# Compilador per el llenguatge PepLang

**19 de gener del 2026**


**Assignatura:**

Compiladors

**Autor:**

Josep Oliver Vallespir - 78222663P

# Introducció

## Objectiu

A n'aquest document explicarem el projecte de l'assignatura de Compiladors del curs 2025-2026.

L'objectiu principal d'aquesta pràctica ha estat el disseny i la implementació d'un compilador per a un llenguatge de programació imperatiu d'alt nivell, en aquest cas el llenguatge **PepLang**, seguint les especificacions detallades a l'enunciat d'aquest projecte.

## Abast del projecte

El desenvolupament d'aquesta pràctica abarca totes les fases principals d'un procés de compilació, que estan dividides en dos grans blocs:

- **Front-end:** inclou l'anàlisi lèxic i sintàctic per validar la forma del codi font, així com l'anàlisi semàntic que garanteix la consistència de tipus i ús correcte dels identificadors. Per aquestes tasques ha estat imprescindible crear una Taula de Símbols i un Gestor d'errors complet.
- **Back-end:** Es centra en la generació de codi. El compilador tradueix el codi font a un codi intermedi, anomenat codi de 3 adreces i, posteriorment, genera el codi assemblador final per el microprocesador MC68k, preparat per ser executat amb l'entorn Easy68k.

## Eines i tecnologies

Per aquesta implementació del compilador s'ha utilitzat el llenguatge de programació Java. Les fases d'anàlisi del codi font s'han dut a terme mitjançant les eines explicades a classe que faciliten molt la construcció del compilador:

- **JFlex:** Emprat per la generació de l'analitzador lèxic, per l'*Scanner.*
- **JavaCUP:** Emprat per la generació de l'analitzador sintàctic, per el *Parser*. Està basat en una gramàtica LALR.
- **Easy68k**: Emprat per l'execució del codi assemblador que genera la fase final del compilador.

## Limitacions del llenguatge

El llenguatge tan sols admet tres tipus primitius, INT, CHAR i BOOL. Per poder treballar amb cadenes de caràcters serà necessari emprar taules de tipus CHAR.

A més, per fer declaracions globals tan sols les podrem declarar de forma CONSTANT. Això implica que no es poden tenir variables globals, sino que tan sols es poden definir constants i taules constants. Per a fer ús de variables s'han de crear dins el main.

També, el llenguatge només admet els comparadors de IGUAL, NO IGUAL, MENOR i MAJOR.

El llenguatge PepLang ofereix dos bucles bàsics el WHILE i el DO...WHILE. Per gestionar increments s'haurà de fer manualment dins el bucle.

La instrucció return només és possibles posarla al final de la funció. No és possible fer un return a la meitat del bloc de codi d'un IF.

La declaració de taules sempre ha de tenir especificat la mida de l'array en temps de compilació i ha de ser un nombre, no serveix una variable o constant definida anteriorment.

Per acabar, només és permeten operacions de SUMA, RESTA, MULT i DIV.

# Anàlisi Lèxic

Per la fase d'anàlisi lèxic s'ha utilitzat l'eina JFlex. Aquesta eina genera un autòmat finit determinista capaç de processar el fitxer d'entrada caràcter a caràcter i agrupar-los en tokens.

El fitxer es troba a *src/frontend/lexic/Lexic.flex.* Les tasques principals de l'escàner són:

- Ignorar els espais en blanc, salts de línia i comentaris de línia (//).
- Per cada patró reconegut, es crida al mètode *symbol(int type, Object val)*, que retorna un objecte *ComplexSymbol.* Aquest objecte conté el tipus de token, la ubicació exacta (línia i columna) per la gestió dels errors, i el valor associat (lexema) en cas de ser un literal o identificador.

