Títol: Entorn per optimització i generació de codi

Volum: 1 de 1
Alumne: Robert Clarisó Viladrosa

Director/Ponent: Jordi Cortadella Fortuny
Departament: LSI
Data: 25 d'abril del 2000

Entorn per optimització i generació de codi

PFC d'Enginyeria Informàtica

Robert Clarisó Viladrosa
Director: Jordi Cortadella Fortuny

AGRAIMENTS

Als meus pares, als meus avis, al meu germà i a la Diana, per la seva pàciencia durant la realització del projecte. Gràcies pel suport que m'heu donat durant tots aquests mesos.

Als meus amics, i en general, a tots els que van cometre el fatídic error de preguntar-me de què tractava el projecte, i van haver de suportar la (llarguíssima) explicació posterior.

A Jordi Cortadella, director del projecte, que em va ajudar molt amb el seus comentaris i suggeriments, sense els quals aquest projecte no hauria estat possible.

A tots vosaltres, gràcies!

ÍNDEX

Capítol 1. Introducció i objectius

1.1 - INTRODUCCIÓ 1

1.2 - COMPONENTS D'UN COMPILADOR FINAL 2

1.3 - REQUERIMENTS DEL COMPILADOR FINAL 4

1.4 - OBJECTIUS I TASQUES DEL PROJECTE 4

1.5 - ORGANITZACIÓ DE LA MEMÒRIA 7

Capítol 2. El codi intermedi

2.1 - PER QUÈ UN CODI INTERMEDI? 9

2.2 - TIPUS DE CODI INTERMEDI 10

2.3 - EL CODI INTERMEDI ESCOLLIT: EL LLENGUATGE IC 11

2.3.1 - Les dades en el llenguatge IC 12
2.3.2 - Les instruccions en el llenguatge IC 13
2.3.3 - Alguns exemples de codi IC 16

2.4 - CONCLUSIONS 18

Capítol 3. Optimització de codi

3.1 - INTRODUCCIÓ 19

3.2 - REQUISITS D'UNA OPTIMITZACIÓ 20

3.3 - NECESSITAT DE L'OPTIMITZACIÓ DE CODI 21

3.4 - TIPUS D'OPTIMITZACIONS DE CODI 22

3.4.1- Optimitzacions dependents / independents de l'arquitectura 22
3.4.2 - Optimitzacions segons la localitat 23

3.5 - ESQUEMA D'UNA OPTIMITZACIÓ 24

3.6 - ANÀLISI DEL FLUXE DE CONTROL 24

3.6.1 - Anàlisi de blocs bàsics 25

3.6.2 - CONSTRUCCIÓ DEL GRAF DE FLUXE DE CONTROL 26
3.6.3 - ANÀLISI DE DOMINADORS 27
3.6.4 - ARBRE DE DOMINADORS 29
3.6.5 - DETECCIÓ DE BUCLES 30
3.6.6 - FRONTERA DE DOMINACIÓ 33

3.7 - ANÀLISI DEL FLUXE DE DADES 34

3.7.1 - LA MEMÒRIA EN L'ANÀLISI DE FLUXE DE DADES 35
3.7.2 - ANÀLISI DE VIDA 35
3.7.3 - CADENES D'ÚS I DEFINICIÓ 37
3.7.4 - ALTRES ANÀLISIS DEL FLUXE DE DADES 38

3.8 - OPTIMITZACIONS DE CODI 39

3.8.1 - ELIMINACIÓ DE CODI INABASTABLE 39
3.8.2 - OPTIMITZACIONS DE SALTS 40
3.8.3 - ELIMINACIÓ DE CODI MORT 41
3.8.4 - PROPAGACIÓ DE CONSTANTS 42
3.8.5 - OPERACIÓ DE CONSTANTS EN TEMPS DE COMPILACIÓ 43
3.8.6 - ELIMINACIÓ DE VARIABLES INÚTILS I NOPS 44
3.8.7 - ELIMINACIÓ DE SUBEXPRESSIONS COMUNES 45
3.8.8 - PROPAGACIÓ DE CÒPIES 47
3.8.9 - OPTIMITZACIONS DE BUCLES 48
3.8.9.1 - CREACIÓ DEL PREENCAPÇALAMENT 50
3.8.9.2 - EXTRACCIÓ D'INSTRUCCIONS INVARIANTS 50
3.8.9.3 - DETECCIÓ DE VARIABLES D'INDUCCIÓ 52
3.8.9.4 - REDUCCIÓ D'INTENSITAT DE VARIABLES D'INDUCCIÓ 54
3.8.9.5 - ELIMINACIÓ DE VARIABLES D'INDUCCIÓ 55
3.8.10 - OPTIMITZACIÓ DEL GRAF DE FLUXE DE CONTROL 57
3.8.11 - ÚNICA ASSIGNACIÓ ESTÀTICA (SSA) 59

3.9 - CONCLUSIONS 62

Capítol 4. Assignació de registres

4.1 - OBJECTIUS DE L'ASSIGNACIÓ DE REGISTRES 63

4.2 - ESTRATÈGIES PER A L'ASSIGNACIÓ DE REGISTRES 64

4.3 - ELS REGISTRES EN L'ARQUITECTURA DESTÍ: MIPS 65

4.4 - ALGORISME D'ASSIGNACIÓ DE REGISTRES UTILITZAT 67

4.4.1 - CONSTRUCCIÓ DEL GRAF D'INTERFERÈNCIA 68
4.4.2 - HEURÍSTIQUES PER A L'ASSIGNACIÓ DE REGISTRES 71
4.4.3 - COLORACIÓ DEL GRAF D'INTERFERÈNCIA 72

4.5 - CONCLUSIONS 78

Capítol 5. Generació de codi

5.1 - OBJECTIUS DE LA GENERACIÓ DE CODI 79

5.2 - ESTRATÈGIES PER A LA GENERACIÓ DE CODI 80

5.3 - EL CODI MÀQUINA DESTÍ: MIPS 82
5.3.1 - ACCÉS A MEMÒRIA 82
5.3.2 - CONVENCIONS D'ÚS 83
5.3.3 - CONJUNT D'INSTRUCCIONS 86
5.4 - PROCÉS DE GENERACIÓ DE CODI MIPS 88
5.5 - UN EXEMPLE DE CODI MIPS 91
5.6 - CONCLUSIONS 94

Capítol 6. Altres optimitzacions

6.1 - DESENROTLLAT DE BUCLES ("LOOP UNROLLING") 95
6.2 - ANÀLISI D'ÀLIES 97
6.3 - OPTIMITZACIÓ INTERPROCEDURAL 99
6.4 - OPTIMITZACIÓ LOCAL 100
6.5 - OPTIMITZACIONS DEPENDENTS DE L'ARQUITECTURA 102
6.5.1 - OPTIMITZACIÓ DE FINESTRA 102
6.5.2 - OPTIMITZACIÓ PER A ARQUITECTURES SEGMENTADES 103
6.5.3 - OPTIMITZACIÓ DELS ACCESSOS A MEMÒRIA 105
6.6 - ON S'HA ARRIBAT? 105
6.7 - CONCLUSIONS 107

Capítol 7. Disseny del projecte

7.1 - ARQUITECTURA DE L'ENTORN 108
7.2- FORMAT DEL DISSENY 109
7.3 - COMPILADOR FINAL 110
7.3.1 - LECTURA DEL CODI IC DES DE FITXER 111
7.3.2 - ADAPTAR CODI IC 112
7.3.3 - OPTIMITZACIÓ DE CODI 113
7.3.3.1 - ORGANITZACIÓ DE LES OPTIMITZACIONS: PRIMERA PROPOSTA 114
7.3.3.2 - ORGANITZACIÓ DE LES OPTIMITZACIONS: PROPOSTA ADOPTADA 116
7.3.3.3 - OPTIMITZACIONS CONCRETES 117
7.3.4 - ASSIGNACIÓ DE REGISTRES 120
7.3.5 - GENERACIÓ DE CODI MÀQUINA 120
7.4 - COMPILADOR FRONTAL 121
7.5 - CONCLUSIONS 122

Capítol 8. Implementació

8.1 - ENTORN DE DESENVOLUPAMENT 124

8.2 - ESTRUCTURA DE DADES DEL LLENGUATGE IC 125

8.2.1 - VARIABLES 126
8.2.2 - OPERANDS 126
8.2.3 - INSTRUCCIONS 127
8.2.4 - BLOC BÀSICS 128
8.2.5 - FUNCIONS 129
8.2.6 - PROGRAMA 129
8.2.7 - MACROS PER RECÒRRER L'ESTRUCTURA 130
8.2.8 - EL TIPUS IC_LIST 132

8.3 - FITXERS DE L'ENTORN 134

Capítol 9. Proves

9.1 - ORGANITZACIÓ DE LES PROVES 139

9.2 - JOCS DE PROVES 140

9.3 - RESULTATS DE LES PROVES 141

9.4 - ANÀLISI DELS RESULTATS 143

9.4.1 - RESULTATS EN FUNCIÓ DEL TIPUS DE PROGRAMA 143
9.4.2 - RESULTATS EN FUNCIÓ DEL TIPUS D'OPTIMITZACIÓ 144

9.5 - CONCLUSIONS FINALS 145

Capítol 10. Planificació i costos

10.1 - PLANIFICACIÓ DEL PROJECTE 146

10.2 - DURADA REAL DEL PROJECTE 152

10.3 - MOTIUS DE LES DESVIACIONS 152

10.4 - ANÀLISI DE COSTOS 153

Annexos

A.1 - ALGORISMES D'OPTIMITZACIÓ 155

A.2 - DEMOSTRACIÓ: EL PROCÉS D'OPTIMITZACIÓ ACABA....186

A.3 - EINES UTILITZADES....191

A.4 - GRAMÀTICA DEL LLENGUATGE IC....194

A.5 - GRAMÀTICA DEL LLENGUATGE CL....198

A.6 - MANUAL D'USUARI DE **CODEGEN**....201

Capítol 11. Bibliografia

COMPILADORS....204

JOCS DE PROVES....205

EINES I LLENGUATGES....205

1 Introducció i objectius

Aquest capítol presenta els objectius d'aquest projecte. Tots els objectius estan relacionats amb el desenvolupament d'un compilador optimitzador: un compilador que no només genera codi a partir del programa d'entrada, sino que realitza un cert treball per a intentar reduir el temps d'execució del programa.

Aquest capítol també introdueix alguns conceptes bàsics sobre codi intermedi, optimització de codi, assignació de registres i generació de codi que seran desenvolupats en els capítols següents. Aquests conceptes són fonamentals per a la comprensió dels objectius del projecte i del treball realitzat.

Aquesta documentació assumeix uns certs coneixements previs sobre llenguatges de programació d'alt nivell per a entendre els exemples plantejats. Altres capítols requereixen un coneixement de conceptes bàsics sobre l'arquitectura d'un computador: què es el codi màquina i el llenguatge assemblador, què és un registre del processador, etc.

1.1 - Introducció

En termes generals, es pot definir un compilador com una aplicació que tradueix els programes que rep com a entrada en un un codi executable per a una arquitectura concreta. Aquesta traducció es pot fer de maneres diferents, donant com a resultat executables diferents que tindran temps d'execució i requeriments de memòria diferents. Un objectiu de tots els compiladors és **produir un codi de sortida el més eficient possible**.

Per a millorar l'eficiència del codi màquina generat per un compilador, es poden utilitzar dos tipus d'aproximacions:

- Estudiar a fons les característiques de la plataforma on s'executarà aquest codi màquina i transformar el codi per a que aprofiti al màxim aquestes característiques. Aquesta aproximació es coneix com **optimització dependent de l'arquitectura**.
- Analitzar el codi d'entrada, buscant instruccions que realitzen càlculs ja realitzats, instruccions inútils o redundants, o altres situacions susceptibles de ser millorades. Després d'això, transformar el codi per a millorar-ne l'eficiència. Donat que aquesta

aproximació no es basa en detalls concrets d'una arquitectura, es coneix com **optimització independent de l'arquitectura**.

A part de la qualitat del codi generat, un altre criteri important en un compilador és la **reusabilitat**. Pot ser necessari escriure més d'un compilador per a un llenguatge font, o escriure diversos compiladors que generin codi per a diferents plataformes. L'objectiu en aquests casos serà reaprofitar part del codi del compilador. Una manera d'afavorir la reusabilitat és dividir el procés de traducció del programa d'alt nivell a codi màquina en dos fases independents:

- **Compilador frontal:** El llenguatge font es tradueix a un llenguatge independent de l'arquitectura i del codi font, anomenat **codi intermedi**. Aquesta fase és dependent del llenguatge font (C, Pascal, ...).
- **Compilador final:** El codi intermedi generat pel compilador frontal es tradueix a codi màquina. Aquesta fase depèn de l'arquitectura per a la qual s'està generant codi (VAX, Pentium, MIPS, Alpha,...).

Si volem optimitzar programes, **el codi intermedi és el punt més molt adequat per a aplicar les optimitzacions independents de l'arquitectura**, ja que és independent del llenguatge font i del codi màquina destí. Això significa que les optimitzacions independents de l'arquitectura es poden reaprofitar en qualsevol compilador que utilitzi el mateix llenguatge intermedi.

Els conceptes de codi intermedi i d'optimització independent de l'arquitectura s'apliquen en molts compiladors comercials existents actualment. De tota manera, aquests compiladors són compiladors comercials i no resulten adequats per a la docència de compiladors (aprenentatge de tècniques d'optimització de codi).. A grans trets, **l'objectiu del projecte serà desenvolupar un compilador optimitzador adequat per a la docència**. Es pot desglossar aquest objectiu en els següents subobjectius:

1. dissenyar un codi intermedi adequat per a ser optimtizat,
2. desenvolupar un compilador final que realitzi optimitzacions independents de l'arquitectura sobre aquest codi intermedi,
3. realitzar una sèrie de proves d'eficiència de les optimitzacions implementades

En els següents apartats es defineix de forma més detallada les tasques a fer en aquest compilador. També es descriuen les característiques que haurien de tenir el codi intermedi i el compilador per a resultar adequats per a la docència. Finalment, es detallen els objectius i tasques del projecte.

1.2 - Components d'un compilador final

L'objectiu d'un compilador final és la traducció del codi intermedi a codi màquina. Aquesta traducció es pot dividir en una sèrie de tasques a realitzar, com ara optimitzar codi o traduir el codi intermedi a instruccions de codi màquina. A la figura 1.1 es pot veure una possible agrupació de les tasques realitzades en un compilador final.

- **Optimització de codi**: El codi intermedi d'entrada s'analitza per a realitzar-hi *transformacions* que puguin millorar la seva eficiència. En realitat, l'optimització de codi no és una única tècnica, sino un conjunt de moltes tècniques diferents. Alguns

exemples són l'optimització de salts, l'eliminació de subexpressions comunes, l'optimització de bucles, etc. Totes aquestes tècniques es descriuen al capítol 3.

- **Assignació de registres**: En temps d'execució, les variables del codi intermedi poden estar emmagatzemades en una posició de memòria o en un registre dins el processador. En aquesta fase es decideix quina és la *ubicació més convenient* de les variables (*memòria o algun registre*). Es pot trobar més informació sobre l'assignació de registres al capítol 4.
- **Generació de codi màquina**: Les instruccions del codi màquina operen amb dades en registre o en memòria, i la decisió sobre la ubicació de cada variable ja ha estat presa en l'assignació de registres. Aquesta fase s'encarrega de *seleccionar les instruccions* de codi màquina que s'utilitzen per a traduir cada instrucció de codi intermedi, i *l'ordre d'avaluació* de les instruccions del codi intermedi. Totes aquestes tasques es descriuen al capítol 5.

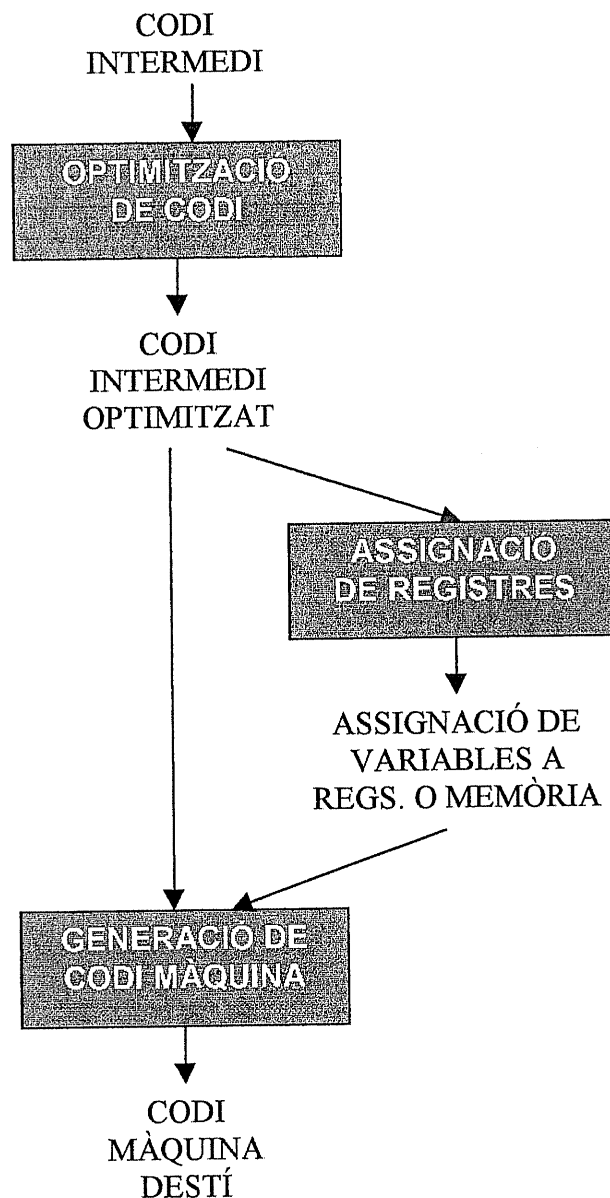


Figura 1.1. Components d'un compilador final

L'optimització de codi i l'assignació de registres constitueixen el nucli de treball d'aquest projecte. Pel que fa a l'optimització de codi independent de l'arquitectura, s'han aplicat moltes *tècniques d'optimització de codi intraprocedural* (optimitzar una funció a cada moment).. Pel que fa a la fase d'assignació de registres, s'ha escollit un algorisme eficient d'assignació de registres global *basat en la coloració de grafs.*

En canvi, la generació de codi realitzada en el projecte ha estat senzilla, al apartar-se dels objectius del projecte realitzar optimitzacions dependents de l'arquitectura. En el capítol 6 algunes optimitzacions dependents de l'arquitectura, com a possible treball futur en el camp de les optimitzacions.

1.3 - Requeriments del compilador final

L'objectiu principal del principal del projecte és el **desenvolupament d'un entorn d'optimització i generació de codi**. Aquest estorn tindrà els següents usos en docència:

- **Aprenentage de les optimitzacions de codi** (en l'àmbit, per exemple de l'assignatura Compiladors-2), estudiant el resultat de les optimitzacions en el codi intermedi. Per aquest motiu es desitja que:
  - El codi intermedi sigui comprensible i pugui ser generat manualment sense dificultat. Per exemple, es desitja que les variables del codi intermedi mantingui els noms que tenien en el programa original.
  - La sortida de l'entorn sigui el més rica possible pel que fa a informació sobre l'anàlisi del programa (fluxe de dades i de control) i les optimitzacions.
  - L'usuari tingui control sobre les optimitzacions que s'apliquen en el programa i pugui seleccionar quines optimitzacions s'apliquen, quan i quantes vegades.
- **Extensió de l'entorn com a part d'una pràctica, un projecte de final de carrera o similar**. Per aquest motiu es desitja que:
  - L'estructura de dades utilitzada pel codi intermedi sigui (relativament) senzilla
  - Les optimitzacions de codis siguin el més independent possible entre elles, de manera que sigui fàcil afegir o treure optimitzacions de l'entorn

El desenvolupament de l'entorn de compilació s'hauria de realitzar en llenguatge C, utilitzant les eines LEX i YACC. El motiu d'aquesta elecció és que els estudiants estan familiaritzats amb aquestes eines (són les eines utilitzades en la docència de Compiladors-1 i Compiladors-2).

1.4 - Objectius i tasques del projecte

El treball realitzat en el projecte està dividit en quatre *etapes* ben diferenciades:

1. **Definició d'un codi intermedi:** Donat que existeixen moltes variants possibles de llenguatges intermedis, una primera part del projecte consisteix en la definició d'un llenguatge adequat per a l'optimització de codi. Aquest codi intermedi ha estat anomenat *llenguatge IC.*
2. **Desenvolupament d'un compilador final (CODEGEN):** Aquesta etapa pretén implementar un compilador final que *no realitzi cap optimització.* Les parts del

compilador desenvolupades en aquesta etapa són l'assignació de registres i la generació de codi màquina a partir de codi IC.

3. **Desenvolupament de les optimitzacions de codi:** Un cop es disposa d'un compilador final que genera codi, s'hi afegeixen les optimitzacions dependents de l'arquitectura. La finalitat de realitzar les optimitzacions en aquesta etapa i no en l'anterior és disposar d'un entorn bàsic que faciliti la implementació d'optimitzacions.
4. **Proves de les optimitzacions:** Després d'implementar les optimitzacions, es desitja provar la seva efectivitat, és a dir, mesurar la millora produida pels programes en el temps d'execució dels programes. Per aquest motiu, s'haurà de *seleccionar un joc de proves representatiu*. Donat que és més fàcil escriure programes complexos en codi font, es realitzarà un *compilador frontal* (**FRONTEND**) que generi codi IC a partir d'un llenguatge font. Finalment, es *mesurarà la millora del temps d'execució* de les optimitzacions en els programes del jocs de prova i *s'analitzaran els resultats*.

El **llenguatge font** de l'entorn serà el **llenguatge CL** (utilitzat en l'assignatura Compiladors-2) ampliat amb funcions, taules i punters. El **llenguatge destí** de l'entorn serà el **llenguatge màquina del MIPS R2000**. Aquesta arquitectura ha estat triada per l'existència d'un simulador de MIPS R2000 (SPIM), que funciona en qualsevol entorn Unix i és de lliure distribució. Finalment, el codi intermedi utilitzat ha estat un llenguatge definit en aquest projecte: el **llenguatge IC**.

Els *productes* obtinguts en aquest projecte són (veure figura 1.2):

1. La definició del llenguatge intermedi IC
2. El compilador frontal, **FRONTEND**, que tradueix codi CL a codi IC
3. El compilador final, **CODEGEN**, que optimitza codi IC i el tradueix a codi màquina per a MIPS R2000.
4. Els jocs de proves en codi CL, i els resultats de les proves de temps d'execució

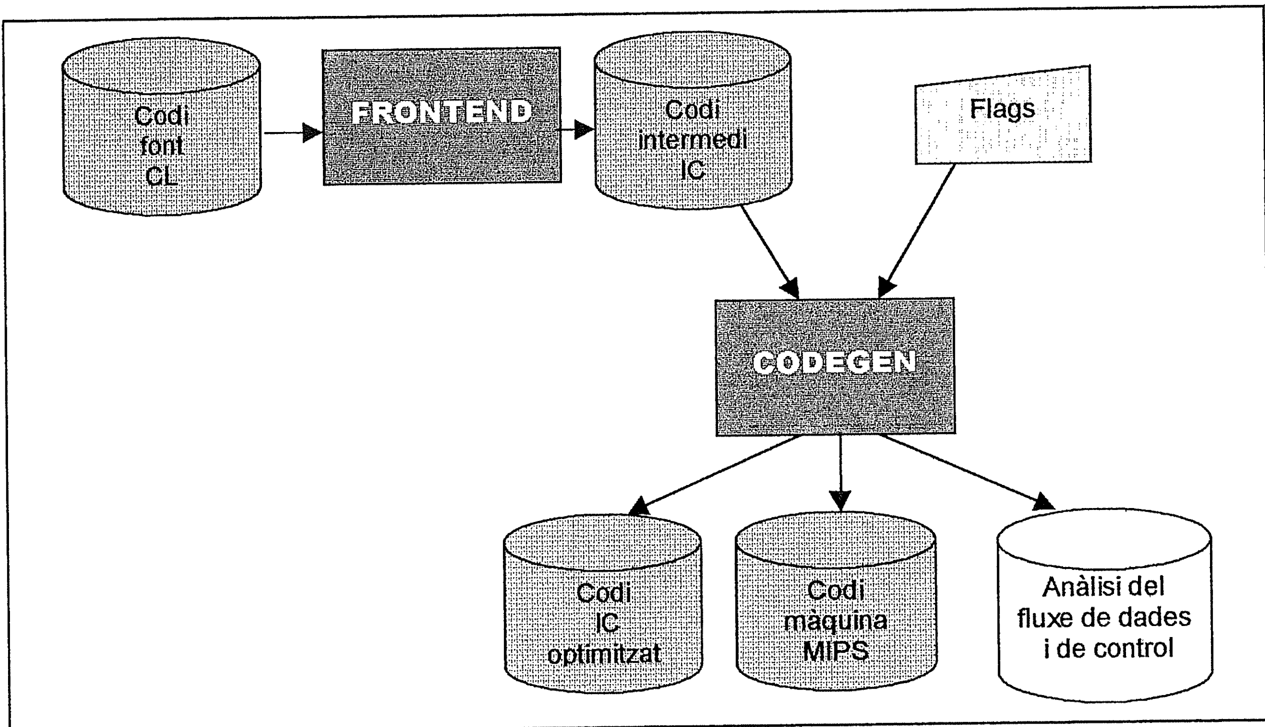


Figura 1.2. Components desenvolupats en el projecte: **FRONTEND** i **CODEGEN**

Per tant, podem identificar les següents *tasques* en el projecte:

- **Estudi de les optimitzacions independents de l'arquitectura**: Estudi de les diferents tècniques existents per a realitzar optimitzacions de codi de manera independent de l'arquitectura. Comprensió de la teoria relacionada, les alternatives possibles i els algorismes existents. Detectar les característiques que ha de tenir un codi intermedi per a permetre aquestes optimitzacions.

- **Estudi dels possibles codis intermedis**: Analitzar les característiques dels diferents codis intermedis, amb els seus avantatges i inconvenients. Tenir en compte també característiques com la comprensibilitat d'un programa escrit en el codi intermedi, la capacitat expressiva (que pugui representar la majoria de construccions d'un llenguatge d'alt nivell) i la possibilitat de realitzar optimitzacions sobre aquest codi.

- **Definició d'un llenguatge de codi intermedi (IC):** Selecció de la millor alternativa d'entre els possibles tipus de codi intermedi. Definició detallada d'un llenguatge intermedi IC del tipus seleccionat, amb un conjunt d'instruccions i la seva semàntica, tipus de dades suportats, etc.

- **Selecció de les optimitzacions de codi a realitzar en el projecte**: Escollir el conjunt d'optimitzacions de codi, d'entre totes les possibles, que seran implementades en aquest projecte. Per a cada optimització escollida, seleccionar l'algorisme a utilitzar. Estudiar la forma de combinar les diferents optimitzacions.

- **Estudi de l'arquitectura MIPS R2000**: Estudi de l'arquitectura destí del nostre compilador final: conjunt d'instruccions del codi màquina, conjunt de registres i convencions d'ús en els registres i les crides a funcions.

- **Estudi de les tècniques d'assignació de registres i generació de codi**: Identificació de les tècniques utilitzades per a fer una bona assignació de registres. Escollir un algorisme senzill per a generar codi màquina. Triar les tècniques i algorismes que s'utilitzaran en el compilador final, tenint en compte les característiques de l'arquitectura MIPS.

- **Desenvolupament d'un compilador final (CODEGEN)**: Dissenyar i implementar un compilador final que no realitzi optimitzacions. Utilitzar en aquest compilador les tècniques d'assignació de registres i generació de codi seleccionades.

- **Provar el compilador final**: Realitzar proves de correctesa en el compilador final desenvolupat per a detectar i corregir els errors. Garantir que les optimitzacions s'afegiran a un entorn lliure d'errors.

- **Implementació de les optimitzacions**: Implementar en el compilador final les optimitzacions de codi independents de l'arquitectura seleccionades prèviament. Verificar el seu correcte funcionament mitjançant proves.

- **Construcció d'un conjunt de jocs de proves**: Seleccionar d'un conjunt de programes escrits en llenguatge d'alt nivell (CL) que resultin significatius per a provar l'efectivitat de les optimitzacions de codi implementades.

- **Desenvolupament d'un compilador frontal (FRONTEND)**: Dissenyar i implementar un compilador frontal que generi codi intermedi IC a partir de codi font CL. Aquest compilador s'utilitzarà per a generar codi intermedi per als jocs de proves.

- **Mesurar el temps d'execució dels jocs de proves amb i sense optimitzacions**: Determinar els temps d'execució del codi MIPS generat a partir dels jocs de proves.

- **Anàlisi del resultat de les proves**: Extreure conclusions dels resultats de les proves. Determinar quins tipus de programes semblen més susceptibles de ser optimitzats i quines optimitzacions semblen més efectives en cada tipus de programa.

- **Redacció de la memòria del projecte**: Documentar les tasques desenvolupades en el projecte i les decisions preses pel que fa a la definició del llenguatge IC, les optimitzacions de codi, etc. Incloure en aquesta documentació el codi desenvolupat en el projecte.

1.5 - Organització de la memòria

Aquest projecte té un component teòric important, perquè ha requerit estudiar el codi intermedi, les optimitzacions dependents de l'arquitectura, l'assignació de registres i la generació de codi. Per aquest motiu, els primers capítols de la memòria es dediquen a descriure els fonaments d'aquests quatre camps. Els següents capítols descriuen el disseny del compilador, la seva implementació i les proves d'efectivitat del compilador. Finalment, els últims capítols es dediquen a la planificació i l'anàlisi de costos del projecte, els annexos i la bibliografia.

- El **capítol 2** tracta en detall el concepte de *codi intermedi*, les seves característiques i beneficis. Després de presentar diferents tipus de codi intermedi, es presenta el codi intermedi escollit: el llenguatge IC.

- El **capítol 3** tracta en profunditat les *optimitzacions de codi*. Per a cada optimització es presenten els fonaments teòrics per al seu càlcul, les diferents alternatives per a realitzar l'optimització, l'alternativa escollida i exemples d'aquesta optimització sobre el llenguatge IC. L'algorisme utilitzat en **CODEGEN** per a efectuar cada optimització es troba en els annexos.

- El **capítol 4** descriu el problema de *l'assignació de registres*, les estratègies possibles per a realitzar l'assignació de registres i les característiques dels registres en l'arquitectura destí, MIPS. També s'inclouen alguns exemples de programes i la seva assignació de registres corresponent.

- El **capítol 5** tracta la *generació de codi màquina*, detallant les característiques de l'arquitectura destí i el procés de traducció de codi IC a codi MIPS.

- El **capítol 6** intenta donar una visió de quines optimitzacions de codi han quedat fora del projecte, indicant breument les seves característiques i com podrien ser realitzades en el llenguatge IC. Aquest capítol intenta donar una visió del possible *treball futur* que es pot realitzar sobre **CODEGEN**.

- El **capítol 7** presenta el *disseny* del compiladors final (**CODEGEN**) i frontal (**FRONTEND**), i el conjunt de mòduls utilitzats per a desenvolupar els dos compiladors. En aquest capítol s'explica com s'han combinat les diferents optimitzacions descrites en el capítol 3 per a realitzar l'optimització d'un programa.

- El **capítol 8** conté la *implementació* dels compiladors final i frontal. Aquí es detalla l'entorn i les eines utilitzades, l'estructura de dades utilitzada per a representar el codi IC, i una breu descripció dels fitxers font.

- El **capítol 9** conté les *proves* que s'han realitzat sobre **CODEGEN** per tal d'avaluar el rendiment de les optimitzacions, així com les conclusions a les que s'ha arribat a partir dels resultats de les proves.

- El **capítol 10** inclou aspectes de la *planificació de treball del projecte i l'anàlisi de costos.* En primer lloc, es presenten els Diagrames de Gantt corresponents a la planificació del projecte i la seva durada real, i els motius de les desviacions en les previsions. Després es presenten els anàlisis de costos del projecte.

