

Compilador de Llenguatge Imperatiu

Grup: "Serveritx"

Membres:

Bernat Parera Servera 41617818P

Pere Roig Verdera 43480786E

Video explicatiu: [Compilador](#)

Índex

Índex.....	0
1. Enunciat.....	1
2. Característiques Implementades.....	2
3. Anàlisis lèxic.....	3
4. Anàlisis sintàctic.....	5
4.1 Gramàtica.....	6
4.2 Errors sintàctics.....	11
5. Anàlisis semàntic.....	12
6. Codi tres adreces.....	16
7. Generació de codi.....	20
8. Optimitzacions.....	0

1. Enunciat

Desenvolupar un compilador per a un llenguatge imperatiu.

La pràctica consisteix en el disseny i implementació d'un compilador per a un llenguatge de programació. Les tasques que haurà de realitzar el processador són:

- Les pròpies de la component front-end:
 - Anàlisi lèxica
 - Anàlisi sintàctica o Anàlisi semàntica
- Les pròpies de la component back-end:
 - Generació de codi intermedi
 - Optimització
 - Generació de codi ensamblador
- S'haurà de contemplar necessàriament la implementació d'una taula de símbols i un gestor d'errors

2. Característiques Implementades

S'ha implementat un llenguatge imperatiu simple.

Aquestes són els diferents apartats s'han implementat sobre el llenguatge.

- Un cos general de programa on hi hagi d'haver els subprogrames, les declaracions i les instruccions del programa de l'estil main de java.
- Definició de subprogrames: funcions o procediments, amb arguments
- Tipus primitius:
 - Enter
 - Lògic booleà (TRUE i FALSE es representa amb -1 i 0 respectivament)
- Tipus definits per l'usuari
 - Tuples
- Valors de qualsevulla dels tipus contemplats
 - Declaració i ús de variables(Els tipus primitius enter i booleà declarats sense assignació tendran el valor 0 i FALSE respectivament a l'estil de Java)
 - Constants
- Operacions:
 - Assignació
 - Condicionals
 - Bucles : while i for
 - Crida a procediments i funcions amb paràmetres
 - Retorn de funcions (el valor de retorn per defecte de les funcions és 0 i FALSE)
- Expressions aritmètiques i lògiques:
 - Fent ús de literals del tipus adient
 - Fent ús de constants i variables
- Operacions d'entrada i sortida
 - Entrada per teclat (nombres enters)
 - Sortida per pantalla
- Operadors :
 - Aritmètics : suma, resta, producte i divisió
 - Relacionals : igual, diferent, major, menor, major o igual, menor o igual
 - Lògics : i, o, no, o-excloent , no-and, no-o-excloent, no-o
- Memòria : S'ha optat per una estructura estàtica en quant a la gestió de la memòria, per tant no es pot plantejar la recursivitat.

3. Anàlisis lèxic

L'anàlisis lèxic s'ha implementat mitjançant JFlex.

Aquests són els tokens y expressions regulars que defineixen el llenguatge.

```
LÈXIC

digit = [0-9]
noZeroDigit = [1-9]
digits = 0 | ({noZeroDigit}{digit}*)
noDigit = 0{digit}+
letter = [A-Za-z]
space = [ ,\t,\r,\n]
id = {letter}({letter}|{digit}|_)*
```

Paraules reservades:

```
functionKey = function
printlnKey = println
returnKey = return
ifKey = if
elseKey = else
forKey = for
whileKey = while
constKey = const
mainKey = main
trueKey = TRUE
falseKey = FALSE
inputKey = input
printKey = print
dot = .
```

Tipus:

```
booleanKey = bool
integerKey = int
tupleKey = tuple
```

Assignacio:

```
equalKey = =
```

Operacions:

```
sumOp = +
minOp = -
mulOp = *
divOp = /
```

LÈXIC

Relacional:

equalRel = ==

notEqualRel = !=

greaterRel = >

lessRel = <

greaterEqRel = >=

lessEqRel = <=

Logic:

andLogic = AND

orLogic = OR

notLogic = NOT

xorLogic = XOR

nandLogic = NAND

norLogic = NOR

xnorLogic = XNOR

Final de línia i coma:

eol = ;

comma = ,

Blocs:

openParenthesis = (

closeParenthesis =)

openBracket = {

closeBracket = }

Comentarios:

comment = \\/[^\n]*

blockComment = \\/ * ([^*]|*[^\\/])*\ *\/

4. Anàlisi sintàctic

Una vegada s'han definit els token hem de verificar la estructura del programa i definir la estructura del llenguatge, és a dir, implementar un analitzador sintàctic.

L'anàlisi sintàctic s'ha implementat mitjançant CUP que consisteix en un generador de parsers LALR(1) que es complementa amb JFlex.

CUP també permet implementar l'anàlisi semàntica i en parlarem a l'apartat corresponent.

Cal destacar els marcadors que hem afegit a diferents parts de la gramàtica, el seu funcionament està explicat a l'apartat d'anàlisi semàntic.