A continuació, es mostra una llista dels tokens principals que s'han definit, classificats per categoria:

| Categoria | Tokens (CUP) | Patró / Exemple |
|---|---|---|
| **Estructura** | PROGRAMA, FUNCIO, RETURN, PROGRAMA_FI | programa, funcio, return, programa_fi |
| **Tipus** | INT, CHAR, BOOL, TAULA | int, char, bool, taula |
| **Control** | IF, ELSE, WHILE, DO | if, else, while, do |
| **Operadors** | SUMA, AND, IGUAL, ASSIGN | +, &&, ==, = |
| **Literals** | NUMERO, CARACTER, BTRUE | [0-9]+, 'c', true |

| | | |
|---|---|---|
| **Identificadors** | ID | [A-Za-z_][A-Za-z0-9_]* |

Un dels aspectes de l'analitzador lèxic és poder detectar caràcters normals i especials que comencen per una barra invertida (\), que a través d'un switch comprovam el valor després de la barra i podem detectar els següents casos: **\n**, **\t**, **\r**, \\ i **\'**. Finalment, es retorna un objecte Character què serà emprat posteriorment.

# Anàlisi Sintàctic

L'analitzador sintàctic s'ha generat mitjançant el JavaCUP. Aquest mòdul rep els tokens de l'escàner i verifica que la seqüència compleixi amb la gramàtica definida, construint l'arbre sintàctic abstracte (AST).

## Gramàtica

L'estructura de la gramàtica segueix l'ordre definit per la regla principal Peplang.

Primerament, tenim les declaracions globals i les taules globals. A continuació, tindrem la secció on aniran totes les funcions o subprograma addicionals que es vulguin crear. I finalment, tindrem el bloc del programa principal.

```
start with Peplang;
Peplang ::= Globs:g Funcs:f Main:m
;
Globs ::= Globs:g Decl_glob:d
|
;
Decl_glob ::= CONSTANT Tipusv:t ID:id ASSIGN Express:e FILINEA
| Decl_glob_taula:d FILINEA
;
Decl_glob_taula ::= CONSTANT TAULA INT ID:id Dims:d DeclTailTaulaInt:tail
| CONSTANT TAULA CHAR ID:id Dims:d DeclTailTaulaChar:tail
| CONSTANT TAULA BOOL ID:id Dims:d DeclTailTaulaBool:tail
;
```

```
Dims ::= LBRACKET Num:n RBRACKET
| Dims:l LBRACKET Num:n RBRACKET
;
Funcs ::= Funcs:fs Func:f
|
;
Func ::= FUNCIO Returnv:rv ID:id LPAREN ParamsOpt:p RPAREN BEGIN Elements:e
Returnfi:ret FUNCIO_FI
;
Returnv ::= Tipusv:t
|
;
ParamsOpt ::= ParamList:p
|
;
ParamList ::= Param:p
| ParamList:pl COMA Param:p
;
Param ::= Tipusv:t ID:id
| TAULA INT ID:id LBRACKET RBRACKET
| TAULA CHAR ID:id LBRACKET RBRACKET
| TAULA BOOL ID:id LBRACKET RBRACKET
;
Returnfi ::= RETURN ExpressOpt:e FILINEA
|
;

Main ::= PROGRAMA BEGIN Elements:e PROGRAMA_FI
;
Elements ::= Elements:es Element:e
|
;
Element ::= Decl_loc:dl
| Instr:i
;
Decl_loc ::= Tipusv:t ID:id DeclTailEscalar:dte FILINEA
| Decl_taula:dt FILINEA
;
DeclTailEscalar ::= ASSIGN Express:e
| ASSIGN LLEGIR LPAREN RPAREN
|
;
```

```
Decl_taula ::= TAULA INT ID:id Dims:d DeclTailTaulaInt:tail
| TAULA CHAR ID:id Dims:d DeclTailTaulaChar:tail
| TAULA BOOL ID:id Dims:d DeclTailTaulaBool:tail
;
DeclTailTaulaInt ::= ASSIGN ArrayLitInt:a
|
;
DeclTailTaulaChar ::= ASSIGN ArrayLitChar:a
|
;
DeclTailTaulaBool ::= ASSIGN ArrayLitBool:a
|
;
ArrayLitInt ::= LBRACKET IntElems:ie RBRACKET
;
IntElems ::= Num:n
| IntElems:ie COMA Num:n
;
ArrayLitChar ::= LBRACKET CharElems:ce RBRACKET
;
CharElems ::= CharLit:c
| CharElems:ce COMA CharLit:cl
;
ArrayLitBool ::= LBRACKET BoolElems:be RBRACKET
;
BoolElems ::= BoolLit:b
| BoolElems:be COMA BoolLit:b
;
Instr ::= If_prog:ifp
| While_prog:wh
| DoWhile_prog:dw
| AssignacioRead FILINEA:ar
| Assignacio:a FILINEA
| Crida_proc:c FILINEA
;
AssignacioRead ::= LValue:lv ASSIGN LLEGIR LPAREN RPAREN
;
Assignacio  ::= LValue:lv ASSIGN Express:e
;
LValue ::= Ref:r
;
Ref ::= ID:id
```

```
| Ref:r LBRACKET Express:e RBRACKET
;
If_prog ::= IF LPAREN Cond:c RPAREN BEGIN Elements:e If_fin:f
;
If_fin ::= IF_FI
| ELSE BEGIN Elements:e ELSE_FI
;
While_prog ::= WHILE LPAREN Cond:c RPAREN BEGIN Elements:e WHILE_FI
;
DoWhile_prog ::= DO BEGIN Elements:e DOWHILE LPAREN Cond:c RPAREN FILINEA
;
Crida_proc ::= IMPRIMIR LPAREN ArgsPrint:ap RPAREN
| CridaBase:c
;
Crida_func ::= CridaBase:c
;
CridaBase ::= ID:id LPAREN ArgsOpt:a RPAREN
;
ArgsPrint ::= ArgList:al
|
;
ArgsOpt ::= ArgList:al
|
;
ArgList ::= Express:e
| ArgList:a COMA Express:e
;
ExpressOpt ::= Express:e
|
;
Cond ::= Express:e
;
Express ::= Express:e1 SUMA Express:e2
| Express:e1 RESTA Express:e2
| Express:e1 MULT Express:e2
| Express:e1 DIV Express:e2
| Express:e1 AND Express:e2
| Express:e1 OR  Express:e2
| Express:e1 IGUAL Express:e2
| Express:e1 NOIGUAL Express:e2
| Express:e1 MENOR Express:e2
| Express:e1 MAJOR Express:e2
```

```
| NOT Express:e
| RESTA Express:e
%prec UMINUS
| LPAREN Express:e RPAREN
| Ref:r
| Num:n
| CharLit:c
| BoolLit:b
| Crida_func:c
;
Num ::= NUMERO:n
;
CharLit ::= CARACTER:c
;
BoolLit ::= BTRUE
| BFALSE
;
Tipusv ::= INT
| CHAR
| BOOL
;
```