- Els **annexos** inclouen els algorismes utilitzats per a realitzar les optimitzacions descrites en el capítol 3, alguns comentaris sobre les eines utilitzades en el desenvolupament del projecte, les gramàtiques dels diferents llenguatges utilitzats (IC i CL), i un petit manual d'usuari de **CODEGEN**.

- Finalment, el **capítol 11** inclou la *bibliografia* del projecte.

2 El codi intermedi

Aquest capítol presenta una tècnica molt utilitzada en el món dels compiladors: la traducció del codi font a un codi abstracte, independent de l'arquitectura destí i el còdi font original. Malgrat que això representa un temps addicional de compilació, té nombrosos avantatges en el reaprofitament del codi del compilador i l'optimització.

Existeixen diferents representacions utilitzades com a codi intermedi. Entre aquestes s'ha triat la representació més adequada i s'ha dissenyat un llenguatge adient, el ***llenguatge IC****. D'aquest llenguatge IC se'n descriu les dades i el conjunt d'instruccions, i es proporcionen alguns exemples d'aquest codi.*

2.1 - Per què un codi intermedi?

L'objectiu d'un compilador és la generació de codi màquina per a una arquitectura concreta a partir d'un codi font. Aquesta traducció normalment es fa en dos passos: generació d'un codi intermedi a partir del codi font, realitzada per un mòdul anomenat **compilador frontal ("front end")**, i generació de codi màquina a partir d'aquest codi intermedi, realitzada per un mòdul anomenat **compilador final ("back end")** (veure Figura 2.1). Aquest codi intermedi és un llenguatge que **abstrau les característiques concretes del llenguatge font original i del codi màquina destí**.

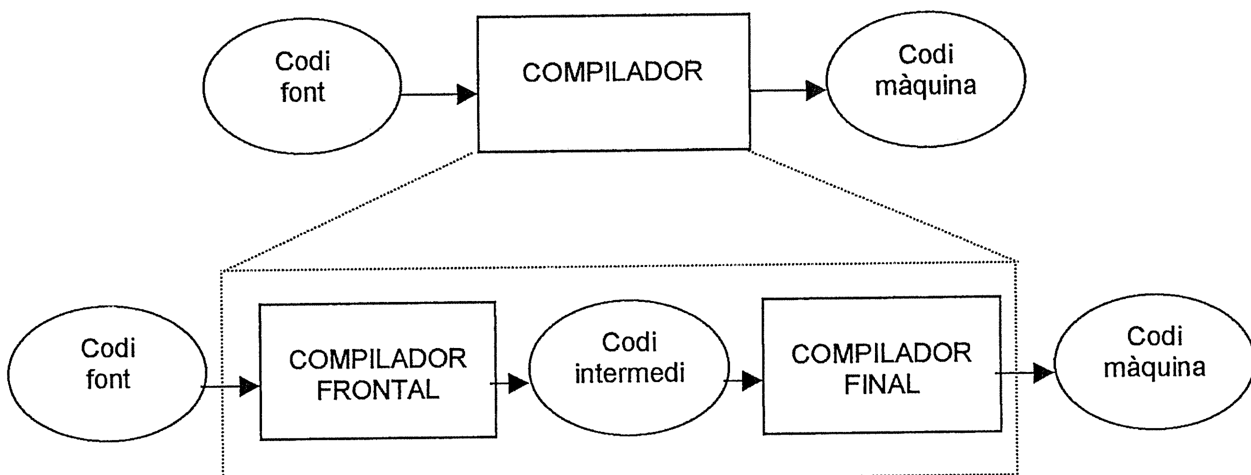


Figura 2.1. El codi intermedi en un compilador

Els motius de fer aquesta divisió en dos fases són:

- Reaprofitar codi quan pretenem canviar el llenguatge font o el codi màquina destí; en aquests casos, **podem reaprofitar el compilador frontal o el compilador final,** respectivament.
- Algunes transformacions del codi milloren el seu rendiment i es poden aplicar independentment de quin sigui el codi font original o el codi màquina destí. Aquestes transformacions, anomenades **optimitzacions de codi**, només s'han de programar un cop en el llenguatge intermedi, i **es poden reaprofitar encara que canviem el llenguatge font o el codi màquina destí**.

Un cop enumerats el motius pels quals es vol incloure un codi intermedi en el procés de compilació, es pot enumerar les **característiques** que serien **desitjables en un bon codi intermedi**.

- Ha de ser **fàcil de generar** a partir dels llenguatges fonts: el compilador frontal ha de senzill.
- Ha de ser **fàcil de traduir** als codis màquina destins: el compilador final ha de ser senzill.
- Ha de ser **independent de l'arquitectura**: no ha de considerar detalls concrets com el número de registres, els modes d'adreçament, com s'implementa la crida a una rutina...
- Les **instruccions** del codi intermedi han de ser **senzilles**. Les instruccions amb molt significat (que generen molts resultats i efectes laterals) dificulten l'optimització de codi.

2.2 - Tipus de codi intermedi

Existeixen diferents representacions del codi intermedi, cadascuna amb les seves característiques. El llistat següent inclou les més habituals així com una breu descripció de les seves característiques i algunes referències on es poden trobar exemples d'aquest tipus de llenguatges.

```
a:= a * b + c - d        (a)

a a b * c + d - :=       (b)

t1 := a * b
t2 := t1+ c              (d)
t3 := t2 - d
a := t3
```

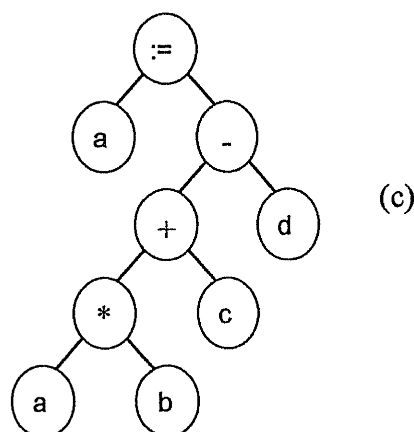


Figura 2.2. Diferents tipus de codi intermedi
(a) codi font original
(b) notació postfixa
(c) arbre sintàctic
(d) codi de tres adreces

- **Arbre sintàctic**

Les instruccions i expressions **es guarden en forma d'arbre**, de manera que cada operador té com a fills els seus operands. Si una expressió té dos subexpressions iguals, pot ser útil representar les dues subexpressions amb el mateix node. En aquest cas, la representació passa de ser un arbre a ser un graf.

L'avantatge d'aquesta representació és que, si no hi ha expressions comunes (és a dir, si és un arbre i no un graf) alguns problemes com ara la generació de codi màquina per a un bloc bàsic o l'assignació de registres tenen solució òptima. A més, els tractaments sobre arbres es poden escriure de manera molt senzilla com a rutines recursives.

Hi ha molts exemples d'arbres sintàctics com a codi intermedi a la literatura. Algunes referències són [APP98] i [FIS88], que presenten un codi intermedi basat en arbres sintàctics i les rutines semàntiques necessàries per a generar codi intermedi a partir del codi font.

- **Notació postfixa**

La notació postfixa **guarda els operadors** d'una expressió **després dels seus operands**, i es pot obtenir a partir d'un recorregut en postordre d'un arbre sintàctic. És una representació difícil de manipular, però ocupa poc espai i és adequada per generar codi per a màquines de pila.

Un exemple de notació postfixa es pot trobar a [AHO90].

- **Codi de tres adreces**

Un programa escrit en codi de tres adreces es **una seqüència d'instruccions en un llenguatge assemblador genèric**, independent de l'arquitectura. Cada instrucció pot tenir fins a **dos operands font** (zero, un o dos) i **un operand destí** (zero o un), i per això es coneix com codi de tres adreces.

Les expressions complexes es trenquen en seqüències d'instruccions de codi de tres adreces, que van guardant els resultats intermedis en variables temporals. Donat que **les instruccions del codi de tres adreces són molt senzilles i que es guarden resultats intermedis de forma explícita**, en aquesta representació és més fàcil l'optimització de codi.

Algunes referències amb exemples de codi de tres adreces són [AHO90] i [FIS88]. A més de la sintaxi del codi de tres adreces, es presenten les rutines semàntiques necessàries per a generar-lo a partir d'un codi font.

2.3 - El codi intermedi escollit: el llenguatge IC

De les tres possibles representacions descrites, la que s'ajusta més a les necessitats del projecte (realitzar optimitzacions de codi intermedi) és el **codi de tres adreces**: està format per una seqüència d'instruccions molt senzilles, i per tant, és fàcil d'optimitzar. El codi de tres adreces que s'ha dissenyat per a aquest projecte s'ha anomenat **llenguatge IC** i té els següents propòsits:

- tenir les característiques d'un bon codi intermedi: fàcilitat de generar a partir del codi font i de traduir al codi màquina destí, independència de l'arquitectura i instruccions senzilles
- representar un conjunt gran de sentències i estructures de dades dels llenguatges imperatius més habituals
- un programa escrit en llenguatge IC hauria de ser fàcil d'entendre
- permetre la realització d'optimitzacions de codi de forma senzilla

2.3.1 - Les dades en el llenguatge IC

Els operands de les instruccions en el llenguatge IC poden ser de dos classes: **variables** o **constants**.

- Les variables estan **representades** en el llenguatge IC **per un identificador**, una cadena de caràcters que comença per una lletra (les paraules reservades del llenguatge IC comencen pel caràcter "."). Una característica important del llenguatge IC és que **les variables** del llenguatge IC **no tenen cap tipus associat**: el tipus està associat a les instruccions.
- Les representacions de constants acceptades en el llenguatge IC són les mateixes que admet el llenguatge C: enteres (2, -40), reals (2.5,.3,-5.6e4,.3E-6), caràcters ('a','\n','\023') i, a més a més, les constants booleanes (.true, .false).

Les variables també es poden classificar en diferents grups, en funció de **l'àmbit de visibilitat** i de **la dada que emmagatzemen**.

- Les **variables locals** només són accessibles dins una funció, i es declaren en la capçalera de la funció. Dins una mateixa funció no poden haver-hi dues variables locals amb el mateix nom. Hi ha tres tipus de variables locals:
  - **Paràmetres** (.parameter): Els paràmetres de la funció s'han de declarar en l'ordre en que es passen al cridar la funció. Els paràmetres d'un procediment només poden ser valors escalars, és a dir, no es poden passar vectors com a paràmetre. Si es vol passar un vector com a paràmetre, s'haurà de passar com a paràmetre l'adreça al vector.
  - **Escalars** (.scalar): Els escalars guarden valors escalars, és a dir, valors que no són vectors. Aquestes variables es podrien considerar els "registres virtuals" del llenguatge IC, donat que les variables s'acabaran representant als registres del codi màquina. En una rutina hi pot haver un nombre ilimitat d'escalars.
  - **Variables estructurades** (.struct): Les variables estructurades del llenguatge IC guarden vectors d'una sola dimensió, guardats a memòria. En una variable estructurada cal indicar el tamany en bytes que ocupa el vector. Si es vol utilitzar una variable estructurada per a guardar una taula de més d'una dimensió, caldrà traduir els índexos de la taula a un índex d'una dimensió i tenir en compte el tamany dels elements guardats per a calcular el tamany de la variable estructurada.
- Les **variables globals** són visibles dins de tot el programa, i es poden declarar en qualsevol punt del programa, dins o fora d'una funció. Si en una funció hi ha una variable local amb el mateix nom que una variable global, les aparicions d'aquell nom en aquella funció faran referència a la variable local. Hi ha tres tipus de variables globals
  - **Escalars** (.gscalar): Iguals als escalars locals però amb un àmbit de visibilitat diferent.
  - **Variables estructurades** (.gstruct): Iguals a les variables estructurades locals però amb un àmbit de visibilitat diferent.

- **Strings** (.gstring): Els strings guarden una cadena de caràcters inicialitzada, que acaba en "\0". Tots els strings són variables globals.

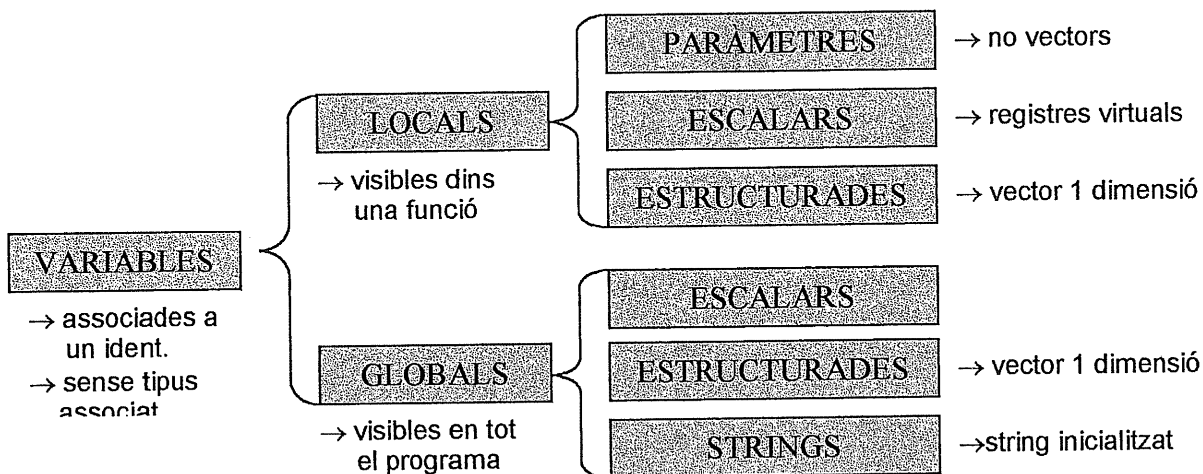
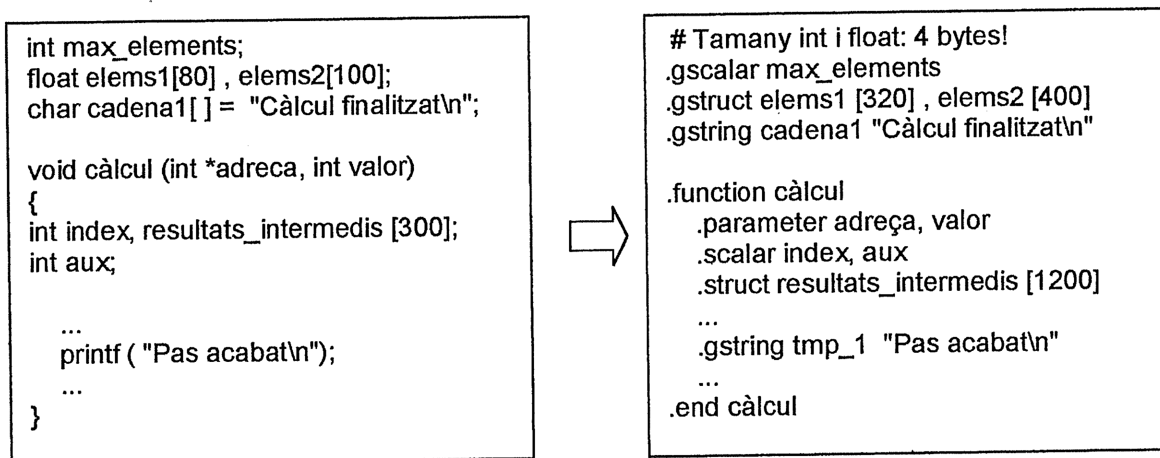


Figura 2.3. Variables en el llenguatge IC

La Figura 2.3 repassa els diferents tipus de variables del llenguatge IC i les seves característiques principals. L'exemple 2.1 ens mostra com traduir variables d'un llenguatge d'alt nivell (C) a variables del llenguatge IC i la sintaxi que tenen les declaracions.



```
int max_elements;
float elems1[80] , elems2[100];
char cadena1[ ] = "Càlcul finalitzat\n";

void càlcul (int *adreca, int valor)
{
int index, resultats_intermedis [300];
int aux;

   ...
   printf ( "Pas acabat\n");
   ...
}
```

```
# Tamany int i float: 4 bytes!
.gscalar max_elements
.gstruct elems1 [320] , elems2 [400]
.gstring cadena1 "Càlcul finalitzat\n"

.function càlcul
   .parameter adreça, valor
   .scalar index, aux
   .struct resultats_intermedis [1200]
   ...
   .gstring tmp_1  "Pas acabat\n"
   ...
.end càlcul
```

Exemple 2.1. Traducció de variables d'alt nivell a variables IC

2.3.2 - Les instruccions en el llenguatge IC

Un programa en llenguatge IC és una **seqüència de funcions**, cadascuna de les quals està formada per una **seqüència d'instruccions**. Tot i que en el cas de les variables es poden declarar variables fora d'una funció (les variables globals), **no poden haver-hi instruccions fora de l'àmbit d'una funció**.

Una instrucció del llenguatge IC consta dels següents elements:

- **L'etiqueta de la instrucció** (.iN:) és opcional, i s'utilitza per a fer referència a una instrucció en un salt
- **Zero, un, dos o tres operands**, dels quals com a màxim un serà l'operand destí

- El **codi d'instrucció**, que ens indica el tipus d'instrucció del que es tracta (aritmética, salt, crida, ...) i **l'operació** en les instruccions aritmétiques (suma, resta, ..)
- El **tipus de la instrucció**, que ens indica quin és el tipus de tots els operands. Els tipus possibles són enter ($i), real ($r), caràcter ($c), booleà ($b) i punter ($p). El tipus per defecte és enter.

El conjunt d'instruccions del llenguatge IC és el següent:

- **Instruccions NOP**

```
.nop
```

Les instruccions NOP (no-operation) són instruccions que no realitzen cap càlcul. El motiu pel qual el llenguatge intermedi té una instrucció d'aquest tipus és per facilitar l'eliminació d'instruccions durant les optimitzacions: si volem eliminar una instrucció, només hem de canviar el seu codi d'operació per NOP.

- **Instruccions aritmètico-lògiques**

```
op1 := op2 ⊕ op3 [tipus]
op1 := ⊗ op2 [tipus]
    on op1, op2, op3 són operands
       ⊕ és un operador binari
       ⊗ és un operador unari
```

Les instruccions aritmètico-lògiques poden tenir els següents operadors:

- aritmètics : suma (+), resta (-), canvi de signe (-), producte (*), divisió (/), mòdul (%)
- lògics: and (&), or (|), xor(^), not (!)
- desplaçaments: a l'esquerra (<<), a la dreta (>>), aritmètic a la dreta (/>)
- relacionals: igualtat (=), desigualtat (!=), major (>,>=), menor (<,<=)

- **Instruccions de còpia**

```
op1 := op2 [tipus]
    on op1, op2 són operands
```

- **Instruccions de salt condicional i incondicional**

```
.goto .iN
.if op1 ≅ op2 .goto .iN [tipus]
    on op1, op2 són operands
       ≅ és un operador relacional
       iN és una etiqueta d'instrucció
```

La següent instrucció a executar-se passa a ser la que té l'etiqueta, que es coneix com destí del salt. En el cas del salt incondiconal, el salt es produeix sempre, i en el cas del salt

condicional només si és certa la condició (en cas contrari, s'executa la següent instrucció). Els operadors relacionals permesos en el salt condicional són: <, >, =, >=, <=, !=.

- **Instruccions de conversió de tipus**

```
op1 := .i2r op2
op1 := .r2i op2
    on op1, op2 són operands
```

Hi ha dos conversions de tipus: d'enter a real (.i2r) i de real a enter (.r2i), realitzant un truncament. No cal indicar el tipus de la instrucció, perquè es coneixen els tipus dels dos operands (real-enter o enter-real).

- **Instruccions de crida a funció i pas de paràmetre**

```
op := func (p1 [tipus], ... ,pn [tipus])  [tipus]
func (p1 [tipus], ... ,pn [tipus])  [tipus]
    on op és un operand
       func és el nom de la funció
       p1, ... , pn són els paràmetres de la crida
```

La crida a una funció es fa a partir de l'identificador i dels seus paràmetres. Cada paràmetre es pot passar amb un tipus diferent, i el tipus de retorn (si n'hi ha) també pot ser d'un tipus diferent.

- **Instruccions de retorn de funció**

```
.return
.return op  [tipus]
    on op és un operand
```

Aquesta instrucció retorna d'una funció (opcionalment, tornant un valor). El significat d'aquesta instrucció és semblant al return de C: després d'executar aquesta instrucció no s'executa la següent, ja que es considera que s'ha acabat la instrucció.

- **Instruccions d'accés indexat a dades**

```
op := opest [opi]    [tipus]   # Llegir memòria
op := &opesc [opi] [tipus]
opest[opi]    := op  [tipus]   # Escriure memòria
&opesc[opi] := op  [tipus]
    on op, opi són operands
       opest és un operand estructurat
       opesc és un operand escalar o paràmetre
```

Aquestes instruccions permeten accedir a les dades d'una variable estructurada. El significat és el següent: opb [opi] accedeix a la posició (byte) opi de la variable

estructurada opb, per a llegir-la o escriure'l; &opb [opi] accedeix a la posició (byte) opi, en que la variable opb conté un punter a una variable estructurada. El motiu de tenir aquest segon tipus d'instruccions és permetre accedir a una variable estructurada passada com a paràmetre (recordem que no es pot passar una variable estructurada com a paràmetre: s'ha de passar l'adreça).

- **Instruccions d'accés a adreces i punters**

```
op1 := & op3              # Obtenir adreça
op1 := *op2  [tipus]      # Llegir memòria
*op1 := op2  [tipus]      # Escriure memòria
     on op1, op2 són operands
        op3 és un operand no constant
```

Aquestes instruccions serveixen per obtenir l'adreça d'un operand i manegar punters. Obtenir l'adreça d'un operand implica que aquest operand passa a estar a memòria, i per tant, no es pot utilitzar amb constants. Totes les variables a les que apliquem l'operador & hauran d'estar emmagatzemades a memòria en algun moment de la seva vida; d'aquesta manera tindran una adreça assignada. Això també inclou les variables escalars, i tot i que són les variables del llenguatge IC que més s'acosten a un "registre" poden estar guardades a memòria.

2.3.3 - Alguns exemples de codi IC

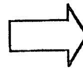
A continuació es presenten alguns exemples de programes i la seva traducció al codi intermedi IC, per a facilitar el domini d'aquest llenguatge.

```
FUNCTION fact  (VAL in INT) RETURN INT
     VARS
          res INT
     ENDVARS

     IF in < 1 THEN
          res:= 1
     ELSE
          res:= in * fact  (in-1)
     ENDIF
     RETURN (res)
ENDFUNCTION
```

```
.function fact
  .parameter in
  .scalar res, tmp_1, tmp_2

            .if in > 1 .goto .i1
            res := 1
            .goto i3
     .i1:
            tmp_1 := in - 1
            tmp_2 := fact (tmp_1)
            res := in * tmp_2
  .i3:
            .return res

.end fact
```

Exemple 2.2. Traducció del factorial al llenguatge IC

```
PROCEDURE copy_array
      (VAL nelem INT,
      REF source ARRAY [0,N-1] OF FLOAT,
      REF dest   ARRAY[0,N-1] OF FLOAT)
      VARS
            i INT
      ENDVARS

      i:= 0
      WHILE (i < nelem) DO
            dest [i] := source[i]
            i:= i+1
      ENDWHILE
ENDPROCEDURE
```



```
.function copy_array
  .parameter nelem, source, dest
  .scalar i,tmp_1,tmp_2,tmp_3

      i := 0
  .i0:
      .if i >= nelem .goto .i1
      # Els enters ocupen 4 bytes
      tmp_1 := i *4
      tmp_2 := &source[tmp_1]  $r
      tmp_3 := i*4
      &dest[tmp_3] := tmp_2  $r
      i := i + 1
      .goto .i0
  .i1:
      .return

.end copy_array
```

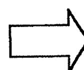
Exemple 2.3. Traducció de la còpia d'un vector al llenguatge IC

```
FUNCTION power
        (VAL mantissa FLOAT,
        VAL exp INT) RETURN FLOAT
    VARS
        res FLOAT
    ENDVARS

    res := 1.0
    WHILE (exp > 0) DO
        res := res * mantissa
        exp:= exp-1
    ENDWHILE
    RETURN res
ENDFUNCTION
```

```
.function power
  .parameter mantissa,exp
  .scalar res,tmp_1,tmp_2,tmp_3

      res := 1.0      $r
  .i0:
      .if exp <= 0 .goto .i1
      res := res * mantissa      $r
      exp:= exp-1
     .goto .i0
  .i1:
      .return res      $r

.end power
```

Exemple 2.4. Traducció de la potència d'un nombre real al llenguatge IC

```
FUNCTION xor
        (VAL op1 BOOL,  VAL op2 BOOL)
        RETURN BOOL

     RETURN (op1 AND NOT(op2)) OR
            (op2 AND NOT(op1))

ENDFUNCTION
```



```
.function xor
  .parameter op1,op2
  .scalar tmp_1,tmp_2,tmp_3, tmp_4,
tmp_5

     tmp_1 := ! op2 $b
     tmp_2 := op1 & tmp_1  $b
     tmp_3 := ! op1 $b
     tmp_4 := op2 & tmp_3  $b
     tmp_5 := tmp_2 | tmp_4 $b
     .return tmp_5 $b

.end xor
```

Exemple 2.5. Traducció de la funció or-exclusiu (XOR) al llenguatge IC

2.4 - Conclusions

Dels diferents tipus de codi intermedi existent, el més adequat per a les optimitzacions sembla el codi de tres adreces, per dos motius:

- **Les seves instruccions són senzilles**. Això vol dir que cada instrucció té un únic resultat i per tant resulta més fàcil interpretar el seu significat quan s'implementen les optimitzacions.
- **Els resultats intermedis es guarden de forma explícita en variables temporals**. Això permetrà reaprofitar resultats intermedis anteriors de forma fàcil.

L'aspecte del llenguatge IC és el d'un llenguatge assemblador senzill. Amb les seves instruccions i modes d'adreçament es poden expressar la majoria de construccions d'alt nivell (constants, variables,arrays, estructures, etc.). Altres estructures com if...else i do...while es tradueixen a base de salts. Aquests salts, així com els resultats intermedis, poden ser una font d'optimització.

La traducció directa d'un codi d'alt nivell a codi intermedi pot produir un codi ineficient. Un exemple molt clar és l'exemple 2.5 vist anteriorment: la funció XOR es pot calcular directament amb una sola instrucció. Un altre exemple es pot veure a l'exemple 2.4: l'expressió i*4 es calcula dos cops dins el bucle, quan es podria reaprofitar el resultat intermedi.

Intentar generar un codi eficient directament a partir del codi font pot complicar molt el compilador frontal. Generar el codi de forma directa i després aplicar optimitzacions sobre el codi intermedi permetrà tenir un compilador frontal senzill.

3 Optimització de codi

L'optimització de codi és un conjunt de tècniques tècniques per a reescriure un programa millorant la seva eficiència i l'espai ocupat. El codi intermedi generat pel compilador frontal contindrà nombroses instruccions redundants o innecessàries, càlculs repetits, ... Aquestes tècniques realitzen anàlisis del programa per a detectar aquestes situacions.

En aquest capítol es descriuen les optimitzacions i anàlisis implementats en el projecte Per a cadascuna, es presenten la teoria en que es basa i les possibles variants. L'algorisme utilitzat en cada cas es pot trobar a l'Annex 1.

Els capítol 6 descriu altres optimitzacions possibles, que no han estat implementades en el projecte i que constitueixen el treball futur per a **CODEGEN** *en el camp de les optimitzacions .*

3.1 - Introducció

L'**optimització de codi** és un conjunt de transformacions que, aplicades a un programa, redueixen l'espai de memòria que necessita o el seu temps d'execució mantenint el seu comportament.

La reducció en el **temps d'execució** es pot aconseguir mitjançant:

- eliminació de càlculs redundants
- trasllat d'instruccions de punts del programa que s'executen freqüentment a altres punts del programa
- substitució de càlculs per altres equivalents de menor cost (reducció d'intensitat)

La reducció de **l'espai de memòria ocupat** s'aconsegueix mitjançant:

- eliminació d'instruccions del programa
- reducció de l'espai ocupat per les variables del programa, minimitzant el nombre de variables que es guarden a memòria

A la majoria dels entorns, el temps d'execució d'un programa és més crític que l'espai ocupat i per tant moltes tècniques es centren en reduir el temps d'execució. Cal tenir en compte, però, que quan eliminem una instrucció d'un programa per fer que s'executi més

ràpid estem reduint la memòria utilitzada. És a dir, **algunes transformacions ens permeten reduir temps d'execució i espai de memòria a la vegada**.

3.2 - Requisits d'una optimització

Totes les optimitzacions de codi han de garantir tres condicions bàsiques:

1. Una optimització ha de **conservar el comportament d'un programa**.

   Això implica que si el programa a optimitzar era correcte, el programa optimitzat també ho serà i donarà el mateix resultat. I si el programa tenia una fallada (accés a memòria invàlid, divisió per zero...), el programa optimitzat també la tindrà.

2. Una optimització ha de **garantir una millora** (bé en espai o en temps d'execució) com a mínim en un percentatge significatiu d'execucions.

   Algunes optimitzacions ens garanteixen un guany en totes les execucions possibles del programa optimitzat.

```
PROCEDURE A (REF X INT)     PROCEDURE AOPT (REF X INT)
  X:= X-1                     X := 5
  X:= 5                     ENDPROCEDURE
ENDPROCEDURE
```

Exemple 3.1. El·liminació de codi mort

En l'exemple 3.1, el resultat de les funcions A i AOPT és el mateix, però AOPT el calcula més ràpid, sigui quin sigui el valor de l'entrada. Això és el més desitjable, però altres optimitzacions (com les optimitzacions de bucles) poden arribar a empitjorar el temps d´execució en alguns casos (per exemple, si el bucle no s'executa mai) encara que en la majoria de casos ens proporcionen un guany de temps substancial.

```
PROCEDURE A (REF X INT)     PROCEDURE AOPT (REF X INT)
  VARS                        VARS
    AUX INT                     AUX INT
  ENDVARS                     ENDVARS
  WHILE (X>0) DO              AUX := 1
    AUX := 1                  WHILE (X>0) DO
    X:= X - AUX                 X:= X - AUX
  ENDWHILE                    ENDWHILE
ENDPROCEDURE                ENDPROCEDURE
```

Exemple 3.2. Trasllat d'expressions invariants

En aquest exemple (3.2) el codi optimitzat és igual al codi original amb una excepció: una instrucció que en A es troba dins el bucle es troba fora en AOPT. Si el bucle s'executa més d'una vegada, AOPT serà més ràpid que A, ja que el seu bucle només conté una instrucció. En canvi, si el bucle no s'executa mai, el programa A serà més ràpid que AOPT. Així doncs, AOPT pot oferir una guany o una pèrdua en temps d'execució en funció del valor de l'entrada. De tota manera, el guany en el cas d'entrar en el bucle (i el

cas més habitual en un bucle és que s'executi com a mínim alguna vegada) compensa amb escreix la pèrdua en l'altre cas.
Com que les optimitzacions de bucles ens ofereixen una millora tan substancial, considerem suficient que una optimització garanteixi millores en mitjana.

3. Una optimització ha de **garantir una millora que compensi el seu temps de càlcul.**

Algunes optimitzacions ofereixen bons resultats amb temps de càlcul petits, però altres requereixen massa temps i memòria en relació al guany que ofereixen. Una manera molt habitual de tractar això consisteix en agrupar les optimitzacions d'un compilador en "nivells d'optimització", de manera que les optimitzacions amb un cost de temps similar es troben el mateix nivell d'optimització. El programador pot seleccionar quin és el grau d'optimització que necessiten els seus programes i per tant, quin nivell d'optimització necessita (més optimització implica més temps de compilació).
Ens podem plantejar escriure un compilador que, donat un programa, escrigui el programa òptim amb un comportament equivalent? La resposta és no. La teoria de la complexitat ens indica que aquest problema és equivalent al problema d'aturada de les màquines de Turing ([APP98]). Llavors quin serà l'objectiu? L'objectiu serà escriure algunes **transformacions concretes que permeten millorar el codi**, encara que **no ens permetin arribar a l'òptim**. A més, donat un conjunt d'optimitzacions, sempre podem afegir-hi altres optimitzacions que proporcionin un codi encara més eficient.