4.1 Gramàtica

```
SINTÀCTIC

PROGRAM → function main openb BLOCK closeb

BLOCK → LINE LINEP
      | lambda

LINEP → LINE LINEP
      | lambda

LINE → DECLARATION semicolon
      | ASSIGNATION semicolon
      | PRINT semicolon
      | INPUT semicolon
      | FUNCALL semicolon
      | RETURN semicolon
      | CONDITIONAL
      | FUNCPROCDECLARATION
      | TUPLE

ID → id

DECLARATION → constt TYPE DECLARATIONP
            | TYPE DECLARATIONP

DECLARATIONP → id
             | ASSIGNATION

ASSIGNATION → ID equal ASSIGNATIONP

ASSIGNATIONP → E
             | TUPLECALL
             | FUNCALL

TYPE → integer
      | bool

TYP → integer
     | bool
     | tuple

FUNCPROCDECLARATION → FUNPROC BODY

FUNPROC → function TYPE id
        | function id

BODY → openp PARAMETERS closep openb BLOCK closeb

PARAMETERS → PARAMETER comma PARAMETERP
           | PARAMETER
           | lambda

PARAMETER → TYPE id

PARAMETERP → PARAMETER comma PARAMETERP
           | PARAMETER
```

SINTÀCTIC

```
IDF → id

FUNCALL → IDF openp PARAMS closep

PARAMS → PARAM PARAM
         | lambda

PARAM → E

PARAMP → comma PARAM PARAMP
        | lambda

PRINT → PRINTP openp E closep

PRINTP → print
        | println

TUPLE → tuple ID openb TUPLEP

TUPLEP → TYP id equal E semicolon TUPLEPP

TUPLEPP → TUPLEP
         | closeb

TUPLECALL → id dot id TCP

TCP → dot id TCP
     | lamda

INPUT → input openp id closep

RETURN → returnt E

E → E sumop E
   | E minop E
   | E mulop E
   | E divop E
   | E and E
   | E or E
   | E xor E
   | E nand E
   | E nor E
   | E xnor E
   | E equalrel E
   | E notequalrel E
   | E greaterrel E
   | E lessrel E
   | E greaterreqrel E
   | E lesseqrel E
   | id
   | minop id
   | number
   | minop number
   | truet
   | falset
   | not id
   | not truet
   | not falset
```



```
SINTÀNTIC

CONDITIONAL → IF
              | WHILE
              | FOR

IF → IFP ELSE

IFP → ift openp CONDITION closep openb BLOCK closeb
    | ift openp CONDITION closep LINE

ELSE → elset openb BLOCK clsoeb
      | elset LINE
      | lambda

WHILE → whilet openp CONDITION closep openb BLOCK closeb

FOR → FORDEC CODE

FORDEC → fort openp DECL comma CONDITIONFOR comma ASSIGN closep

CODE → open BLOCK closeb

DECL → DECLARATION
      | lambda

ASSIGN → ASSIGNATION
        | lambda

CONDITION → E

CONDITIONFOR → CONDITION
```

Explicació de la gramàtica de dalt a baix

Un programa és un main on dins del main hi ha un bloc de codi.

Un bloc de codi es basa en 0 a n línees de codi.

Una línia de codi tant pot ser:

- Una declaració
- Una assignació
- Una impressió per pantalla
- Una entrada de dades per terminal
- Crida a funció
- Un retorn de valors (tenint en compte que ha de estar dins una funció)
- Un condicional (if, while o for)
- Una declaració de funció
- Una declaració d'una tupla

Una declaració tant pot ser una constant com una variable, és d'un tipus (tipat estàtic) i tant pots fer la declaració amb o sense assignació (tenint en compte que en cas de ser una declaració sense assignació se assignarà el valor 0 o FALSE segons el tipus).

Una assignació tant pot ser una línia per ella mateixa, per exemple: `a = 2`. O també part de una declaració, per exemple: `int a = 2`.

Per altre banda el '2' dels exemples pot ser tres possibilitats

- Una expressió: `2`, `2+2`, `3*5+2*a`.
- Una crida a una tupla: `tupla.a` (on 'a' és un element de la tupla 'tuple').
- Una crida a una funció: `suma1(1)`

Cada una d'aquestes està explicada més avall.

TYPE i TYP son la producció per indicar el tipus a una declaració, TYPE s'usa a la declaració de funcions, declaracions de constants i variables i de paràmetres, mentre que TYP, que conté també el tipus tuple només s'usa dins les declaracions de les tuples per poder inserir una tupla dins una altre.

Una declaració de una funció i d'un procediment té la mateixa estructura, la capçalera (FUNPROC) i un cos (BODY)

La capçalera indica si es tracta d'una funció o un procediment, la diferència està en que una funció tindrà un tipus. Per altre banda el cos de la declaració a nivell de sintàctic és el mateix tant si és una funció com un procediment (la diferència està en el semàntic, una funció ha de tenir mínim un retorn obligatoriament)

La producció PARAMETERS permet declarar funcions i procediments amb paràmetres, un paràmetre és un tipus seguit del seu identificador.

Una declaració d'una funció quedaria així: `function TIPUS nom(int a, bool b){BLOC}`, on tindrem tants paràmetres com es desitgi, amb el tipus escollit. TIPUS és opcional, en el cas de una funció TIPUS indicarà el tipus del retorn i en el cas d'un procediment s'ometrà.

La crida a funcions es basa en l'identificador de la funció a la que volem cridar seguit de paràmetres, el tipus i correctesa dels paràmetres ho deixam al semàntic amb l'ajuda de la taula de símbols.

Les crides es troben en dos contexts, com una línia (procediments o funcions) o a una assignació (només funció).

Posteriorment trobem la sortida per pantalla on tenim la opció de mostrar per pantalla normal o fer-ho amb un bot de línia al final, en el cas de mostrar un booleà per pantalla, mostrarem un -1 pel valor TRUE i un 0 pel valor FALSE.