A continuació, explicarem les principals produccions de la gramàtica de Peplang:

1. **Estructura general del programa:**
   El punt d'entrada de la gramàtica implica que un programa s'ha de dividir en 3 blocs seqüencials obligatoris, tot i que els dos primers poden ser buids.

   ```
   Peplang ::= Globs:g Funcs:f Main:m
   ```

   *Globs*, es el bloc de declaracions globals, *Funcs*, el de declaració de funcions i *Main*, el bloc principal d'execució. Aquesta separació simplifica la gestió dels àmbits, ja que assegura que totes les variables globals i funcions estiguin declarades abans de processar el bloc principal.

2. **Declaracions Globals:**
   Una característica de Peplang és que les declaracions globals estan restringides a ser constants, això es reflexa en la gramàtica, on el terminal CONSTANT és obligatori.

```
Decl_glob ::= CONSTANT Tipusv:t ID:id ASSIGN Express:e FILINEA
| Decl_glob_taula:d FILINEA
```

Això implica que no es poden tenir variables globals que canvien de valor, només s'admeten valors fixos, que poden ser accessibles desde qualsevol punt del programa.
Aquestes declaracions poden ser de tipus valor o de tipus taula.

3. **Definició de funcions:**
   Les funcions tenen una estructura encapsulada que defineix clarament la signatura, el cos i el valor de retorn.

   ```
   Func ::= FUNCIO Returnv:rv ID:id LPAREN ParamsOpt:p RPAREN BEGIN
   Elements:e Returnfi:ret FUNCIO_FI
   ```

   *Returnv*, és el tipus de retorn (pot estar buit), *ParamsOpt* és la llista de paràmetres que se li passen a la funció, aquests són específics de cada funció. També tenim el cos que son *Elements*, aquests poden ser instruccions i declaracions locals. Finalment, tenim *Returnfi*, que obliga que aquesta instrucció aparegui al final del la funció si *Returnv* no és buit.

4. **Instruccions de control de flux:**
   S'han definit estructures típiques. Cada una d'elles genera un Node diferent de l'AST per gestionar la generació d'etiquetes i salts posteriorment.