3.3 - Necessitat de l'optimització de codi

La millora del temps d'execució i/o espai ocupat per un programa es pot aconseguir a nivell algorísmic, **triant l'algorisme més eficient** per resoldre un problema donat. Per exemple, en el cas de l'ordenació d'un vector, triar un algorisme o un altre (quicksort, bombolla, mergesort, ...) pot fer canviar molt el temps d'execució. Però aquestes millores queden a càrrec del programador, perquè un compilador no pot resoldre el problema "trobar l'algorisme més eficient per solucionar un problema donat".

Però alguns aspectes, com ara l'accés a les variables del programa, accés a vectors, punters, etc. són **transparents al programador** en alguns llenguatges de programació d'alt nivell. Aquestes aspectes representen un número de càlculs gens despreciable que pot afectar sensiblement al rendiment d'un programa, i és responsabilitat del compilador optimitzar-ne el càlcul.

A més, totes les **característiques pròpies de la màquina** en que s'executarà el programa (modes d'adreçament, instruccions complexes) també són transparents al programador i per tant són responsabilitat del compilador, que ha d'utilitzar-les de la forma més convenient possible.

Finalment, el programador normalment intenta escriure **codi de forma clara i comprensible**, evitant instruccions que aconseguirien una major eficiència a canvi de dificultar la comprensió del programa. Un bon compilador hauria de permetre als programadors abstreure's dels detalls d'eficiència i centrar-se en la claredat del codi, garantint que la fase d'optimització generarà un codi eficient.

Per tant, alguns aspectes de l'eficiència d'un programa són responsabilitat del programador (com ara la sel·lecció del algorisme a aplicar). Però **és necessària la intervenció**

del compilador, per permetre al programador l'obtenció d'un codi eficient malgrat que alguns aspectes siguin transparents al programador (estructures del llenguatge utilitzat o característiques de l'arquitectura) o es vulgui escriure un codi clar i comprensible.

3.4 - Tipus d'optimitzacions de codi

Hi ha diversos criteris per classificar les optimitzacions de codi:

- Dependents / independents de l'arquitectura (sobre quin llenguatge treballem)
- Localitat de l'optimització (on s'apliquen les optimitzacions)

3.4.1- Optimitzacions dependents / independents de l'arquitectura

Algunes optimitzacions es poden fer sobre un codi intermedi, independentment de la màquina sobre la que s'executarà el programa. Per exemple, si una instrucció no s'executa mai, es pot eliminar sense tenir en compte el codi màquina al que es traduirà l'expressió. Aquestes optimitzacions s'anomenen **independents de l'arquitectura**.

```
.function A                    .function AOPT
  .parameter X                   .parameter X
    X:= X-1                      .return 5
    X:= 5                      .end AOPT
    .return X
.end A
```

Exemple 3.3. El·liminació de codi mort en codi intermedi

L'exemple 3.3 es la traducció a codi intermedi de l'exemple 3.1. Aquest codi intermedi és independent de les característiques de l'arquitectura, i per tant podem realitzar optimitzacions que no requereixin informació específica d'una màquina concreta. En aquest cas, com que sabem que el valor calculat per la instrucció X:= X-1 no serà utilitzat, podem eliminar la instrucció. Això és vàlid en qualsevol arquitectura i en qualsevol arquitectura representarà una reducció del temps d'execució.

D'altra banda, quan estem treballant sobre un codi màquina concret, podem fer ús d'instruccions especials de la màquina (modes d'adreçament complexos, instruccions d'autoincrement, ...) per realitzar el mateix càlcul de forma més eficient. Aquestes optimitzacions són **dependents de l'arquitectura**.

```
MOV reg3, reg1                 MOV reg3,reg1
MOV reg1, reg2                 XCHG reg1,reg2
MOV reg2, reg3
```

Exemple 3.4. Optimizació de codi màquina d'un 80386

Aquest exemple ens mostra una optimització realitzada sobre el codi màquina d'un PC 386. El que fa aquest codi és assignar al registres reg3 i reg2 el contingut inicial de reg1, i al registre reg1 hi assigna el contingut inicial de reg2 (la instrucció MOV mou el contingut d'un

registre a un altre registre). És a dir, aquest codi assigna a reg3 el contingut inicial de reg1 i després intercanvia els registres 1 i 2. En el codi màquina del 80386 tenim una instrucció que intercanvia registres de forma més eficient que dos MOV: XCHG. Per tant, el codi de la dreta calcula el mateix resultat que el codi de l'esquerra de forma més eficient. Només podem fer aquesta substitució quan estem treballant amb el codi màquina d'una plataforma concreta, ja que és l'únic moment en que sabem de quines instruccions disposem.

3.4.2 - Optimitzacions segons la localitat

Podem classificar les optimitzacions en funció dels fragments del programa que es consideren a l'hora de fer una optimització.

Les **optimitzacions de finestra (peephole)** recorren el programa examinant només un fragment, la **finestra**, de poques instruccions de longitud. El seu mètode de funcionament es basa en el "pattern matching": trobar un patró d'instruccions en la finestra i substituir-lo per un altre que és equivalent però més eficient. Aquest tipus d'optimitzacions són molt utilitzades en el cas de les optimitzacions dependents de l'arquitectura.

L'exemple 3.4 és una optimització de finestra, amb tamany de finestra 3. Explorem el codi examinant blocs de 3 instruccions i si trobem una expressió com la de l'esquerra la substituim per l'expressió de la dreta.

Les **optimitzacions locals** divideixen els procediments del programa en conjunts d'instruccions que sempre s'executaran consecutivament (anomenats **blocs bàsics**) i intenten trobar l'ordre d'avaluació òptim d'aquestes instruccions que redueix el temps d'execució. [AHO90] conté el mètode utilitzat per a optimitzacions locals: generació de codi a partir del DAG (graf dirigit i acíclic) del bloc bàsic.

a:= 1	a:=1
b:= c + d	f:= c + d
f:= a * b	d:= b-1
e:= f +2	e:=d
d:= f -1	b:=2
e:= b-1	
b:=2	

Exemple 3.5. Optimizació local d'un bloc bàsic

Aquest exemple 3.5 mostra un conjunt d'instruccions que tenen una seqüència d'instruccions equivalent i més eficient. Aquestes optimitzacions es basen en analitzar les instruccions del bloc, veure quines calculen la mateixa expressió i, finalment, escriu-les totes en un ordre que minimitzi el nombre d'instruccions a executar.

Les **optimitzacions globals** analitzen alhora totes les instruccions d'un procediment. El seu objectiu és detectar instruccions redundants o inútils que es troben en blocs bàsics diferents, i detectar les parts del programa que s'executen més freqüentment per tal d'intentar reduir la seva longitud.

Totes aquestes optimitzacions són **intraprocedurals**, és a dir, només examinen les instruccions dins d'una funció. Un darrer tipus d'optimitzacions són les **interprocedurals**,

que examinen diversos procediments al mateix temps. Aquestes optimitzacions permeten, per exemple, l'anàlisi de variables globals del programa.

3.5 - Esquema d'una optimització

Totes les optimitzacions segueixen el mateix patró de dos fases:

- **Anàlisi del programa**: examinem el programa per tal d'extreure informació sobre com modificar-lo sense canviar el seu comportament
- **Transformació del programa**: modificació del programa que intenta reduir el temps d'execució o la memòria ocupada; podem assegurar que el comportament del programa no ha variat gràcies a la informació recollida a la fase prèvia d'anàlisi

Hi ha dos grans famílies d'anàlisi del programa, que fan referència al tipus d'informació que es recull sobre el programa: l'anàlisi del fluxe de dades i l'anàlisi del fluxe de control.

L'anàlisi del fluxe de dades consisteix en examinar els valors que poden prendre les variables durant l'execució del programa. Permet detectar, per exemple:

- expressions formades només per constants, que poden ser calculades en temps de compilació
- assignacions de valors a variables que no s'utilitzen i que, per tant, poden ser eliminades (eliminació de codi mort)
- expressions disponibles en un moment del programa i que, per tant, no s'han de recalcular (eliminació de subexpressions comunes)

L'anàlisi del fluxe de control consisteix en examinar les diferents seqüències del fluxe de control del programa, és a dir, l'ordre en que s'executen les instruccions del programa. De cara a fer aquest anàlisi ens podem centrar en les instruccions que canvien el fluxe de control (salts) ignorant la resta d'instruccions. Algunes informacions que podem obtenir amb aquest anàlisi són:

- els blocs bàsics, és a dir, els conjunts d'instruccions que sempre s'executen consecutivament
- detecció de dominadors, és a dir, blocs bàsics que sempre s'executen abans que altres
- detecció de bucles
- detecció de blocs bàsics que no són executats mai, i per tant poden ser eliminats (eliminació de codi inabastable)
- detecció de salts a altres salts, que es poden optimitzar (optimització de salts)

3.6 - Anàlisi del fluxe de control

L'objectiu de l'anàlisi del fluxe de control és determinar el **fluxe de control**, l'ordre d'execució de les instruccions en un programa, ignorant els càlculs. El fluxe de control en un programa en principi és linial: s'executa una instrucció i després la següent. Però hi ha algunes instruccions, els **salts**, que modifiquen el fluxe de control, fent que la següent

instrucció a executar-se sigui una altra, el **destí del salt**, que no té perquè ser la següent instrucció en ordre.

L'anàlisi del fluxe de control **es centrarà en els salts i les instruccions destí dels salts,** les úniques instruccions rellevants per considerar la seqüència d'execució de les instruccions en el programa. L'anàlisi del programa es farà de forma incremental:

- primer es separen les instruccions en conjunts que sempre s'executaran de forma consecutiva **(blocs bàsics)**
- després es construeix el **graf de fluxe de control**, que indica l'ordre en que es poden executar els diferents blocs bàsics
- a partir del graf de fluxe de control, l'estructura fonamental, es faran altres anàlisis: per exemple, l'**anàlisi de dominadors**, necessari per poder realitzar la **detecció de bucles**.

3.6.1 - Anàlisi de blocs bàsics

L'objectiu d'aquest anàlisi consisteix a detectar en el codi del programa aquells conjunts d'instruccions que sempre s'executaran de forma consecutiva, els **blocs bàsics**. En un bloc bàsic, només hi ha una entrada de fluxe de control i és a la primera instrucció, i només hi ha una sortida del fluxe de control a la darrera instrucció.

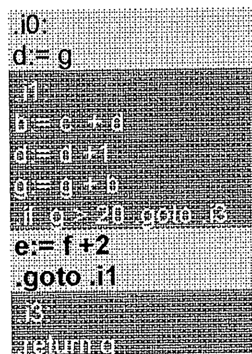
La primera instrucció d'un bloc bàsic s'anomena **líder** del bloc bàsic. Si determinem el conjunt de líders d'una funció, haurem trobat tots els blocs bàsics d'una funció: a partir d'un conjunt de líders, un bloc bàsic és el conjunt d'instruccions que va d'un líder fins al següent (incloent el primer líder i sense incloure el segon).

Per trobar els líders dels blocs bàsics es fa servir la definició de bloc bàsic:

- la primera instrucció d'una funció és un líder
- tota instrucció destí d'un salt és un líder
- tota instrucció que segueix a un salt o .return és un líder

Cal destacar que després d'una crida a funció no hi ha un nou bloc bàsic, donat que després de la crida a la funció sempre s'executarà la instrucció posterior a la crida. Les úniques instruccions rellevants a l'hora de trobar els líders són els salts i els .return.

```
.i0:
d:= g
.i1:
b:= c  + d
d:=d + 1
g:= g + b
.if  g > 20 .goto .i3
e:= f +2
.goto .i1
.i3:
.return g
```



Exemple 3.6. Detecció de blocs bàsics en un fragment de codi

Les seqüències d'instruccions entre els líders són els blocs bàsics, i verifiquen la propietat que **cap instrucció del bloc bàsic és un salt (excepte potser la darrera) i cap instrucció del bloc bàsic és destí d'un salt (excepte potser la primera)**.

En l'exemple 3.6 podem trobar els següents líders: i0 (per ser la primera instrucció), i1 (per ser destí d'un salt), e:= f+2 (per ser una instrucció posterior a un salt) i i3 (per ser posterior a un salt i destí d'un salt). Amb aquests quatre líders podem construir els quatre blocs bàsics que es veuen a la columna de la dreta. Podem comprovar que només tenen un punt d'entrada (la primera instrucció) i un punt de sortida (la última).

El fet de tenir les instruccions agrupades en blocs bàsics ens permet ignorar les instruccions que no siguin salts, alhora que ens permet accelerar moltes de les optimitzacions, que poden treballar sobre (pocs) blocs bàsics en comptes de fer-ho sobre (moltes) instruccions.

3.6.2 - Construcció del graf de fluxe de control

L'objectiu és emmagatzemar els blocs bàsics en una estructura que ens permeti accedir ràpidament als successors i predecessors d'un bloc bàsic donat. Aquesta estructura, en forma de graf, s'anomena graf de fluxe de control ("control-flow graph") i conté la següent informació:

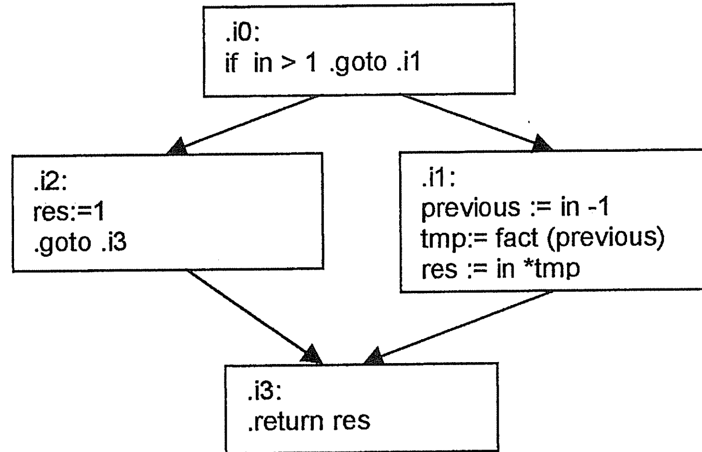
- cada node del graf és un bloc bàsic. A partir d'ara utilitzaré de forma indistinta node per referir-me a bloc bàsic quan parli del graf de fluxe de control
- existeix una aresta del node i al node j si el bloc bàsic j es pot executar després del i, ja sigui perquè hi ha un salt de i a j o perquè j és el bloc bàsic següent a i. Es diu que **"i és predecessor de j"** i **"j és successor de i"**.

Per a la següent fase de l'anàlisi del fluxe de control (l'anàlisi de dominadors) ens convé que el graf de fluxe de control contingui un **node inicial sense cap predecessor**. Aquest node representa el punt d'entrada a la funció. Per tant, quan construim el graf de fluxe de control., hi afegirem aquest node inicial.

En l'algorisme usat per construir el graf de fluxe de control suposarem que tot bloc bàsic té una etiqueta associada, l'etiqueta del líder del bloc bàsic. D'aquesta manera, sabem quin bloc bàsic és el destí del salt consultant l'etiqueta del salt i buscant el bloc bàsic amb aquesta etiqueta.

El funcionament de l'algorisme es pot comprovar amb un exemple senzill. El programa inicial de l'exemple 3.7 és el factorial recursiu, que ja ha estat dividit en blocs bàsics. Hi ha quatre blocs bàsics, un per cada etiqueta: i0, i1, i2, i3. L'algorisme només afegeix arestes entre els nodes que es poden executar un darrera l'altre: els nodes 2 i 1 es poden executar després del node 0, i el node 3 es pot executar després del 2 o del 1. Com es pot comprovar, no s'ha d'afegir cap aresta entre els nodes 2 i 1 encara que estan en ordre consecutiu en el codi, perquè el node 2 acaba en un salt incondicional i per tant el node 1 mai s'executarà després del node 2. Finalment, afegir com a comentari que el node 0 ja és un node sense predecessors i per tant no caldria afegir cap node inicial.

```
.function fact
 .parameter in
 .scalar previous, tmp, res

 .i0:
     .if in > 1 .goto .i1
 .i2:
     res := 1
     .goto .i3
 .i1:
     previous := in - 1
     tmp := fact(previous)
     res := in * tmp
 .i3:
     .return res

.end fact
```

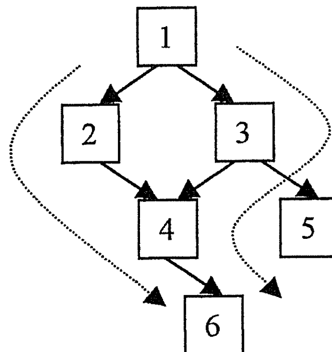


Exemple 3.7. Construcció del graf de fluxe de control

3.6.3 - Anàlisi de dominadors

Dins el graf de fluxe de control, existeixen relacions entre els blocs bàsics de la forma "sempre que s'executa el bloc bàsic j, el bloc bàsic i s'ha executat abans". Aquestes relacions s'anomenen relacions de **dominació**, i s'utilitzen per detectar bucles i realitzar altres optimitzacions.

Formalment, **un node i domina un node j (i DOM j)** si **qualsevol** camí que va del node inicial al node j passa per i.



Exemple 3.8. Dominadors d'un node

En l'exemple 3.8, el node 6 és dominats pels nodes 1 i 4, perquè tot camí del node inicial (1) al node 6 passa per aquests nodes. En canvi, no és dominat pel node 2, perquè hi ha un camí 1,3,4,6.

Aquesta relació i DOM j compleix una serie de propietats:

- és un ordre reflexiu parcial, o sigui,
  - és reflexiva, és a dir, tot node es domina a sí mateix

Dem: sigui a un node del graf de fluxe de control. En tot camí del node inicial al node a es troba el node a, i per tant, tot node a es domina a sí mateix.

- és antisimètrica, és a dir, no hi ha dos nodes a, b tals que a DOM b i b DOM a.

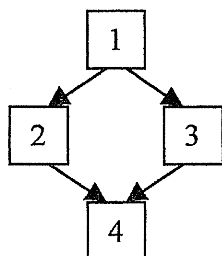
Dem: siguin a i b dos nodes diferents que es dominen mútuament. Siguin l i m les distàncies mínimes del node inicial a a i b. Per reducció a l'absurd:
Si $l \leq m$ (en cas contrari només cal intercanviar a i b), llavors considerem la posició de b en el camí mínim [node inicial, a]. b ha de trobar-se abans que a en aquest camí, i per tant està a distància menor que l del node inicial. Com que $d < l \leq m$ i sabem que la distància mínima del node inicial a b és m, $d \geq m$ i $d < m$. **Contradicció.**

- és transitiva, és a dir, si a DOM b i b DOM c, llavors a DOM c

Dem: siguin els nodes a, b, c tres nodes tals que a DOM b i b DOM c. Això vol dir que en tot camí del node inicial al node b hi ha a, i que en tot camí del node inicial a c hi ha b. Com que en tot camí del node inicial a c hi ha b, també hi haurà a. Per tant a domina a c.

- és parcial, és a dir, poden existir dos nodes a,b tals que ni a DOM b ni b DOM a.

Dem: en el següent graf de fluxe de control, els nodes 2 i 3 no es dominen.



El node inicial és el node 1. En els camins del node 1 al 2 no hi ha el node 3 i , contràriament, en el camí entre els nodes 1 i 3 no hi ha mai el 2. Per tant, ni 2 DOM 3 ni 3 DOM 2. Finalment només cal dir que aquest graf de fluxe de control és possible: és l'obtingut en l'exemple 3.7

- els dominadors d'un node han de ser ancestres d'aquell node, però no tots els ancestres d'un node el dominen

Dem: Un node només pot dominar un altre si està en algun camí del node inicial al node. I per estar-hi ha de ser ancestre del node, considerant com a ancestres d'un node el mateix node, els seus predecessors, els predecessors dels seus predecessors, etc. A més, en la demostració de la propietat anterior podem veure que 1 DOM 4 però ni 2 DOM 4 ni 3 DOM 4, i per tant no tots els ancestres d'un node el dominen.

- el node inicial només és dominat per ell mateix

Dem: Per la primera propietat, sabem que el node inicial es domina a sí mateix. Per la propietat anterior, sabem només els ancestres d'un node el poden dominar, i com el node inicial té un sol ancestre (ell mateix) només és dominat per ell mateix.

La definició de dominador ens ofereix un problema: els nodes que no tenen cap predecessor (i tots els seus successors) no tenen cap camí des del node inicial fins a ells. Aquests nodes, anomenats **codi inabastable**, compliquen el nostre anàlisi de dominadors

perquè són dominats per tots els nodes del graf de fluxe de control: com que no hi ha cap camí del node inicial fins a ells, podem dir que qualsevol node es troba en tots els camins. Per evitar que es produeixi això, **els nodes inabastables s'haurien d'eliminar** abans d'iniciar l'anàlisi de dominadors. Això és pot fer perquè els nodes inabastables no s'arribaran a executar mai, i d'això se n'encarrega l'optimització "eliminar codi inabastable".

El conjunt de dominadors d'un node es pot **definir de forma recursiva**, a partir del conjunt de dominadors dels predecessors d'un node. Expressar una propietat d'un node en forma d'equacions recursives que utilitzen el valor per als altres nodes és molt habitual (gairebé tots els anàlisis del fluxe de dades es poden expressar així) i ens proporciona una forma directa per calcular dades dels nodes.

En el cas dels dominadors, les equacions són les següents:

- Si "n" és el node inicial, $\text{Dominadors}[n] = \{n\}$
- Si "n" no és el node inicial $\text{Dominadors}[n] = \{n\} \cup \left(\bigcap_{p \in Pred[n]} \text{Dominadors}[p] \right)$

Cal inicialitzar els conjunts de dominadors amb tots els nodes del graf. A cada iteració els conjunts s'aniran reduint (perquè es va fent intersecció de conjunts) i quan arribem a una situació en que no hi ha canvis tindrem els conjunts de dominadors. Evidentment l'ordre en que es considleren els nodes és important, i pot reduir molt el nombre d'iteracions necessàries per calcular el conjunt. Intuitivament es pot veure que si hem calculat el conjunt de dominadors per als predecessors d'un node, podrem calcular els dominadors d'un node directament. Per tant, **l'ordre en que ens interessa visitar els nodes** per calcular els conjunts de dominadors és **l'ordre** que obtenim recorrent el graf **en profunditat**. Basat en aquesta ordenació, hi ha un algorisme que calcula els dominadors d'un graf de control de fluxe en temps quasi-lineal, l'algorisme de Lengauer i Tarjan [APP98].

L'algorisme escollit per calcular els dominadors es basa en les equacions però explora els blocs bàsics en ordre linial, és a dir, l'ordre literal en que es troben en el programa.

3.6.4 - Arbre de dominadors

Quan accedim als dominadors d'un node, és habitual utilitzar el concepte de **dominador immediat**. En el graf de control de fluxe tot node n excepte el node inicial té un dominador immediat IDOM(n), un node diferent de n que domina a n però que no domina a cap altre dominador de n. Es pot demostrar que tot node del graf de control de fluxe té un i només un dominador immediat, excepte el node inicial que no en té cap.

Dem [APP98]: Per demostrar que tot node té un i només un dominador immediat, cal veure que si dos nodes a i b dominen un node c, o bé a DOM b o bé b DOM a. D'aquesta manera podrem establir que sempre hi haurà un dels dominadors que serà un dominador immediat (i com que tots els nodes excepte l'inicial i els inabastables tenen dominadors diferents d'ells, haurem acabat la demostració.

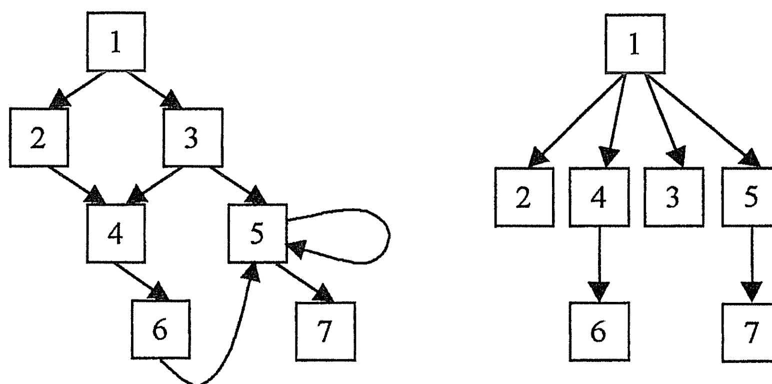
Per reducció a l'absurd: suposem que a i b dominen a c, però que ni a DOM b ni a DOM c. Com que no a DOM b, ha d'haver-hi algun camí del node inicial a b que no passi per a. Però com que a DOM c, això vol dir que a es troba en tots els camins que van del node b al node c (si no fos així, el node a no dominaria a c). De la mateixa manera, el node b es troba en tots els camins que van del node a al node c (repetint el raonament anterior).

Llavors en tot camí del node a al node c hi ha el node b i en tot camí del node b al node c hi ha el node a. Per tant, per anar de a fins a c s'ha de passar per b; però abans el camí haurà de passar per b; però llavors abans haurà de passar per a; etc. Això és un bucle infinit i no hi ha cap camí del node a fins al c, tot i que a DOM c. **Contradicció.**

La informació sobre els dominadors es sol presentar en forma d'**arbre de dominadors**:

- L'arrel de l'arbre és el node inicial
- Els fills de cada node n són els nodes m tals que IDOM (m) = n.
- La representació és un arbre perquè cada node (excepte l'arrel) té un i només un pare.

L'exemple següent (3.9), ens mostra l'arbre de dominadors d'un graf de control de fluxe. Com es pot veure, el pare en l'arbre de dominadors no té perquè ser un dels predecessors del node. Com es pot veure, el node 1 no és predecessor del node 4 però és el seu dominador immediat, perquè els nodes 2 i 3 no dominen al node 4.



Exemple 3.9. Arbre de dominadors

3.6.5 - Detecció de bucles

El temps d'execució dels programes es dedica de forma majoritària a l'execució de **bucles** (**loops** en anglés). Tots els llenguatges procedurals d'alt nivell tenen alguna estructura del llenguatge del tipus for, while ... do, repeat ... until, etc. Aquestes estructures són molt importants a l'hora d'optimitzar el codi d'un programa: aconseguir reduir el temps de càlcul d'una iteració del bucle pot ser molt important, ja que un bucle pot tenir un nombre molt gran d'iteracions.

El pas previ a l'optimització del bucle és **detectar els fragments del programa que corresponen a un bucle**. En un llenguatge d'alt nivell la detecció de bucles és directa a partir de les estructures del llenguatge. En canvi, en el codi intermedi només observem un conjunt de gotos, i la detecció de bucles és més complicada, però és possible.

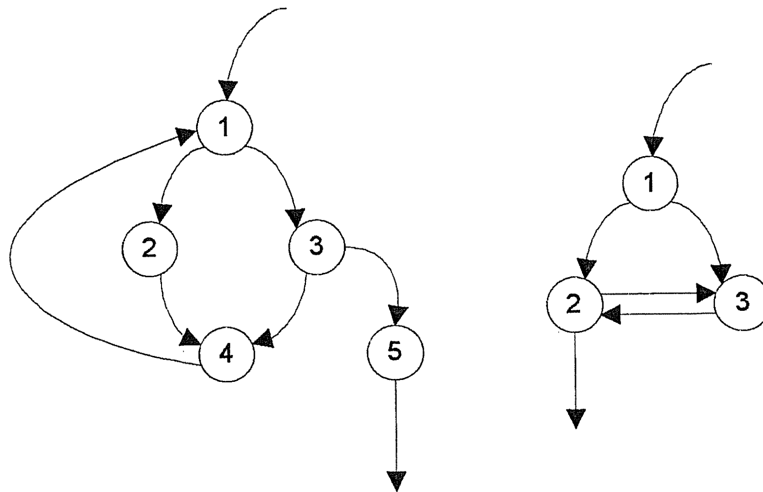
En primer lloc cal distingir dos tipus de grafs de fluxe de control, els **reduibles** i els **irreduibles**.

Formalment, un graf de fluxe de control G és reduible sí i només si es poden particionar les arestes del graf en dos conjunts disjunts, les arestes de retrocés i les arestes d'avançada, de manera que:

- Les arestes d'avançada formen un graf acíclic en el que cada node pot ser abastat des del node inicial G
- Les arestes de retrocés són només aquelles en que el node destí domina al node origen

Els grafs de fluxe de control reduibles són els més habituals, i són els que s'obtenen a partir de llenguatges de programació estructurat, és a dir, amb instruccions del tipus if...then...else, while, continue o break, però sense **gotos**. Els llenguatges que tenen gotos poden produir grafs de fluxe de control irreduibles.

Els grafs reduibles tenen molts avantatges enfront dels grafs no reduibles. En primer lloc, **permeten calcular les equacions del fluxe de dades de forma molt eficient**: en comptes de calcular les equacions fins que no hi hagi canvis, es pot fixar l'ordre en que cal avaluar els nodes per reduir el nombre de càlculs (veure comentaris sobre l'ordre en profunditat a 3.6.3). I un altre avantatge, que resulta molt important a l'hora d'analitzar els bucles, és que tots els bucles tenen un únic punt d'entrada, anomenat **encapçalament** (**header** en anglés).



Exemple 3.10. Bucles en un graf de fluxe de control

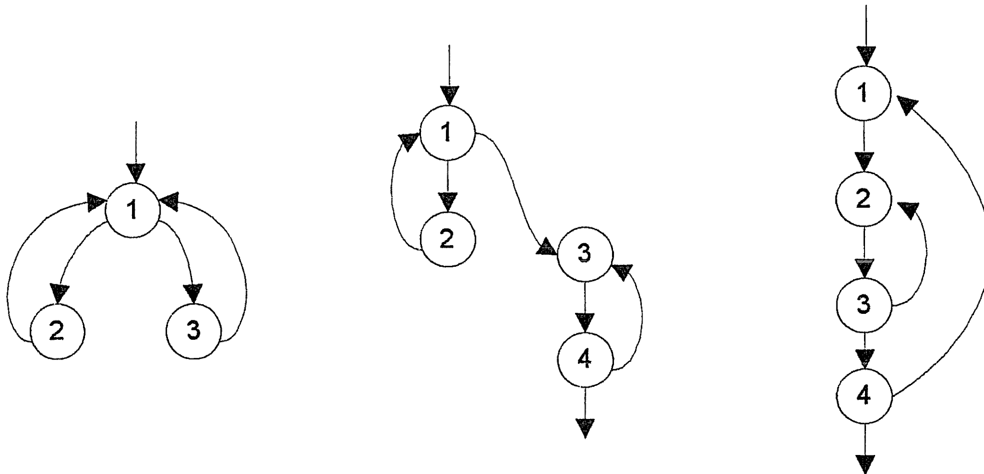
L'exemple 3.10 ens mostra dos bucles diferents. El bucle de l'esquerra té un únic punt d'entrada, el node 1. En canvi es pot entrar al bucle de l'esquerra (format pels nodes 2 i 3) per qualsevol dels dos nodes. El graf de fluxe de control de la dreta és irreduible mentre que el de l'esquerra és reduible. Cal fer notar que la restricció d'un únic punt d'entrada no s'aplica a un únic punt de sortida: en un graf de fluxe de control reduible un bucle pot tenir més d'un punt de sortida.

Per detectar els bucles en el graf de fluxe de control es fa servir la informació sobre els dominadors. A més d'un encapçalament, tot bucle ha de tenir una **aresta de retrocés**, és a dir, una aresta m → n tal que n DOM m. Aquesta és l'aresta que ens permet tornar a l'encapçalament per fer una nova iteració del bucle.

El **bucle natural** associat a una aresta de retrocés n → h és el conjunt de nodes x tals que h DOM x i existeix un camí des de x fins a n que no conté h. L'encapçalament del bucle és el node h, i per tant l'encapçalament del bucle domina tots els nodes del bucle.

