Memoria práctica 4:

Procesadores de Lenguajes:

Integrante: Suhuai Chen

Indice:

1. Vinculación

```
var ts // La tabla de símbolos

vincula(prog(Bloq)):
        ts = creaTS() // Crea una tabla de símbolos vacía.
        vincula(Bloq)

vincula(bloq(DecsOp,InstrsOp):
        abreAmbito(ts)
        vincula(DecsOp)
        vincula(InstrsOp)
        cierraAmbito(ts)

vincula(si_decs(Decs)):
        vincula1(Decs)
        vincula2(Decs)

vincula(no_decs()): noop
```

1.1. Declaración primera pasada

```
vincula1(muchas_decs(Decs,Dec)):
        vincula1(Decs)
        vincula1(Dec)

vincula1(una_dec(Dec)):
        vincula1(Dec)

vincula1(dec_variable(Tipo,Id)):
        vincula1(Tipo)
        if contiene(ts,Id) then
                error
        else
                inserta(ts,Id,$)
        end if



vincula1(dec_tipo(Tipo,Id)):
        vincula1(Tipo)
        if contiene(ts,Id) then
                error
        else
                inserta(ts,Id,$)
        end if

vincula1(dec_proc(Id, ParsFOp, Bloq)):
        if contiene(ts,Id) then
                error
        else
                // el nombre del Proc queda en el ámbito del padre
                inserta(ts,Id,$)
                abreAmbito(ts)
                vincula1(ParsFOP)
                vincula2(ParsFOP)
                vincula(Bloq)
                cierraAmbito(ts)
        end if

vincula1(si_parsF(ParsF)):
        vincula1(ParsF)
```

```
vincula1(no_parsF()): noop

vincula1(muchos_parsF(ParsF, ParF)):
        vincula1(ParsF)
        vincula1(ParF)

vincula1(un_parsF(ParF)):
        vincula1(ParF)

vincula1(paramF(id, Tipo)):
        vincula1(Tipo)
        If contiene(ts, id) then
                error
        else
                inserta(ts, id, $)
endIf


vincula1(param(id, Tipo)):
        vincula1(Tipo)
        If contiene(ts, id) then
                error
        else
                inserta(ts, id, $)
endIf



//vinculacion Tipo

vincula1 (tipo_struct(Campos)):
        vincula1(Campos)

vincula1(muchos_campos(Campos, Campo)):
        vincula1(Campos)
        vincula1(Campo)

vincula1(un_campo(Campo)):
        vincula1(Campo)

vincula1(crea_campo(Tipo, id)):
        vincula1(Tipo)
```

```
vincula1(tipo_iden(id)):
        $.vinculo = vinculoDe(ts, id)
        If $.vinculo != dec_tipo(_,_) then      // dec_tipo(_,_): declared customized type
                error
        end if


vincula1 tipo_circum(Tipo):                     // ref(_) : customized type
        If Tipo != tipo_iden() then
                vincula1(Tipo)
        end if

vincula1 tipo_lista(Tipo, literalEntero): noop
        vincula(Tipo)

vincula1 tipo_int(): noop
vincula1 tipo_real(): noop
vincula1 tipo_bool(): noop
vincula1 tipo_string(): noop
```

1.2 Declaración Segunda Pasada

```
vincula2(muchas_decs(Decs,Dec)):
        vincula2(Decs)
        vincula2(Dec)

vincula2(una_dec(Dec)):
        vincula2(Dec)

vincula2(dec_variable(Tipo,Id)):
        vincula2(Tipo)

vincula2(dec_tipo(Tipo,Id)):
        vincula2(Tipo)

vincula2(dec_proc(Id, ParsFOp, Bloq)): noop

vincula2(si_parsF(ParsF)):
        vincula2(ParsF)

vincula2(no_parsF()): noop
```

```
vincula2(muchos_parsF(ParsF, ParF)):
        vincula2(ParsF)
        vincula2(ParF)

vincula2(un_parsF()):
        vincula2(ParF)

vincula2(paramF(id, Tipo)):
        vincula2(Tipo)

vincula2(param(id, Tipo)):
        vincula2(Tipo)


//vinculacion Tipo

vincula2 (tipo_struct(Campos)):
        vincula2(Campos)

vincula2(muchos_campos(Campos, Campo)):
        vincula2(Campos)
        vincula2(Campo)

vincula2(un_campo(Campo)):
        vincula2(Campo)

vincula2(crea_campo(Tipo, id)):
        vincula2(Tipo)

vincula2(tipo_iden(id)): noop

vincula2 tipo_circum(Tipo):
        If Tipo == tipo_iden(Id) then
                Tipo.vinculo = vinculoDe(ts, Id)
                If Tipo.vinculo != dec_tipo(_,_) then      // can also just check vinculo null
                        error
                end if
        else
                vincula2(Tipo)
        end if

vincula2 tipo_lista(Tipo, literalEntero): noop
vincula2 tipo_int(): noop
vincula2 tipo_real(): noop
```

vincula2 tipo_bool(): noop
vincula2 tipo_string(): noop

1.3 Instruccion

vincula(**si_instrs**(Instrs)):
    vincula(Instrs)

vincula(**no_instrs**(Instrs)): noop

vincula(**muchas_instrs**(Instrs, Instr)):
    vinculas(Instrs)
    vincula(Instr)

vinculas(**una_instr**(Instr)):
    vincula(Instr)

vincula(**instr_eval**(Exp)):
    vincula(Exp)

vincula(**instr_if**(Exp, Bloq)):
    vincula(Exp)
    vincula(Bloq)

vincula(**instr_ifelse**(Exp, Bloq1, Bloq2)):
    vincula(Exp)
    vincula(Bloq1)
    vincula(Bloq2)

vincula(**instr_while**(Exp, Bloq)):
    vincula(Exp)
    vincula(Bloq)

vincula(**instr_read**(Exp)):
    vincula(Exp)

vincula(**intr_write**(Exp)):
    vincula(Exp)