   - **Condicional**, la producció *If_fin* gestiona la part opcional de l'else, permetent condicionals simples o compostos:
     ```
     If_prog ::= IF LPAREN Cond:c RPAREN BEGIN Elements:e If_fin:f
     ```

   - **Bucles,** aquestes regles permeten la iteració basada en una condició booleana *Cond.* En la part d'elements hi hauria un bloc complet.
     ```
     While_prog ::= WHILE LPAREN Cond:c RPAREN BEGIN Elements:e WHILE_FI

     DoWhile_prog ::= DO BEGIN Elements:e DOWHILE LPAREN Cond:c RPAREN
     FILINEA
     ```

5. **Assignacions i Expressions:**
   L'assignació és la instrucció bàsica per modificar l'estat del programa. La gramàtica distingeix entre assignació d'expressions i la lectura per teclat.

```
Assignacio  ::= LValue:lv ASSIGN Express:e

AssignacioRead ::= LValue:lv ASSIGN LLEGIR LPAREN RPAREN
```

   *LValue*, representa el que hi ha a l'esquerra, pot ser un identificador o un accés a array; ID o ID[Express]. *Express*, defineix tota la lògica d'operacions (aritmètiques, lògiques i relacionals). La precedència d'operadors s'ha resolt mitjançant les directives de *precedende* de CUP, així evitar ambigüitats.

6. **Crides a Funcions i Procediments:**
   Aquí, es diferencia entre la crida com a instrucció i la crida com a part d'una expressió (funció que retorna valor).

```
Crida_proc ::= IMPRIMIR LPAREN ArgsPrint:ap RPAREN
| CridaBase:c

Crida_func ::= CridaBase:c

CridaBase ::= ID:id LPAREN ArgsOpt:a RPAREN
```

   Això ens permet que IMPRIMIR es tracti com una instrucció nativa especial, mentre que la resta de crides *cridaBase* segueixen la sintaxi normal.

## Altres aspectes

A més, s'ha optat per la construcció d'un arbre explícit, on cada regla de producció té associada una acció semàntica que instancia una classe específica del paquet *frontend/ast*.

- Un exemple seria: `Espress:e1 SUMA Express:e2` crea un nou objecte `Node_Express(Operador.SUMA, e1, e2)`

També, s'han sobreescrit dos mètodes de recuperació d'errors de CUP per integrar-los al nostre gestor d'errors:

- ***syntax_error()***: detecta l'error però intenta recuperar-se per continuar l'anàlisi i trobar més errors.
- ***report_error()***: el que fa és afegir l'error al nostre gestor d'errors.
- ***unrecovered_syntax_error()***: es llança quan el parser no pot continuar.

# Anàlisi Semàntic

Aquesta fase s'encarrega de validar el significat del programa, assegurant que les construccions sintàctiques siguin lògiques. En el nostre compilador, aquesta fase es realitza mitjançant un recorregut de l'Arbre Sintàctic Abstracta (AST), on cada node valida els seus fills i propaga la informació de tipus cap els pares.

Els punts més importants d'aquesta part són la gestió d'àmbits i la comprovació de tipus, fent ús de la Taula de Símbols generada.

## Taula de Símbols

La classe TaulaSimbols actua com un punt central d'informació de tot el compilador. S'ha dissenyat utilitzant una estructura de piles per suportar la recursivitat i els àmbits locals.

### Estructura de dades

El sistema basat en piles, simula el comportament d'entrada i sortida dels blocs.

- **pilaAmbits:** cada element de la pila representa un àmbit o bloc. El fons de la pila és sempre un àmbit global i el cim de la pila sempre serà l'àmbit local actual. Quan es cerca un símbol amb *cercarSimbol* es recorre la pila de dalt a baix, això permet que una variables local "oculti" una variable global amb el mateix nom.
- **pilaOffsets**: juntament amb els àmbits, també es gestiona una pila de desplaçaments de memòria. Això ens permet calcular automàticament l'offset de cada variabla local respecte al punter de marc, és imprescindible per a la generació de codi posterior.

### Persistència

Un problema que tenim quan feim un *sortirBloc,* la informació de les variables locals es perd. Per solucionar això s'ha inclòs una llista *taulaCompleta* que emmagatzema tot l'històric de símbols que s'han declarat, independentment del seu àmbit. Això ens permet poder guardar la taula de símbols a un fitxer i també poder recuperar informació de variables locals antigues.

