tipo\_vacio}

A -> T id {insertatipoTS(id.pos,T.tipo); insertaDespTS(id.pos,desp); desp=desp++} K {A.Tipo = T.tipo x K.tipo}

A -> lambda {A.Tipo = tipo\_vacio}

K -> , T id {insertatipoTS(id.pos,T.tipo); insertaDespTS(id.pos,desp); desp=desp++} K1 {K.Tipo = T.tipo x K1.tipo}

K -> lambda {K.Tipo = tipo\_vacio}

X -> E {X.Tipo = E.Tipo}

X -> lambda {X.Tipo = tipo\_vacio}

C -> {B.switch = C.switch; C1.switch = B.switch} B C1

{C.Tipo = if(B.Tipo == tipo\_OK) then C1.Tipo else tipo\_error;

C.TipoRet = if(B.tipoRet == C1. tipoRet) then B.tipoRet

Else if(B.tipoRet == tipo\_vacio) then C1.tipoRet

Else if(C1.tipoRet == tipo\_vacio) then B.tipoRet

Else tipo\_error}

C -> lambda {C.tipo = Tipo\_OK; C.TipoRet = tipo\_vacio}

**E -> R \_E** {E.Tipo = if(\_E.Tipo== tipo\_vacio) then R.Tipo else

if(\_E.Tipo== bool AND R.Tipo == bool) then bool else

tipo\_error }

**\_E -> || R \_E** {\_E.Tipo = if( (R.Tipo==bool) AND (\_E1.Tipo==bool OR tipo\_vacio) )then bool else

tipo\_error }

**\_E -> lambda** {E.Tipo = tipo\_vacio }

E puede ser: int, string, bool, tipo\_error

\_E puede ser: bool, vacío, tipo\_error

R -> U \_R {R.Tipo = if(\_R.Tipo== tipo\_vacio) then U.Tipo else

if(\_R.Tipo== bool AND U.Tipo == bool) then bool else

tipo\_error }

**\_R -> && U \_R** {\_R.Tipo = if( (U.Tipo==bool) AND (\_R1.Tipo==bool OR tipo\_vacio) )then bool else tipo\_error }

**\_R -> lambda**  {R.Tipo = tipo\_vacio }

R puede ser: int, string, bool, tipo\_error

\_R puede ser: bool, vacío, tipo\_error

**U -> V \_U**  {U.Tipo = if(\_U.Tipo== tipo\_vacio) then V.Tipo else

if(\_U.Tipo== bool AND V.Tipo == int) then bool else

tipo\_error }

\_**U -> == V \_U** {\_U.Tipo=if( (V.Tipo==int) AND (\_U1.Tipo==tipo\_vacio) )then bool else

tipo\_error }

\_**U -> != V \_U**  {\_U.Tipo=if( (V.Tipo==int) AND (\_U1.Tipo==tipo\_vacio) )then bool else

tipo\_error }

**\_U -> lambda** {\_U.Tipo = tipo\_vacio }

U puede ser: int, string, bool, tipo\_error

\_U puede ser: bool, vacío, tipo\_error

**V -> W \_V** {V.Tipo = if(\_V.Tipo== tipo\_vacio) then W.Tipo else

if(\_V.Tipo== bool AND W.Tipo == int) then bool else

tipo\_error }

**\_V -> > W \_V** {\_V.Tipo=if( (W.Tipo==int) AND (\_V1.Tipo==tipo\_vacio) )then bool else

tipo\_error }

\_**V -> < W \_V** {\_V.Tipo=if( (W.Tipo==int) AND (\_V1.Tipo==tipo\_vacio) )then bool else

tipo\_error }

**\_V -> lambda**  {\_V.Tipo = tipo\_vacio }

V puede ser: int, string, bool, tipo\_error

\_V puede ser: bool, vacío, tipo\_error

**W -> Z \_W** {W.Tipo = if(\_W.Tipo== tipo\_vacio) then Z.Tipo else

if(\_W.Tipo== int AND Z.Tipo == int) then int else

tipo\_error }

**\_W -> + Z \_W** {\_W.Tipo = if( (Z.Tipo==int) AND (\_W1.Tipo==int OR tipo\_vacio) )then int else

tipo\_error }

**\_W -> - Z \_W** {\_W.Tipo = if( (Z.Tipo==int) AND (\_W1.Tipo==int OR tipo\_vacio) )then int else

tipo\_error }

**\_W -> lambda** {\_W.Tipo = tipo\_vacio }   
W puede ser: int, string, bool, tipo\_error

\_W puede ser: int, vacío, tipo\_error

**Z -> G \_Z** {Z.Tipo = if(\_Z.Tipo== tipo\_vacio) then G.Tipo else

if(\_Z.Tipo== int AND G.Tipo == int) then int else

tipo\_error }

**\_Z -> \* G \_Z** {\_Z.Tipo = if( (G.Tipo==int) AND (\_Z1.Tipo==int OR tipo\_vacio) ) then int else

tipo\_error }

**\_Z -> / G \_Z** {\_Z.Tipo = if( (G.Tipo==int) AND (\_Z1.Tipo==int OR tipo\_vacio) ) then int else

tipo\_error }

**\_Z -> % G \_Z1** {\_Z.Tipo = if( (G.Tipo==int) AND (\_Z1.Tipo==int OR tipo\_vacio) ) then int else

tipo\_error }

**\_Z -> lambda** {\_Z.Tipo = tipo\_vacio }

Z puede ser: int, string, bool, tipo\_error

\_Z puede ser: int, tipo\_vacio, tipo\_error

**G -> ! G1** {G.Tipo = if(G1.Tipo==bool) then bool else tipo\_error}

**G -> id \_G** {G.Tipo = if(\_G.Tipo== tipo\_vacio) then buscaTipoTS(id.pos) else

If (buscaTipoTS(id.pos)==\_G.Tipo->t then t else tipo\_error) }

**G -> ( E )** {G.Tipo = E.Tipo}

**G -> cteent**  {G.Tipo = int}

**G -> CAD** {G.Tipo = string}

**G -> ctebool** {G.Tipo = bool}

**\_G -> ( L )** {\_G.Tipo = L.Tipo}

**\_G -> lambda**  {\_G.Tipo = tipo\_vacio }

G puede ser: int, string, bool, tipo\_error

\_G puede ser: tipo\_vacio, arg1 x arg2 x … x argN

D -> case cteent : {C.switch = D.switch.} C D1 {D.tipo = if(C.tipo == tipo\_OK AND D1.tipo = tipo\_ok then tipo\_OK else tipo\_error)}

D -> default : {C.switch = D.switch.} C {D.tipo = C.tipo}

}