Els bucles d'un graf de control de fluxe poden estar relacionats entre sí. Doncs els bucles naturals tenen una propietat que ens ajuda a relacionar-los: donats dos bucles, poden donar-se tres circumstàncies:

- o bé els dos bucles tenen el mateix encapçalament
- o bé tenen un encapçalament diferent i són disjunts
- o bé tenen un encapçalament diferent i un bucle està contingut (imbricat) en l'altre. Es pot determinar que un bucle està imbricat en un altre perquè l'encapçalament d'un node domina l'altre



Exemple 3.11. Bucles naturals

L'exemple 3.11 ens mostra les tres possibles situacions en que poden estar dos bucles. En el graf de l'esquerra hi ha dos bucles, (1,2) i (1,3), que comparteixen el mateix encapçalament, el node 1. En el graf de la dreta els dos bucles són (2,3) i (1,2,3,4). El bucle (2,3) està imbricat en l'altre, i això es pot detectar perquè el seu encapçalament, 2, és dominat per l'encapçalament de l'altre bucle, 1 (1 DOM 2). Finalment, en el graf del centre hi ha dos bucles sense cap relació.

L'ordre en el que ens interessa optimitzar els bucles és el següent:

- si un bucle està imbricat en l'altre, optimitzarem primer el bucle més intern. Això permet que al optimitzar el bucle més exterior, el bucle intern ja estigui optimitzat i les optimitzacions que es fagin sobre el bucle exterior no redueixin el temps de càlcul del bucle intern (que s'executarà més vegades que el bucle extern).
- si dos bucles són disjunts, ens és indiferent l'ordre en que els optimitzem, perquè no tenen res a veure entre ells.
- si dos bucles tenen el mateix encapçalament, no podem saber quin dels dos bucles és el més intern. Per tant, es consideraran els dos bucles com un de sol amb l'encapçalament comú i que conté la reunió dels nodes dels dos bucles.

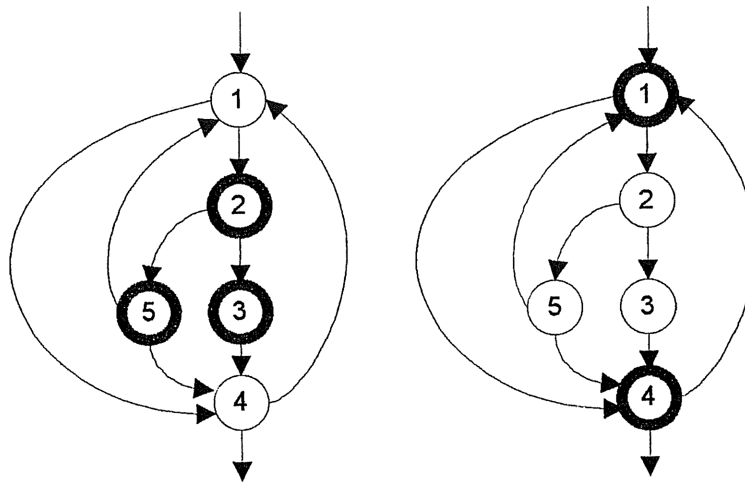
Per tant, l'algorisme de detecció de bucles tindrà dues fases: una que **detectarà els nodes que formen part d'un bucle i els seus encapçalaments**, i una segona fase que **fusionarà els bucles amb el mateix encapçalament i ordenarà els bucles de** forma que analitzem primer els bucles més interns.

3.6.6 - Frontera de dominació

Per a una optimització concreta (la transformació a la forma SSA, o forma estàtica amb una única assignació), ens cal conèixer més informació del graf de fluxe de control. Fins ara coneixem el conjunt de nodes del graf dominats per un node concret. Ara es necessita conèixer quins nodes queden just fora d'aquesta regió dominada, és a dir, la frontera d'aquesta regió dominada. Això ens coneix com **frontera de dominació (dominance frontier** en anglès), i ens indica dels nodes als que hem d'afegir informació perquè no dominen directament; no cal afegir informació a tots els nodes no dominats del graf, només cal fer-ho a aquesta frontera.

La definició de frontera de dominació es fa a partir dels dominadors. Es diu que un node n domina estrictament un node m si n DOM m i n ≠ m. La frontera de dominació d'un node n és el conjunt dels nodes x tals que **n domina un predecessor de x**, però **n no domina estrictament a x**.

L'exemple 3.12 ens mostra la frontera de dominació del node 2. A l'esquerra tenim el conjunt de nodes dominats pel node 2 (2,3 i 5). El node 2 no domina ni el node 1 ni el node 4, ja que els camins 1 i 1,4 permeten arribar als dos nodes sense passar per 2. A la dreta tenim la seva frontera de dominació, formada pels nodes 1 i 4: 2 domina a 5 (predecessor dels nodes 1 i 4) però no domina estrictament a 1 ni a 4. Cal tenir en compte que el node 2 podria formar part de la frontera de dominació, ja que 2 no es domina estrictament a sí mateix.



Exemple 3.12. Frontera de dominació d'un graf

El conjunt de nodes que formen part de la frontera de dominació es pot calcular a partir d'**equacions recursives**, de la mateixa manera que els conjunts de dominadors.

- **DF [n]** = frontera de dominació del node n
- **DF_{local} [n]** = conjunt de successors de n que no són dominats estrictament per n
- **DF_{up} [n]** = nodes de la frontera de dominació de n no dominats pel dominador immediat de n
- La relació entre els diferents conjunts és la següent:

$$DF[n] = DF_{local}[n] \cup \left(\bigcup_{c \text{ tals que } n = idom(c)} DF_{up}[c] \right)$$

3.7 - Anàlisi del fluxe de dades

L'anàlisi del fluxe de dades preten recollir informació sobre les variables del programa durant l'execució. L'objectiu és obtenir la informació necessària per poder realitzar optimitzacions sobre el programa.

L'estructura de la qual es parteix per realitzar l'anàlisi del fluxe de dades és el graf de fluxe de control, on cada node és un bloc bàsic. Sobre aquesta estructura expressarem la informació del fluxe de dades en forma d'**equacions** (tal i com feiem en l'anàlisi de dominadors): aquestes equacions defineixen el valor d'una dada en un cert bloc bàsic utilitzant les dades dels blocs bàsics adjacents, predecessors o successors.

L'anàlisi del fluxe de dades es basa en dos conceptes: ús i definició. Una instrucció **defineix** una variable **quan canvia el seu valor**. Una instrucció **usa** una variable **quan consulta el seu valor sense modificar-lo**. El mateix concepte pot ser usat en els blocs bàsics: es diu que una variable és usada en un bloc bàsic quan alguna de les instruccions del bloc bàsic usa la variable, i que una variable és definida en el bloc bàsic quan alguna de les instruccions del bloc bàsic defineix la variable.

```
.i0:
    .if a > b .goto .i2
.i1:
    max := b
    .goto .i3
.i2:
    max := a
.i3:
    val := elevar_quadrat (max)
```

Exemple 3.13. Úsos i definicions

L'exemple 3.13 ens mostra un fragment de programa format per 4 blocs bàsics (0,1,2,3), que calcula el màxim.Les variables a i b no són definides en tot l'exemple, però sí que són usades: la variable a és usada en els blocs bàsics 0 i 2, i la variable b és usada en els blocs bàsics 0 i 1. La variable max és definida en els blocs bàsics 1 i 2 i és usada en el bloc bàsic 3. Finalment, la variable val és definida en el bloc bàsic 3 i no s'usa enlloc. Com es pot veure, les assignacions a:= x defineixen la variable a, però hi ha altres instruccions (com ara les crides a funció) que poden definir una variable.

L'anàlisi del fluxe de dades es realitza de forma incremental, com l'anàlisi del fluxe de control. Les diferents fases són les següents:

- en primer lloc, s'examina en quins punts del programa el valor d'una variable es consultarà posteriorment i en quins punts no es tornarà a consultar **(anàlisi de vida)**
- després es contrueixen les **cadenes d'ús i definició**, que indiquen en cada instrucció quin és el proper ús de les variables definides i l'anterior definició de les variables usades
- un cop tenim aquesta informació, podem realitzar altres anàlisis com les **expressiones disponibles** en un punt determinat del programa. Tota aquesta informació ens permetrà realitzar les optimitzacions sobre el programa

3.7.1 - La memòria en l'anàlisi de fluxe de dades

Abans de començar amb els diferents anàlisis del fluxe de dades, cal realitzar una definició molt més precisa del concepte d'ús i definició per tenir en compte dos aspectes: **la memòria i els punters**.

El llenguatge intermedi inclou instruccions per obtenir l'adreça d'una variable (a := &b) i per assignar un valor a l'adreça apuntada per una variable (a := *b). Si no disposem de més informació, hem de considerar que **un punter pot apuntar a qualsevol adreça de la memòria** i per tant pot definir / utilitzar qualsevol variable de la memòria.

```
.i0:
    .if a > b .goto .i2
.i1:
    max := &b
    .goto .i3
.i2:
    max := &a
.i3:
    val := elevar_quadrat (max)
```

Exemple 3.14. Úsos i definicions considerant la memòria

En l'exemple 3.14 podem veure un programa similar a l'anterior exemple però que ara utilitza punters. El problema és el següent: la crida a elevar_quadrat defineix la variable apuntada per max? En principi no ho sabem, i hauríem d'examinar aquesta funció per veure-ho. I si defineix la variable apuntada per max, quina defineix? Tampoc ho sabem, perquè max pot apuntar a a o b. En aquest cas no és tan clar quines variables defineix la crida a elevar_quadrat.

Definirem el conjunt de variables en memòria d'una funció com el conjunt de les **variables estructurades, les variables globals i aquelles variables a les que agafem l'adreça** (a:= &b). Llavors, una instrucció que consulta la memòria usarà totes les variables en memòria, i una instrucció que defineixi la memòria definirà totes les variables en memòria.

Poden usar la memòria	Poden definir la memòria
a := *b a:= b[x] a:= &b[x] crida_a_funció ()	*a := b a[x] := b &a[x] := b crida_a_funció ()

Aquesta aproximació és molt conservadora i es pot millorar, intentant analitzar on apunta cada punter. Això es coneix com **anàlisi d'àlies**, i consisteix a assignar a cada punter un conjunt d'àlies (adreces de memòria a les que pot apuntar). Una assignació a un punter no defineix tota la memòria, només el conjunt d'àlies als que podia apuntar el punter.

3.7.2 - Anàlisi de vida

L'objectiu de l'anàlisi de vida és detectar a cada punt del programa **quines variables contenen un valor útil**. Aquestes variables s'anomenen **variables vives**. Es diu que una

variable està viva en un punt del programa si aquesta variable pot ser consultada posteriorment sense tornar a ser definida.

a:= 1 c:= 0 b:= a +1	a:=1 a:=2 b:=a	a:= 1 .if c < 3 .goto .i3 a:= b +c .i3: d:= a
(a)	(b)	(c)

Exemple 3.15. Variables vives en un programa

En l'exemple 3.15 es tracta de comprovar si la variable a està viva després de la instrucció a:= 1. En el cas (a) la variable a està viva, ja que és utilitzada a b:= a + 1 sense tornar a ser definida. En el cas (b), en canvi, no està viva, ja que és redefinida a a:= 2 sense tornar a ser utilitzada. En canvi, la variable a sí que està viva després de a:= 2. En el darrer cas, la variable a està viva: pot ser consultada en d:= a. Cal notar que si es produeix el salt la variable seria definida, però tot i així hem de considerar que la variable està viva, donat que **pot** ser utilitzada.

Per determinar si una variable està viva, tenim dos mètodes, com en tots els anàlisis del fluxe de dades: **expressar el problema en forma d'equacions recursives** i resoldre-les, o bé, cercant, per a cada definició d'una variable, les seves properes definicions i usos per veure si la variable està viva. Donat que en el nostre llenguatge el nombre de variables (i d'assignacions a variables) és potencialment gran, és millor el primer mètode, al tractar totes les variables i tot el programa al mateix temps.

Per a cada bloc bàsic, definim una serie de conjunts:

- **use[n]** = conjunt de variables usades al bloc bàsic n abans de ser definides a n
- **def[n]** = conjunt de variables definides al bloc bàsic n
- **in[n]** = conjunt de variables vives al principi del bloc bàsic n
- **out[n]** = conjunt de variables vives al final del bloc bàsic n

Informalment, una variable està viva al principi del bloc bàsic n si és utilitzada pel bloc bàsic sense ser definida (o sigui, si és a use), o si és utilitzada per algun succesor del bloc bàsic n sense ser definida a n. Això s'expressa formalment en les equacions recursives:

- $in[n] = use[n] \cup (out[n] - def[n])$
- $out[n] = \bigcup_{s \in succ[n]} in[s]$

Aplicant aquestes equacions es poden calcular els conjunts in i out per a cada bloc bàsic: inicialitzant els conjunts in i out a $\varnothing$ i iterant amb aquestes equacions s'arriba a la solució. A [APP98] es troba la demostració que aquest algorisme acaba, i a [AHO95] i [APP98] es troben consideracions sobre l'ordre en que s'han d'avaluar aquestes equacions per reduir el temps d'execució.

Un exemple d'execució d'aquest algorisme es pot veure en el següent fragment de programa (exemple 3.16), en que es pot veure l'evolució dels conjunts in i out fins a arribar a la solució final. Cal destacar, per exemple, que en el bloc bàsic 1 la variable e no està dins el conjunt use, donat que abans de ser utilitzada en el bloc bàsic és definida.

```
.i1:
a:=1
f:= c + d
d:= b-1
e:=d
b:=2
.if e < b .goto .i3
.i2:
g:= a * f
d:= g +b
.i3:
e:= d +b
```

use(1) = {b,c,d}
def(1) = {a,b,d,e,f}

use(2) = {a,b,f}
def(2) = {d,g}

use(3) = {d,b}
def(3) = {e}

Exemple 3.16. Anàlisi de vida en un programa

	1a iteració		2a iteració		3a iteració	
Blocs bàsics	in	out	in	out	in	out
1	{b,c,d}	{}	{b,c,d}	{a,b,d,e,f}	{b,c,d}	{a,b,d,e,f}
2	{a,b,f}	{}	{a,b,f}	{d,b}	{a,b,f}	{d,b}
3	{d,b}	{}	{d,b}	{}	{d,b}	{}

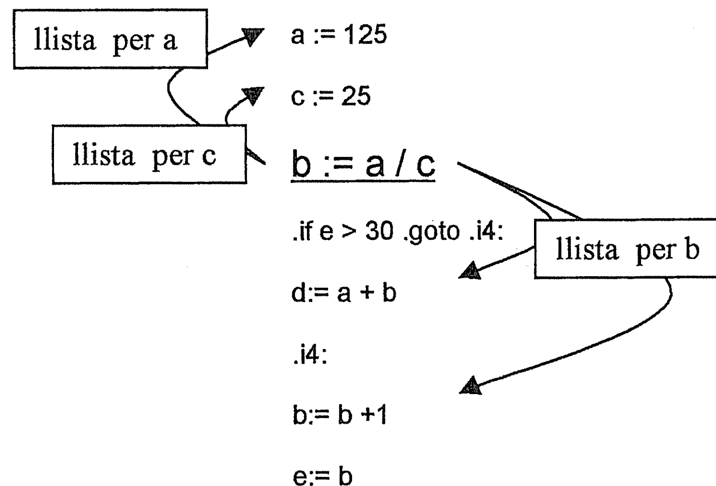
La informació sobre les variables vives en cada punt del programa es fa servir per a dos propòsits:

- a **l'assignació de registres**, per detectar quan **dues variables poden compartir un registre**. Si dues variables estan vives en un mateix punt del programa, llavors no poden estar en el mateix registre.
- Si una instrucció assigna un valor a una variable i la variable no està viva després de l'assignació, significa que mai es consultarà el valor assignat. Així, pot ser que la instrucció sigui inútil, ja que el seu resultat és ignorat, i pugui ser eliminada. Per detectar millor aquestes situacions i accedir més ràpidament a les instruccions es fan servir les **cadenes d'ús i definició**, construides a partir de la informació de vida de les variables.

3.7.3 - Cadenes d'ús i definició

Una forma eficient d'accedir a la informació produida per l'anàlisi del fluxe de dades és la construcció de les cadenes d'ús i definició. Es vol tenir associada a cada instrucció la següent informació:

- per a cada variableV definida per una instrucció I, es vol la llista d'instruccions que consulten la variable V. Més concretament, es volen aquelles instruccions que consulten el valor de V assignat en la instrucció I. Aquesta llista serà la **cadena d'usos**, les instruccions que usen el valors generats per la instrucció I.
- per a cada variable V consultada per una instrucció I, es vol la llista d'instruccions que defineixen la variable V. Més concretament, es volen aquelles instruccions que generen el valor consultat en la instrucció I. Aquesta serà la **cadena de definicions**, les instruccions que generen els valors usats per la instrucció I.



Exemple 3.17. Cadenes d'ús i definició

En l'exemple 3.17 es poden observar les cadenes d'ús (esquerra) i definició (dreta) per a una instrucció determinada. Hi ha una llista associada a cada operand de la instrucció i cada operand pot tenir diversos usos / definicions. Si un operand de la instrucció és constant, llavors no té associada cap llista de definicions (evidentment,tampoc cap llista d'usos, perquè una instrucció no pot donar valor a un operand constant).

Per calcular les cadenes d'ús i definició, s'utilitza informació sobre l'**anàlisi de vida** i les **definicions disponibles** (un cop sabem que una variable està viva, observem quines són les instruccions que han definit un valor en aquesta variable). Les definicions disponibles en un moment del programa es poden calcular a partir d'equacions (com l'anàlisi de vida) o bé directament: en un punt del programa on es volen trobar les definicions disponibles d'una variable V, es retrocedeix fins a trobar les últimes definicions de V.

Les cadenes d'ús s'utilitzen directament en moltes optimitzacions (com l'eliminació de codi mort) i representen una **forma eficient d'accés a la informació del fluxe de dades**. Un inconvenient d'aquesta estructura és que **consumeix molta memòria** (diverses llistes per a cada instrucció), especialment si les cadenes d'ús i definició són molt llargues. Una solució per a aquest problema és la transformació a SSA (veure 3.8.11), que es basa en reescriure el programa de manera que només hi hagi una definició per a cada variable. Això converteix la cadena de definicions en una única variable, reduint el cost de memòria.

3.7.4 - Altres anàlisis del fluxe de dades

Per a realitzar altres optimitzacions podem necessitar més informació sobre l'estat de les variables. Un exemple és el de les **expressions disponibles**: conèixer, en cada punt del programa, quines expressions ja han estat avaluades i es poden reaprofitar sense recalcular-les. Com ja hem vist en l'anàlisi de vida i la construcció de les cadenes d'ús i definició, la informació del fluxe de dades es pot calcular de dues maneres:

- com a **equacions recursives** que cal resoldre. D'aquesta manera el problema queda resolt en tots els punts del programa al mateix temps.

- realitzant **recorreguts** en el programa en cada punt en que necessitem la informació. El tipus de recorregut concret i les condicions d'aturada depenen del problema a resoldre

El criteri que ens permet decidir quina de les dues opcions és millor és el següent: els recorreguts seran més aconsellables si es vol informació en **pocs punts del programa** i quan la **distància a recòrrer sigui petita**.

En el cas de les expressions disponibles el recorregut a realitzar és el següent: quan trobem una expressió busquem en les instruccions anterior fins a trobar

- una instrucció que calculi la mateixa expressió (expressió disponible)
- o bé una instrucció que defineixi alguns dels operands (expressió no disponible)

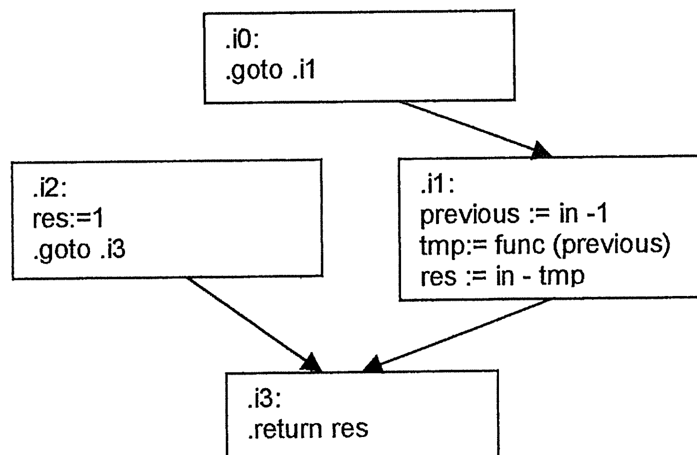
Com que s'espera que el rang de vida de les variables sigui curt, les cerques a realitzar seran curtes i per tant he triat els recorreguts per a calcular les expressions disponibles (veure eliminació de subexpressions comunes a 3.8.7). Les equacions recursives per a resoldre aquest problema es poden trobar a [AHO90] i [APP98], i en totes dues referències apareixen altres problemes formulats en forma d'equacions.

3.8 - Optimitzacions de codi

3.8.1 - Eliminació de codi inabastable

En el graf de fluxe de control hi ha un node inicial que no té cap predecessor, i que és el primer node que s'executa quan es crida la funció. Però en el graf de fluxe de control poden haver-hi altres nodes sense predecessors. El significat d'aquests nodes és ben diferent: no s'arribaran a executar mai. Aquest nodes es coneixen com **codi inabastable i poden ser eliminats, reduint el tamany del programa i sense afectar-ne el comportament**. De tota manera, aquesta optimització no redueix el temps d'execució, ja que les instruccions eliminades no s'haguessin executat mai.

Informalment, l'algorisme per a eliminar codi inabastable és el següent: esborrem els nodes que no tenen cap predecessor, comprovem si al fer-ho deixem altres nodes sense predecessors i si és així els tornem a esborrar, ...



Exemple 3.18. Codi inabastable

L'exemple 3.18 mostra un graf de fluxe de control on el node 2 és inabastable, i per tant pot ser eliminat. Després d'eliminar el node 2, el node 3 continúa tenint predecessors i per tant no pot ser eliminat.

Aquest codi inabastable pot aparèixer en el programa per diverses causes. Una és l'ús de **sentències del tipus if (DEBUG) {}** , que deixen d'executar-se si es canvia el valor d'una constant i que serveixen normalment per a depurar el programa o configurar-lo. Una altra causa és que apareguin com a producte **d'altres optimitzacions** (com l'optimització de salts), que poden canviar els salts dins el programa fent que un bloc bàsic que abans era abastable deixi de ser-ho.

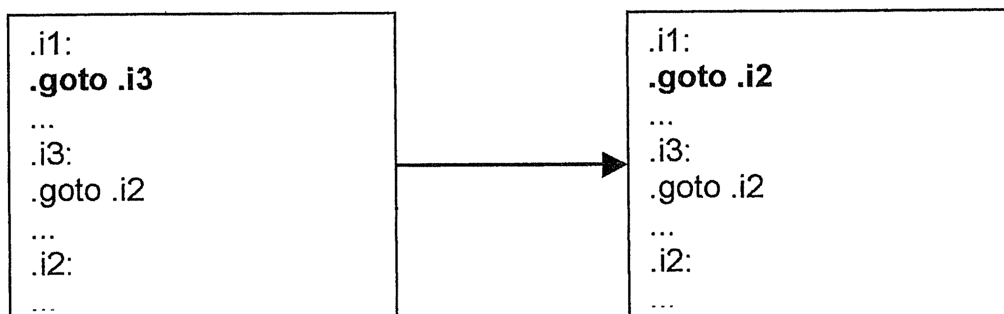
3.8.2 - Optimitzacions de salts

La traducció d'un llenguatge d'alt nivell a codi intermig **produeix moltes instruccions de salt**: les estructures if ... then ... else, while ... do ... , for ..., etc. són reduides en el codi intermedi a salts condicionals i incondicionals.

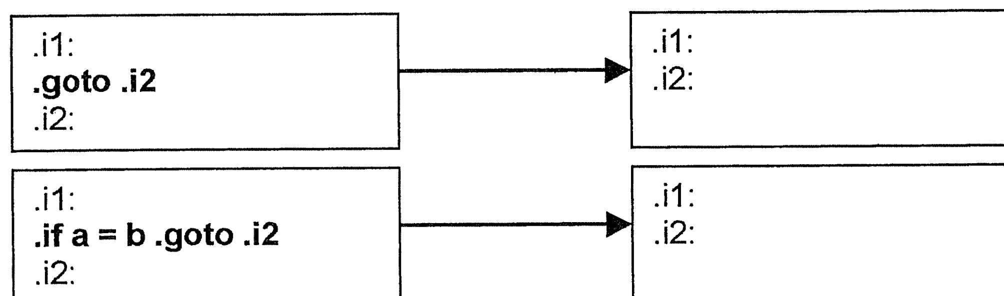
L'objectiu de l'optimització de salts és **traduir una seqüència de salts a una altra equivalent que tingui menys cost en temps d'execució**. Existeixen moltes optimitzacions de salt possibles, i la diferència consisteix en els patrons de salts que s'intenten optimitzar. Les situacions que s'han optimitzat són les següents:

- salts incondicionals a altres salts incondicionals

És possible que el destí d'una instrucció de salt incondicional sigui una altra instrucció de salt incondicional. En aquest cas, es pot substituir l'etiqueta de destí del salt inicial per l'etiqueta de destí del salt final, **reduint el temps d'execució en una unitat**.

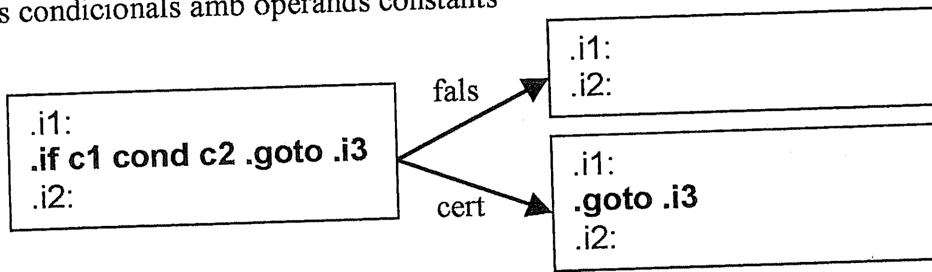


- salts a la següent instrucció



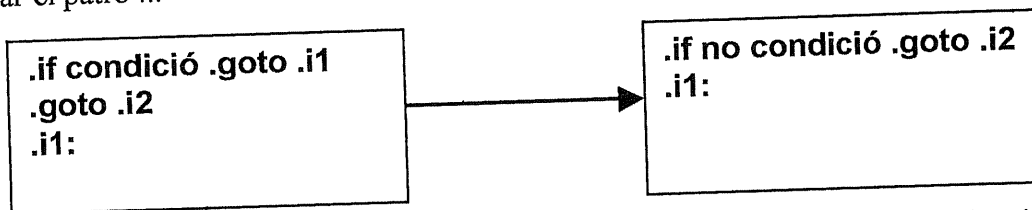
Si tenim una instrucció de salt (condicional o incondicional) a la següent instrucció del programa podem eliminar el salt. El programa tindrà el mateix comportament i un temps d'execució menor.

- salts condicionals amb operands constants



Si els dos operands d'un salt condicional són constants, podem avaluar la condició de salt en temps de compilació i substituir el salt condicional per un salt incondicional (si la condició de salt és certa) o eliminar el salt (si la condició de salt és falsa).

- buscar el patró ...



En aquest cas, la seqüència de salts a optimitzar és més específica, i redueix el temps d'execució en cas que la condició de salt sigui falsa (dos instruccions de salt passen a ser una sola instrucció).

De la mateixa manera que s'han optimitzat aquests patrons es podrien buscar patrons més complicats per optimitzar. Només cal tenir en compte que el temps de localitzar aquests patrons sigui compensat per la millora oferida: el millor és buscar patrons **molt usuals, fàcils de detectar** i **que ofereixin grans beneficis** pel que fa al temps de càlcul.

L'aplicació d'aquestes optimitzacions pot provocar que alguns blocs bàsics passin a ser inabastables, podent-se aplicar l'optimització "eliminar codi inabastable". Es molt usual que aplicar una optimització permeti aplicar una altra optimització. De tota manera, sempre arriba un punt en que no es poden aplicar més optimitzacions (veure la demostració a l'Annex A.2).

3.8.3 - Eliminació de codi mort

L'objectiu de l'eliminació de codi mort és la **supressió d'instruccions que calculen un valor que després no serà utilitzat**. Aquestes instruccions s'anomenen **codi mort**, i poden ser eliminades si no tenen cap efecte lateral com ara accessos incorrectes a memòria (accessos a NULL), excepcions aritmètiques (divisió per zero), o canviar el contingut de la memòria (crides a funció). De la mateixa manera, cal evitar eliminar les instruccions que no defineixen cap operand (com ara .goto), que sempre tindran una llista d'usos buida.

```
a:= 1          a:= c / d      a:= funcio (a,c)
a:= b + 2      a:= b + 2      a:= b + 2

(a)            (b)            (c)
```

Exemple 3.19. Exemples d'eliminació de codi mort

En els tres casos de l'exemple 3.19 la primera instrucció defineix el valor de la variable a i aquesta definició no és consultada posteriorment. Per tant, en els tres casos, la primera instrucció dels programes és codi mort. Pot ser eliminada aquesta instrucció sense risc? En el cas (a) és així, donat que a:= 1 no té efectes laterals. En canvi, en el cas (b), a:= c / d podria causar una divisió per zero i no pot ser eliminada, ja que canviaria el comportament del programa. Finalment, en el cas (c) no podem eliminar la crida a la funció, perquè dins de la funció es podrien modificar el contingut de la memòria (per exemple, si b és una variable global).

Per a detectar les instruccions que poden ser considerades codi mort s'utilitzen les **cadenes d'ús i definició** (veure 3.7.3). Si la cadena d'usos d'una instrucció és buida, llavors és codi mort i pot ser eliminada si no té efectes laterals. Quan eliminem una instrucció, podem seguir les cadenes d'ús i definició per veure quines instruccions passen a ser codi mort al eliminar aquesta instrucció.

Moltes optimitzacions intenten reduir la llista d'usos d'una instrucció per a que aquesta es converteixi en codi mort i pugui ser eliminada. Algunes d'aquestes optimitzacions són la **propagació de constants** (que vol eliminar les instruccions var:= constant), la **propagació de còpies** (que vol eliminar les instruccions var1:= var2) i **algunes optimitzacions de variables d'inducció**.

3.8.4 - Propagació de constants

L'objectiu d'aquesta optimització és **eliminar les instruccions d'assignació de constants a variables** (var:= const). La manera de fer-ho és **buscar totes les instruccions a les que només arriba la definició de var guardada en var:= const, i substituir la variable var per la constant**. Si això ho podem fer en totes les instruccions de la llista d'usos de var:= const, aquesta instrucció es convertirà en codi mort i podrà ser eliminada.

.i0:	.i0:
d:= 0	d:= 0
.i1:	.i1:
f:=1	f:= 1
b:= 2 + d	b:= 2 + d
d:=d + 1	d:= d +1
.if g > 20 .goto .i3	.if g > 20 .goto .i3
e:= f +2	e:= 1 +2
.goto .i1	**.goto .i1**
.i3:	.i3:
.return g	.return g

Exemple 3.20. Propagació de constants en codi intermedi

En l'exemple 3.20 hi ha dos instruccions d'assignació de constants: d:=0 i f:= 1. Pel que fa a la primera, les instrucció que usen aquesta definició són: b:= 2+d i d:= d+1, però no es pot fer la propagació de constants perquè aquestes dues instruccions també reben la definició de d feta a d:= d+1. En canvi, amb la instrucció f:=1 es pot fer la substitució a e:=f+2 perquè només hi arriba una definició de f, la de f:=1. La instrucció f:=1 ja no és utilitzada i podrà ser eliminada per l'eliminació de codi mort.

La substitució es podrà fer en totes les instruccions excepte en un cas: la instrucció var1:=&var2. Les constants no tenen cap adreça de memòria, i per tant no es pot substituir var2 per cap constant.

3.8.5 - Operació de constants en temps de compilació

L'objectiu d'aquesta optimització és **detectar les instruccions aritmético-lògiques** (i també altres instruccions com les conversions de tipus) **que poden ser calculades en temps de compilació**, i substituir la instrucció que calcula el resultat per una instrucció que assigna el resultat a una variable.

$$\text{var} := \text{op1} \otimes \text{op2} \Rightarrow \text{var} := \text{resultat}$$

Amb aquesta optimització obtenim dos avantatges. Per una banda, fem **una reducció d'intensitat**: transformem una instrucció aritmética en una instrucció de còpia, que té un cost menor (o igual en el pitjor dels casos) en qualsevol arquitectura. Per altra banda, **després** d'aquesta optimització **es poden aplicar altres optimitzacions**: si el resultat és una constant apliquem propagació de constants (veure 3.8.4) i si és una variable apliquem propagació de còpies.