### Gestió de declaracions i àmbits

- Entrar bloc (*entrarBloc*): quan l'analitzador entra a una funció o estructura de control, es fa un push d'una nova taula buida a la pila. Si és una funció, es reinicia el comptador d'offsets locals a 0.
- Afegir simbol (*afegirSimbol*): abans d'inserir una nova variable, es verifica que no existeixi ja en l'àmbit actual. S'assigna el seu tipus, categoria i es calcula la seva ocupació en bytes.
- Sortir bloc (*sortirBloc*): en finalitzar el bloc, es fa pop de la taula actual, alliberant les variables locals de l'àmbit de visibilitat, però mantenint-les a la *taulaCompleta*.

### Verificació de tipus

La validació de tipus s'implementa principalment a la classe *Node_Express*. El compilador de Peplang és **fortament tipat,** la qual cosa significa que no permet operacions entre tipus incompatibles.

El mètode *getTipusSimbol(TaulaSimbols ts)* implementa la lògica de comprovació de tipus:

- **Aritmètica i Lògica**
  - **Operacions aritmètiques**: requereixen si o si dos operands i que siguin exactament del mateix tipus (INT). Si es detecta un BOOL o CHAR, es llança un error semàntic i es retorna un tipus de símbol ERROR.
  - **Operacions lògiques**: només operen sobre expressions que el resultat sigui un BOOL.
- **Comparacions i Relacionals**

Les operacions com IGUAL (==) o NOIGUAL (!=) permeten comparar entre dos enters o dos booleans, pero exigeixen que ambdós costats siguin del mateix tipus.

Per els operands d'ordre (<, >), es restringeix l'ús a tipus ordenables com INT i CHAR.

- **Referències**

  Quan l'expressió és una variable (node REF), es consulta la Taula de Símbols per assegurar-se que la variable ha estat declarada i per recuperar el seu tipus associat per propagar-lo cap amunt a l'arbre.

## Gestió d'Errors Semàntics

Quan detectam un error, empram el GestorError. A diferència dels errors sintàctics que a vegades aturen l'execució, els errors semàntics es registren però intenten permetre que l'anàlisi continuï per detectar tants errors com sigui possible en una sola compilació.

L'error típic inclou el tipus d'error, a n'aquest cas tipus SEMÀNTIC, la línia on s'ha produït i el missatge que descriu l'error.

## Classes associades a la Taula de Símbols

1. **Classe Símbol**

   Aquesta classe és fonamental que encapsula tota la informació associada a un identificador. Degut a la complexitat del compilador, la classe compte camps per a diferents propòsits, perquè s'han de gestionar desde variables simples fins arrays i funcions.

   Emmagatzema principalment el nom, el tipus de símbol i la categoría del símbol. Per la part de gestió de memòria, té tres camps crítics com n'és l'*offset* (direcció relativa de la variable dins la pila), l'*àmbit* (nom de l'àmbit al qual pertany) i *esGlobal* (indicador booleà per saber si el símbol és global o no).

   A més, conté atributs per les estructures complexes, com *esArray* i *midaArray* per gestionar la reserva d'espai dins la memòria. I també, emmagatzema *midaFrame*

que és el total de bytes de les variables locals i, la *llistaParametres*, necessària per validar les crides.

2. **Classe Paràmetre**

   Aquesta classe serveix per definir els arguments d'una funció. A diferència d'un símbol genèric, un paràmetre està lligat a una posició específica dins la crida.

   L'atribut *posicio*, indica l'índex que ocupa en la llista d'arguments, l'*offset* és el desplaçament calculat (essencial perquè el codi assemblador pugui trobar-lo dins la pila).

3. **Enum TipusSimbol**

   Defineix els tipus de dades suportats com INT, BOOL, CHAR, TAULA_INT, TAULA_BOOL i TAULA_CHAR, VOID, NULL i ERROR.

   Inclu el mètode getMidaBytes(), que retorna l'espai que ocupa cada tipus bàsic: 4 bytes per INT i BOOL i 1 byte per CHAR.