vincula(**instr_nl**()):
    noop

vincula(**instr_new**(Exp)):
    vincula(Exp)

```
vincula(instr_del(Exp)):
        vincula(Exp)

vincula(instr_call(Id, ParsReOp)):
        if vinculoDe(ts, Id) != dec_proc(_,_,_)
                error
        else
                vincula(ParsReOp)
                // inserta(ts,Id,$)
        end if


vincula(instr_bloque(Bloq)):
        vincula(Bloq)

//parametroReales
vincula(si_parsRe(ParsRe)):
        vinclula(ParsRe)

vincula(no_parsRe()): noop

vincula(muchos_parsRe(ParsRe, Exp)):
        vincula(ParsRe)
        vinclula(Exp)

vincula(un_parRe(Exp)):
        vinclula(Exp)

//Exp
vincula(iden(Id)):
        $.vinculo = vinculoDe(ts,Id) // Recupera la referencia asociada a id en la tabla de símbolos ts. Si no está, devuelve ⊥.
        if $.vinculo == ⊥ then
                error
        end if

// PRIORIDAD 0
vincula(asig(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)

// PRIORIDAD 1
vincula(menorI(Opnd0,Opnd1)):
        vincula(Opnd0)
```

```
        vincula(Opnd1)
vincula(menor(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)
vincula(mayorl(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)
vincula(mayor(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)
vincula(igual(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)

vincula(distint(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)

// PRIORIDAD 2
vincula(suma(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)
vincula(resta(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)

// PRIORIDAD 3
vicula(multiplicacion(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)
vincula(division(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)
vincula(modulo(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)

// PRIORIDAD 4
vincula(and(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)
vincula(or(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)
```

```
// PRIORIDAD 5
vincula(negacion(Opnd)):
        vincula(Opnd)
vincula(menorUnario(Opnd)):
        vincula(Opnd)

// PRIORIDAD 6
vincula(indexacion(Opnd0,Opnd1)):
        vincula(Opnd0)
        vincula(Opnd1)
vincula(acceso(Opnd0,Opnd1String)):
        vincula(Opnd0)
vincula(indireccion(Opnd0)):
        vincula(Opnd0)

// PRIORIDAD 7
vincula(lit_entero(num)): noop
vincula(lit_real(num)): noop
vincula(true()): noop
vincula(false()): noop
vincula(lit_cadena(num)): noop
vincula(null()): noop
```

2. Comprobación de tipos

Funciones extra:

- ambos-ok(_,_). Ok si están bien, error en caso contrario
- aviso-error(_,_)
- aviso-error(_). Error

tipado(**prog**(Bloq)):
    tipado(Bloq)
    $.tipo = Bloq.tipo

tipado(**bloq**(DecsOp,InstrsOp):
    tipado(DecsOp)
    tipado(InstrsOp)
    $.tipo = ambos-ok(DecsOp.tipo,InstrsOp.tipo)

2.1. Declaración

tipado(**si_decs**(Decs)):
    tipado(Decs)
    $.tipo = Decs.tipo

tipado(**no_decs**()): $.tipo = ok

tipado(**muchas_decs**(Decs,Dec)):
    tipado(Decs)
    tipado(Dec)
    $.tipo = ambos-ok(Decs.tipo,Dec.tipo)

tipado(**una_dec**(Dec)):
    tipado(Dec)

```
        $.tipo = Dec.tipo

tipado(dec_variable(Tipo,Id)):
        $.tipo = tipado(Tipo)

tipado(dec_tipo(Tipo,Id)):
        $.tipo = tipado(Tipo)

tipado(dec_proc(Id, ParsFOp, Bloq)):
        tipado(ParsFOp)
        tipado(Bloq)
        $.tipo = ambos-ok(ParsFOp.tipo,Bloq.tipo)
tipado(si_parsF(ParsF)):
        tipado(ParsF)
        $.tipo = ParsF.tipo

tipado(no_parsF()): $.tipo = ok
tipado(muchos_parsF(ParsF, ParF)):
        tipado(ParsF)
        tipado(ParF)
        $.tipo = ambos-ok(ParsF.tipo,ParF.tipo)

tipado(un_parsF(ParF)):
        tipado(ParF)
        $.tipo = ParF.tipo

tipado(paramF(id, Tipo)):
        $.tipo = tipado(Tipo)

tipado(param(id, Tipo)):
        $.tipo = tipado(Tipo)

tipado (tipo_struct(Campos)):
        crea_hashmap(nm)
        tipado(Campos)
        $.tipo = Campos.tipo

tipado(muchos_campos(Campos, Campo)):
        tipado(Campos)
        tipado(Campo)
        $.tipo = ambos-ok(Campos.tipo, Campo.tipo)

tipado(un_campo(Campo)):
        tipado(Campo)
```

```
        $.tipo = Campo.tipo

// Las definiciones de tipos registro no tienen campos duplicados
tipado(crea_campo(Tipo, id)):
        if contiene(nm, id) then
                error
                $.tipo = error
        else
                put(nm, id, $)
                $.tipo = tipado(tipo)
        end if


// Los vínculos de los nombres de tipo utilizados en las declaraciones de tipo deben ser
declaraciones type (parece que ya se comprobó en vinculación)
tipado(tipo_iden(id)):
        return ok

tipado tipo_circum(Tipo):
        return tipado(Tipo)


// El tamaño de los tipos array es siempre un entero no negativo
tipado tipo_lista(Tipo, literalEntero):
        if int(literalEntero < 0) then
                error
                return error
        else
                return tipado(Tipo)
        end if

tipado tipo_int(): return ok
tipado tipo_real(): return ok
tipado tipo_bool(): return ok
tipado tipo_string(): return ok
```

2.2. Instrucciones

```
tipado(si_instrs(Instrs)):
        tipado(Instrs)
        $.tipo = Instrs.tipo
```

```
tipado(no_instr(Instrs)): $.tipo = ok

tipado(muchas_instrs(Instrs, Instr)):
        tipado(Instrs)
        tipado(Instr)
        $.tipo = ambos-ok(Instrs.tipo,Instr.tipo)

tipado(una_instr(Instr)):
        tipado(Instr)
        $.tipo = Instr.tipo
tipado(instr_eval(Exp)):
        tipado(Exp)
        If Exp.tipo == error then
                $.tipo = error
        else
                $.tipo = ok
        end if

tipado(instr_read(Exp)):
        tipado(Exp)
        tipoExp = ref!(Exp.tipo) // create a temporal value to save time
        If (tipoExp == int || tipoExp == real || tipoExp == string) && es-designador(Exp) then
                $.tipo = ok
        else
                aviso-error(Exp)
                $.tipo = error
        end if

tipado(intr_write(Exp)):
        tipado(Exp)
        tipoExp = ref!(Exp.tipo) // create a temporal value to save time
        If (tipoExp == int || tipoExp == real ||
          tipoExp == bool ||  tipoExp == string) &&  es-designador(Exp) then
                $.tipo = ok
        else
                $.tipo = error
        end if

tipado(instr_new(Exp)):
        tipado(Exp.tipo)
        If  ref!(Exp.tipo) == tipo_circum(T) then
                $.tipo = ok
        else
```

```
                aviso-error(ref!(Exp))
                $.tipo = error
        end if

tipado(instr_del(Exp)):
        If  ref!(Exp) == tipo_circum(T) then
                $.tipo = ok
        else
                aviso-error(ref!(Exp))
                $.tipo = error
        end if
```

```
tipado(instr_call(Id, ParsReOp)):
        let dec_proc (_, ParsFOp, _) == vinculoDe(ts, Id) in
                if ParsFOp == no_parsF() && ParsReOp == no_parsRe() then
                        $.tipo = ok
                else if ParsFOp == si_parsF(PF) && ParsReOp == si_parsRe(PR) then
                        if comprobar_parametros(PF, PR) then
                                $.tipo = ok
                        else
                                $.tipo = error
                        end if
                else
                        $.tipo = error
                end if
        end let
```

comprobar_parametros(PF, PR): // resursivo

```
        if PF == muchos_parsF(ParsF, ParF) && PR == muchos_parsRe(ParsRe, ParRe) then

                return comprobar_paramatros(ParsF, ParsRe) && comprobar_parametro(ParF,
        ParRe)

        else if PF == un_parF(ParF) && PR == un_parRe(ParRe) then

                return comprobar_parametro(ParF, ParRe)

        else

                return false

        end if
```