Seguit trobem la declaració de tuples, la qual es una estructura per declarar variables agrupades. Un exemple de declaració de tupla és `tuple t{int a = 2;}` on podríem declarar tants atributs com volguem dels tipus: `int`, `bool` i `tuple`. Per afegir una tupla a una altre ha de ser declarada abans.

Les següents produccions manegen les crides a tuples, per accedir a cada un dels paràmetres d'una tupla, aquesta producció només es troba a la dreta d'una assignació. És a dir, que per usar un paràmetre d'una tupla a una funció o a una operació hauriem de declarar una variable auxiliar amb la crida a la tupla i després usar aquella variable auxiliar. Amb l'exemple anterior la crida al membre a de la tupla t seria t.a, per altre banda suposem que tenim la tupla anterior i una altre tupla: tuple t2{tuple altre= t}, la cridada: t2.altre.a, funcionaria retornant el valor de a de la tupla t.

Seguit trobem la entrada per teclat la qual té un paràmetre que és la variable(int) a la que assignarem el valor introduït per teclat, un exemple és: input(a);.

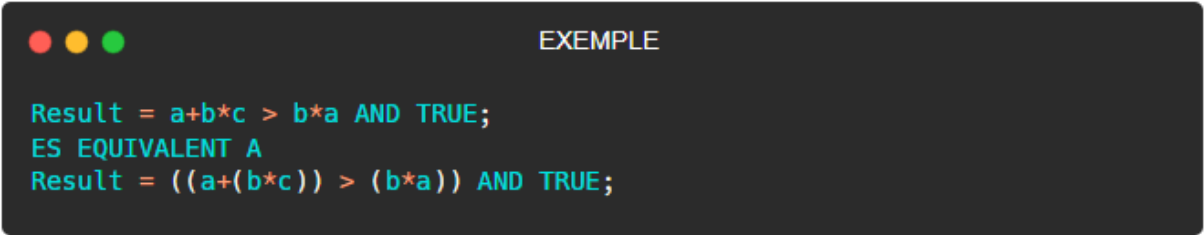
La següent producció és un return el qual va seguit d'una expressió, serveix per retornar un valor a una funció per exemple: return 1+2; o return TRUE;

A continuació ve la producció més gran: les expressions. Cal destacar que hem decidit per fer expressions sense parentesis ja que tot i ser més còmode operar amb ells son prescindibles a l'hora que podem declarar variables. Si volem escriure res=(a+b)*c es pot convertir en int aux = a+b; res=aux*c i donarà el mateix resultat. Tot i això les nostres expressions suporten les operacions aritmètiques: suma, resta, multiplicació i divisió. Per la part booleana: and, or, xor, nand, nor, xnor i per la part relacional, ==, !=, <, <=, > i >=. A més es pot negar un terminal booleà o identificador, per exemple: NOT TRUE, NOT a, i també, es pot operar amb el negatiu un nombre o identificador, per exemple -2.

És important destacar que no es poden escriure signes seguits.

Ahora de assegurar que no hi ha dos signes seguits com per exemple: 2- - 4 s'encarregarà el semàntic.

L'ordre de precedència és: Primer de tot els relacionals ja que si tenim 2+1>2, el resultat que volem és saber si 2+1 és major que 2. Posteriorment tendrem els booleans ja que TRUE OR FALSE == FALSE volem que l'ordre sigui (TRUE OR FALSE) == FALSE. Finalment serà la prioritat de les operacions aritmètiques, primer multiplicació i divisió i finalment sumes i restes. Per juntar tot això tenim que si volguessim fer:



```
Result = a+b*c > b*a AND TRUE;
ES EQUIVALENT A
Result = ((a+(b*c)) > (b*a)) AND TRUE;
```

Ja que la prioritat va d'esquerra a dreta i agrupam per la prioritat que hem dit (relacionals, booleans, aritmètics) i per a =1, b=2, c =3 Result és TRUE

Finalment ens trobem amb els condicionals. Tenim 3 condicionals i gràcies a aquesta pràctica hem entès que els tres tipus són el mateix amb les seves particularitats i ho demostrarem a mesura que avança la documentació.

Per començar l'if seria el més senzill si no fos per la clausula else del final i l'if i l'else sense bloc de codi (només línia). Veim que el if es separa en if else.

L'if té una condició (una expressió booleana) seguit de un bloc de codi o una línia de codi i un else (o no).

L'else per la seva part pot no existir, ser un else d'una línia o ser un else d'un bloc de codi.

El següent és el while. Aquest hem trobat que en total és el més senzill ja que no és més que un if sense else que al final de la seva execució torna a fer el if del principi. Entendre-ho així ens ha permès ahorrar codi amb la producció "CONDITION" (Sobretot a l'hora de generar el codi tres adreces).

El canvi més gran és el bucle for. Aquest és un bucle while però amb una declaració abans i una assignació que es fa cada pic que s'executa. A nivell de semàntic no té molta complicació, és el mateix que el while, però afegint una declaració omissible a l'esquerra de la condició i una assignació també omissible a la dreta de la condició separat per comes, tot i que per a la generació de codi 3 adreces la condició de de la producció del for és una producció diferent per el funcionament diferent que té aquest condicional.