4. **Enum Categoria**

   Permet determinar el "rol" d'un identificador. Els valors possibles son CONSTANT, VARIABLE, PARAMETRE, FUNCIO, PROCEDIMENT, TEMPORAL i NULL. El tipus TEMPORAL el feim servir per el codi de 3 adreçes per generar les variables temporals.

5. **Classe TipusUtils**

   La funció principal de la classe és actuar com un conversor entre les representacions textuals dels tipus i els valors interns d'enumeració TipusSimbol.

   *getTipusBaseDesdeNomBase(String base)*: transforma les cadenes que venen de la gramàtica (com "INT", "BOOL" o "CARACTER") en els seus corresponents TipusSimbol (INT, BOOL, CHAR).

   *getTipusArrayDesdeNomBase(String base):* permet obtenir directament el tipus compost corresponent (TAULA_INT, TAULA_BOOL, etc.) sense haver de crear lògica complexa al CUP.

*getTipusBaseDeTipusArray(TipusSimbol tArray):* realitza l'operació inversa: donat un tipus de taula, retorna el tipus dels elements que conté (per exemple, de TAULA_INT retorna INT).

# Codi Intermedi

Entre l'anàlisi semàntica i la generació de codi màquina, el compilador realitza una traducció a una representació intermèdia anomenada Codi de 3 Adreces (C3A).

### Representació

S'ha implementat un sistema basat en Quàdruples, on cada instrucció està formada per quatre elements bàsics, més una etiqueta opcional. Aquesta estructura està definida a la classe *C3a_Instr*:

- **Etiqueta**: Identificador opcional (ex: e1:) que marca el punt d'entrada per als salts.
- **Operació (Codi)**: L'acció a realitzar (suma, salt condicional, assignació, etc.). S'ha utilitzat un Enum per garantir la integritat dels tipus.
- **Arguments (Arg1 i Arg2)**: Els operands, que poden ser variables del codi font, constants literals o variables temporals generades pel compilador.
- **Destí**: El lloc on s'emmagatzema el resultat de l'operació o, en el cas de salts, l'etiqueta on saltar.

El gestor global d'aquesta fase és la classe **C3a**, que manté un ArrayList de totes les instruccions generades.

### Gestió de Temporals i Etiquetes

Com que el C3A "aplana" les expressions complexes (ex: a = b + c * d), el compilador necessita crear variables auxiliars per guardar els resultats intermedis.

- **Variables Temporals:** El mètode *novaTemp()* genera identificadors únics (t0, t1, t2...) de manera seqüencial. Aquestes variables s'utilitzen per guardar el resultat de c * d abans de sumar-lo amb b.
- **Etiquetes:** Per gestionar el control de flux (bucles i condicionals), el mètode *novaEtiqueta()* genera marcadors (e0, e1...) que serviran de destí per a les instruccions GOTO o salts condicionals.

**Conjunt d'Instruccions**

Les instruccions intermèdies s'han dissenyat per poder representar qualsevol algorisme de manera simple, però sense arribar a ser assemblador.

Les principals categories que tenim són:

- **Aritmètiques i Lògiques**: ADD, SUB, PROD, DIV, AND, OR, NOT, NEG.
- **Transferència de Dades:** COPY (assignació simple), IND_VAL i IND_ASS (per accedir a arrays mitjançant índexs: x = y[i] o y[i] = x).
- **Control de Flux:**
    - Incondicional: GOTO, SKIP (etiqueta buida).
    - Condicional: IF_EQ, IF_NE, IF_LT, IF_GT. Aquests salts avaluen una condició i, si es compleix, salten a l'etiqueta destí.
- **Gestió de Procediments:**
    - PMB (Preàmbul): Marca l'inici d'una funció i la reserva d'espai a la pila.
    - PARAM_S / PARAM_C: Pas de paràmetres (simple o complex).
    - CALL: Crida efectiva a la subrutina.
    - RET: Retorn de funció.

**Generació Recursiva i Backpatching**

La generació del codi es realitza recorrent l'AST. Cada node té un mètode *generaCodi3a(C3a codi)* que afegeix les instruccions necessàries a la llista global i retorna el nom de la variable (temporal o no) on ha deixat el resultat.

Per solucionar el problema dels salts endavant (quan s'ha de generar un salt a una etiqueta que encara no sabem on es col·locarà), s'ha implementat una tècnica de

Backpatching. Els mètodes *modificaDesti* o *modificaArg1* permeten actualitzar una instrucció ja generada un cop es coneix l'etiqueta de destí correcta.