comprobar_parametro(PF, PR):

```
        let PR == un_parRe(E) in

                if PF == param(Id, T) then
                        return compatible(T, E.tipo)

                else
                        let PF == paramF(Id, T) in
                                if ! es-designador(E) then
                                        return false
                                end if

                                if T ==tipo_real() then
                                        return ref!(E.tipo) == tipo_real()
                                else
                                        return compatible(T, E.tipo)
                                end if
                end if
        end let

tipado(instr_if(Exp, Bloq)):
        tipado(Exp)
        tipado(Bloq)
        if ref!(Exp.tipo) == tipo_bool() && Bloq.tipo == ok then
                $.tipo = ok
        else
                aviso-error(Exp)
                $.tipo = error
        end if

tipado(instr_ifelse(Exp, Bloq1, Bloq2)):
        tipado(Exp)
        tipado(Bloq1)
        tipado(Bloq2)
        if ref!(Exp.tipo) == tipo_bool() && Bloq1.tipo == ok && Bloq2.tipo == ok then
                $.tipo = ok
        else
                aviso-error(Exp)
                $.tipo = error
        end if
```

```
tipado(instr_while(Exp, Bloq)):
        tipado(Exp)
        tipado(Bloq)
        if ref!(Exp.tipo) == tipo_bool() && Bloq.tipo == ok then
                $.tipo = ok
        else
                aviso-error(Exp)
                $.tipo = error
        end if


tipado(instr_nl()): $.tipo = ok


tipado(instr_bloque(Bloq)):
        tipado(Bloq)
        $.tipo = Bloq.tipo


//parametroReales
tipado(siParsRe(ParsRe)):
        tipado(ParsRe)
        $.tipo = ParsRe.tipo

tipado(noParsRe(ParsRe)):
        $.tipo = ok

tipado(muchosParsRe(ParsRe, Exp)):
        tipado(ParsRe)
        tipado(Exp)
        $.tipo = ambos-ok(ParsRe.tipo,Exp.tipo)

vinclula(unParRe(Exp)):
        tipado(Exp)
        $.tipo = Exp.tipo


tipado(asig(E0,E1)):
        tipado(E0)
        tipado(E1)
        if es-designador(E0) && compatible(E0.tipo, E1.tipo) then
                $.tipo = E0.tipo
        else
```

```
            aviso-error(E0.tipo, E1.tipo)
            $.tipo = error
        end if


// binario aritmético
tipado(suma(E0,E1)):
        $.tipo = tipado-bin-aritmetico(E0,E1)

tipado(resta(E0,E1)):
        $.tipo = tipado-bin-aritmetico(E0,E1)

tipado(multiplicacion(E0,E1)):
        $.tipo = tipado-bin-aritmetico(E0,E1)

tipado(division(E0,E1)):
        $.tipo = tipado-bin-aritmetico(E0,E1)


// modulo
tipado(modulo(E0,E1)):
        tipado(E0)
        tipado(E1)
        if ref!(E0.tipo) == int && ref!(E1.tipo) == int then
            $.tipo = int
        else
            aviso-error(E0.tipo, E1.tipo)
            return error
        end if


//operador logico
tipado(and(E0,E1)):
        $.tipo = tipado-bin-logico(E0,E1)

tipado(or(E0,E1)):
        $.tipo = tipado-bin-logico(E0,E1)


// operador relacional
tipado(menorl(E0,E1)):
        $.tipo = tipado-bin-relacional(E0,E1)

tipado(menor(E0,E1)):
```

```
        $.tipo = tipado-bin-relacional(E0,E1)

tipado(mayorI(E0,E1)):
        $.tipo = tipado-bin-relacional(E0,E1)

tipado(mayor(E0,E1)):
        $.tipo = tipado-bin-relacional(E0,E1)


// operador relacional especial
tipado(igual(E0,E1)):

        $.tipo = tipado-bin-relacional(E0,E1)

tipado(distint(E0,E1)):
        $.tipo = tipado-bin-relacional(E0,E1)


// operador unario
vincula(negacion(E)):
        tipado(E)
        T = ref!(E.tipo)
        if T == bool then
                $.tipo = bool
        else
                aviso-error(E)
                $.tipo = error
        end if

vincula(menorUnario(E)):
        tipado(E)
        T = ref!(E.tipo)
        if T == int then
                $.tipo = int
        elif T == real then
                $.tipo = real
        else
                aviso-error(E)
                $.tipo = error
        end if


// Designador
tipado(indexacion(E0,E1)):
```

```
        tipado(E0)
        tipado(E1)
        T0 = ref!(E0)
        T1 = ref!(E1)
        if  T0 == tipo_lista(T0') && T1 == int then
                $.tipo = T0'
        else
                aviso-error(T0, T1)
                $.tipo = error
        end if


tipado(acceso(E,C)):
        tipado(E)
        T = ref!(E)
        if T = tipo_struct(_)  && contiene(T.nm, C) then
                $.tipo = vinculoDe(T.nm, C).tipo
        else
                aviso-error(T)
                $.tipo = error
        end if



tipado(indireccion(E)):
        tipado(E)
        if  ref!(Exp) == tipo_circum(T) then
                        $.tipo = T
                else
                        aviso-error(T)
                        $.tipo = error
                end if



// valores
tipado(iden(Id)):
        let $.vinculo = Dec_var(T,I) in
                $.tipo = T
        end let
tipado(lit_entero(num)): $.tipo = int
tipado(lit_real(num)): $.tipo = real
tipado(true()): $.tipo = bool
tipado(false()): $.tipo = bool
```

tipado(**lit_cadena** (num)): $.tipo = string

tipado-bin-aritmetico(E0,E1):

```
tipado(E0)
tipado(E1)
T0 = ref!(E0.tipo)
T1 = ref!(E1.tipo)
if T0 == int && T1 == int then
        return int
else if (T0 == int || T0 == real) &&
        (T1 == int || T1 == real) then
        return real
else
        aviso-error(T0, T1)
        return error
end if
```

tipado-bin-logico(E0,E1):

```
tipado(E0)
tipado(E1)
T0 = ref!(E0.tipo)
T1 = ref!(E1.tipo)
if T0 == bool && T1 == bool then
        $.tipo = bool
else
        aviso-error(T0, T1)
        return error
end if
```

tipado-bin-relacional(E0,E1):

```
tipado(E0)
tipado(E1)
T0 = ref!(E0.tipo)
T1 = ref!(E1.tipo)
if ((T0 == int || T1 == real ) && (T1 == int || T1 == real)) ||
  (T0 == bool && T1 ==bool) ||
  (T0 == string && T1 ==string) then
                return bool
```

```
        else
                aviso-error(E0.tipo, E1.tipo)
                return error
        end if

tipado-bin-relacional-especial(E0,E1):
        tipado(E0)
        tipado(E1)
        T0 = ref!(E0.tipo)
        T1 = ref!(E1.tipo)
        if (T0 == tipo_circum() || T0 == null) && (T1 == tipo_circum() || T1 == null) then
                return bool

        // this part is basically copied from above. I' m not using the function above directly
        because I don't want to call tipado(E0) and tipado (E1) twice

        else if ((T0 == int || T1 == real ) && (T1 == int || T1 == real)) ||
          (T0 == bool && T1 ==bool) ||
          (T0 == string && T1 ==string) then
                        return bool
        else
                aviso-error(E0.tipo, E1.tipo)
                return error
        end if
```