4.2 Errors sintàctics

El compilador no és capaç de recuperar-se de els errors sintàctics, informa d'on es troba el primer error i, en general, informació sobre el caràcter de l'error, però en cas d'haver-hi varis el procés s'aturarà al primer i s'haurà de corregir el primer i tornar a compilar el programa per detectar el segon.

5. Anàlisi semàntic

L'Anàlisi semàntic és la comprovació de la correctesa del codi tot i estar ben estructurat.

L'exemple més senzill és `int a = TRUE;` A nivell sintàctic és correcte ja que `TRUE` és una expressió i i la declaració d'`a` està ben feta. El problema és que `TRUE` no és un enter, doncs tenim un error de tipus en la declaració d'`a`. Aquest exemple veim que és senzill però si ens trobem amb el següent codi. Com ho feim?

```
EXAMPLE

int a = 2;
bool b = a;
```

Sabem que la primera sentència és correcte ja que 2 és un enter. La segona línia ja no ho podem saber ja que. Com sabem si '`a`' és un booleà?

La resposta radica en la Taula de Símbols. És una estructura de dades on emmagatzemarem tots els símbols que trobem tals com: variables, constants, tuples (amb els seus camps) i funcions (amb els seus paràmetres). Implementar la taula de símbols va ser un dels reptes més grans de la pràctica ja que primer vam haver de comprendre els conceptes teòrics per després aplicar-los a la pràctica.

Abans d'entrar amb la nostra implementació cal explicar el concepte de nivell, o almenys el que nosaltres hem entès. Cada pic que entrem dins un bloc de codi (BLOCK) entrem a un nou nivell, això ens permetrà declarar una variable que ja existeix. per exemple:

```
EXAMPLE

function main{
    int a = 2
    if(a==2){
        int a = 3;
        print(a);
    }
    print(a);
}
```

Aquest programa seria correcte i mostra: 32, ja que al entrar dins l'`if`, entrem dins un altre nivell i podem declarar una altra variable '`a`' però al sortir de l'`if`, eliminam aquesta nova '`a`' i tornam a la del nivell 1.

Al nostre cas vam optar per una solució fidel a la teoria vista a classe i tenim tres taules:

- Taula d'àmbits (ta): la qual marca la darrera posició de la taula de expansió escrita al nivell.
- Taula de expansió (te): Aquí es emmagatzemaran els símbols no accessibles per el programador directament (tals com paràmetres de funcions i elements de les tuples) i també els símbols que se amaguen per canvi de nivell, per exemple, si tenim declarada una variable '`a`' a nivell de `main` però després tenim un paràmetre '`a`' dins una funció a l'hora de estar dins la funció, la variable '`a`' quedarà emmagatzemada a la taula de expansió i el paràmetre '`a`' serà el accessible (com veim a l'exemple però amb una sentència `if`).

- Taula de descripció (td): aquí s'emmagatzemaràn els símbols accessibles per el programador al nivell actual. Es podria dir que llevant parametres de funcions i tuples aquesta és la taula important i les altres dues son suplementàries per manejar els nivells

Tenim diferents procediments i funcions per a la gestió semàntica que s'aniràn mencionant i explicant a mesura que s'usen.

Regles semàntiques:

Recopilació de les regles semàntiques més interessants:

BLOCK:

Tots els bloc van precedits del marcador M0 el qual la seva única funció és entrar dins el nou nivell a la taula de símbols i al sortir del bloc és surt del nivell.

Per altre banda també s'observa un codi que es repeteix per les diferents produccions el qual compara line i linep el qual serveix per el return.

FUNCIONS I RETURN:

Per gestionar els return se'ns va ocórrer una solució per saber si una funció te mínim un return. Cada pic que hi hagi un retorn, aquest "pujarà" el tipus del retorn i compararem el tipus dins tot el bloc de codi. En el cas de trobar dos retorns de tipus diferents tenim un error semàntic, però tant si només hi ha un tipus o si no hi ha cap, seguirem normal. En el cas de que sortim de un bloc de codi i tenim que el valor de aquell bloc és diferent que buit sabem que dins aquell bloc hi ha un retorn, i si, aquell bloc no és d'una funció això és un error semàntic (return fora de funció). També ens haurem d'assegurar de que el tipus del retorn és el mateix que el de la funció.

Com que no ens podem assegurar que tot i tenir un return aquest s'executi varem pensar que el retorn "estàndard" és 0 o FALSE segons el tipus de la funció.

Exemple en el que una funció té un return però no s'executarà i retornarà el valor estàndard:



```

function main{
  function int a(){
    if(1==2){
      return 22;
    }
  }
}

```

DECLARACIÓ I ASSIGNACIÓ:

Per les declaracions i assignacions l'anàlisi semàntic radica en comprovació de tipus usant la taula de símbols. A l'assignació també es comprova que no estem editant una constant. A més hem de comprovar si ja existeix un símbol amb aquest identificador i a les declaracions al cas d'existir tindrem un error (declarar un símbol ja declarat) i l'assignació si no existeix també tindrem un error (usar identificador sense existir).