# Codi Assemblador

Aquesta darrera fase de compilació és la traducció del C3A a un llenguatge màquina executable. S'ha escollit l'arquitectura MC68000 i el simulador Easy68k com a plataforma de destí. S'ha elegit així perquè és el llenguatge assemblador que més per mà es té.

### Gestió de la memòria

El compilador implementa un model de gestió de memòria basat en pila per suportar la recursivitat i les crides a funcions. S'utilitza el registre A6 com a Frame Pointer.

- **Entrada a funció (PMB):** quan es processa la instrucció preàmbul, es genera la instrucció LINK A6, #-mida. Això guarda l'antic A6 a la pila i reserva espai per a totes les variables locals i temporals de la funció en una sola passa.
- **Sortida de funció (RET):** es genera la seqüència UNLK A6 seguit de RTS, que allibera l'espai local i restaura el punter del marc anterior.

### Traducció d'Operands

El mètode *traduirOperand* resol la complexitat de determinar l'adreça efectiva per a cada cas:

- **Literals:** Si l'operand és un nombre, s'afegeix el prefix # (ex: #5).
- **Variables Globals:** S'accedeix directament per la seva etiqueta (definida a la secció de dades amb DS.L).
- **Variables Locals i Temporals**: Es calculen amb un desplaçament negatiu respecte a A6 (ex: -4(A6)). Els temporals es tracten com a variables locals addicionals.

- **Paràmetres:** Es calculen amb un desplaçament positiu respecte a A6 (ex: 8(A6)), ja que es troben "sota" l'adreça de retorn a la pila.

### Patrons d'Instruccions

La traducció d'instruccions és pràcticament directa, però amb algunes particularitats de l'arquitectura:

- **Aritmètica (MUL/DIV):** El MC68000 té limitacions en multiplicar i dividir a 32 bits directament. S'han utilitzat les instruccions MULS.W i DIVS.W, que operen amb entrades de 16 bits.
- **Control de Flux:** Les instruccions condicionals del C3A (IF_EQ, IF_LT...) es tradueixen en dues instruccions de màquina: una comparació CMP.L seguida d'un salt condicional (BEQ, BLT, etc.).
- **Entrada / Sortida:** S'han implementat subrutines que fan ús de les interrupcions del simulador (TRAP #15):
    - **Lectura:** Tasca 4 (MOVE.L #4, D0).
    - **Escriptura:** Tasca 3 per nombres i Tasca 6 per caràcters.

### Estructura del Fitxer Resultant

El fitxer .X68 generat segueix una estructura perque l'el codi assemblador sigui correcte: una capçalera: ORG $1000 i inicialització de la pila (LEA STACK_TOP, A7). Una secció de dades on es reserva espai per variables globals i la pila del sistema. Una secció de codi on apareixen les intruccions del programa principal i subrutines. I finalment, el codi de les funcions de llegir i imprimir que s'afegeixen al final.

# Conclusió

El desenvolupament d'aquest projecte ha permès assolir l'objectiu principal de l'assignatura: la implementació d'un compilador complet i funcional per al llenguatge Peplang, capaç de traduir codi font d'alt nivell a codi màquina per a l'arquitectura MC68000.

Durant el procés, s'ha constatat la importància d'una arquitectura modular. La separació clara entre el Front-end (anàlisi lèxica, sintàctica i semàntica) i el Back-end (generació de codi) ha facilitat la detecció d'errors i la depuració. Eines com JFlex i JavaCUP han estat fonamentals per automatitzar les primeres fases, permetent centrar els esforços en la lògica semàntica.

Un dels reptes més significatius ha estat el disseny de la Taula de Símbols i la gestió dels àmbits mitjançant piles, així com la traducció al Codi de 3 Adreces (C3A).

Finalment, cal destacar que el desenvolupament s'ha centrat prioritàriament en la correcció funcional i la robustesa del compilador. Per aquest motiu, no s'han aplicat tècniques d'optimització, optant per una generació de codi directa i segura. Aquest projecte ens ha proporcionat una visió profunda del món dels compiladors, des de la comprovació de tipus fins a la manipulació directa de la pila i els registres del processador.