Les operacions que poden ser realitzades en temps de compilació poden ser agrupades en tres categories:

- les operacions en que **tots els operands són constants**. En aquest grup s'inclouen les instruccions aritmético-lògiques i les instruccions de conversions de tipus. El resultat de l'operació serà sempre una constant. Alguns exemples són:
  a := 2 + 2 ⇒ a:= 4
  b:= .i2r 3 ⇒ b:= 3.0 $r
  c:= .r2i 2.0 ⇒ c:= 2

- les operacions que **tenen un operand constant que és el neutre o l'element absorvent de l'operador**. En aquest cas es proporciona la llista completa d'operadors i substitucions:

Operador	Element		Exemples	
	Neutre	Absorvent	Inicial	Optimitzada
+	0	No	a:= b + 0	a:= b
-	0	No	a:= b - 0	a:= b
*	1		a:= b * 1	a:= b
		0	a:= b * 0	a:= 0
/	1	No	a:= b / 1	a:= b
MOD	1	No	a:= b % 1	a:= 0

AND	cert	fals	a:= b & .true a:= b & .false	a:= b a:= .false
OR	fals	cert	a:= b | .false a:= b | .true	a:= b a:= .true
XOR	fals	No	a:= b ^ .false	a:= b

- Les instruccions de multiplicació per constants que siguin potència de 2 pot ser implementada com un LSHIFT de manera més eficient (desplaçar m possicions equival a multiplicar per 2^m). Uns exemples són:

```
a:= b * 2   ⇒  a:= b << 1
a:= b * 8   ⇒  a:= b << 3
a:= b * 7   ⇒  No hi ha substitució possible
```

Aquesta optimització resulta molt més efectiva quan s'aplica juntament amb la propagació de constants. L'exemple 3.21 mostra com, aplicant alternativament la propagació de constants i l'operació de constants en temps de compilació, seguides d'eliminació de codi mort, es poden eliminar un gran nombre de càlculs.

```
a:= 1          a:= 1          a:= 1         a:= 1          a:= 1        a:= 1
b:= a + 2      b:= 1 + 2      b:= 3         b:= 3          b:= 3        b:= 3
c:= 3 *a   ⇒   c:= 3 * 1  ⇒   c:= 3    ⇒    c:= 3     ⇒    c:= 3    ⇒   c:= 3    ➡
d:= b + c      d:= b + c      d:= b +c      d:= 3 + 3      d:= 6        d:= 6
.return d      .return d      .return d     .return d      .return d    .return 6     .return 6
```

⇒ **Propagació de constants** ⇒ **Operació de constants** ➡ **Eliminació de codi mort**

Exemple 3.21. Operació de constants i propagació de constants

3.8.6 - Eliminació de variables inútils i NOPs

Aquesta optimització pretèn **eliminar les variables locals de la funció** (només les variables escalars i estructurades, mai els paràmetres) **que no són utilitzades i les instruccions NOP**.

Per a fer-ho, es fa un recorregut de la funció generant una llista de les variables utilitzades (usades o definides). En el mateix recorregut, s'eliminen les instruccions NOP que es troben. Quan acaba el recorregut, s'eliminen les variables locals que no es troben en la llista de variables utilitzades.

```
.function exemple             .function exemple
  .parameter p1, p2             .parameter p1,p2
  .scalar a, b, c, d            .scalar a,c
  .struct e[40]
                                a:= p1 * 3
  a := p1 * 3                   c:= p1 +2
  .nop                          .return c
  c:= p1 + 2                  .end exemple
  .return c
.end exemple
```

Exemple 3.22. Eliminació de variables inútils i NOPs

L'exemple 3.22 mostra el resultat que pot produir-se al aplicar aquesta optimització a una funció. Hi ha variables que no són utilitzades en la funció (possiblement a causa d'altres optimitzacions realitzades) i per tant poden ser eliminades. Cal veure que encara que no s'utilitza el paràmetre p2, no es poden canviar els paràmetres de la funció: només es poden eliminar les variables escalars i estructurades.

Amb aquesta optimització obtenim dos tipus de millores: **en temps d'execució** (al eliminar les NOPs) i **en memòria**, perquè el programa té menys variables i per tant, ocupa menys espai a la pila.

3.8.7 - Eliminació de subexpressions comunes

L'objectiu d'aquesta optmització és **detectar les instruccions que calculen una expressió computada prèviament i encara disponible**, i **evitar la repetició dels càlculs**.

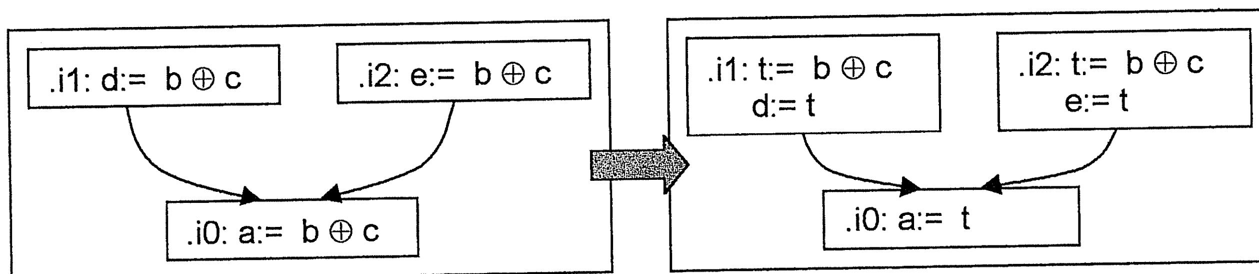
Quan es parla d'expressions, no ens referim només a les instruccions aritmètiques. Una expressió serà qualsevol instrucció que produeixi un valor. En el llenguatge IC, les instruccions que produeixen un valor són les següents instruccions:

a:= b ⊕ c .... instruccions aritmético-lògiques
a:= *b ....instruccions de lectura de memòria
a:= b[c]
a:= &b[c]
a:= .i2r b .... instruccions de conversió de tipus
a:= .r2i b

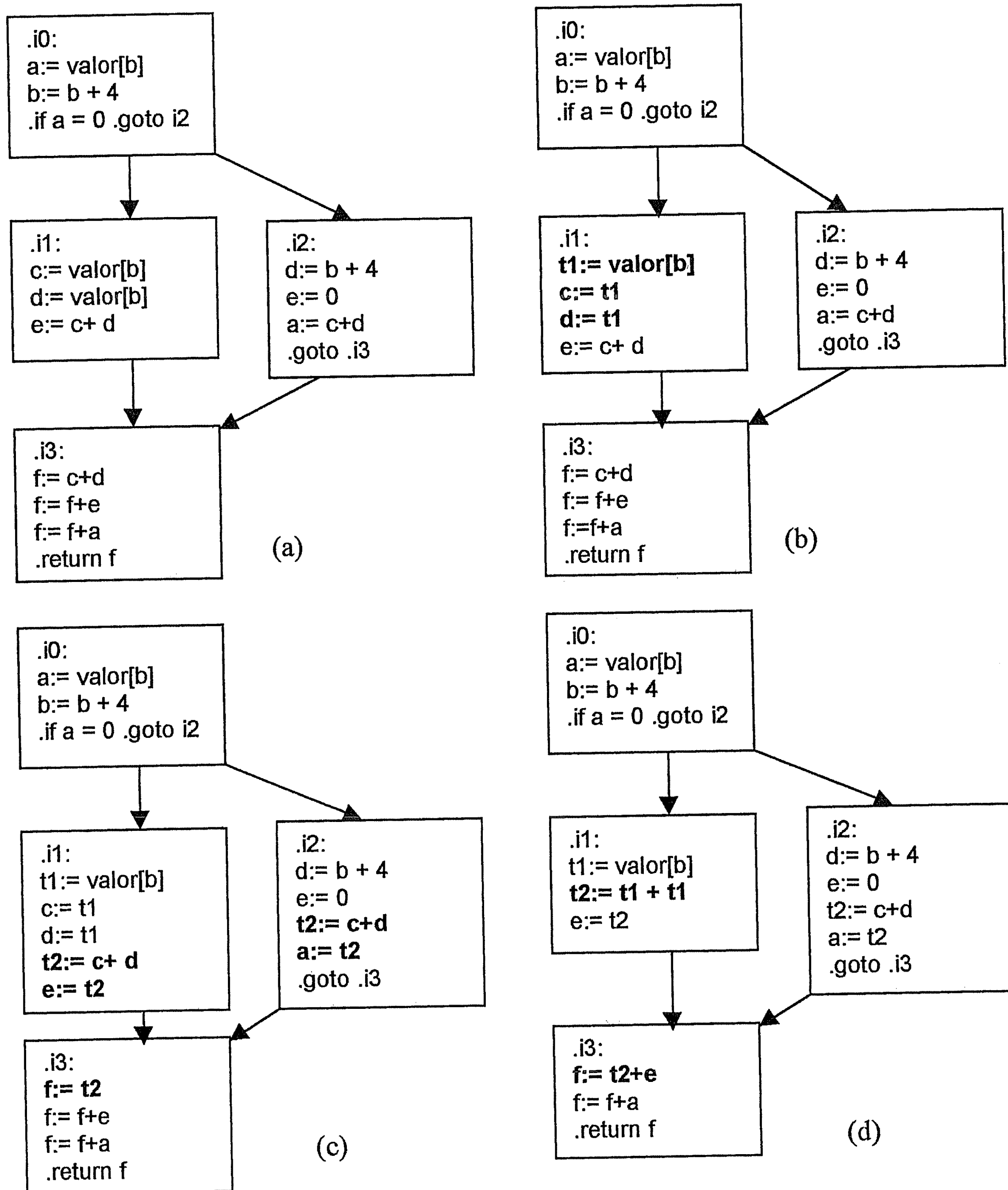
Cada cop que trobem una d'aquestes expressions, hem de comprovar si estem en les condicions indicades anteriorment: que l'expressió hagi estat computada prèviament i que encara estigui disponible.

- DISPONIBLE: Una expressió deixa d'estar disponible quan una instrucció **defineix algun dels operands de l'expressió** (es diu que **la instrucció mata l'expressió**). Si l'expressió és una lectura de memòria, qualsevol instrucció que defineixi la memòria mata l'expressió.
- COMPUTADA PRÈVIAMENT: Una expressió ha estat computada prèviament si **en tots els camins** que arriben a la instrucció, hi ha una instrucció que calculi la mateixa expressió, i posteriorment, no hi ha cap instrucció que mati l'expressió.

Si trobem una instrucció i0 que calcula una expressió, de manera que en tots els camins que arriben a la instrucció hi ha una instrucció (i1, i2, ...) que calcula l'expressió i després d'aquesta no hi ha cap instrucció que mati l'expressió, realitzem **l'eliminació de subexpressions comunes**:



El resultat de l'optimització és que es substitueix una instrucció que calcula una expressió (a:= b ⊕ c) per una instrucció de còpia (a:=t), de cost menor. Però al mateix temps s'han afegit dues instruccions de còpia (d:=t i e:=t). De tota manera, això no té perquè representar un cost addicional, perquè l'optimització de propagació de còpies s'encarrega d'eliminar les instruccions var1 := var2, i pot ser que aconseguim eliminar totes les instruccions de còpia que hem afegit, fins i tot a:=t.



Exemple 3.23. Eliminació de subexpressions comunes en etapes:
(a) codi original
(b) després d'eliminar l'expressió comuna valor[b]
(c) després d'eliminar l'expressió comuna c+d
(d) després de propagar còpies i eliminar codi mort

En l'exemple 3.23 podem veure com s'aplica l'algorisme d'eliminació de subexpressions comunes a un programa concret. Les expressions que es calculen més d'una vegada en el programa són: b+4, valor[b] i c+d .

- En primer lloc, trobem l'expressió comuna b+4 quan estem al bloc i2. Intentem veure si l'expressió estava calculada prèviament, però trobem b:= b+4, que mata l'expressió perquè assigna un valor a la variable b. Per tant, no es pot fer cap eliminació de subexpressions comunes per l'expressió b+4.
- L'expressió valor[b] la trobem al mòdul i1 en c:= valor[b], però al retrocedir trobem la instrucció b:= b+4, que mata l'expressió. En canvi, en d:= valor[b] sí que podem fer eliminació de subexpressions comunes, ja que trobem l'expressió calculada en la instrucció anterior.
- L'expressió c+d es troba al bloc i3. Retrocedint per tots els camins que poden portar a la instrucció (blocs i1 i i2) trobem l'expressió calculada i disponible, de manera que podem efectuar l'eliminació
- Finalment, es pot veure quin és l'efecte d'aplicar la propagació de còpies (i l'eliminació de codi mort), ja que s'eliminen gran part de les instruccions de còpia afegides per l'eliminació de subexpressions. A més, és possible que la fase d'assignació de registres aconsegueixi eliminar les instruccions de còpia restants.

El procediment per a buscar les expressions comunes és el següent: un cop trobem una expressió, **fem una cerca** cap enrere per tots els camins que arriben a la instrucció. Si en algun camí no trobem una instrucció que calculi l'expressió o trobem alguna instrucció que mati l'expressió, aturem la cerca i passem a examinar la següent expressió. També es poden calcular les expressions disponibles en un punt del programa en forma **d'equacions recursives**, similars a les utilitzades en l'anàlisi de vida: ([AHO90],[APP98])

in [n], out [n] - expressions disponibles al principi i final del bloc bàsic n, respectivament
gen [n] - expressions generades (calculades i no matades posteriorment) al bloc bàsic n
kill [n] - expression matades al bloc bàsic n.

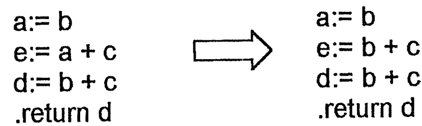
- in [n] = $\varnothing$ si n és el node inicial
- in [n] = $\bigcap_{p \in \text{predecessors}[n]} (\text{out}[n])$ si n no és el node inicial
- out [n] = gen [n] $\cup$ (in [n] - kill [n])

L'últim aspecte que cal tenir en compte abans d'escollir un algorisme és la comparació de les expressions: alguns operands són commutatius, i per tant expressions com "b + c" i "c + b" haurien de ser considerades la mateixa expressió. És per això que, com un pas previ a l'optimització de subexpressions comunes, **escrivim les expressions de forma canònica**, de manera que les expressions equivalents tinguin una representació única i sigui més fàcil la comparació. El que fem és **triar un ordre concret per als operands en les expressions que tenen operands commutatius**: si un dels operands és una constant, serà l'operand de la dreta; si els dos operands són variables, establim un ordre total entre elles (en funció del tipus de variable i l'adreça de memòria on es troba) de manera que l'operand més gran és el de la dreta. D'aquesta manera, tota expressió estarà representada de manera única.

3.8.8 - Propagació de còpies

L'objectiu d'aquesta optimització és **eliminar les instruccions** var1:=var2 (anomenades instruccions **de còpia**). El procediment per a fer-ho és buscar les instruccions que usen

aquesta instrucció i substituir (sempre que sigui possible) la variable var1 per la variable var2; si podem fer aquesta substitució (anomenada **propagació de la còpia**) en totes les instruccions que usen var1:= var2, la instrucció de còpia es convertirà en codi mort i podrà ser eliminada.



Exemple 3.24. Propagació de còpies

Es pot veure en l'exemple 3.24 com la instrucció de còpia es converteix en codi mort i pot ser eliminada. Quines condicions s'han de verificar per a poder propagar la còpia? Si I és la instrucció de còpia a:= b i J és la instrucció que usa la instrucció I, **es pot propagar la còpia si es compleix**:

- La instrucció I ha de ser l'única definicó de la variable a que arriba a J
- No hi ha cap assignació a la variable b en cap dels camins que van de I fins a J (fins i tot els que passen per J més d'una vegada)
- J no pot ser una instrucció e:= & a. Això passa perquè e:= &a i e:= &b mai tindran el mateix valor, encara que a i b continguin el mateix valor.
- Si la instrucció I és a:= a, llavors es pot eliminar directament.

Els avantatges de la propagació de còpies són dos: per una banda, la propagació de còpies **permet detectar noves subexpressions comunes** (com es pot veure en l'exemple 3.24). Per altra banda, **elimina instruccions** i per tant, **millora el temps d'execució**. De tota manera, la propagació de còpies **no** pot eliminar totes les instruccions de còpia, i per això l'assignació de registres té una fase encarregada d'eliminar aquestes instruccions de còpia (veure el capítol 4, assignació de registres).

3.8.9 - Optimitzacions de bucles

El bucles són fragments de codi que es poden executar un gran nombre de vegades (iteracions) en un programa, i per això, la seva eficiència és determinant en el temps d'execució total del programa.

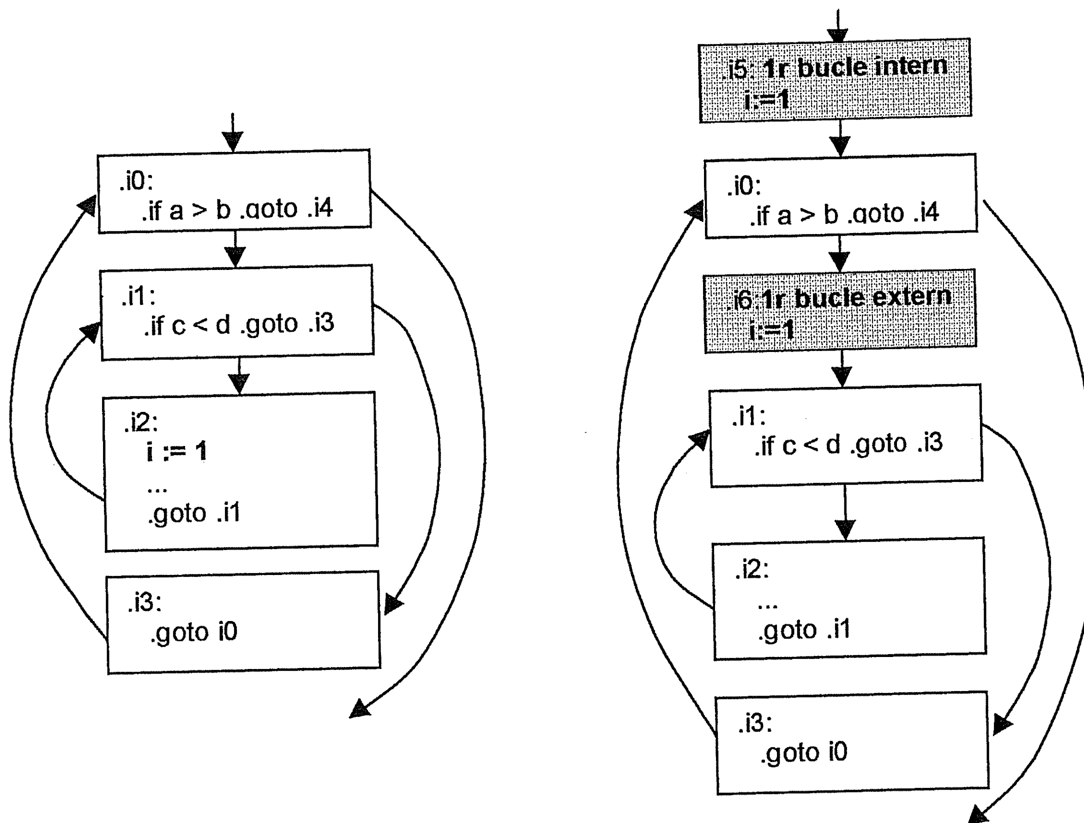
A l'apartart 3.6.5 hem vist com detectar bucles i hem presentat les seves característiques: tenen un únic punt d'entrada (**encapçalament**), que domina tots els nodes del bucle, i han de tenir una o més **arestes de retrocés**, que van d'un node del bucle a l'encapçalament. Pel que fa als punts de sortida, un bucle en pot tenir més d'un.

Considerant un bucle aïllat, les optimitzacions que hi podem realitzar són les següents:

- Les instruccions dins el bucle s'executaran moltes vegades, i per tant, si hi ha alguna instrucció que **calcula el mateix valor** a cada volta del bucle es pot extreure del bucle i calcular-se abans del encapçalament. Així, les instruccions només s'executen una vegada, en comptes de fer-ho a cada iteració. Aquesta optimització s'anomena **extracció d'instruccions invariants**, i té un pas previ de **detecció d'instruccions invariants**. L'exemple 3.25 mostra l'extracció d'una instrucció invariant.

- En els bucles acostuma a haver-hi **variables d'inducció**, que guarden informació relativa al nombre d'iteracions realitzades fins al moment. Un exemple molt usual són les variables i utilitzades en els bucles de tipus for. Després de **detectar** aquestes variables, podem fer una **reducció d'intensitat** dels càlculs que utilitzen aquestes variables i potser podem arribar a **eliminar** totes les assignacions que s'hi fan.

Els bucles poden estar **imbricats** (un bucle inclós dins un altre), i això s'ha de tenir en compte a l'hora d'optimitzar. S'ha de seguir un ordre a l'hora d'optmitzar bucles imbricats: **no optimitzem un bucle fins que no hem optimitzat els bucles imbricats dins seu**. El motiu d'aquest ordre és que quan optimitzem el bucle inferior, podem extreure'n càlculs que també puguin ser extrets del bucle exterior; si optimitzem primer el bucle exterior, els càlculs extrets al bucle interior no seran examinats posteriorment. A l'apartat 3.6.5 ja vam presentar l'algorisme que ordenava els bucles seguint aquest ordre.

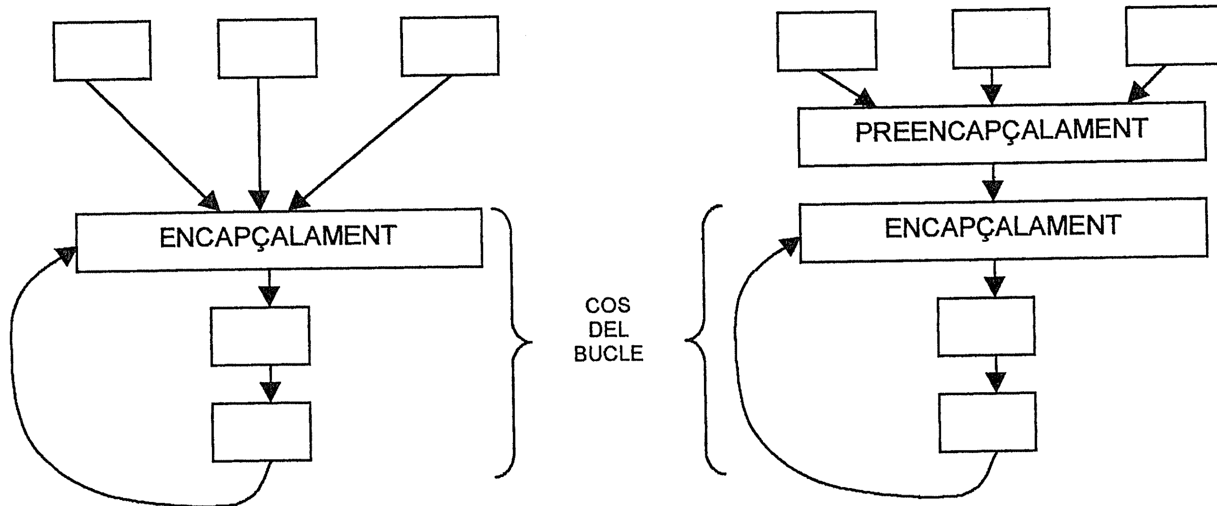


Exemple 3.25. Bucles aniuats

L'exemple 3.25 mostra perquè és important l'ordre quan optimitzem els bucles. Hi ha dos bucles imbricats, d'encapçalaments i0 i i1, i en el bucle més intern hi ha una instrucció invariant: i:=1. Sota certes condicions, podem extreure la instrucció del bucle, com es pot veure a l'exemple de la dreta (considerant només un dels blocs marcats, i5 o i6). Doncs bé, si optimitzem primer el bucle més intern i després el més extern, ens trobaríem en la situació indicada pel node i5 (el node i6 no existiria): la instrucció i:=1 seria calculada un sol cop en tots els dos bucles. Si optimitzem primer el bucle extern i després l'intern, estaríem en la situació indicada pel node i6 (el node i5 no existiria): la instrucció i:= 1 seria calculada a cada iteració del bucle exterior.

3.8.9.1 - Creació del preencapçalament

En les optimitzacions de bucles, de vegades volem crear instruccions (o moure instruccions existents) de manera que sempre **s'executin abans d'executar el bucle** i **només s'executin una vegada**. Com que el bucle té un únic punt d'entrada, l'encapçalament, podem aconseguir els nostres objectius creant un node nou, el **preencapçalament**, que s'executi abans del encapçalament la primera vegada que entrem en el bucle.



Si hem de moure una instrucció fora del bucle, la situarem al final del preencapçalament. Les instruccions del preencapçalament s'executaran una vegada (no a cada iteració), però això podria arribar a representar una pèrdua d'eficiència si el bucle no s'arriba a executar mai. De tota manera, el gran benefici que obtenim si el bucle s'executa moltes vegades ens compensa aquesta possibilitat.

3.8.9.2 - Extracció d'instruccions invariants

Algunes instruccions d'un bucle tenen operands que no es modifiquen dins del bucle, i per tant calculen a cada iteració el mateix valor. Aquestes instruccions s'anomenen **instruccions invariants.** Alguns exemples molt senzills són i:= 2, b := 2 + 3, però hi ha més expressions invariants a part de les que tenen operands constants.

Formalment, una instrucció es considera invariant si cadascun dels seus operands compleix una de les tres condicions següents:

- l'operand és constant
- la llista de definicions de l'operand només conté instruccions que no són del bucle
- l'operand només té una definició, que és dins el bucle i és invariant

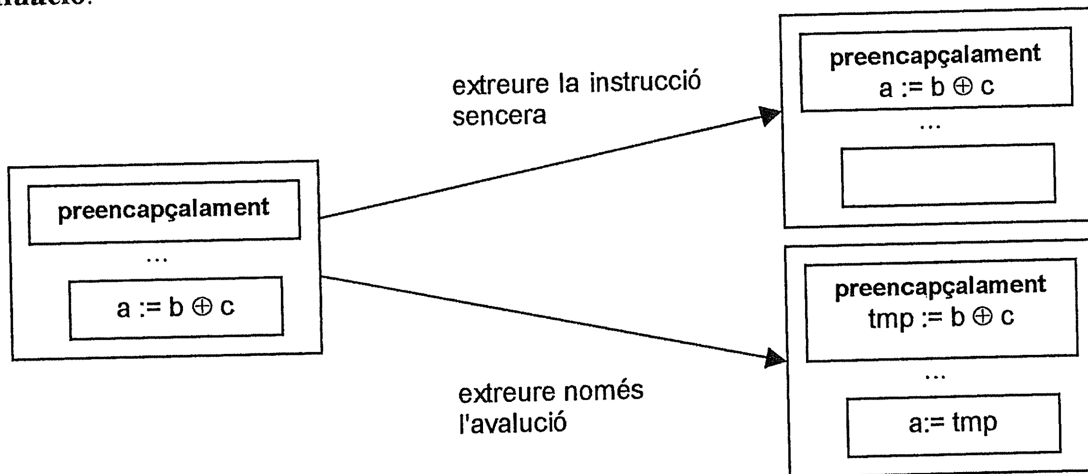
Un cop sabem quines instruccions són invariants, podem realitzar **l'extracció d'expressions invariants**, que consisteix en moure algunes instruccions invariants fora del bucle. Hi ha una sèrie de condicions que s'han de verificar per a poder extreure una instrucció del bucle: si la instrucció a extreure, I , defineix la variable V, llavors:

0- cap dels operands de I han de ser definits dins el bucle

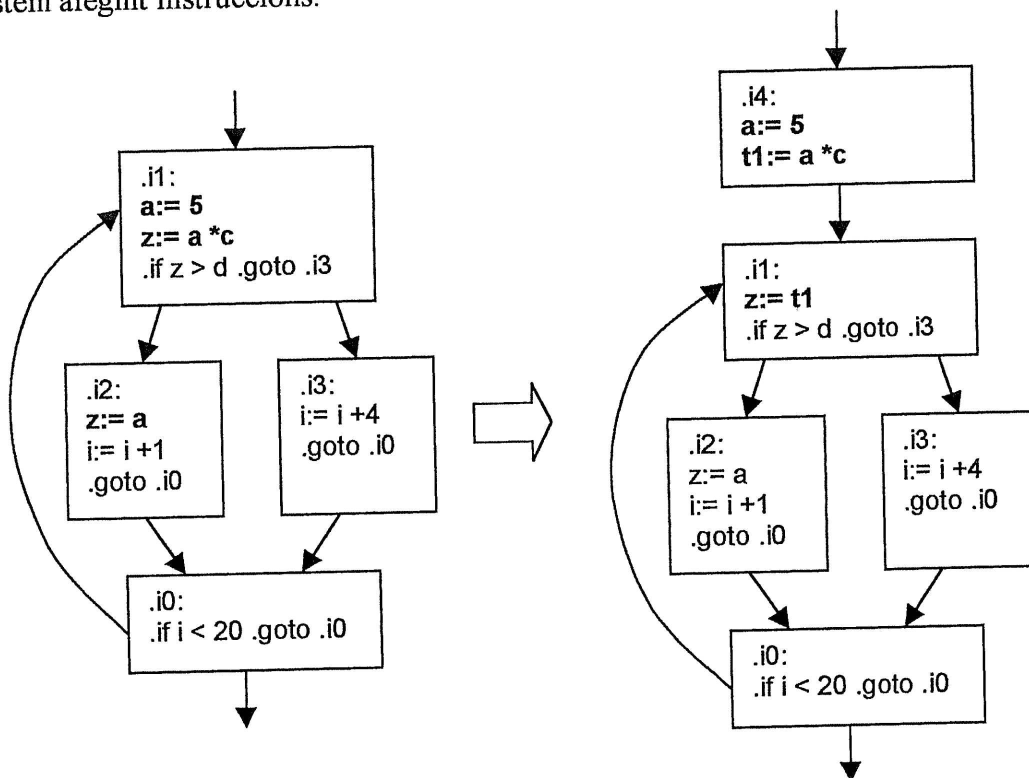
1- la instrucció I no té cap efecte lateral (definir memòria, excepcions aritmètiques)

2- la instrucció I ha de dominar totes les sortides del bucle
3- la única definició de la variable V dins el bucle és la instrucció I
4- totes les instruccions que usen V dins el bucle tenen I com única definició a la llista de definicions

L'extracció d'instruccions invariants es pot fer de dues maneres: si es verifiquen les condicions 0, 2, 3 i 4 anteriors, llavors la instrucció I pot ser **extreta** des del bucle fins al final del preencapçalament. Si només es verifiquen les condicions 0 i 1 i la instrucció no és una còpia, llavors no podem extreure la instrucció de forma segura, però en podem **extreure l'avaluació**.



La propagació de còpies i l'assignació de registres es poden encarregar d'eliminar les instruccions de còpia afegides quan només fem l'extracció de l'avaluació. Però encara que no aconseguim eliminar les instruccions de còpia, el temps d'execució continúa sent més petit, tot i que estem afegint instruccions.



Exemple 3.26. Extracció de codi invariant

Ja hem vist un exemple molt senzill d'extracció de codi invariant a 3.25, però ara en veurem un altre més complet (exemple 3.26). En l'exemple 3.26, per tal d'aplicar l'extracció de codi invariant, primer examinem el bucle buscant el codi invariant. La primera instrucció invariant és a:= 5, que té tots els operands constants. Després trobem com a instrucció invariant z:= a*c, que té un operand definit en una instrucció invariant (a) i un altre operand (c) que només es defineix fora del bucle. Finalment, tenim z:= a, que té un operand definit en una instrucció invariant.