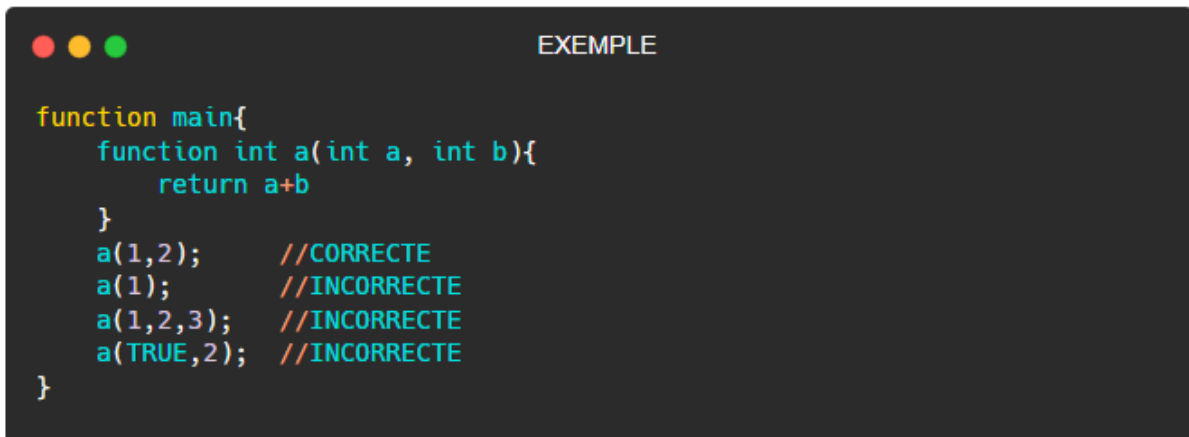
DECLARACIÓ DE FUNCIONS I PROCEDIMENTS:

Apart del que ja hem explicat dels returns per les funcions i procediments hem de afegir-los a la taula de símbols i després afegir cada un dels paràmetres de la funció a la taula d'expansió. Per això hem usat una variable global anomenada upperID per passar el nom de la funció a les produccions del parametres (ja que hem d'afegir cada parametre un a un). Al final i per poder usar els parametres cridam un mètode de la taula de símbols el anomenat setParams el qual posa els paràmetres de la funció a la taula de símbols per poder usarlos a dins la funció o el procediment.

També hem de tenir en compte que les funcions i procediments només podran ser declarades a nivell de main i no es pot accedir a una funció si no ha estat declarada abans (no funciona cridar a una funció declarada més endavant)

CRIDA FUNCIONS I PROCEDIMENTS:

Usant dues estructures auxiliars comprovam que els paràmetres inserits a la crida son correctes (en tipus) i son el nombre adequat (que el nombre de paràmetres inserits és correcte).



```
EXAMPLE

function main{
  function int a(int a, int b){
    return a+b
  }
  a(1,2);      //CORRECTE
  a(1);        //INCORRECTE
  a(1,2,3);    //INCORRECTE
  a(TRUE,2);   //INCORRECTE
}
```

PRINT:

A aquest cas està tot delegat a les expressions ja que l'únic que podem fer és mostrar una expressió a excepció que si la expressió és un identificador tot sol i aquest és una tupla.

INPUT:

Es comprova que l'identificador on s'emmagatzema l'input sigui un enter.

DECLARACIÓ DE TUPLES:

El primer que feim és afegir la tupla a la taula de símbols i per a cada camp de la tupla es comprova la correctesa del tipus i s'afegeix a la taula de símbols.

CRIDA A TUPLES:

Per cada una de les crides (el nombre de punts a la crida) es comprova si la tupla a la que se està intentant accedir té aquell camp i si encara no és la darrera crida ens hem de assegurar que sigui una tupla. En el cas de la darrera crida (darrer punt) el valor que retorna ha de ser booleà o enter ja que hem considerat que no podem fer assignació de tuples.

EXPRESSIONS:

Les expressions per molt llargues que siguin les separem en diferents operacions de un o dos operands segons les regles que hem definit a l'apartat anterior. A totes les produccions es fa el mateix canviat lleugerament entre els diferents tipus de operacions i tenim:

- Operacions aritmètiques: $E + E$, $E - E$, $E * E$ i E / E . Per aquests casos ens hem d'assegurar que totes les E siguin enters. A més a més ens hem d'assegurar que la E de la dreta de la operació no sigui un negatiu ($E \rightarrow -id \mid -number$) ja que així ens asseguram que podem posar $a = -2$ però no podem posar $a = 2+-2$.
- Operacions booleanes: $E \text{ op } E$ amb totes les operacions booleanes descrites anteriorment on només hem d'assegurar que les E son del tipus bool.
- Operacions relacionals: $E \text{ rel } E$ amb tots els relacionals hem de comprovar que les E siguin del mateix tipus i que siguin enters. Excepte per: $E == E$ i $E != E$ ja que també poden operar amb booleans, per aquells casos només hem de mirar que no siguin tuples.
- Operacions individuals: 'NOT' i '-' davant de un identificador, nombre, TRUE o FALSE només hem de comprovar que siguin el tipus correcte (bool per 'NOT' i int per '-').

Com que les totes les operacions que no son individuals els seus operands son E , podem combinar les operacions com volguem com ja hem vist a l'exemple del sintàctic que combinam tots els tipus d'operacions, això fa que les expressions del nostre llenguatge ,tot i no tenir parèntesis, siguin molt completes.

CONDICIONALS:

A l'apartat semàntic tractarem els condicionals per igual ja que l'únic que comprovam és que la condició sigui una expressió booleana (CONDITION).