2.3. Métodos Extra

```
ambos-ok(T0,T1):
        if T0 != error && T1 != error then
                return ok
        else
                return error
        endif

aviso-error(T0,T1):
        if T0 != error && T1 != error then
                error
        endif

aviso-error(T):
        if T != error then
                error
```

```
        endif

ref(T):
  let T.vinculo = Dec_tipo(T',T) in
        return T'


ref!(T):
 if T == Ref(I) then

        let T.vinculo = Dec_tipo(T',I) in

                return ref!(T')

        end let

else

        return T

end if
```

es-designador(E):
        return E == iden(_) || E == indexacion(_,_) || E == acceso(_,_) || E = indireccion(_,_)

compatible(T1, T2):

        Θ = {T1 == T2}

        unificable(T1, T2)

unificable(T1, T2):

```
        let T1' = ref!(T1) T2' = ref!(T2) in
                if T1' == T2' then
                        return true

                else if T1' == tipo_real()  && T2' == tipo_int()
                        return true

                else if T1' == tipo_circum(_) && T2' == tipo_null() then
                        return true

                else if T1' == tipo_lista(C1, N1) && T2' == tipo_lista(C2, N2) then
                        if int(N1) != int(N2) then
                                return false
```

```
                    else
                        return unificable(C1, C2)
                    end if

                else if T1' == tipo_struct (C1) && T2' == tipo_struct (C2) then
                    return comprobar_campos(C1,C2)

                else if T1' == tipo_circum (C1) && T2' == tipo_circum (C2) then
                    return unificable(C1, C2)

                else
                    return false

                end if
        end let
```

son_unificables(T1,T2):

if (T1==T2) ∉ **Θ then**

    Θ = **Θ** ∪ {T1==T2}

    return unificables(T1,T2)

else

    return true

end if

comprobar_campos(C1, C2): // campos de struct, resursivo

```
if C1 == muchos_campos(Campos1, Campo1) && C2 == muchos_campos(Campos2, Campo2)
then

    let Campo1 == crea_campo(T1, _) && Campo2 == crea_campo(T2, _) in

        return son_unificables(T1, T2) && comprobar_campos(Campos1, Campos2)

else if C1 == un_campo(Campo1) && C2 == un_campo(Campo2) then

    let Campo1 == crea_campo(T1, _) && Campo2 == crea_campo(T2, _) in

        return son_unificables(T1, T2)

else

    return false
```

3. Asignación de espacio

Asignación de espacio: tenéis que asignar el nivel de cada variable, así como el nivel de los procedimientos, y el tamaño de los datos locales de dichos procedimientos. La asignación de espacio para decProc no tiene demasiado sentido. **¿Dónde se asignan las direcciones a los parámetros formales? ¿Cómo se consigue que las direcciones de los objetos que viven en los procedimintos sean direcciones relativas -se comience a contar de nuevo desde 0?** A los campos de los struct hay que asignarles desplazamientos (no tiene sentido incrementar allí direcciones)... En general hay bastantes defectos. Tenéis que revisar todo el procesamiento con cuidado.

3.1. Declaración primera pasada

```
var dir= 0;
var max_dir = 0;
var nivel = 0;

asig-espacio(prog(Bloq)):
        asig-espacio(Bloq)

asig-espacio(bloq(DecsOp,InstrsOp):
        dir_ant = dir
        asig-espacio(DecsOp)
        asig-espacio(InstrsOp)
        dir = dir_ant

asig-espacio(si_decs(DecsOp)): //2 pasada
        asig-espacio1(Decs)
        asig-espacio2(Decs)

asig-espacio(no_decs()): noop


asig-espacio1(muchas_decs(Decs,Dec)):
        asig-espacio1(Decs)
        asig-espacio1(Dec)

asig-espacio1(una_dec(Dec)):
        asig-espacio1(Dec)

asig-espacio1(decVariable(Tipo,Id)): int a
```

```
        $.dir = dir //para id
        asig-tam(Tipo)
        $.nivel = nivel
        inc_dir(Tipo.tam)


asig-espacio1(decTipo(Tipo,Id)): //type
        asig-tam(Tipo)


asig-espacio1(decProc(Id, ParsFOp, Bloq)):
        dir_ant = dir
        max_dir_ant = max_dir
        nivel++
        $.nivel = nivel
        dir = 0
        max_dir = 0
        asig-espacio1(ParsFOp)
        let Bloq == bloq(DecsOp, InstrsOp) in
                asig-espacio(DecsOp)
                asig-espacio2(DecsOp)
                asig-espacio(InstrsOp)
        $.tam = dir
        dir = dir_ant
        max_dir = max_dir_ant
        nivel - -


asig-espacio1(si_parsF(ParsF)):
        asig-espacio1(ParsF)

asig-espacio1(no_parsF()): noop

asig-espacio1(muchos_parsF(ParsF, ParF)):
        asig-espacio1(ParsF)
        asig-espacio1(ParF)

asig-espacio1(un_parF(ParF)):
        asig-espacio1(ParF)

asig-espacio1(paramF(id, Tipo)):
        $.dir = dir
        asig-tam1(Tipo)
        $.nivel = nivel
```

```
        inc_dir(1)

asig-espacio1(param(id, Tipo)):
        $.dir = dir //para id
        asig-tam1(Tipo)
        $.nivel = nivel
        inc_dir(Tipo.tam)

//Tipo

asig-tam1(tipo_struct(Campos)):
        dir_ant = dir
        dir = 0
        asig-espacio1(Campos)
        $.tam = dir
        dir = dir_ant

asig-espacio1(muchos_campos(Campos, Campo)):
        asig-espacio1(Campos)
        asig-espacio1(Campo)

asig-espacio1(un_campo(Campo)):
        asig-espacio1(Campo)

asig-espacio1(crea_campo(Tipo, id)):
        $.desp = dir
        asig-tam1(Tipo)
        dir += T.tam

asig-tam1(tipo_iden(id)):
        let $.vinculo = dec_tipo(T,id) in
                $.tam= T.tam

asig-tam1(tipo_lista(Tipo, literalEntero)):
        asig-tam1(Tipo)
        $.tam = Tipo.tam * parse_int(literalEntero)


asig-tam1(tipo_circum(Tipo, literalEntero)): //puntero
        if Tipo !=  tipo_iden(id) then
                asig-tam1(T)
        endif
        $.tam = 1
```

```
asig-tam1(tipo_int()):
        $.tam =1

asig-tam1 (tipo_real()):
        $.tam =1

asig-tam1 (tipo_bool()):
        $.tam =1

asig-tam1 (tipo_string()):
        $.tam =1
```