Després hem de considerar, una a una, si podem extreure les diferents instruccions del bucle. La instrucció a:= 5 no té cap operand definit dins el bucle (condició 0), no té cap efecte lateral (condició 1), domina la única sortida del bucle que és i0 (condició 2), és la única definició de la variable a dins el bucle (condició 3) i totes les instruccions que usen a en el bucle usen aquesta definició de a (condició 4). Per tant, aquesta instrucció verifica totes les condicions i pot extreta al preencapçalament.

En canvi, z:= a*c i z:=a verifiquen les condicions 0 (després de l'extracció de a:= 5) i 1, però no són les úniques definicions de la variable z en el bucle (condició 3). Per tant, aquestes instruccions no es poden treure del bucle, però sí que es pot treure l'avaluació en el cas de la instrucció z:= a*c, ja que verifica les condicions 0 i 1 i no és una còpia.

Un detall molt important és que **si el bucle és de tipus while ... do**, és a dir, l'encapçalament del bucle és una sortida del bucle, **resulta molt difícil extreure instruccions**, donat que les instruccions difícilment dominaran l'encapçalament (condició 2 per a poder extreure instruccions). En aquest cas, es pot intentar **transformar el bucle en un bucle de tipus do ... while** (veure 3.8.10) o bé extreure les avaluacions, que no necessiten verificar la segona condició.

Finalment, només comentaré que a [AHO90] apareixen variants de les condicions 0-4 per a determinar si es poden extreure instruccions d'un bucle. En aquesta referència es citen diferents criteris per a relaxar aquestes condicions i possibles riscos d'aquestes relaxacions.

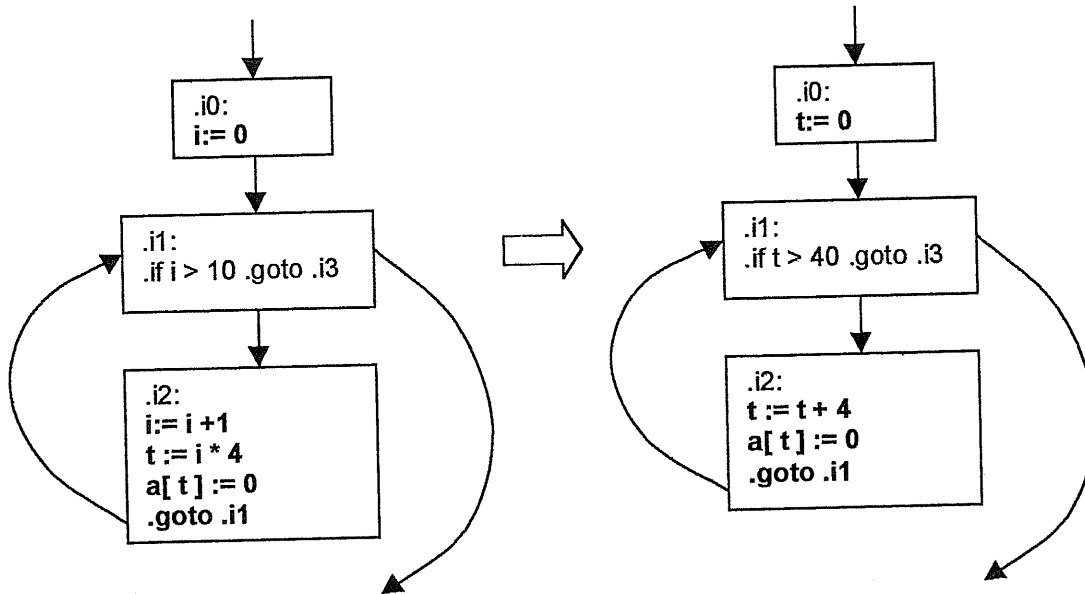
3.8.9.3 - Detecció de variables d'inducció

Una variable V s'anomena **variable d'inducció bàsica** d'un bucle B si en el bucle B **les úniques definicions de V són increments o decrements en valors invariants**, és a dir, només es defineix v ens instruccions de la forma v:= v+k o v:= v-k on k és invariant en el bucle.

L'exemple 3.26, vist anteriorment a l'extracció d'instruccions invariants, conté una variable bàsica d'inducció: la variable i, que només es definida en i:= i +1 i a i:= i+4. Cal destacar que en comptes de ser incrementada per una constant, podria ser incrementada per un valor invariant qualsevol (com ara la variable a). També cal destacar **que no es forçan un únic increment de la variable o que la variable sempre sigui incrementada o decrementada pel mateix valor**. D'aquesta manera, una variable que no sigui la variable d'un for també pot ser una variable d'inducció bàsica.

També es consideren variables d'inducció aquelles variables que utilitzen una altra variable d'inducció en el càlcul del seu valor. Una variable V s'anomena **variable d'inducció derivada de la famíla de W** en el bucle B si:

- només hi ha una definició de V en el bucle B, de la forma v:= w + k , v:= w -k, v:= w * k o v:= w << k, on k és un operand invariant en B i W és una variable d'inducció (bàsica o derivada) en B.
- si W és una variable d'inducció derivada en la família de U, també s'ha de complir que només arribi a la definició de V la definició de W de dins el bucle, i que no hi hagi cap definició de U en cap camí entre la definició de W i la definició de V.



Exemple 3.27. Anàlisi de variables d'inducció i el seu objectiu

L'exemple 3.27 ens mostra un bucle que posa a 0 els elements d'un vector, cadascun dels qual ocupa 4 posicions. Per a calcular la posició en el vector, utilitzem una variable i (variable d'inducció bàsica) i una variable t que té un valor calculat a partir del valor de i (variable d'inducció derivada). Al bucle de la dreta podem veure quin serà el nostre objectiu quan analitzem aquestes variables d'inducció: **expressar els càlculs de lès variables d'inducció derivades de manera que no depenguin de cap altre variable d'inducció i intentar eliminar els càlculs que afectin a variables d'inducció inútils**.

Les variables d'inducció poden ser caracteritzats per una **terna** que ens indica com calcular el valor de la variable. La detecció de variables d'inducció consistirà en **detectar quines variables són variables d'inducció i calcular les seves ternes associades**.

La terna (w,a,b) associada a la variable v ens indica que la variable v es calcula de la següent manera: v := a*w+b, on w és una variable d'inducció bàsica. Les variables d'inducció bàsiques w tenen una terna **(w,1,0)**, ja que w:= 1*w+0. En el cas de les variables derivades, el càlcul de la seva terna es fa a partir de la terna de la variable d'inducció utilitzada:

Càlcul de la terna de la variable d'inducció derivada t a partir de la variable d'inducció v, de terna (w,a,b)

instrucció que defineix t	**terna de t**
t:= v + k	(w,a,b+k)
t:= v - k	(w,a,b-k)
t:= v * k	(w,a*k,b*k)
t:= v << k	(w,a<<k,b<<k)

En l'exemple 3.27, tenim dos variables d'inducció. La variable i és bàsica, i per tant, tindrà com a terna (i,1,0). La variable t és una variable derivada, calculada a partir de t:= 4*i, i per tant tindrà com a terna (i,4*1,4*0), o sigui, (i,4,0).

Les dades que guardem per a cada variable són les següents: la **variable**, la **terna associada**, i **la instrucció que defineix la variable d'inducció** (en el cas de les variables d'inducció derivades).

3.8.9.4 - Reducció d'intensitat de variables d'inducció

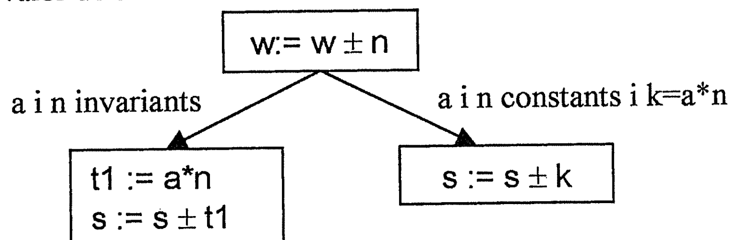
En general, la reducció d'intensitat consisteix en la substitució d'una expressió per una altra menys costosa. En les variables d'inducció, l'objectiu és **substituir les multiplicacions** utilitzades en el càlcul de les variables d'inducció derivades **per sumes i restes**, com ja s'ha vist en l'exemple 3.27.

La manera de fer-ho serà calcular les variables d'inducció derivades directament, en lloc de fer-ho a partir de les variables d'inducció bàsiques. D'aquesta manera, **pot ser que les** variables d'inducció bàsiques deixin d'utilitzar-se en el bucle i **puguin ser eliminades**.

Si tenim una variable d'inducció derivada V de la família de W, de terna (w,a,b) i definida a la instrucció I, llavors cal fer els següents passos:

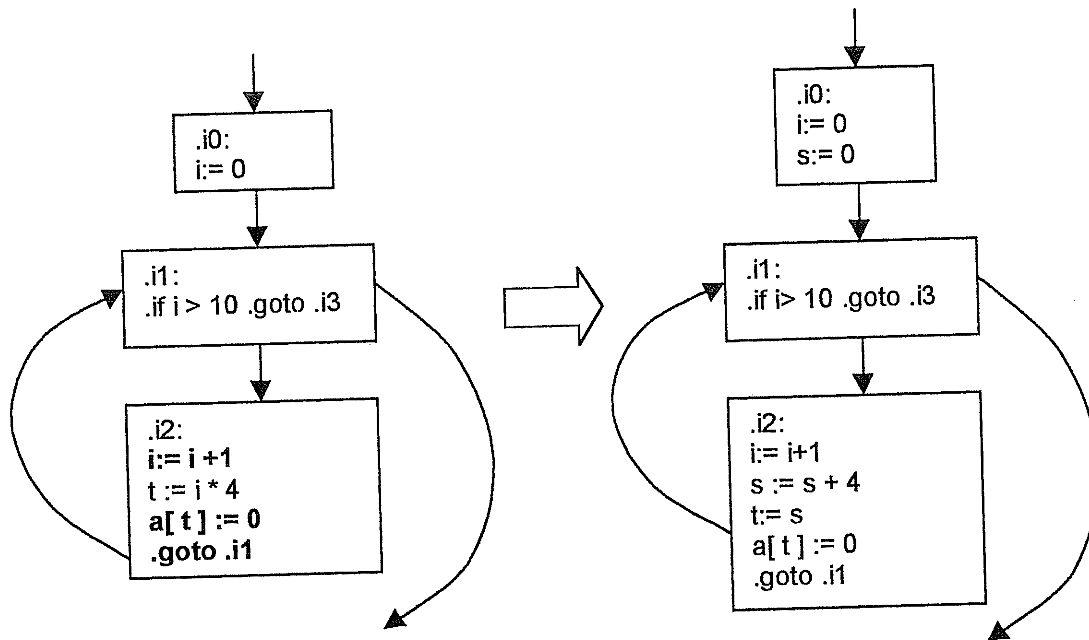
- Crear la nova variable S que guardarà el valor de la variable V. En cas que dos o més variables derivades comparteixin la mateixa terna, poden compartir la mateixa variable
- Substituir la instrucció que calcula V en funció de W per v:= s.
- Inicialitzar el valor de la variable S al final del preencapçalament amb

  s := a * w
  s := s + b
- Cada cop que s'assigna un valor a la variable W s'han de d'afegir instruccions per a calcular el nou valor de s:



La variable afegida, s, serà una nova variable d'inducció amb la mateixa terna que v.
La instrucció t1 := a*n no està afegint una multiplicació al bucle, perquè els dos operands de la multiplicació són invariants i per tant la instrucció (o potser només l'avaluació) serà extreta mitjançant **l'extracció d'expressions invariants**.

Un resultat d'aplicar aquesta optimització es pot veure a l'exemple 3.28. En aquest exemple, només hi ha una variable d'inducció (t), que té una terna (i,4,0). La nova variable que creem serà s, i l'haurem d'inicialitzar al preencapçalament i modificar-la cada cop que modifiquem la variable i. Finalment, només falta modificar la instrucció que defineix t (t:= 4*i) per a canviar-la t:=s.



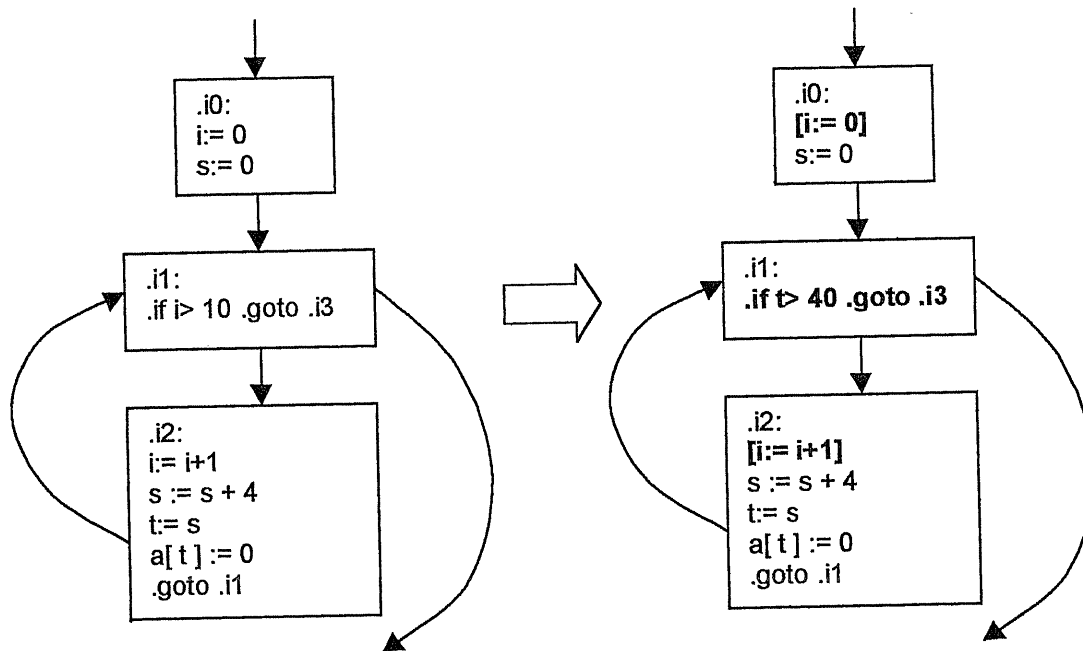
Exemple 3.28. Reducció d'intensitat de variables d'inducció

Després de totes les modificacions realitzades al codi de l'exemple no sembla que hi hagi hagut cap millora en el codi, però això és perquè es poden aplicar altres modificacions per a millorar-lo: propagació de còpies per eliminar t:=s, i eliminació de variables d'inducció per a eliminar la variable i.

3.8.9.5 - Eliminació de variables d'inducció

En algunes ocasions, les variables d'inducció d'un bucle **només s'utilitzen per a calcular variables derivades i per a comprovar les condicions d'acabament del bucle**. La reducció d'intensitat de variables d'inducció ha fet que no s'utilitzin les variables d'inducció per a calcular altres variables derivades, i per tant pot ser que les variables d'inducció només s'utilitzin per a comprovar les condicions d'acabament. En aquest cas, si podem reescriure la condició d'acabament del bucle de manera que fagi referència a una variable derivada en comptes de referir-se a la variable inútil, podrem eliminar les instruccions que modifiquen el valor de la variable bàsica dins el bucle. Aquest procés es coneix com **eliminació de variables d'inducció**.

Podem examinar el cas de l'exemple 3.29: la variable bàsica i només s'utilitza en la inicialització, en l'increment en cada iteració del bucle i en la comprovació d'acabament. De tota manera, sabem que la variable s té, en cada moment del bucle, un valor igual a 4*i. Això ens permet reescriure la condició d'acabament, fent que les instruccions entre corxets puguin ser eliminades.



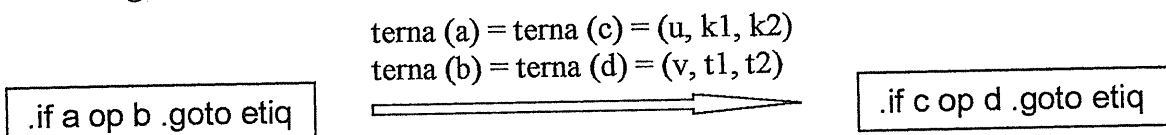
Exemple 3.29. Eliminació de variables d'inducció

Diem que una variable d'inducció és **inservible** en un bucle si **només s'utilitza en una definició de sí mateixa**. Podem eliminar totes les instruccions que defineixen aquesta variable en el bucle si el valor d'aquesta variable no és utilitzat fora del bucle.

Una variable d'inducció és **gairebé inservible**. si només **s'utilitza en definicions de sí mateixa i comparacions amb valors invariants**. Si es poden reescriure les comparacions per a que usin altres variables d'inducció la variable esdevé inservible i pot ser eliminada si no s'usa el valor d'aquesta variable fora del bucle.

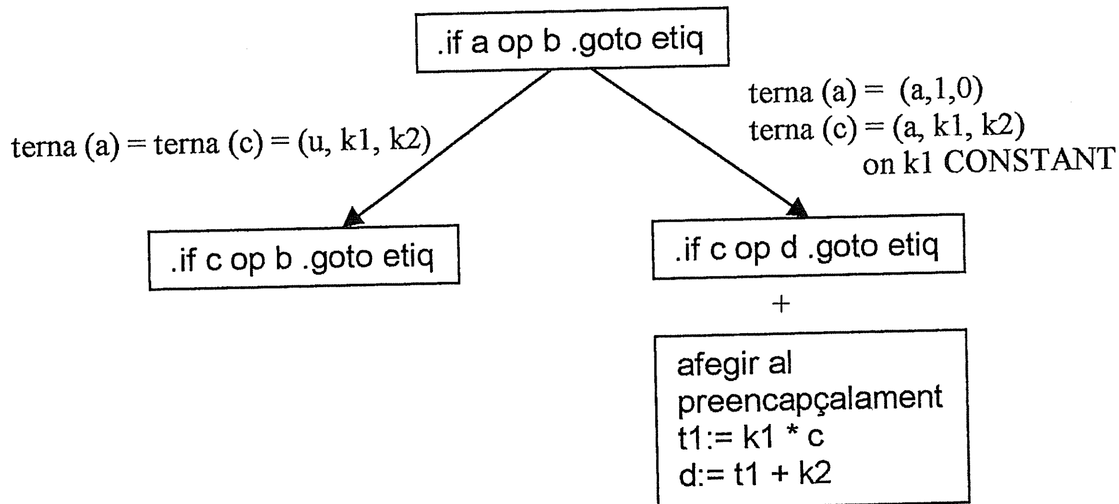
Per a poder reescriure una comparació .if a op b .goto etiq , on considerem que A és la variable d'inducció, hem de considerar les següents coses:

- Hi ha dos casos de comparacions ben diferenciats: comparar A **amb un operand invariant** o bé comparar-la **amb una altra variable d'inducció**.
- En el cas de comparar A amb una altra variable d'inducció, el procediment és el següent: busquem un parell de variables C i D (que no siguin inservibles ni gairebé inservibles) de manera que C té la mateixa terna que A i D té la mateixa terna que B. El canvi que cal fer serà el següent:



  Si no trobem un parell de variables que tinguin les ternes adequades, no reescriurem la comparació (veure comentaris al final d'aquest apartat).
- En el cas que B sigui un operand invariant forçarem que B sigui o bé una constant o bé estigui definit fora del bucle per a limitar el número de casos possibles. El que hem de fer és buscar una variable C que tingui la mateixa variable bàsica que A i que no sigui inútil ni gairebé inútil:

- Si C té la mateixa terna que A, llavors només hem de substituir la variable A per C en la comparació.
- Si C no té la mateixa terna que A, considerarem només els casos en que A sigui una variable bàsica i C sigui una variable derivada d'aquesta variable bàsica. En aquest cas, si c = k1*a + k2 substituirem l'operand B per d = k1*b + k2.



També hem de forçar que k1 ha de ser una constant positiva. En cas que no fos així, s'hauria de canviar l'operador de comparació: $a < b \rightarrow -a \geq -b$. Per a saber si hem de fer aquest canvi, k1 en el triple ha de ser una constant.

Un comentari que cal fer sobre la reescriptura de comparacions és que **hi ha molts casos possibles**. A més, el fet de **permetre ternes que continguin expressions invariants** en comptes de limitar a expressions constants **dificulta la reescriptura de comparacions**, que en determinats casos necessiten comprovacions del tipus "es pot fer la comparació si aquesta constant és divisible per aquesta altra". Aquesta comprovació es pot realitzar en temps de comprovació en el cas de les constants, però no es pot fer per als operands invariants. A [APP98] hi ha un procediment que pot reescriure més comparacions que les aquí descrites, sempre i quan els operands de les ternes i de la comparició siguin únicament constants (no es permeten operands invariants).

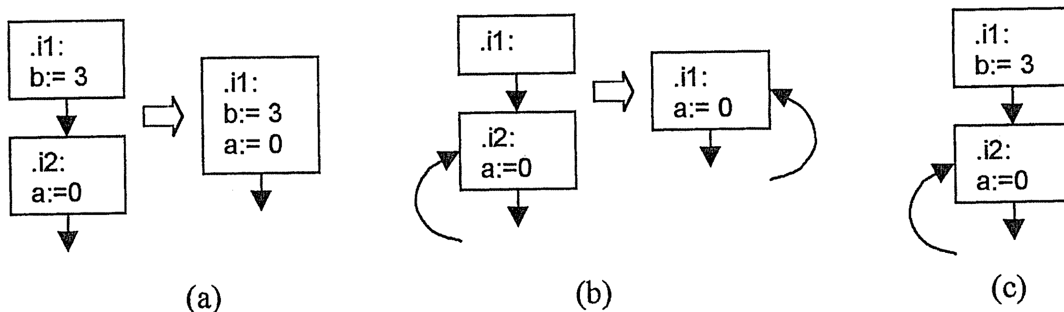
3.8.10 - Optimització del graf de fluxe de control

Moltes optimitzacions anteriors poden modificar el graf de fluxe de control (eliminació de salts, creació d'un preencapçalament, ...). Després de les modificacions, és possible que el graf de fluxe de control es pugui reescriure de manera que hi hagi menys blocs bàsics o menys salts.

Una primera optimització del graf de fluxe de control fa referència als blocs bàsics consecutius. Si tenim **dos blocs bàsics consecutius**, B_1 i B_2, **podrem fusionar (unir) les instruccions** dels dos blocs bàsics si es verifiquen totes les condicions:

- B_1 no acaba en un salt (condicional o incondicional) o instrucció de retorn (.return)
- o bé B_2 nomes té a B_1 com a predecessor o bé B_1 no conté instruccions

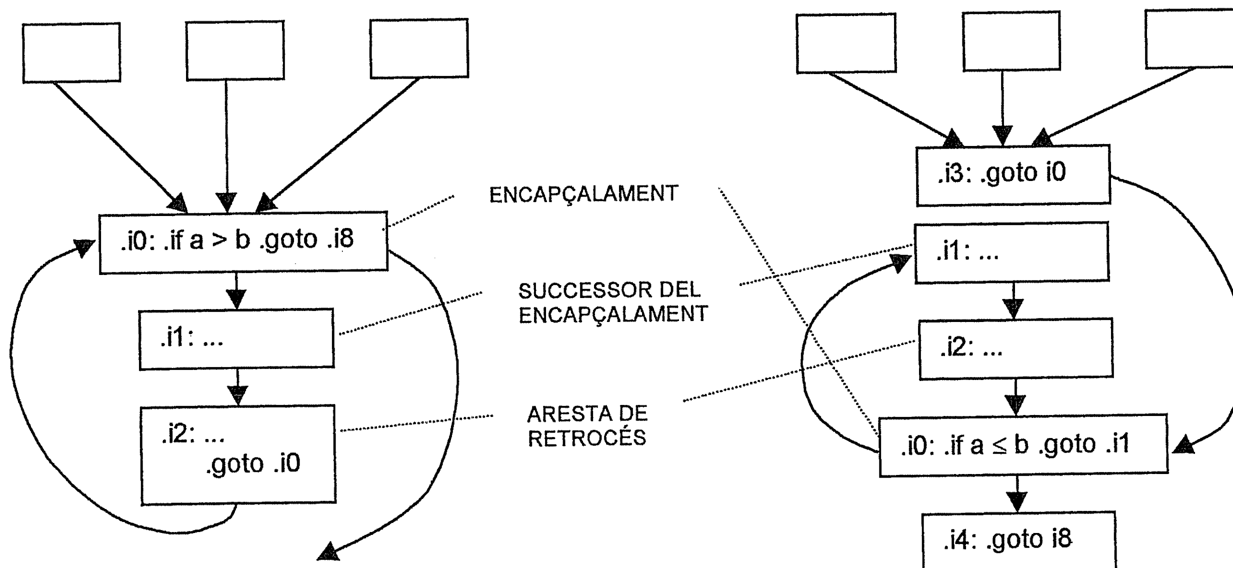
- si B_1 és el primer bloc bàsic del graf de fluxe de control, llavors B_2 només té a B_1 com a predecessor. Aquesta condició serveix per a seguir forçant que el bloc bàsic inicial del graf de fluxe de control no tingui cap predecessor.



Exemple 3.30 Fusió de blocs bàsics consecutius

En l'exemple 3.30 podem veure dos casos senzills de fusió de blocs bàsics consecutius. En l'exemple (a), el primer bloc bàsic (i1) no acaba en cap instrucció de salt o retorn i el segon bloc bàsic (i2) només té a i1 com a predecessor. Es segueixen totes les condicions i per tant podem procedir a la fusió. En el cas (b), el primer bloc bàsic no té instruccions i el segon té més d'un predecessor; podem fer la fusió si i1 no és el primer bloc bàsic del graf de fluxe de control (recordem que volem que el primer bloc bàsic no tingui cap predecessor). El tercer cas (c) és una situació en que no podem fusionar blocs bàsics consecutius: i2 té diversos predecessors i i1 té instruccions.

L'objectiu d'aquesta primera optimització és **reduir el nombre de blocs bàsics** del graf de fluxe de control **de cara a facilitar l'anàlisi del codi** (menys blocs bàsics significa un cost menor). La segona optimització del graf de fluxe de control té un objectiu diferent: **reescriure els bucles de tipus** while...do de manera que siguin bucles de tipus do...while **per a reduir el número de salts que s'executen a cada iteració** del bucle. A més d'això els bucles do...while permeten una millor optimització de bucles (veure 3.8.9.2).



Exemple 3.31 Transformació de while..do a do...while

En un bucle while...do, l'encapçalament del bucle és també una sortida del bucle. Per a poder realitzar una optimització, s'han de complir les condicions següents:

- l'encapçalament del bucle ha de ser una sortida del bucle
- l'encapçalament no ha de tenir com a predecessor cap node del bucle
- alguna aresta de retrocés ha de ser un salt incondicional

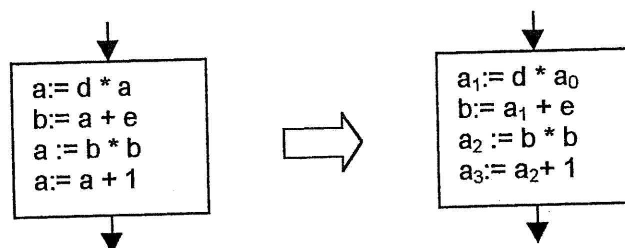
Després de realitzar aquestes comprovacions, podem realitzar la transformació de bucle while...do a do...while. La transformació consisteix en moure l'encapçalament després de l'aresta de retrocés i2, i canviar la condició de salt (abans saltàvem per sortir del bucle, ara saltem per a fer una nova iteració. D'aquesta manera es pot eliminar el salt incondicional del bloc i2 (.goto .i0, i **només es produeix un salt per iteració** (.if a ≤ b .goto .i1), **a diferència dels dos salts per iteració anteriors** (.if a > b .goto .i8 i .goto .i0).

En la transformació també cal afegir dos blocs bàsics (i3 i i4) per a controlar l'entrada i la sortida del bucle. Aquestes instruccions de salt fan que el programa sigui més lent si el bucle no s'executa (encara que el benifici obtingut en cada iteració del bucle compensa aquesta possibilitat), tot i que és possible que els salts incondicionals puguin ser eliminats per l'optimització de salts.

3.8.11 - Única assignació estàtica (SSA)

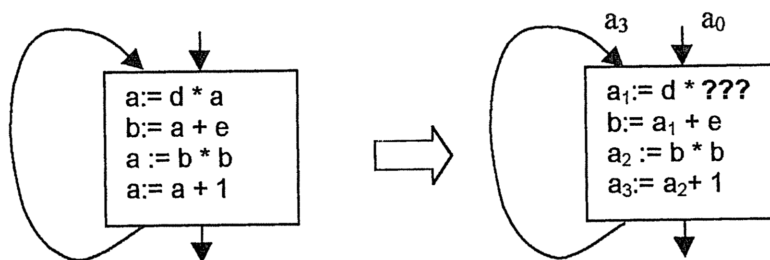
En l'anàlisi del fluxe de les dades s'utilitzaven les cadenes d'ús i definició. Els algorismes que utilitzen aquestes cadenes es poden reescriure de manera molt més eficient si garantim que les cadenes de definició només contenen un element.

La transformació d'un programa a la **forma d'única assignació estàtica** (Static Single Assignment form o SSA) pretèn **reescriure el codi** del programa de manera que **cada variable escalar només sigui definida en una instrucció** del programa. Si una variable té més d'una definició en el programa, s'assigna un **número de versió** a cada definició; dues versions d'una mateixa variable es poden considerar dues variables diferents.



Exemple 3.32 Transformació d'un bloc bàsic a forma SSA

Com es veu a l'exemple 3.32, assignar noves versiona a cada definició d'una varaible és un procés trivial. En canvi, **decidir quina versió d'una variable s'ha d'usar** en un punt del programa **és complicat**, ja que pot ser que diferents versions de la mateixa variable estiguin actives (veure exemple 3.33).



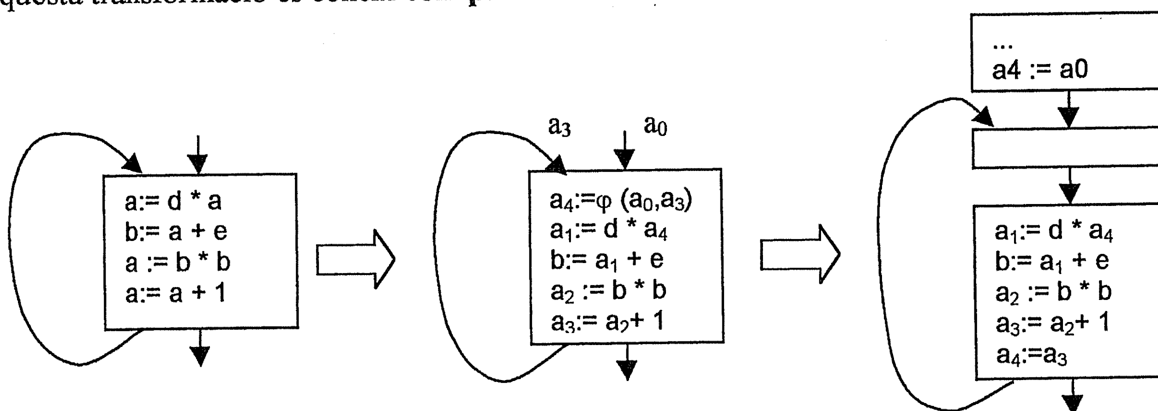
Exemple 3.33 Un conflicte de versions en SSA

La solució a aquest problema de versions és la **creació d'un tipus d'instrucció fictícia,** anomenada **funció φ** o senzillament φ. Una instrucció $a_i := \varphi(a_1, a_2, \ldots, a_n)$ en un bloc bàsic b amb predecessors $p_1, p_2, \ldots, p_n$ **assigna** a la variable a_i **un valor diferent en funció del predecessor de b que haguem executat abans d'executar b**:

- $a_i := \varphi(a_1, a_2, \ldots, a_n) \Leftrightarrow a_i := a_1$ si abans d'executar b s'ha executat p_1.
- $a_i := \varphi(a_1, a_2, \ldots, a_n) \Leftrightarrow a_i := a_2$ si abans d'executar b s'ha executat p_2.