FOR:

El for te la excepció que al tenir una declaració i una assignació hem de assegurar-nos que siguin correctes. La declaració ho comprova la producció de DECLARATION i l'assignació la de ASSIGNATION, però ens hem de assegurar que no esteim declarant una constant dins el for, que la declaració tengui una assignació (podriem no fer-ho ja que se inicialitzaria al valor 0 però trobem que no queda bé) i ens hem de assegurar que l'assignació no sigui una tupla

6. Codi tres adreces

El codi de tres adreces fa d'unió entre el front-end del compilador i el back-end, la classe CodeGenerator s'encarrega d'aportar els mètodes necessaris per produir el codi que, posteriorment, s'usarà per tenir el codi executable final.

El codi de tres adreces està format per una llista d'objectes "code" que consisteixen en una col·lecció de enters de la següent forma:

ID Operació	Operador 1	Operador2	Destí
-------------	------------	-----------	-------

Les diferents operacions tenen la següent estructura:

Operació	Operador 1	Operador2	Destí
COPY	Variable temporal / identificador		Variable temporal / identificador
ADD, SUB, PROD, DIV	Variable temporal / identificador / valor	Variable temporal / identificador / valor	Variable temporal / identificador
COPYVAL	Valor		Identificador
AND, OR , NOT, XOR , NAND, XNOR, NOR	Variable temporal / identificador / valor	Variable temporal / identificador / valor	Variable temporal / identificador
SKIP, GOTO			Etiqueta
LT,LE,EQ,NE,GE,GT	Variable temporal / identificador / valor	Variable temporal / identificador / valor	Variable temporal / identificador
IF			Etiqueta
INPUT (valor per teclat)			Variable temporal / identificador
PRINT,PRINTLN			Variable temporal / identificador
PMB (preambul)			Etiqueta
CALL			Etiqueta
ASSIGNATIONCALL		Etiqueta	identificador
RTN(return)			Variable temporal / identificador

CONDICIONALS:

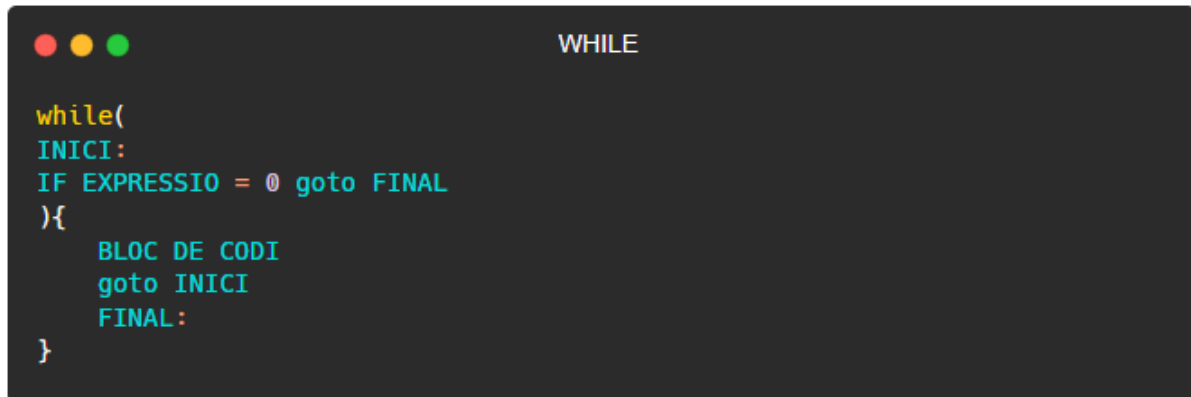
Per els condicionals trobam que hem d'explicar a part la estructura dels bots. Començant per el més senzill.

Els següents diagrames representen com s'ha inserit el codi tres adreces per els condicionals:

ETIQUETA: representarà una instrucció skip amb un nom de etiqueta inventat per fer-ho més visual

goto ETIQUETA representarà un bot incondicional a la etiqueta

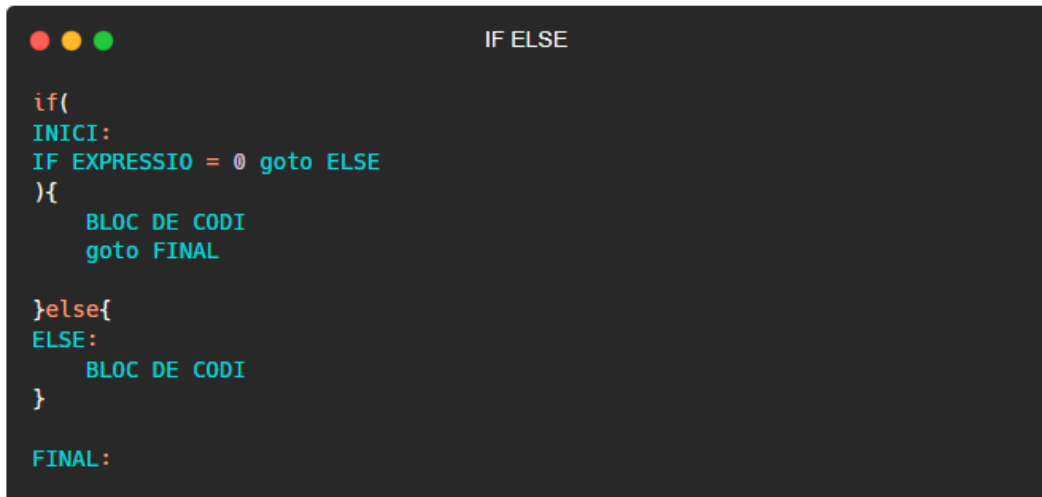
IF EXPRESSIO = 0 goto ETIQUETA si la expressió és falsa bot a la etiqueta



```
WHILE

while(
INICI:
IF EXPRESSIO = 0 goto FINAL
){
    BLOC DE CODI
    goto INICI
FINAL:
}
```

El bucle while veim que només necessita dues etiquetes, el primer que farà és que si l'expressió és 0 (FALSE) botarà al final, metre que si no és 0, executarà el bloc de codi. Quan finalitza el bloc de codi botarà a INICI i tornarà a fer el if del principi. Executarà el bloc de codi infinitament fins que la expressió sigui 0.