3.2. Declaración Segunda pasada

```
asig-espacio2(muchas_decs(Decs,Dec)):
        asig-espacio2(Decs)
        asig-espacio2(Dec)

asig-espacio2(una_dec(Dec)):
        asig-espacio2(Dec)

asig-espacio2(decVariable(Tipo,Id)):
        asig-tam2(Tipo)

asig-espacio2(decTipo(Tipo,Id)): //type
        asig-tam2(Tipo)

asig-espacio2(decProc(Id, ParsFOp, Bloq)): noop

asig-espacio2(si_parsF(ParsF)):
        asig-espacio2(ParsF)

asig-espacio2(no_parsF()): noop

asig-espacio2(muchos_parsF(ParsF, ParF)):
        asig-espacio12(ParsF)
        asig-espacio2(ParF)

asig-espacio2(un_parsF()):
        asig-espacio2(ParF)
```

```
asig-espacio2(paraF(ParF)):
        asig-espacio2(ParF)

asig-espacio2(parF(id, Tipo)):
        asig-tam2(Tipo)

//Tipo

asig-tam2(tipo_struct(Campos)):
        asig-espacio2(Campos)

asig-espacio2(muchos_campos(Campos, Campo)):
        asig-espacio2(Campos)
        asig-espacio2(Campo)

asig-espacio2(un_campo(Campo)):
        asig-espacio2(Campo)

asig-espacio2(crea_campo(Tipo, id)):
        asig-tam2(Tipo)


asig-tam2(tipo_iden(id)): noop

asig-tam2(tipo_lista(Tipo, literalEntero)):
        asig-tam1(Tipo)


asig-tam2(tipo_circum(Tipo)): //puntero
        if Tipo ==  ref(id) then
                let Tipo.vinculo = dec_tipo(T',id) in
                        Tipo.tam = T'.tam
        else
                asig-tam2(Tipo)
        endif

asig-tam1(tipo_int()):
        noop

asig-tam1 (tipo_real()):
        noop

asig-tam1 (tipo_bool()):
        noop
```

asig-tam1 (tipo_string()):
    Noop

3.3. Instruccion

asig-espacio(**si_instrs**(Instrs)):
    asig-espacio(Instrs)

asig-espacio(**no_instrs**(Instrs)): noop

asig-espacio(**muchas_instrs**(Instrs, Instr)):
    asig-espacios(Instrs)
    asig-espacio(Instr)

asig-espacios(**una_instr**(Instr)):
    asig-espacio(Instr)

asig-espacio(**instr_eval**(Exp)): noop

asig-espacio(**instr_if**(Exp, Bloq)):
    asig-espacio(Bloq)

asig-espacio(**instr_ifelse**(Exp, Bloq1, Bloq2)):
    asig-espacio(Bloq1)
    asig-espacio(Bloq2)

asig-espacio(**instr_while**(Exp, Bloq)):
    asig-espacio(Bloq)

asig-espacio(**instr_read**(Exp)): noop

asig-espacio(**intr_write**(Exp)): noop

asig-espacio(**instr_nl**()): noop

asig-espacio(**instr_new**(Exp)): noop

asig-espacio(**instr_del**(Exp)): noop

asig-espacio(**instr_call**(Id, ParsReOp)): noop

```
asig-espacio(instr_bloque(Bloq)):
        asig-espacio(Bloq)
```

3.4 Métodos Extra

```
inc_dir(inc):
        dir += inc
        if dir > max_dir then
                max_dir = dir
```

4. Descripción del repertorio de instrucciones de la máquina-p

Valores

- apila-int(V): apila un número entero.
- apila-real(V): apila un número real.
- apila-bool(V): apila un booleano
- apila-cadena(V): apila una cadena
- apila-null: apila *null*.
- dup: Consulta el valor v de la cima de la pila de evaluación, y apila de nuevo dicho valor (es decir, duplica la cima de la pila de evaluación)
- in2real: desapila un número int de la pila y apila su número real equivalente

Operadores Básicos

desapila Vo (y V1), apila el resultado de la operación

- menorI: **v0 <= v1**
- menor: **v0 < v1**
- mayorI: **v0 >= v1**
- mayor: **v0 > v1**
- igual: **v0 == v1**
- distint: **v0 != v1**
- suma: **v0 + v1.**
- resta: **v0 - v1**
- multiplicacion: **v0 * v1**
- division: **v0 / v1**
- modulo: **v0 % v1**
- and: **v0 and v1**
- or: **v0 or v1**

- menosUnario: **-Vo**
- negacion: **!Vo**

Direcciones

solo τ es un parámetro, otros son de la pila

- fetch(d): con d una dirección. Devuelve el valor almacenado en la celda direccionada por d
- store(d,v): con d una dirección, y v un valor. Almacena v en la celda direccionada por d
- copy(d, d', τ): con d y d' direcciones, y τ un tipo. Copia el valor del tipo τ que se encuentra almacenado a partir de la dirección d' en el bloque que comienza a partir de la dirección d
- alloc(τ): con τ un tipo. Se reserva una zona de memoria adecuada para almacenar valores del tipo τ. La operación en sí devuelve la dirección de comienzo de dicha zona de memoria.
- dealloc(d,τ): con d una dirección, y τ un tipo. Se notifica que la zona de memoria que comienza en d y que permite almacenar valores del tipo τ queda liberada.
- indx(d,i,τ): con d una dirección, i un valor, y τ un tipo. Considera que, a partir de d, comienza un array cuyos elementos son valores del tipo τ, y devuelve la dirección de comienzo del elemento i-esimo de dicho array.
- acc(d,c,τ), con d una dirección, c un nombre de campo, y τ un tipo record. Considera que, a partir de d, está almacenado un registro de tipo τ, que contiene un campo c. Devuelve la dirección de comienzo de dicho campo.

Pila de registros de activación

- activa(n,t,d): Reserva espacio en el segmento de pila de registros de activación para ejecutar un procedimiento que tiene nivel de anidamiento n y tamaño de datos locales t. Así mismo, almacena en la zona de control de dicho registro d como dirección de retorno. También almacena en dicha zona de control el valor del display de nivel n. Por último, apila en la pila de evaluación la dirección de comienzo de los datos en el registro creado.
- apilad(n): Apila en la pila de evaluación el valor del display de nivel n
- desapilad(n): Desapila una dirección d de la pila de evaluación en el display de nivel n.
- desactiva(n,t): Libera el espacio ocupado por el registro de activación actual, restaurando adecuadamente el estado de la máquina. n indica el nivel de anidamiento del procedimiento asociado; t el tamaño de los datos locales. De esta forma, la instrucción: (i) apila en la pila de evaluación la dirección de retorno; (ii) restaura el valor del display de nivel n al antiguo valor guardado en el registro;

(iii) decrementa el puntero de pila de registros de activación en el tamaño ocupado por el registro.

Saltos

- ir-a(D): Salta a la dirección *D*.
- ir-v(D): Desapila un valor de la pila (tiene que ser booleano) y salta si es verdadero.
- ir-f(D): Desapila un valor de la pila (tiene que ser booleano) y salta si es falso.
- ir-ind: Desapila una dirección d de la pila de evaluación, y realiza un salto incondicional a dicha dirección.