  ...
- $a_i := \varphi(a_1, a_2, \ldots, a_n) \Leftrightarrow a_i := a_n$ si abans d'executar b s'ha executat p_n.

Si es vol expressar les instrucions φ a partir de les instruccions habituals, es pot fer a partir d'instruccions de còpia a partir del seu significat. Per a evitar que s'executin instruccions de còpia inútils, es força que **no hi hagi cap aresta entre un bloc bàsic amb més d'un successor a un bloc bàsic amb més d'un predecessor**. Si un bloc no verifica aquesta propietat, creem un node predecessor al qual arriben tots els predecessors d'aquest node. Aquesta transformació es coneix com **partició d'arestes**.



Exemple 3.34 Ús de les instruccions φ i el seu significat

A l'exemple 3.34 podem veure com s'utilitzen les instruccions φ per a resoldre el problema anterior amb les versions (exemple 3.33). A més, es pot veure com traduir φ a les instruccions de còpia equivalents, forçant la propietat d'un únic predecessor o successor.

Els avantatges de traduir un programa a la forma SSA són:

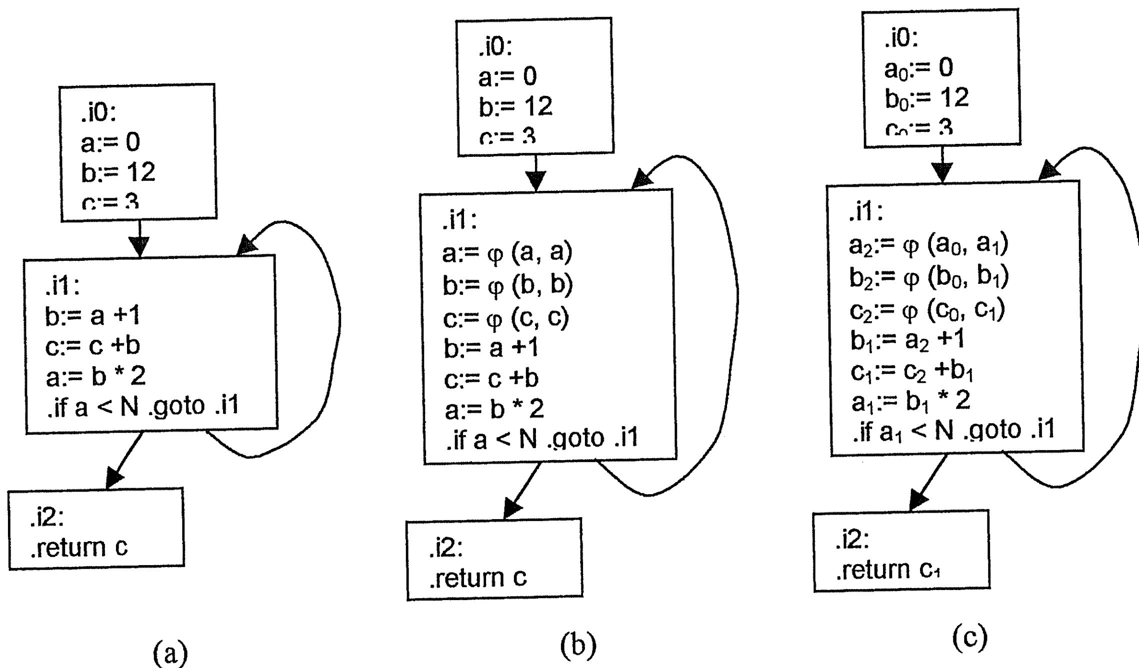
- Totes les **optimitzacions** realitzades a partir de l'anàlisi del fluxe de dades (propagació de còpies i constants, eliminació de codi mort, ...) **es poden implementar de forma més eficient en SSA** (veure els algorismes a [APP98]).
- **L'espai** utilitzat **per** a guardar **les cadenes d'ús i definició es redueix** considerablement (una única definició per a cada variable)

- Quan assignem versions a una variable, estem creant noves variables que estan vives en fragments del programa més petits que la variable original . Això **facilita l'assignació de registres**.
- A part dels esmentats aquí, hi ha altres algorismes i usos de la forma SSA. Per exemple [CHOI96] descriu un algorisme per a calcular la SSA de manera incremental és a dir, com garantir que un programa continúa estant en forma SSA després d'una transformació donada; [CHOW96] conté una extensió de la SSA per a incloure en l'anàlisi de la SSA les referències a memòria.

Les optimitzacions del compilador no estan implementades utilitzant SSA i per tant només s'utilitza SSA per a intentar reduir el rang de vida de les variables i facilitar així l'assignació de registres.

El procés de transformació a SSA té dues fases: traducció del programa a SSA (**afegint instruccions φ i assignant versions a les variables**) i traducció de SSA al conjunt d'instruccions inicial (**eliminant les instruccions** φ). La segona fase, l'eliminació de les instruccions φ, és molt senzilla i com ja s'ha vist es realitza traduint les φ per instruccions de còpia. En canvi, la primera fase presenta un problema important: on s'han d'afegir instruccions φ.

Les instruccions φ s'insereixen al principi d'alguns blocs bàsics però no en tots. Només s'han d'inserir instruccions φ en aquells blocs bàsics als que puguin arribar diverses versions d'una mateixa variable: és a dir, on no hi ha cap definició de la variable que domini el bloc bàsic. Per a determinar aquests blocs bàsics utilitzarem el concepte de **frontera de dominació**, definit a 3.6.6.



Exemple 3.35 Transformació d'un programa a SSA
(a) programa original
(b) després d'inserir les instruccions φ
(c) després d'assignar versions

A 3.35 es pot veure un exemple d'aplicació d'aquest algorisme: el primer pas afegeix instruccions φ només allà on són necessàries (en aquest cas només hi ha un problema de versions al bloc i1). Posteriorment s'assigna una nova versió a cada definició de la variable i es modifica cada ús de la variable de manera que utilitza la darrera versió disponible de la variable.

3.9 - Conclusions

Hem vist que una optimització és una transformació del codi intermedi que millora la seva eficiència o temps de càlcul. És molt important destacar que **només es millora la qualitat del codi**: el cost de trobar un codi òptim és prohibitiu en la majoria dels casos.

Les situacions a optimitzar són molt variades: expressions comunes, salts, bucles, instruccions de còpia, ... En alguns programes les optimitzacions seran més efectives que en altres, i un tipus d'optimització pot ser més eficient que altres. El capítol 8 conté algunes proves de les optimitzacions que mostren la seva eficiència.

A més de les optimitzacions descrites en aquest capítol, n'existeixen moltes altres. El capítol 6 descriu algunes d'aquestes optimitzacions. Cal tenir en compte que sempre es poden inventar noves optimitzacions per aplicar a un programa. Només cal que aquesta optimització compleixi els requisits de tota optimització (veure 3.9):

- **caràcter conservador**: es modificarà el programa només si s'està completament segus que no canviarà el seu comportament.
- **garantia de millora en el codi generat:** no es demana optimalitat!
- **temps de càlcul raonable**

El fet de **disposar d'una fase d'optimització permet abstreure els detalls d'eficiència d'altres mòduls del compilador,** i per tant, fer-los més senzills. Això vol dir, per exemple, que quan generem codi intermedi no cal complicar aquesta generació per a produir un codi intermedi eficient: podem utilitzar un procés de generació de codi intermedi senzill, tenint la seguretat que l'optimització posterior en millorarà la qualitat.

L'optimització de codi també té els seus inconvenients. Resulta evident que **l'optimització augmenta considerablement el temps de compilació, perquè és un procés costós**. Per aquest motiu la majoria de compiladors treballen normalment sense optimitzacions (per a generar codi de forma ràpida en les primeres versions d'un programa) a menys que l'usuari ho indiqui a través de flags (generar un codi eficient en la versió definitiva del programa). Per tant, **és important que l'usuari d'un compilador sàpiga seleccionar el nivell d'optimització que necessiten els seus programes**.

4 Assignació de registres

Abans de poder generar codi màquina a partir del codi intermedi, cal decidir on s'emmagatzemaran les variables del codi intermedi. Una variable podrà estar guardada en memòria o en un registre dins el processador, i triar una o altra opció pot afectar molt el temps d'execució del programa.

Aquest capítol aprofondeix en aquest procés de decisió sobre les variables, anomenat ***assignació de registres****. Per una banda es descriuen els objectius que volem aconseguir aquesta assignació, les diferents maneres d'aconseguir-los i l'algorisme escollit. També es presenten les característiques de l'arquitectura destí (MIPS) necessàries per a prendre aquesta decisió.*

4.1 - Objectius de l'assignació de registres

L'assignació de registres és la primera fase dependent de l'arquitectura en el compilador final. Aquesta fase parteix d'un programa, amb un conjunt de variables, que s'ha d'executar sobre una arquitectura concreta amb un conjunt limitat de registres, i intenta **decidir una bona manera de situar les variables en els registres**. El problema de l'assignació de registres és NP-complet, és a dir, l'algorisme que calcula l'assignació òptima té en el cas pitjor un cost exponencial. Per tant no intentarem buscar l'assignació de registres òptima, només una bona assignació. **La qualitat de l'assignació de registres té un impacte molt important en l'eficiència del codi màquina generat**.

En temps d'execució, els operands han d'estar o bé en un registre del processador o bé en una posició de memòria. En algunes arquitectures (anomenades "load-store"), tots els operands han d'estar en registre, i per a utilitzar un valor de memòria cal llegir-lo, operar-lo i tornar-lo a guardar. Sigui quin sigui el tipus d'arquitectura, **operar amb un registre del processador és molt més eficient que fer-ho amb un operand en memòria**, però no sempre és possible tenir totes les variables en registre. Algunes situacions ens poden portar a situar una variable a memòria:

- Una variable ha d'estar a memòria obligatòriament en el moment en que alguna instrucció agafa la seva adreça (var2:= &var1).

- Si no disposem de prou registres per a guardar totes les variables a memòria, les variables restants hauran d'estar a memòria.

Una **assignació de registres** és una funció

assig: variables x instrucció→ {reg_1,... reg_n, memòria}

que donada una variable del nostre programa i un punt del programa (instrucció), o bé ens dirà que la variable s'ha de situar en memòria o bé ens dirà que ha d'anar a un registre i en aquest cas ens dirà quin. **Si dues variables no contenen un valor útil al mateix temps en cap punt del programa es poden assignar al mateix registre**. Dues variables que estan vives en un mateix punt del programa es diu que **interfereixen**, i no poden tenir el mateix registre assignat.

Els **objectius** d'una assignació de registres seran:

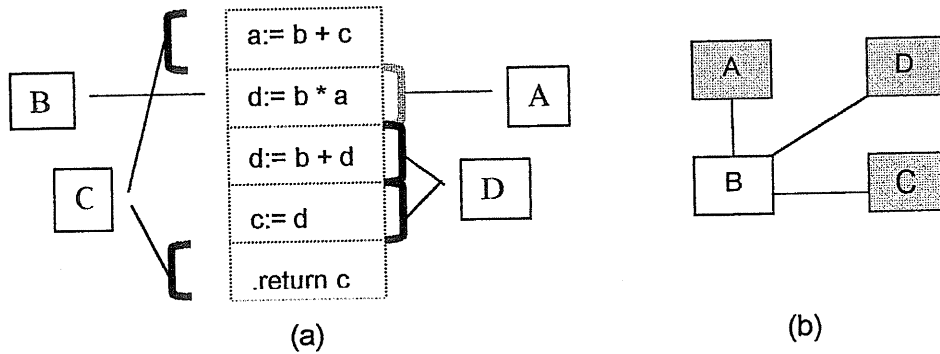
- **Minimitzar el número d'accessos a memòria per a llegir les variables**. Això implica dues coses: que el màxim número possible de variables estaran guardades en registre, i que es triaran els tipus de registres més adequats per a cada variable (veure apartat 4.2).
- **Eliminar tantes instruccions de còpia com sigui possible**. Una instrucció a:= b es pot eliminar si les variables a i b estan assignades al mateix registre, però no sempre es podrà assignar el mateix registre a les dues variables.
- **El primer objectiu té prioritat sobre el segon**: és molt pitjor tenir una variable a memòria que haver d'executar una instrucció de còpia. Cal recordar que el temps d'un accés a memòria pot ser equivalent a executar desenes d'instruccions.

4.2 - Estratègies per a l'assignació de registres

Si el codi intermedi és un arbre sintàctic, l'assignació de registres és més senzilla. L'algorisme Sethi-Ullman [AHO90] proporciona l'ordre d'avaluació i l'assignació de registres que minimitza el número de registres en una expressió. En canvi, si el codi intermedi és un codi de tres adreces, l'assignació de registres és més complexa.

Una estratègia que utilitzen alguns compiladors senzills per a generar codi màquina és fer-ho a partir dels **blocs bàsics** [AHO90]: s'estudia quina assignació de registres és més convenient dins un bloc bàsic i després del bloc bàsic es guarden els valors calculats en memòria. Això pot provocar un nombre excessiu d'accessos a memòria, i per a resoldre aquest problema s'acostuma a utilitzar una **assignació de registres global**, que utilitza informació sobre tot el programa.

Una tècnica coneguda per a fer assignació de registres global és la **coloració de grafs**. A partir d'un programa es construeix un **graf d'interferència** usant **l'anàlisi de vida**. Aquest graf té com a nodes dos variables del programa, i existeixen dues arestes entre un parell de variables si aquestes interfereixen (és a dir, estan vives simultàneament en el mateix punt del programa).



Exemple 4.1 Assignació de registres per coloració de grafs
(a) programa amb anàlisi de vida
(b) graf d'interferència associat al programa

A l'exemple 4.1 podem veure un programa senzill a partir del qual s'ha construït un graf d'interferència. La variable B interfereix amb totes les altres variables del programa, i per això hi ha una aresta entre la variable B i la resta de variables del programa.

El problema de d'assignar el mínim número de registes a les variables és equivalent al problema de colorejar el graf d'interferència amb el mínim número de colors. En el problèma de la colorabilitat, podem assignar el mateix color a dos nodes del graf sempre i quan no hi hagi una aresta entre ells. En l'exemple 4.1, necessitem dos colors per a colorejar el graf i per tant necessitem 2 registres per a executar les instruccions: un per a la variable b i un altre per a les variables a,c,d. Aquesta coloració, a més, permet eliminar la instrucció c:= d, donat que les variables c i d comparteixen el mateix registre.

El problema de colorejar un graf d'interferència també és NP-complet, és a dir, té un cost exponencial en el pitjor dels casos. Per això buscarem un algorisme per a colorejar que tingui un cost polinòmic, encara que no proporcioni la coloració òptima.

4.3 - Els registres en l'arquitectura destí: MIPS

Abans de continuar, cal conèixer més detalls sobre l'arquitectura destí: és necessari recordar que la fase d'assignació de registres és la primera fase del compilador final que depèn de l'arquitectura.

L'arquitectura destí és el **processador MIPS R2000**. Des del punt de vista dels registres, aquest processador és **una arquitectura "load-store"**: només pot accedir a memòria a través d'operacions de "load" (llegir memòria) i "store" (escriure memòria). La resta d'operacions (aritmètiques, salts condicionals,...) han de tenir tots els seus operands a memòria. Com és habitual en les màquines "load-store", l'arquitectura MIPS té un gran nombre de registres: 32 de propòsit general i 32 per a operacions de punt flotant.

Pel que fa als registres de propòsit general, MIPS té una **convenció en l'ús de registres** [LAR98], que es seguida per tots els compiladors en les màquines MIPS i que garanteix que rutines compilades per un compilador puguin ser cridades des de rutines d'un altre compilador.

Nom del registre	Ús	Nom del registre	Ús
$zero	constant 0	$at	reservat
$v0-$v1	avaluació d'expressions i retorn de funció	$k0-$k1	reservats
		$gp	àrea global
$a0-$a3	paràmetres de funció	$sp	punter a la pila
$t0-$t9	registre temporal ("callee-save")	$fp	punter al bloc d'activació anterior
$s0-$s6	registre salvat ("caller-save")	$ra	adreça de retorn

Els registres que ens interessen de cara a l'assignació de registres són els següents:

- $a0, $a1, $a2 i $a3 **guarden els quatre primers paràmetres de la funció**. La resta de paràmetres estaran emmagatzemats a memòria. Si en alguna instrucció de la funció agafem l'adreça (&) d'algun dels quatre primers paràmetres, l'haurem de posar a memòria.
- $t0, $t1, $t2, $t3, $t4, $t5, $t6, $t7, $t8 i $t9 són **registres temporals**. Aquests registres guarden valors de variables locals a la funció, i **no es mantenen després d'una crida**: si volem que el valor d'un registre temporal no canvii després d'una crida l'hem de guardar a memòria abans de la crida i recuperar-lo després d'una crida.
- $s0, $s1, $s2, $s3, $s4, $s5 i $s6 són **registres salvats**. Aquests registres guarden valors de variables locals a la funció que es mantenen després d'una crida. De tota manera, si una funció vol utilitzar un d'aquests registres ha de salvar el seu contingut a memòria al principi de la funció i recuperar-lo de memòria al final.

Per una banda, **si una variable està viva durant una crida a funció, ens interessa guardar-la en un registre salvat**, perquè així no caldrà salvar-la i restaurar-la a cada crida: només l'haurem de salvar un cop, al principi i al final de la funció. Per altra banda, **si una variable no està viva durant cap crida a funció, ens interessa que estigui en un registre temporal**, perquè així no l'haurem de salvar.

(a)
```
.function PROC1
  .scalar a, b, c
  ....
  c := PROC2 (b)
  ...
.end PROC1
```

(b)
```
.function PROC1
  .scalar a, b, c
  salvar registre reg $s0
  ....
  c := PROC2 (b)
  ...
  restaurar registre $s0
.end PROC1
```

(c)
```
.function PROC1
  .scalar a, b, c, d
  ....
  salvar registre $t0
  c := PROC2 (b)
  restaurar registre $t0
  ...
.end PROC1
```

Exemple 4.2 Registres temporals i salvats
(a) Programa IC
(b) Variable a assignada al registre salvat $s0
(c) Variable a assignada al registre temporal $t0

L'exemple 4.2 ens mostra com s'haurien de salvar els registres en el cas de guardar una variable en un registre salvat o en un registre temporal. Aquestes instruccions per a salvar el registre no apareixen el llenguatge IC (només apareixen en el codi màquina), i només es mostren per a diferenciar els dos tipus de registres.

4.4 - Algorisme d'assignació de registres utilitzat

L'algorisme utilitzat per a l'assignació de registres realitza una **assignació de registres global basada en la coloració de grafs, utilitzant estratègies per a eliminar instruccions de còpia** [APP98]. **Cada variable en registre estarà en el mateix registre en tota la funció.**

Els registres de l'arquitectura destí han estat parametritzats: l'algorisme de coloració treballa amb una arquitectura on els P primers paràmetres es passen per registre, hi ha T registres temporals i hi ha S registres salvats. Canviant els valors de P, T i S podem adaptar l'algorisme d'assignació de registres a una altra arquitectura; per exemple, si no es passa cap paràmetre per registre podem fer P = 0. Els valors d'aquests paràmetres per a l'arquitectura MIPS (segons s'ha vist a l'apartat 4.3) són: P = 4, T = 10, S = 7.

Aquesta parametrització no és prou general per a detectar situacions més complexes: instruccions que sempre consulten el seu valor d'un registre concret, instruccions que sempre guarden el seu valor en un altre registre concret, ... Aquestes situacions no es produeixen en l'arquitectura MIPS, però poden produir-se en altres arquitectures, com ara els Intelx86. Per a poder parametritzar aquestes situacions, caldria disposar d'un model complet de l'arquitectura destí que inclogui registres i el conjunt d'instruccions. Un exemple d'aquesta tècnica es pot trobar a [GCC].

No totes les variables del nostre programa poden ser assignades a registre. S'ha prés la decisió que les següents variables estiguin assignades a memòria incondicionalment:

- totes les variables globals (.gscalar, .gstruct i .gstring)
- totes les variables estructurades (.struct)
- totes les variables escalars (.scalar) i paràmetres a les que agafem l'adreça en una instrucció addr := &var
- tots els paràmetres per sobre del quart

Els quatre primers paràmetres de la funció ja tenen un registre assignat. Per tant, les úniques variables a les que podem assignar registres seran els escalars als que no agafem l'adreça. Els **criteris a seguir** per assignar registre a una variable V són:

- sempre que sigui possible, assignar V a algun registre
- no assignar a V el mateix registre que ja tingui una variable que interfereixi amb V
- si V està viva durant alguna crida, intentar assignar-la als registres $s0-$s6
- si no està viva durant cap crida, intentar assignar-la als registres $a0-$a3 o $t0-$t3
- si una variable W no interfereix amb V i hi ha una instrucció de còpia entre V i W, intentar assignar el mateix registre a V i W

Per a realitzar l'assignació de registres, hem de realitzar una sèrie de passos previs. En primer lloc, cal **construir el graf d'interferència** a partir de l'anàlisi de vida del programa. Després hem de tenir una sèrie **d'heurístiques** que ens indiquin quina variable assignar a memòria en cas que hi hagi conflictes entre diverses variables i quines instrucció de còpia són les primeres que hem d'intentar eliminar. Després d'aquests càlculs previs, es realitzarà la coloració del graf d'interferència obtenint l'assignació de registres.

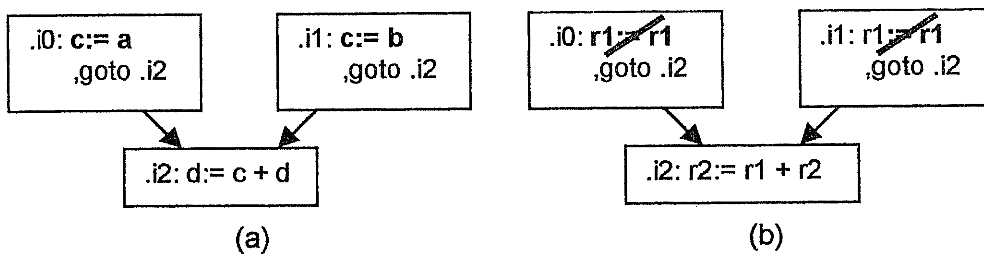
4.4.1 - Construcció del graf d'interferència

El graf d'interferència es construeix a partir de l'anàlisi de vida del programa. Com a primera aproximació, cada **node** del graf representa una **variable** del programa i cada **aresta** representa una restricció del tipus *"aquestes dues **variables** estan **vives simultàneament** en algun punt del programa"*. Si dues variables (nodes) tenen una resta entre elles en el graf, es diu que **interfereixen**.

A aquesta primera aproximació cal afegir-li tres matitzacions:

- **el tractament especial de les instruccions de còpia**

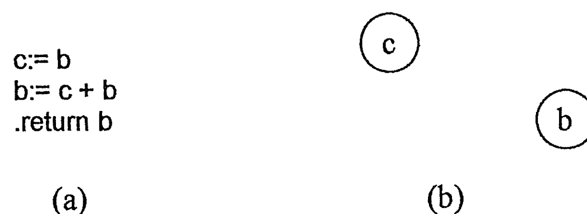
La propagació de còpies és una optimització que permetia eliminar instruccions de còpia del programa. De tota manera, existeixen instruccions que no poden ser eliminades per la propagació de còpies, però que sí es poden eliminar assignant el mateix registre a les variables implicades.



Exemple 4.3 Una situació en que la propagació de còpies no és efectiva
(a) no es pot fer propagació de còpies
(b) resultat si fem una bona assignació de registres

L'exemple 4.3 mostra una situació en la que no es pot utilitzar la propagació de còpies: la variable c en la instrucció d:= c + d té dues definicions (les dues instruccions de còpia) i per tant cap de les dues còpies es pot propagar. Però si assignem el mateix registre r1 a les variables a, b, i c llavors les dues instruccions de còpia es poden eliminar (r1:=r1 no té cap efecte).

El procediment per a eliminar instruccions de còpia serà assignar a les dues variables involucrades el mateix color (registre) al colorejar el graf. Evidentment, si les dues variables involucrades interfereixen no els hi podrem assignar donar el mateix color. Per aquest motiu, si dues variables a i b interfereixen només en instruccións de còpia a:=b, no afegirem cap arc entre els nodes a i b al graf d'interferència.



Exemple 4.4 Les instruccions de còpia en el graf d'interferència
(a) programa original
(b) graf d'interferència

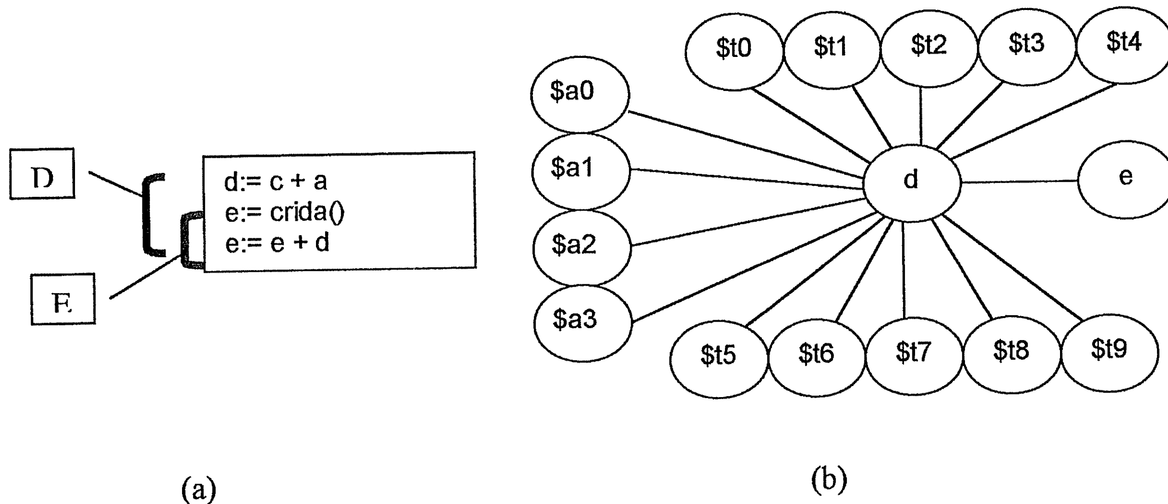
En l'exemple 4.4, les variables c i b estan vives després de c:= b i per tant segons la definició estricta hauríem d'afegir una aresta en el graf d'interferència entre c i b. No ho fem així per a permetre que c i b estiguin en el mateix registre i c:=b pugui ser eliminada.

- **els nodes precolorejats**

Algunes variables del programa tenen assignat un registre abans de començar la coloració (per exemple, els quatre primers paràmetres estan assignats als registres $a0-$a3). El que es fa és afegir al graf d'interferència un node per cada registre de l'arquitectura que s'usi de manera explícita o necessitem per expressar interferències (veure apartat següent). Es considera que **un node precolorejat interfereix amb tots els altres nodes precolorejats.** En el nostre cas, **hi ha un node precolorejat per a cada registre temporal**: $a0-$a3 i $t0-$t9.

- **la generalització del concepte d'interferència**

Anteriorment ja s'ha indicat la diferència entre registres temporals i registres salvats, i el fet que una variable que està viva durant una crida a funció s'hauria d'assignar a un registre salvat. La manera de representar això en un graf d'interferència amb nodes precolorejats és la següent: en cada variable que estigui viva durant una crida a funció afegim arestes a tots els registres temporals. Això vol dir que els crides a funció fan interferir totes les variables vives amb els registres temporals, i per tant les variables vives durant una crida no podran ser assignades a un registre temporal. S'ha près la decisió que si alguna d'aquestes variables no pot ser assignada a un registre salvat perquè no queden més registres, serà assignada a memòria.



Exemple 4.5 Les crides a funció en el graf d'interferència
(a) programa
(b) graf d'interferència associat

<u>Algorisme</u> construir graf d'interferència
<u>Entrada</u>
Graf de fluxe de control amb informació de variables vives al final de cada bloc (b.out): cfg
<u>Sortida</u>

```
          Graf d'interferència: igraf

     igraf := crear_graf_buit();
     inicialitzar (igraf, cfg);
     per a cada bloc bàsic b del graf cfg
          var_vives := copiar_llista (b.out);
          per a cada instrucció ins del bloc bàsic b en ordre invers
               afegir_arestes_instrucció (igraf, ins, var_vives);
          fper
     fper

inicialitzar (igraf)
     per a cada registre r temporal
          nou_node (igraf, r, r);
     fper
     per a cada  paràmetre p de la funció
          i := posició de p en la llista de paràmetres;
          si i ≤ P llavors (* P és el nombre de paràmetres passats per registre*)
               nou_node (igraf, p, color_iessim( i ) );
          sino
               assignar_a_memòria (p);
          fsi
     fper
     per a cada escalar c de la funció
          si agafem l'adreça de c llavors
               assignar_a_memòria(c);
          sino
               nou_node (igraf, c, SENSE_COLOR);
          fsi
     fper


afegir_arestes_instrucció (igraf, ins,var_vives)
     dst := variable definida per la instrucció;
     op1, op2, op3 := variables usades per la instrucció;
     si ins és una crida a funció llavors
          per a cada variable v de la llista var_vives
               per a cada registre r temporal
                    afegir_aresta (igraf, v, r);
               fper
          fper
     sino
          si ins defineix alguna variable llavors
               si ins és una instrucció de còpia llavors
                    còpia:= op1;
               sino
                    còpia .= NULL;
               fsi
               per a cada variable v de la llista var_vives
                    si v ≠ còpia llavors
```

```
                afegir_aresta (igraf, dst, v);
            fsi
        fper
    fsi
fsi
var_vives := (var_vives - dst) ∪ op1 ∪ op2 ∪ op3;
```

4.4.2 - Heurístiques per a l'assignació de registres

Durant el procés d'assignació de registres, hem de prendre dos tipus de decisions. En primer lloc, si no podem posar totes les variables en registre caldrà posar alguna variable a memòria, i **cal decidir quina variable s'assignarà a memòria**. A més, si podem eliminar instruccions de còpia **cal decidir quines instruccions de còpia s'eliminaran primer**.

El criteri per a escollir la variable que s'assignarà a memòria és **triar la variables menys utilitzada en el programa**. Per a calcular la utilització d'una variable tenim una heurística sobre el número de vegades que s'executarà una instrucció: si una instrucció no està dins un bucle, considerarem que s'executarà una vegada; si està dins un bucle, considerarem que s'executarà 10 vegades; si està dins de dos bucles imbricats, 100 vegades; si són tres bucles, 1000 vegades, etc.

usos (i, v) := # de vegades que la variable v és utilitzada a la instrucció i
k (i, cfg) := # de bucles en que es troba la instrucció i

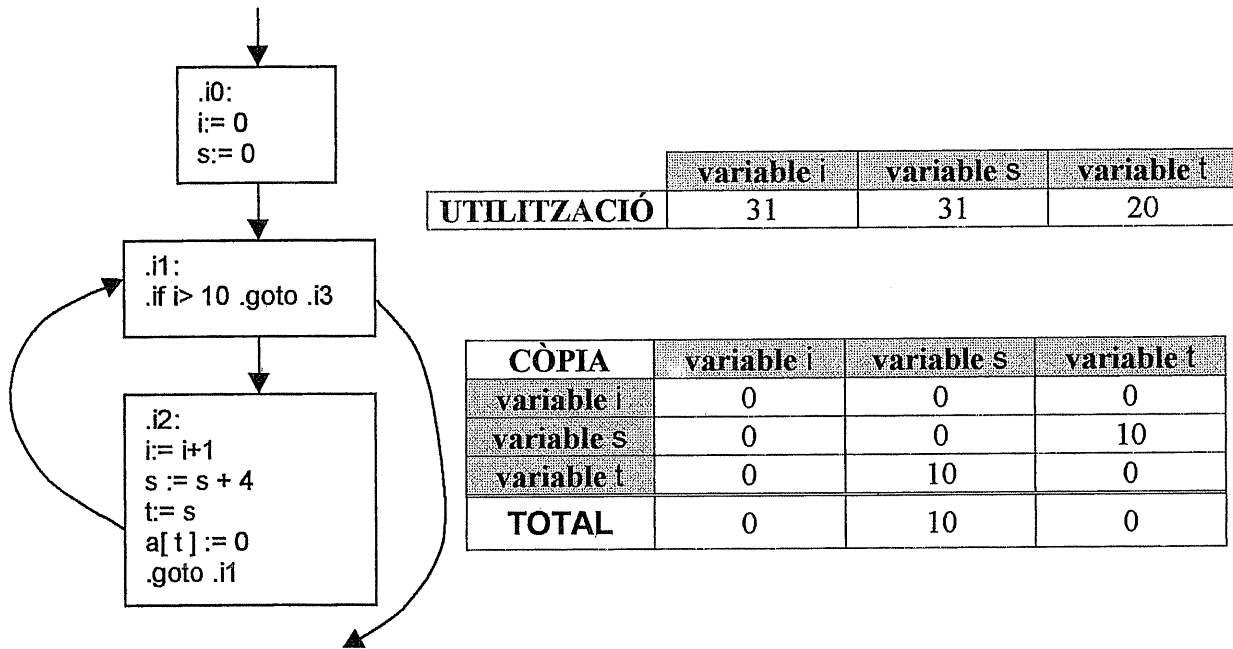
$$\textbf{utilització}\ (v, cfg) := \sum_{\forall i \in cfg} usos(i,v) * 10^{k(i,cfg)}$$

Es calcula la utilització per a totes les variables del programa, i si s'ha de triar una variable per a situar en memòria, s'escull la variable amb una utilització més petita. A [APP98] es descriu una altra heurística, que utilitza el número d'arestes que té una variable en el graf d'interferència.