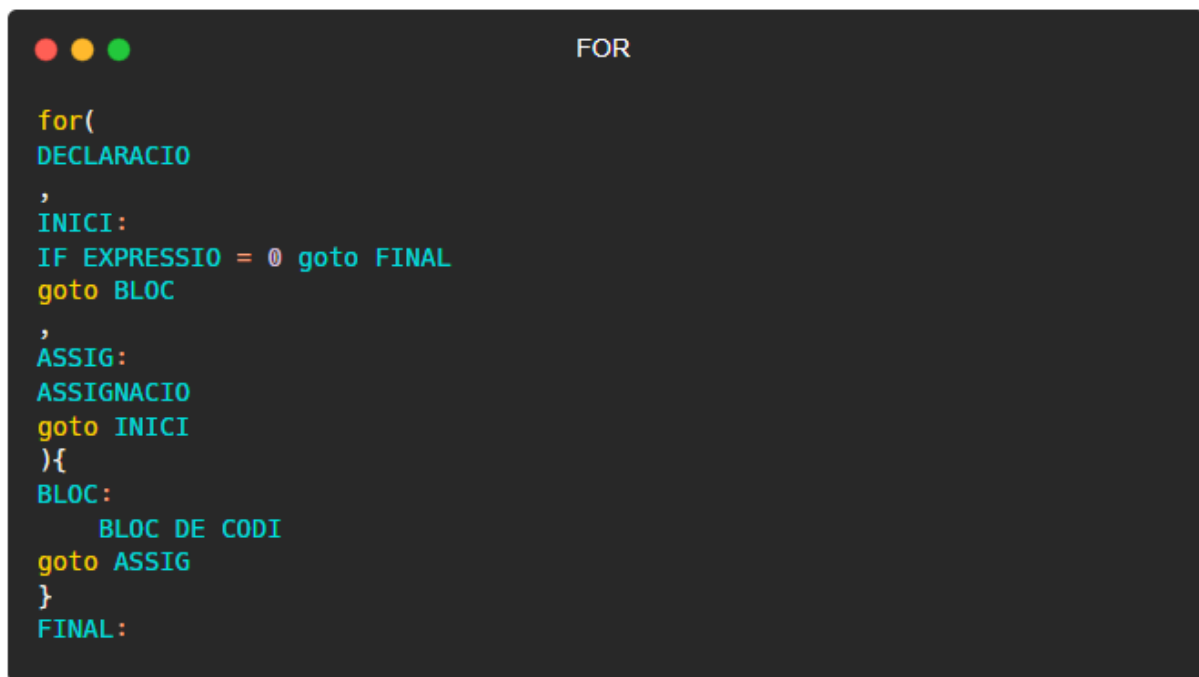


```
if(  
INICI:  
IF EXPRESSIO = 0 goto ELSE  
)  
{  
    BLOC DE CODI  
    goto FINAL  
  
}else{  
ELSE:  
    BLOC DE CODI  
}  
  
FINAL:
```

Al cas if else vem com necessitem una etiqueta més tot i que si tenim prou vista vem com la de INICI no serveix per res. Aquesta etiqueta surt del plantejament de que tots els condicionals son el mateix amb diferents característiques. A més hem de tenir en compte que al codi que convertirem a ensamblador la etiqueta INICI no existirà ja que a la optimització de codi s'eliminarà sempre.

Doncs partint de la base que INICI no fa res tenim dues etiquetes. ELSE és a la que es botarà en el cas que la expressió sigui falsa i FINAL a la que botarem quan acabem de executar el codi de l'if.

En el cas de no tenir else es quedaria igual el codi amb dues etiquetes una rere l'altre. En el cas de tenir if o else del tipus d'una línia funciona exactament igual.



```
for(  
DECLARACIO  
,  
INICI:  
IF EXPRESSIO = 0 goto FINAL  
goto BLOC  
,  
ASSIG:  
ASSIGNACIO  
goto INICI  
) {  
BLOC:  
    BLOC DE CODI  
goto ASSIG  
}  
FINAL:
```

El cas del for és el més complex de tots ja que tot i ser un while al final de cada iteració hem de realitzar l'assignació. El fluxe és el següent:

Primer de tot fem la declaració i passada la declaració se situa la etiqueta INICI passat l'inici ve IF expressió és FALSE botam al final de com a tots els condicionals, en el cas que sigui TRUE botam al bloc de codi (ja que l'assignació no s'ha de fer fins el final) executam tot el bloc de codi i al final fem un bot incondicional a ASSIG on s'executarà l'assignació per finalment fer un bot incondicional al inici (i tornada a començar).

En el cas que no hi hagi assignació o declaració es queda molt de codi sense usar, per això, les optimitzacions que hem triat funcionen per millorar totes aquestes instruccions que sobren.