I/O

- read(T): Lee T
- write(V): Imprime V
- nl: Imprime una nueva linea

Otros

- stop:Detiene la máquina.
- Error: Da error
- Desapila: Desapila un valor del run-time stack y no hace nada

5. Procesamiento de **Etiquetado** y **Generación** de **Código**

Operación **emit** usada en la especificación denotacional de la traducción. **emit** es una primitiva que genera (emite) una instrucción de la máquina virtual

5.1 Métodos Principales

En azul: Constructor
var etq = 0
var sub_pendientes = pila-vacia()

Etiquetado	Generación de Código

prog: Bloq → Prog

```
etiquetado(prog(Bloq)):
        $.prim = etq
        let Bloq = bloq(DecsOp, InstrsOp) in
          recolecta_procs(DecsOp)
        etiquetado(Is)
        etq++
        while !es-vacia(sub_pendientes)
           sub = desapila(sub_pendientes)
           let sub == dec_proc(Id, ParsFOp, Bloq) in:
              etq++
              let Bloq == bloq(DOp, IOp) in
                 recolecta_procs(DOp)
                 etiquetado(IOp)
              etq += 2
              end let
           end let
        end while
        $.sig = etq
```

```
gen-cod(prog(Bloq)):
        let Bloq = bloq(DecsOp, InstrsOp) in
         recolecta_procs(DecsOp)
         gen-cod(InstrsOp)
         emit stop()
         while !es-vacia(sub_pendientes)
            sub = desapila(sub_pendientes)
            let sub == dec_proc(Id, ParsFOp, Bloq) in:
               emit desapilad(sub.nivel)
               let Bloq == bloq(DOp, IOp) in
                  recolecta_procs(DOp)
                  gen-cod(IOp)
               emit desactiva(sub.nivel, sub.tam)
               emit ir-ind()
               end let
            end let
         end while
         gen-cod(Bloq)
```

bloq: DecsOp x InstrsOp → Bloq

```
etiquetado(bloq(DecsOp, InstrsOp)):
        $.prim = etq
        recolecta_procs(DecsOp)
        etiquetado(InstrsOp)
        $.sig = etq
```

```
gen-cod(bloq(DecsOp, InstrsOp)):
        recolecta_procs(DecsOp)
        gen-cod(InstrsOp)
```

siInstrs: Instrs → InstrsOp

```
etiquetado(si_instrs(Is)):
        $.prim = etq
        etiquetado(Is)
        $.sig = etq
```

```
gen-cod(si_instrs(Is)):
        gen-cod(Is)
```

noInstrs: → InstrsOp

```
etiquetado(no_instrs()): noop
```

```
gen-cod(no_instrs()): noop
```

muchasInstrs: Instrs x Instr → Instrs

```
etiquetado(muchas_instrs(Is,I)):
```

```
gen-cod(muchas_instrs(Is,I)):
```

$.prim = etq etiquetado(Is) etiquetado(I) $.sig = etq	gen-cod(Is) gen-cod(I)
unaInstr: Instr → Instrs	
etiquetado(**una_instr**(I)): $.prim = etq etiquetado(I) $.sig = etq	gen-cod(**una_instr**(I)): gen-cod(I)
instrEval: Exp → Instr	
etiquetado(**instr_eval**(Exp)): $.prim = etq etiquetado(Exp) etq ++ $.sig = etq	gen-cod(**instr_eval**(Exp)): // discard the result of Exp gen-cod(Exp) emit pop
instrIf: Exp x Bloq → Instr	
etiquetado(**instr_if**(Exp, Bloq)): $.prim = etq etiquetado(Exp) etiquetado-acc-val(Exp) etq++ etiquetado(Bloq) $.sig = etq	gen-cod(**instr_if**(Exp,Bloq)): gen-cod(Exp) gen-acc-val(Exp) emit ir-f($.sig) gen-cod(Bloq)
instrIfElse: Exp x Bloq x Bloq → Instr	
etiquetado(**instr_ifelse**(Exp, Bloq1, Bloq2)): $.prim = etq etiquetado(Exp) etiquetado-acc-val(Exp) etq++ etiquetado(Bloq1) etq++ etiquetado(Bloq2) $.sig = etq	gen-cod(**instr_ifelse**(Exp,Bloq1,Bloq2)): gen-cod(Exp) gen-acc-val(Exp) emit ir-f(Bloq2.prim) gen-cod(Bloq1) emit ir-a($.sig) gen-cod(Bloq2)
instrWhile: Exp x Bloq →Instr	
etiquetado(**instr_while**(Exp, Bloq)):	gen-cod(**instr_while**(Exp,Bloq)):

$.prim = etq etiquetado(Exp) etiquetado-acc-val(Exp) etq++ etiquetado(Bloq) etq++ $.sig = etq	gen-cod(Exp) gen-acc-val(Exp) emit ir-f($.sig) gen-cod(Bloq) emit ir-a($.prim)
instrRead: Exp → Instr	
etiquetado(**instr_read**(Exp)): $.prim = etq etiquetado(Exp) etq++ $.sig = etq	gen-cod(**instr_read**(Exp)): gen-cod(Exp) emit read(ref!(Exp.tipo))
intrWrite: Exp → Instr	
etiquetado(**instr_write**(Exp)): $.prim = etq etiquetado(Exp) etiquetado-acc-val(Exp) etq++ $.sig = etq	gen-cod(**instr_write**(Exp)): gen-cod(Exp) gen-acc-val(Exp) emit write
instrNl: -> Instr	
etiquetado(**instr_nl**(Exp)): etq ++	gen-cod(**instr_nl**(Exp)): emit nl
instrNew: Exp → Instr	
etiquetado(**instr_new**(Exp)): $.prim = etq etiquetado(Exp) etq +=2 $.sig = etq	gen-cod(**instr_new**(Exp)): gen-cod(Exp) emit alloc(Exp.tipo) emit store
instrDel: Exp → Instr	
etiquetado(**instr_del**(Exp)): $.prim = etq etiquetado(Exp) etq+=7	gen-cod(**instr_del**(Exp)): gen-cod(Exp) emit dup emit apila-int(-1)

$.sig = etq	emit igual
	emit ir-v($.sig-1)
	emit dealloc(Exp.tipo)
	emit ir-a($.sig)
	emit Error
	
	/*
	if cima(run-time-stack) != -1 then
	emit dealloc(Exp.tipo)
	else
	emit Error
	*/

instrCall: string x ParsReOp → Instr	

etiquetado(**instr_call**(Id, Pars)):
$.prim = etq
Etq++
if !ParsReOp == no_parsOp() then
let $.vinculo.parsFOp == si_parsF(ParsF) && ParsReOp == si_parsRe(ParsRe) in
etiquetado-paso-params(ParsF, ParsRe)
end let
end if
$.sig = etq
	gen-cod(**instr_call**(Id, ParsReOp)):
	
	emit activa($.vinculo.nivel, $.vinculo.tam, $.sig)
	if !ParsReOp == no_parsOp() then
	let $.vinculo.parsFOp == si_parsF(ParsF) && ParsReOp == si_parsRe(ParsRe) in
	gen-paso-params(ParsF, ParsRe)
	end let
	end if
	emit ir-a($.vinculo.prim)

instrBloque: Bloq → Instr	
etiquetado(**instr_bloque**(Bloq)): $.prim = etq etiquetado(Bloq) $.sig = etq	gen-cod(**instr_bloque**(Bloq)): gen-cod(Bloq)
siParsRe: ParsRe → ParsReOp	
etiquetado(**si_parsRe**(Pars)): $.prim = etq etiquetado(Pars) $.sig = etq	gen-cod(**si_parsRe**(Pars)): noop
noParsRe: → ParsReOp	
etiquetado(**no_parsRe**(Is)): noop	gen-cod(**no_parsRe**()): noop

<table>
<tr><td colspan="2">muchosParsRe: ParsRe x Exp → ParsRe</td></tr>
<tr><td>etiquetado(muchos_parsRe(Pars,Exp)): noop</td><td>gen-cod(muchos_parsRe(Pars,Exp)): noop</td></tr>
<tr><td colspan="2">unParRe: Exp → ParsRe</td></tr>
<tr><td>etiquetado(un_parRe(Exp)): noop</td><td>gen-cod(un_parRe(Exp)): noop</td></tr>
<tr><td colspan="2">asig: Exp x Exp → Exp</td></tr>
<tr><td>etiquetado(asig(Exp1, Exp2)):
$.prim = etq
etiquetado(Exp1)
etq ++
etiquetado(Exp2)
if Exp1.tipo = real && Exp2.tipo == int then
etiquetado-acc-val(Exp2)
etq += 2
else
if es-designador(Exp2) then
etq ++
else
etq ++
end if
end if
etq ++
// the code above is redundant but it keeps the structural correspondence with gen-cod</td><td>gen-cod(asig(Exp1,Exp2)):

gen-cod(Exp1)
emit cup
gen-cod(Exp2)
if Exp1.tipo = real && Exp2.tipo == int then
gen-acc-val(Exp2)
emit int2real
emit store
else
if es-designador(Exp2) then
emit copy(Exp1.tipo)
else
emit store
end if
end if
emit(fetch)
// exp2 could be clase asig but basically it’s a designador</td></tr>
<tr><td colspan="2">menorI: Exp x Exp → Exp</td></tr>
<tr><td>etiquetado(menorI(Exp1, Exp2)):
$.prim = etq
etiquetado-cod-opnds($)
etq ++
$.sig = etq</td><td>gen-cod(menorI(Exp1,Exp2)):

gen-cod-opnds($)
emit menorI</td></tr>
<tr><td colspan="2">menor: Exp x Exp → Exp</td></tr>
<tr><td>etiquetado(menor(Exp1, Exp2)):
$.prim = etq
etiquetado-cod-opnds($)
etq ++
$.sig = etq</td><td>gen-cod(menor(Exp1,Exp2)):

gen-cod-opnds($)
emit menor</td></tr>
</table>

mayorI: Exp x Exp → Exp	
etiquetado(**mayorI**(Exp1, Exp2)): $.prim = etq etiquetado-cod-opnds($) etq ++ $.sig = etq	gen-cod(**mayorI**(Exp1,Exp2)): gen-cod-opnds($) emit mayorI
mayor: Exp x Exp → Exp	
etiquetado(**mayor**(Exp1, Exp2)): $.prim = etq etiquetado-cod-opnds($) etq ++ $.sig = etq	gen-cod(**mayor**(Exp1,Exp2)): gen-cod-opnds($) emit mayor
igual: Exp x Exp → Exp	
etiquetado(**igual**(Exp1, Exp2)): $.prim = etq etiquetado-cod-opnds($) etq ++ $.sig = etq	gen-cod(**igual**(Exp1,Exp2)): gen-cod-opnds($) emit igual
distint: Exp x Exp → Exp	
etiquetado(**distint**(Exp1, Exp2)): $.prim = etq etiquetado-cod-opnds($) etq ++ $.sig = etq	gen-cod(**distint**(Exp1,Exp2)): gen-cod-opnds($) emit distint
suma: Exp x Exp → Exp	
etiquetado(**suma**(Exp1, Exp2)): $.prim = etq etiquetado-cod-opnds-aritmetico($) etq ++ $.sig = etq	gen-cod(**suma**(Exp1,Exp2)): gen-cod-opnds-aritmetico($) emit suma
resta: Exp x Exp → Exp	
etiquetado(**resta**(Exp1, Exp2)): $.prim = etq	gen-cod(**resta**(Exp1,Exp2)):

etiquetado-cod-opnds-aritmetico($) etq ++ $.sig = etq	gen-cod-opnds-aritmetico($) emit resta
multiplicacion: Exp x Exp → Exp	
etiquetado(**multiplicacion**(Exp1, Exp2)): $.prim = etq etiquetado-cod-opnds-aritmetico($) etq ++ $.sig = etq	gen-cod(**multiplicacion**(Exp1,Exp2)): gen-cod-opnds-aritmetico($) emit multiplicacion
division: Exp x Exp → Exp	
etiquetado(**division**(Exp1, Exp2)): $.prim = etq etiquetado-cod-opnds-aritmetico($) etq ++ $.sig = etq	gen-cod(**division**(Exp1,Exp2)): gen-cod-opnds-aritmetico($) emit division
modulo: Exp x Exp → Exp	
etiquetado(**modulo**(Exp1, Exp2)): $.prim = etq etiquetado-cod-opnds($) etq ++ $.sig = etq	gen-cod(**modulo**(Exp1,Exp2)): gen-cod-opnds($) emit modulo
and: Exp x Exp → Exp	
etiquetado(**and**(Exp1, Exp2)): $.prim = etq etiquetado-cod-opnds($) etq ++ $.sig = etq	gen-cod(**and**(Exp1,Exp2)): gen-cod-opnds($) emit and
or: Exp x Exp → Exp	
etiquetado(**or**(Exp1, Exp2)): $.prim = etq etiquetado-cod-opnds($) etq ++ $.sig = etq	gen-cod(**or**(Exp1,Exp2)): gen-cod-opnds($) emit or

negacion: Exp → Exp	
etiquetado(**negacion**(Exp)): $.prim = etq etiquetado-cod-opnds($) etiquetado-acc-val(Exp) etq ++ $.sig = etq	gen-cod(**negacion**(Exp)): gen-cod(Exp) gen-acc-val(Exp) emit negacion
menosUnario: Exp → Exp	
etiquetado(**menosUnario**(Exp)): $.prim = etq etiquetado(Exp) etq-acc-val(Exp) etq++ $.sig = etq	gen-cod(**menosUnario**(Exp)): gen-cod(Exp) gen-acc-val(Exp) emit menosUnario
indexacion: Exp x Exp → Exp	
etiquetado(**indexacion**(Exp1, Exp2)): $.prim = etq etiquetado(Exp1) etiquetado(Exp2) etq-acc-val(Exp2) etq++ $.sig = etq	gen-cod(**indexacion**(Exp1,Exp2)): gen-cod(Exp1) gen-cod(Exp2) gen-acc-val(Exp2) emit indx(Exp1.tipo())
acceso: Exp x string → E	
etiquetado(**acceso**(Exp)): $.prim = etq etiquetado(Exp) etq++ $.sig = etq	gen-cod(**acceso**(Exp,Id)): gen-cod(Exp) emit acc(Exp.tipo())
indireccion: Exp → Exp	
etiquetado(**indireccion**(Exp)): $.prim = etq etiquetado(Exp)	gen-cod(**indireccion**(Exp)): gen-cod(Exp) emit dup

etq +=7
$.sig = etq
	emit apila-int(-1)
	emit igual
	emit ir-v($.sig-1)
	emit fetch
	emit ir-a($.sig)
	emit Error
	