7. Generació de codi

Per generar el codi ensamblador s'ha fet ús de easy68k. En el nostre cas no s'ha usat cap llibreria externa.

Per generar el codi ensamblador es processa el codi de tres adreces en ordre i s'escriu l'equivalent en ensamblador a un fitxer de text. Per aconseguir això fa ús del propi codi de tres adreces i altres estructures de dades, com la taula de procediments, una taula de etiquetes una variable sp que guarda l'apuntador actual de la pila del 68k i un índex de etiquetes auxiliars.

La classe AssemblyGenerator.java és l'encarregada d'aquesta funció.

Els codis per a cada operació són els següents:

- **Pre (Guardar els operands 1 i 2 a dos registres de dades, usat a varies codificacions)**

MOVE.L	Operand 1	D0
MOVE.L	Operand 2	D1

- **Post (Guardar el registre D0 a la destinació, usat a varies codificacions)**

MOVE.L	D0	Destinació
--------	----	------------

- **Operacions aritmètiques**

Pre()

ADD.L	D1	D0
SUB.L	D1	D0
DIVU	D1	D0
MULU	D1	D0

Post()

- **Operacions Lògiques**

Pre()

Comparador()

CMPL

D0	D1
----	----

BEQ /BNE
/BGE/BGT/BLT

	Etiqueta
--	----------

Conditional()

AND.L

D1	D0
----	----

OR.L

D1	D0
----	----

EOR.L

D1	D0
----	----

Post()

- Per aplicar "NAND" "XNOR" "NOR" és fa una combinació de les anteriors expressions
- Per aplicar la negació lògica s'usa un XOR amb "FFFFFF" és a dir -1

MOVE.L

#-1	D1
-----	----

EOR.L

D1	D0
----	----

- **Skip**

Etiqueta:

--	--

- **Goto**

BRA		Etiqueta
-----	--	----------

- **Assignacions (copyval)**

MOVE.L	#ValorOperand1	D0
MOVE.L	D0	Destinació

- **Assignacions (assigcall)**

si hi ha paramatres a tenir en compte:

ADD.L	4*(sp+1)	A7
JSR L	ValorOperand 2	
MOVE.L	-(A7)	VDestinació
ADD.L	#8	A7

- **Return**

si el valor de la destinacio == -1

MOVE.L	#0	-(A7)
--------	----	-------

si el valor de la destinacio != -1

MOVE.L	DestinacióVar	-(A7)
--------	---------------	-------

ADD.L	#4	A7
-------	----	----

RTS		
-----	--	--

- **TRAP 15**

- **Input**

MOVEQ	#4	D0
TRAP	#15	A7
MOVE.L	D1	Destinació

- **Print**

MOVE.L	Operació1 (Missatge)	D0
EXT.L	D1	
MOVE.L	#3	D0

TRAP	#15	
------	-----	--

- **Println el mateix que print + :**

MOVE.L	#11	D0
MOVE.W	#\$00FF	D1
TRAP	#15	
ADD.W	#1	D1
AND.W	#\$00FF	D1
TRAP	#15	

- **Conditional**

FALSEn:		
MOVE.L	#0	D0
BRA	AUXn	
TRUEn:		
MOVE.L	#-1	D0
BRA	AUXn	

- **Call**

first = true;
if(sp > 0){

ADD.L	#4*(sp+1)	A7
-------	-----------	----

}

JSR	Destinació
-----	------------

sp = 0;

- **PMB**

```
for(int i = 1; i<parameters.lenght;i++){  
    aux = parameters[i];  
    sp++
```

MOVE.L	-(A7)	Vaux
--------	-------	------

```
}
```

ADD.L	#sp*4	A7
-------	-------	----

```
sp =0;
```

8. Optimitzacions

S'han implementat 4 optimitzacions de mireta, unes modificant el codi de 3 adreces una vegada generat i altres s'implementen mitjançant la pròpia generació de codi.

1. Brancaments adjacents:

S'eviten les construccions de seqüències d'instruccions ineficients de la forma

```
if condició goto e1
goto e2
e1: skip
...
e2:skip
```

ja que els salts condicionals tenen el seu propi codi de tres adreces que, traduint a codi 68k amb el mètode "IF" de assemblyGenerator, evita salts innecessaris.

Després mitjançant el mètode "Optimization" a la classe CodeGenerator s'esborren totes les etiquetes que no siguin referenciades per cap salt.

2. Brancaments sobre brancaments:

El mètode "Optimization" a la classe CodeGenerator també canvia la destinació de salts derivats de condicions per evitar recórrer instruccions innecessàries.

```
if condició goto e1
...
skip: e1
goto e2
```

passa a:

```
if condició goto e2
...
skip: e1
goto e2
```

3. Assignació booleanes:

Mitjançant la instrucció de tres adreces "copyval" s'hi assignen el valor booleà directament evitant codi de la forma :

```
if b = -1 goto e1
```

```
goto e2  
e1: skip  
a = -1  
goto e3  
e2: skip  
a=0  
e3: skip
```

Directament se l'hi assigna el valor 0 o -1 a la direcció pertinent

4. Eliminació de codi inaccessible

El mètode noAccesCode a la classe CodeGenerator elimina les instruccions entre cada sentència de salt i la pròxima etiqueta. Així el codi inaccessible de la forma

```
if condicio goto eN  
... (no hi ha etiquetes de salt)  
eX: skip  
...
```

queda de la forma:

```
if condicio goto eN  
eX: skip  
...
```