	/*
	if cima(run-time-stack) != -1 then
	emit fetch
	else
	emit Error
	*/

identificador: string → Exp	
etiquetado(**identificador**(Id)): $.prim = etq etiquetado-acc-id($.vinculo) $.sig = etq	gen-cod(**identificador**(Id)): gen-acc-id($.vinculo)
literalEntero: string → Exp	
etiquetado(**literalEntero**(N)): $.prim = etq etq++ $.sig = etq	gen-cod(**literalEntero**(N)): emit apila-int(int(N))
literalReal: string → Exp	
etiquetado(**literalReal**(n)): $.prim = etq etq++ $.sig = etq	gen-cod(**literalReal**(N)): emit apila-real(float(N))
true: → Exp	
etiquetado(**true**()): $.prim = etq etq++ $.sig = etq	gen-cod(**true**()): emit apila-bool(true)
false: → Exp	
etiquetado(**false**()):	gen-cod(**false**()):

$.prim = etq etq++ $.sig = etq	emit apila-bool(false)
literalCadena: string → Exp	
etiquetado(**literalCadena**(N)): $.prim = etq etq++ $.sig = etq	gen-cod(**literalCadena**(N)): emit apila-cadena(N)
null: → Exp	
etiquetado(**null**()): $.prim = etq etq++ $.sig = etq	gen-cod(**null**()): emit apila-null

5.2 Métodos Extra

recolecta_procs(si_decs(Decs)): noop	recolecta_procs(si_decs(Decs)): noop
recolecta_procs(si_decs(Decs)): recolecta_procs(Decs)	recolecta_procs(si_decs(Decs)): recolecta_procs(Decs)
recolecta_procs(muchas_decs(Decs, Dec)): recolecta_procs(Decs) recolecta_procs(Dec)	recolecta_procs(muchas_decs(Decs, Dec)): recolecta_procs(Decs) recolecta_procs(Dec)
recolecta_procs(una_dec(Dec)): recolecta_procs(Dec)	recolecta_procs(una_dec(Dec)): recolecta_procs(Dec)
recolecta_procs(dec_variable(T, Id)): noop	recolecta_procs(dec_variable(T, Id)): noop
recolecta_procs(dec_tipo(T, Id)): noop	recolecta_procs(dec_tipo(T, Id)): noop
recolecta_procs(dec_proc(Id, ParsFOp, Bloq)): apila(sub_pendientes, $)	recolecta_procs(dec_proc(Id, ParsFOp, Bloq)): apila(sub_pendientes, $)
etiquetado-cod-opnds(ExpBin(Opnd0, Opnd1)): etiquetado(Opnd0) etiquetado-acc-val(Opnd0)	gen-cod-opnds(ExpBin(Opnd0, Opnd1)): gen-cod(Opnd0) gen-acc-val(Opnd0)

etiquetado(Opnd1) etiquetado-acc-val(Opnd1)	gen-cod(Opnd1) gen-acc-val(Opnd1)

etiquetado-cod-opnds-aritmetico(ExpBin(Opnd0, Opnd1)):
etiquetado(Opnd0)
etiquetado-acc-val(Opnd0)
if ref!(ExpBin.tipo) == tipo_real(_) && ref!(Opn0.tipo) == tipo_int(_) then
etq++
end if
etiquetado(Opnd1)
etiquetado-acc-val(Opnd1)
if ref!(ExpBin.tipo) == tipo_real(_) && ref!(Opnd1.tipo) == tipo_int(_) then
etq++
end if
	gen-cod-opnds-aritmetico(ExpBin(Opnd0, Opnd1)):
	gen-cod(Opnd0)
	gen-acc-val(Opnd0)
	if ref!(ExpBin.tipo) == tipo_real(_) && ref!(Opn0.tipo) == tipo_int(_) then
	emit int2real
	end if
	gen-cod(Opnd1)
	gen-acc-val(Opnd1)
	if ref!(ExpBin.tipo) == tipo_real(_) && ref!(Opnd1.tipo) == tipo_int(_) then
	emit int2real
	end if

etiquetado-acc-val(E): if es-designador(E) then etq++	gen-acc-val(E): if es-designador(E) then emit fetch

etiquetado-paso-params(PF, PR):
if PF == muchos_parsF(ParsF, ParF) && PR == muchos_parsRe(ParsRe,E) then
etiquetado-paso-params(ParF, ParsRe)
etiquetado-paso-param(ParF,E)
else let PF == un_parF(ParF) && PR == un_parRe(E) in
etiquetado-paso-param(ParF, E)
end let
end if
	gen-paso-params(PF, PR):
	if PF == muchos_parsF(ParsF, ParF) && PR == muchos_parsRe(ParsRe, E) then
	gen-paso-param(ParF, E)
	gen-paso-params(ParsF, ParsRe)
	else let PF == un_parF(ParF) && PR == un_parRe(E) in
	gen-paso-param(ParF, E)
	end let
	end if

etiquetado-paso-param(ParF, E):
etq += 3
etiquetado(E)
if ParF == param(T, id) then
if T == tipo_real(_) && E.tipo == tipo_int() then
etiquetado-acc-val(E)
etq += 2
	gen-paso-param(ParF, E):
	emit dup
	emit apila-int(ParF.dir)
	emit suma
	gen-cod(E)
	if ParF == param(T, id) then
	if T == tipo_real(_) && E.tipo == tipo_int() then

else if es-designador(E) then
etq++
else
etq++
end if
else
// paramF
etq++
	gen-acc-val(E)
	emit int2real
	emit store
	else if es-designador(E) then
	emit copy(T)
	else
	emit store
	end if
	else
	// paramF
	emit store

etiquetado-acc-id(dec_var(T, id)):
if $.nivel == 0 then
etq++
else
etiquetado-acc-var($)
end if
	gen-acc-id(dec_variable(T, id)):
	if $.nivel == 0 then
	emit apila-int($.dir)
	else
	gen-acc-var($)
	end if

etiquetado-acc-id(param(T, id)): etiquetado-acc-var($)	gen-acc-id(param(T, id)): gen-acc-var($)
etiquetado-acc-id(paramF(T, id)): etiquetado-acc-var($) etq++	gen-acc-id(paramF(T, id)): gen-acc-var($) emit fetch
etiquetado-acc-var(V): etq+=3	gen-acc-var(V): emit apilad(V.nivel) emit apila-int(V.dir) emit suma