Pel que fa a les instruccions de còpia fem servir la mateixa heurística: guardarem una heurística per a cada parella de variables, que indicarà **el número de vegades que s'executaran les instruccions de còpia en les que estan implicades les dues variables**. Quan hem d'eliminar instruccions de còpia escollirem les variables que tinguin instruccions de còpia que s'executaran més vegades.

còpia (i,v1,v2) := 1 si ins és (v1:=v2) o (v2:=v1) , 0 en cas contrari
k (i, cfg) := # de bucles en que es troba la instrucció i

$$\textbf{còpia}\ (v1,v2, cfg) := \sum_{\forall i \in cfg} còpia(i,v1,v2) * 10^{k(i,cfg)}$$



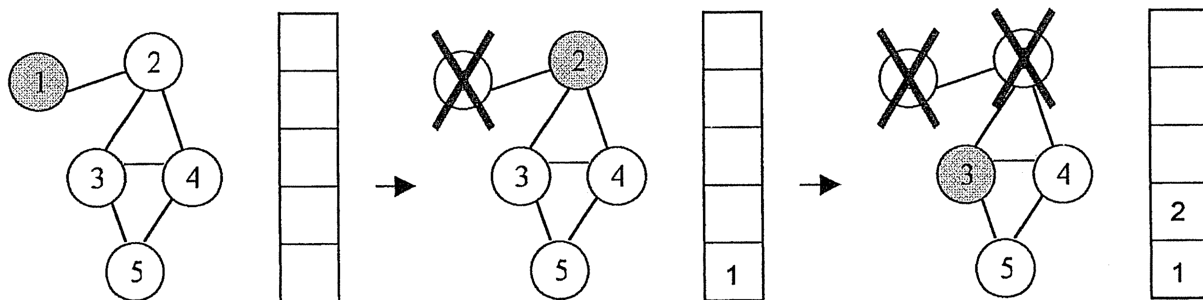
	variable i	variable s	variable t
UTILITZACIÓ	31	31	20

CÒPIA	variable i	variable s	variable t
variable i	0	0	0
variable s	0	0	10
variable t	0	10	0
TOTAL	0	10	0

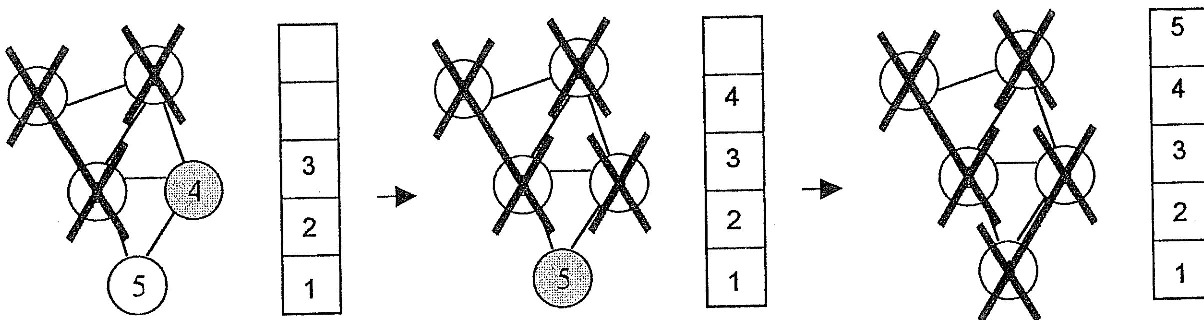
Exemple 4.6 Heurístiques en un programa

4.4.3 - Coloració del graf d'interferència

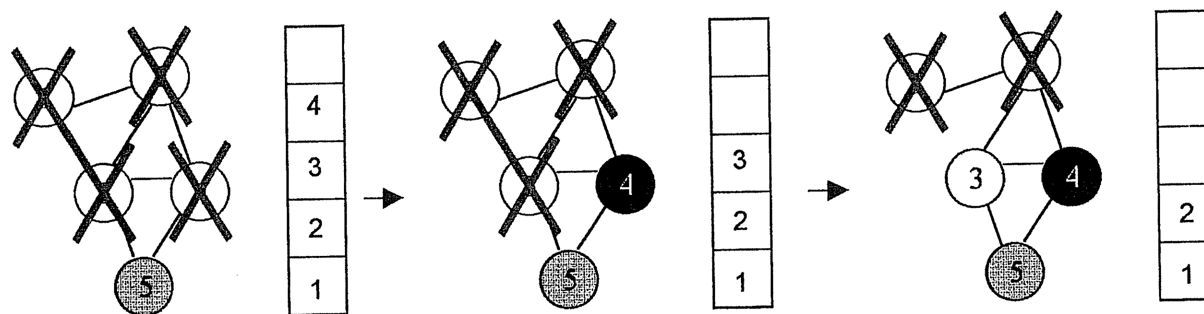
L'algorisme de coloració del graf d'interferència utilitzat és el de **coloració per simplificació**. El procediment per colorejar el graf és el següent: si tenim K colors (registres), anem escollint a cada pas un node del graf amb menys de K nodes adjacents, l'eliminem del graf (**simplifiquem**) i el posem en una pila. Quan s'han eliminat tots els nodes del graf, es van desempilant un a un i es reconstrueig el graf, assignant un color a cada node. Com que en el moment de treure'ls del graf tenien menys de K nodes adjacents, és segur que podrem assignar-li un dels K colors disponibles.



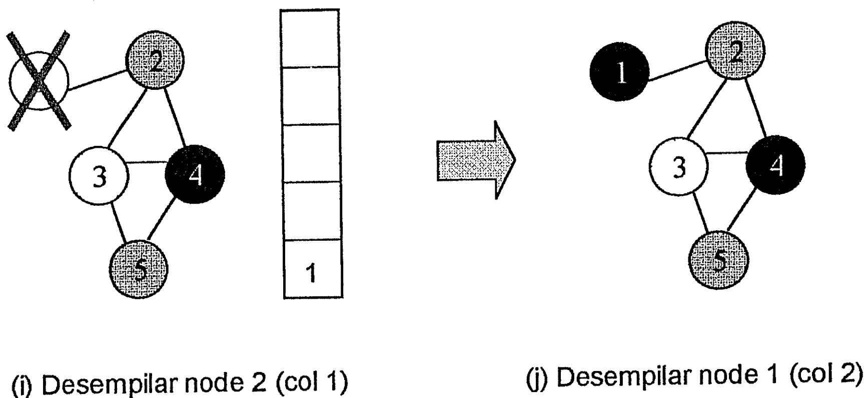
(a) Simplificar el node 1 (b) Simplificar el node 2 (c) Simplificar el node 3



(d) Simplificar el node 4 (e) Simplificar el node 5



(f) Desempilar node 5 (col 1) (g) Desempilar node 4 (col 2) (h) Desempilar node 3 (col 3)



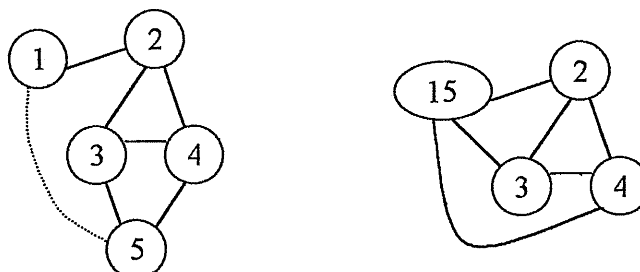
(i) Desempilar node 2 (col 1) (j) Desempilar node 1 (col 2)

Exemple 4.7 Coloració per simplificació amb K=3 colors

Si en algun moment de l'algorisme no podem triar node amb menys de K nodes adjacents (veïns), triem el node que té l'heurística d'utilització més baixa i el posem a la pila, indicant que és un candidat a ser assignat a memòria. Al desempilar-lo tindrà més de K veïns, i pot ser que no li poguem assignar cap color i hagi d'anar a memòria però també pot ser que algun dels seus veïns tinguin el mateix color i pugui ser assignat a registre.

Falta detallar com eliminarem instruccions de còpia. El procediment es basa es diferenciar dos tipus de nodes en el graf: els que representen variables involucrades en instruccions de còpia i els que no. Només s'aplicarà la simplificació sobre variables no involucrades en instruccions de còpia.

Per a eliminar una instrucció de còpia, dues variables han de tenir el mateix color. Això només es pot fer si les dues variables no interfereixen. El mètode per assignar el mateix color a dues variables es basa en **fusionar** els nodes corresponents a les dues variables. Com que només hi haurà un node que representarà a les dues variables, els hi assignarem el mateix color.



Instrucció de còpia entre les variables 1 i 5

Exemple 4.8 Fusió de nodes involucrats en instruccions de còpia

Quan fusionem dos nodes, el node resultant té com a arestes la unió de les arestes del node original. Això vol dir que el nou node interfereix amb més variables, i per tant, podria ser que al fusionar les variables no poguessim simplificar la variable i haguéssim d'assignar-la a memòria. Per això, **només fusionarem dues variables si estem segurs que fer-ho no modifica la colorabilitat del graf**: si abans de fusionar el graf es podia colorejar amb K colors, després de fusionar també ha de ser possible. Per a comprovar això es poden fer servir dues estratègies [APP98]:

- **Briggs** (estratègia escollida): podem fusionar dos nodes A i B si el node resultant AB té menys de K nodes adjacents de grau major o igual a K
- **George**: podem fusionar dos nodes A i B si cada node adjacent al node A o bé també és adjacent a B o bé té grau menor o igual a K

```
Algorisme coloració de registres
Entrada
    Graf de fluxe de control amb informació d'anàlisi de vida: cfg
    Número de colors disponibles : K
Sortida
    Assignació de registres: assig

igraf := construir el graf d'interferència ;
heuristiques := calcular  heurístiques ;
fer
    si és possible simplificar un node llavors
        simplificar (igraf,pila_nodes,K);
    sino si és possible fusionar dos nodes llavors
        fusionar (igraf, heurístiques);
    sino si alguna variable restant participa en instruccions de còpia llavors
        congelar (heurístiques);
    sino
        assignar_a_memòria (igraf, heurístiques,K);
    fsi
mentre alguna variable de VARS encara no estigui a registre o a memòria;
assig := assignar_colors (igraf,pila_nodes);
retorna assig;
```

```
simplificar (igraf, pila_nodes, K)
    v := seleccionar un node de "igraf" no precolorejat,
                                          i que no participi en instruccions de còpia
                                          i amb menys de K nodes adjacents a g;
    si v existeix llavors
        empilar (pila_nodes, v);
        eliminar v del graf  igraf;
    sino
        (* no  és possible simplificar *)
    fsi

fusionar (igraf, heurístiques)
    (* Unir dos nodes que representen variables involucrades en instruccions *)
    (* de còpia i que no interfereixen *)
    max := 0; actual := ∅;
    per a cada parella de variables {v1,v2}  del graf igraf
        si heurística.còpia (v1,v2) > max i
          el test de Briggs indica que la fusió de v1 i v2 és segura i
          v1 i v2 no interfereixen llavors
             actual := {v1,v2};
        fsi
    fper
    si actual = ∅ llavors
        (* No és possible fusionar dos nodes *)
    sino
        fusionar les variables (nodes) en el graf;
        recalcular les heurístiques del nou node i els seus adjacents;
    fsi

congelar(heurístiques)
    (* Deixar de considerar determinades instruccions de còpia que no han *)
    (* pogut ser eliminades mitjançant la fusió *)
    mentre cap variable hagi deixat de participar en instruccions de còpia fer
        deixar de considerar instruccions de còpia
             ≡ posar zeros a la taula d'heurístiques;
    fmentre

assignar_a_memòria (igraf,heurístiques,K)
    (* Simplificar un node encara que és possible que sigui assignat a *)
    (* memòria; fem això quan no podem ni simplificar, ni fusionar, ni congelar *)
    v := seleccionar el node del graf amb K o més veïns,
                                      i no precolorejat,
                                      i no involucrat en instruccions de còpia,
                                      i heurística d'utilització mínima;
    simplificar el node v;

assignar_colors (igraf, pila_nodes)
    (* Els registres temporals tenen els colors més baixos de l'interval 0.K *)
    (* i els registres salvats tenen els més alts: $a0-$a3, $t0-$t9, $s0-$s6 *)
```

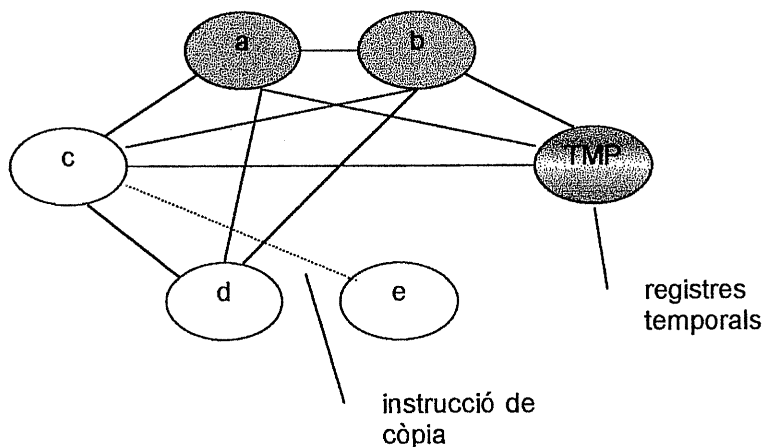
```
mentre no buida (pila_nodes) fer
    v := cim (pila_nodes);
    desempilar (pila_nodes);
    tornar a posar v al graf d'interferència;
    color_actual := 0;
    possible := fals;
    mentre color_actual ≠ K i no possible fer
        si algun node w adjacent a v té color (w) = color_actual llavors
            color_actual := color_actual +1;
        sino
            possible := cert;
        fsi
    fmentre
    si color_actual = K llavors
        assig (v) := MEMÒRIA; (* No hi ha cap registre disponibles *)
    sino
        assig (v) := REGISTRE (color_actual);
    fsi
fmentre
retorna assig;
```

El comportament obtingut pel que fa a l'assignació de registres temporals i salvats és el que es buscava des d'un principi:

- si una variable està viva durant una crida a procediment, al graf d'interferències interferirà amb tots els registres temporals, i per tant serà assignada a un registre salvat
- si una variable no està viva durant una crida a procediment, intentarem assignar-li colors començant pel 0, 1, ... Com que hem indicat que els primers colors corresponen a registres temporals, segurament li assignarem un registre temporal a aquesta variable. Només s'assignarà aquesta variable a un registre salvat si interfereix amb variables que estiguin en tots els registres temporals, i això és molt difícil pel gran nombre de registres temporals de l'arquitectura MIPS.

Per acabar, s'exposa un exemple complet d'assignació de registres, mostrant l'aplicació de les diferents fases de l'algorisme (exemple 4.9).

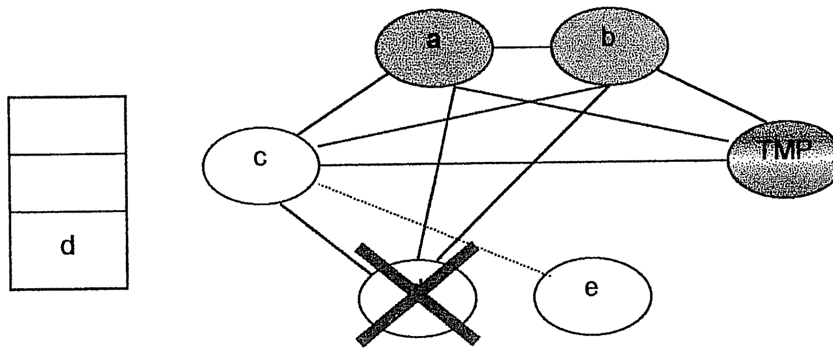
```
.gscalar valor

.function dummy
  .parameter a,b
  .scalar c, d, e

.i1:  c := a - 3
      .if a < 0 .goto .i0
.i2:  d := b * a
      valor := d
      d := dummy (b,a)
      e := d + c
      c := e
.i0:  .return c
.end dummy
```



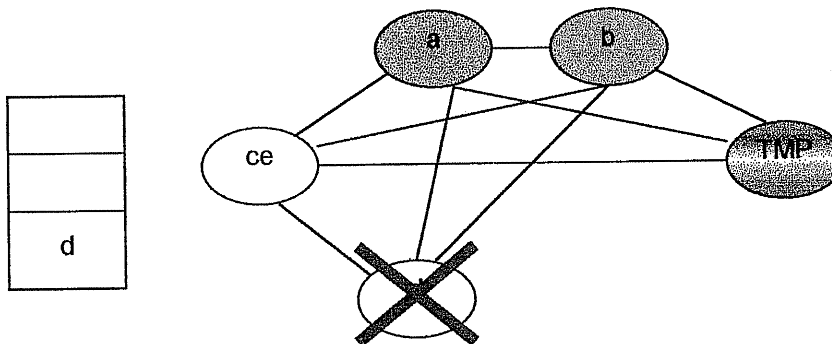
Exemple 4.9 Programa i graf d'interferència en l'exemple

En el graf d'interferència, hi ha cinc variables: a, b, p2, sca1, sca2. Per a simplificar el graf, els nodes precolorejats que representen els registres temporals s'han dibuixat com un sol node (TMP). Les variables a i b, al ser els dos primers paràmetres de la funció estaran assignats a registre i per tant els seus nodes estan precolorejats amb els colors 0 i 1 (registres $a0 i $a1).

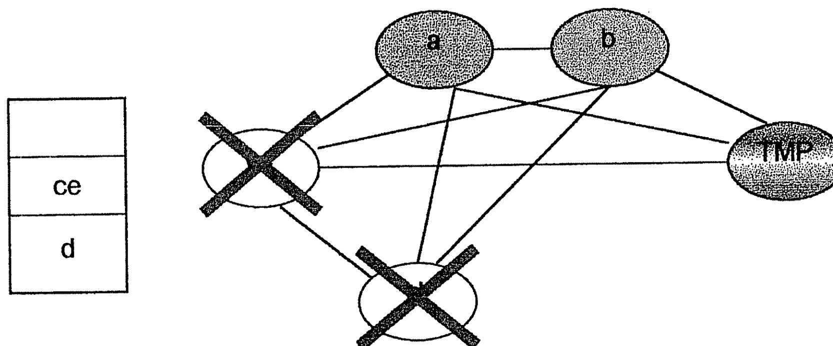
El número de registres disponibles per assignar registres a variables és: K= 21 ($a0-$a4, $t0-$t9, $s0-s6). El procediment que es segueix per a colorejar el graf és el següent:



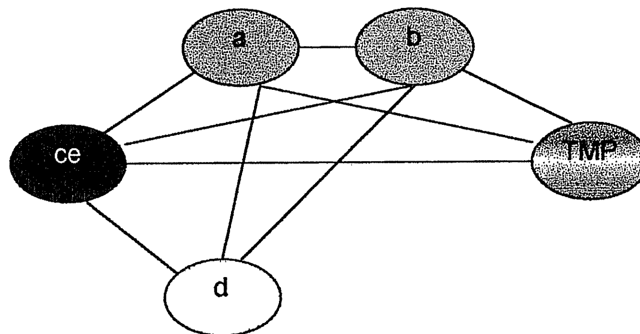
(a) Simplificar: Tots els nodes tenen menys de K = 21 veïns, però només n'hi ha un que no estigui precolorejat ni participi es cap instrucció de còpia: d. El simplifiquem i no podem seguir simplificant.



(b) Fusionar: Els nodes (c,e) són els únics nodes participants en instruccions de còpia i per tant són els que tenen l'heurística de còpia més alta. Fusionem els dos nodes en un de sol: ce.



(c) Simplificar: Tornen a haver-hi nodes per simplificar: ce pot ser simplificat.. Tots els nodes que queden en el graf estan precolorejats, i per tant no queda cap node que no hagi estat assignat. Passem a l'assignació de colors



(d) Assignar colors: Desempilem els nodes que hem anat guardant a la pila en cada simplificacio. El node ce intefereix amb tots els registres temporals i per tant li assignem el color del primer registre salvat: \$s0. El node d interfereix amb les variables a i b i per tant no pot usar els registres \$a0 ni \$a1: usarà el registre \$a2.

a	→	\$a0	d	→	\$a2
b	→	\$a1	e	→	\$s0
c	→	\$s0			

Eliminada una instrucció de còpia
Només cal salvar/restaurar un registre: \$s0

(e) Assignació de colors obtinguda

Exemple 4.10 Passos per a l'assignació de registres

4.5 - Conclusions

Els accessos a memòria són costosos, i per tant, situar correctament les dades en els registres és molt important per al temps d'execució. De tota manera, l'assignació de registres òptima és molt costosa de trobar i per això caldrà buscar una solució subòptima en un temps raonable. Una tècnica possible per a això és l'assignació de registres basada en coloració de grafs.

L'elecció en l'assignació de registres no es limita a triar "*registres o memòria?*". No tots els registres es comporten de la mateixa manera (paràmetres, temporals o salvats), i cal decidir quin tipus de registre és més adient durant l'assignació de registres.

L'arquitectura concreta amb la que es treballa, MIPS, disposa d'un gran nombre de registres i per tant és poc probable que alguna variable hagi d'anar a memòria. El que sí és important en aquesta arquitectura és decidir quines variables es guarden en registres salvats i quines variables es guarden en registres temporals. Escollir una opció o l'altra pot representar haver de salvar registres a memòria o no haver de fer-ho en absolut.

5 Generació de codi

El codi intermedi s'ha de traduir a codi màquina per poder ser executat. L'assignació de registres ja ha proporcionat la localització deles variables del codi intermedi: queda per decidir com es traduiran les instruccions del codi intermedi a instruccions de codi màquina. Per a fer això s'han de tenir en compte les característiques de l'arquitectura per a aprofitar-les al màxim.

En aquest capítol es presenten els diferents algorismes per a generació de codi màquina que s'utilitzen habitualment, l'algorisme escollit i les característiques rellevant del processador MIPS, i exemples del codi màquina generat.

5.1 - Objectius de la generació de codi

El codi intermedi optimitzat resultant de les fases anteriors **s'ha de traduir a un codi màquina eficient**, que utilitzi al màxim les possibilitats de l'arquitectura destí. El procés de generació de codi màquina es pot separar en tres fases:

- **assignació de registres**, que escull quins operands es guarden en registre i quins operands es guarden en memòria. Aquesta fase ja ha estat tractada al capítol 4
- **selecció d'instruccions**, que tradueix les instruccions de codi intermedi a instruccions de codi màquina. La complexitat d'aquesta fase depèn del conjunt d'instruccions de l'arquitectura destí.
- **optimització del codi màquina** (opcional), que intenta millorar la qualitat del codi màquina produit. Típicament, aquesta és una optimització de finestra (veure 3.4.2) que busca patrons d'instruccions per a substituir-los per altres patrons més eficients.

Els objectius d'aquesta fase de generació de codi seran dos: per una banda, **generar un codi màquina el més eficient possible**, i per altra banda **respectar les convencions d'ús de registres i crides a funció.**

El problema "donat un codi intermedi, generar el codi màquina equivalent i òptim" és un problema NP-complet, és a dir, només es pot calcular de manera exponencial. D'aquesta manera, **l'objectiu d'aquesta fase serà generar un codi màquina eficient**, no un codi màquina òptim. Per a generar un codi màquina eficient, caldrà tenir en compte les característiques concretes de l'arquitectura:

- **selecció de l'ordre d'avaluació:** En algunes arquitectures, si alterem l'ordre de càlcul d'una seqüència d'instruccions de codi intermedi, pot ser que millori la qualitat del codi màquina produit.
- **els modes d'adreçament:** En el codi màquina, es poden indicar les adreces de memòria de maneres diferents (indicant el valor de l'adreça, indicant un registre que conté un apuntador a l'adreça, ...) , conegudes com modes d'adreçament. Utilitzar un mode d'adreçament o un altre representa un cost de temps diferent.
- **el conjunt d'instruccions:** No totes les instruccions del codi màquina triguen el mateix temps a executar-se; pot ser que utilitzar unes instruccions en comptes d'unes altres ens proporcioni un codi més eficient i per això cal decidir quines instruccions utilitzarem. En les arquitectures RISC, les instruccions del codi màquina són molt senzilles, i per això és fàcil identificar quan s'ha d'utilitzar cada instrucció. En canvi, les arquitectures CISC (com ara Intel) ofereixen una conjunt d'instrucions molt complex. Això vol dir que hi ha instruccions molt complexes, que poden realitzar càlculs molt complicats de forma molt eficient. El problema de les arquitectures CISC és decidir quan podem utilitzar aquestes instruccions complexes.

A més d'aconseguir un codi màquina eficient, **per a cada arquitectura existeix un conjunt de convencions d'ús**. Aquestes convencions indiquen com utilitzar els registres, com construir els blocs d'activació en les crides a procediment, etc. L'objectiu d'aquestes convencions és que siguin usades per tots els compiladors, de manera que una rutina compilada amb un cert compilador pugui ser cridada per una rutina compilada des d'un altre compilador. És important que en la generació de codi màquina es respectin aquestes convencions.

5.2 - Estratègies per a la generació de codi

A l'hora de de realitzar la traducció de codi intermedi a codi màquina hi ha diferents mètodes que podem seguir: generar codi instrucció a instrucció, generar codi a nivell de bloc bàsic o utilitzar un algorisme de reescriptura d'arbres.

- **generar codi instrucció a instrucció + optimització de finestra**

El procediment més senzill per a generar codi màquina per a un conjunt d'instruccions de codi intermedi és generar codi màquina per a la primera instrucció, després per a la segona, ... El problema d'aquest mètode és que es poden generar instruccions redundants o ineficients. Per aquest motiu, un procés de generació de codi instrucció a instrucció acostuma a anar seguit per una optimització de finestra, per a millorar la qualitat del codi produit.

```
a := b + c  →  Guardar b al registre reg(b)
               Guardar c al registre reg(c)
               si a està en un registre llavors
                   add reg(a), reg(b), reg(c)
               sino
                   add $v0, reg(b), reg(c)
                   sw  $v0, addr(a)
```

Exemple 5.1 Generació de codi instrucció a instrucció

L'exemple 5.1 ens proporciona un exemple de codi que pot ser subòptim. Si la següent instrucció a a:=b+c utilitza la variable a, no sabrà que ja està guardada al registre temporal $v0. Això pot ser resolt fàcilment amb una optimització de finestra

L'avantatge d'aquest mètode és que és molt senzill, i el seu inconvenient és que la qualitat del codi màquina produit podria ser millorada amb algun dels mètodes següents.

- **generació de codi a nivell de bloc bàsic ([AHO90])**

Existeixen algorismes de programació dinàmica que permeten **generar codi de manera òptima per a un bloc bàsic** (eliminant codi mort, subexpressions comunes, triant l'ordre òptim d'avaluació, ...), així com també existien algorismes d'assignació de registres a nivell de bloc bàsic. Un compilador que no realitzi cap optimització global de codi intermedi pot utilitzar assignació de registres i generació de codi a nivell de bloc bàsic per a obtenir un codi màquina de millor qualitat.

```
t1: = 4 * i
a1 := b [t1]
t2 := 4 * i
a2 := c [t2]
t4 .= a1 + a2
```

```
mul $t1, 4, i
lw $a1, b + ($t1)
lw $a2, c + ($t1)
add $t4, $a1, $a2
move $t2, $t1
```

(a) (b) (c)

Exemple 5.2 Generació de a nivell de bloc bàsic
(a) bloc bàsic
(b) DAG associat al bloc bàsic
(c) codi màquina generat

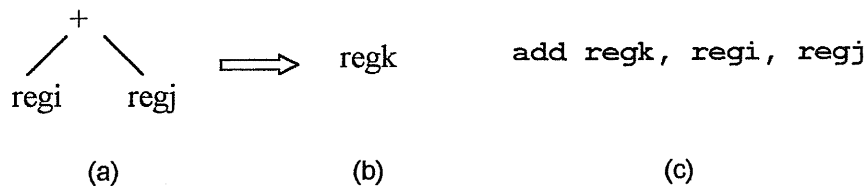
Un avantatge d'aquest mètode és que permet tenir en compte l'ordre d'avaluació de les instruccions, realitzant al mateix temps moltes optimitzacions sobre el codi del bloc bàsic. Per això és un mètode molt utilitzat en els compiladors actuals.

- **utilitzar un algorisme de reescriptura d'arbres ([AHO90], [APP98])**

Si el codi intermedi està escrit en forma d'arbres sintàctics, podem expressar les instruccions del codi màquina en forma de patrons. Cada patró ens indica un subarbre del qual parteix, el subarbre que genera i les instruccions de codi màquina produides per aquell patró. Un cop tenim aquests patrons, busquem a l'arbre del codi intermedi els nostres patrons. Cada cop que trobem un patró substituim el subarbre original per el subarbre destí i produim la llista d'instruccions de codi màquina associades al patró.

L'avantatge d'aquest mètode és que és possible expressar instruccions molt complexes gràcies a aquests patrons. Un inconvenient del mètode és que es necessiten molts patrons per a expressar el conjunt d'instruccions d'un codi màquina concret. Pot ser necessari un

pas previ de reescriptura de l'arbre de codi intermedi, per a que aquest estigui en una forma més o menys canònica (cada construcció expressada de forma única) i es redueixi el número de patrons a buscar ([APP98]).



(a) (b) (c)

Exemple 5.3 Generació de codi reescrivint subarbres
(a) subarbre original
(b) subarbre destí
(c) codi màquina associat

5.3 - El codi màquina destí: MIPS

Les característiques de l'arquitectura MIPS descrites en aquest capítol estan extretes de [LAR98], que descriu un simulador del processador MIPS R2000 anomenat SPIM.

El processador MIPS R2000 segueix una arquitectura RISC, i per tant **les instruccions del seu codi màquina són molt senzilles**, amb pocs modes d'adreçament. MIPS té diversos processadors, entre ells un coprocessador per a realitzar càlculs de punt flotant, amb un banc de registres propi. Caldrà executar instruccions especials per a carregar els registres del processador i executar les instruccions de punt flotant.

Altres arquitectures (Intel x86) tenen només dos operands: un operand que és a la vegada font i destí, i un operand font. En canvi, la majoria d'**instruccions del codi màquina de MIPS tenen tres operands**: **un operand destí** (que pot ser diferent dels altres dos) i **dos operands font**. Això facilita molt la generació de codi màquina a partir del code IC, que és un codi de tres adreces.

5.3.1 - Accés a memòria

MIPS és una arquitectura "load-store". Això vol dir que les úniques instruccions que poden accedir a memòria són les que fan un load (llegir un valor de memòria) o store (escriure un valor a memòria).

Durant l'execució d'un programa, la memòria està dividida en quatre grans conjunts d'adreces consecutives, anomenades **segments**:

- el **segment de text**, que guarda el codi del programa que s'està executant
- el **segment de dades estàtic**, que guarda valors durant tot el temps d'execució del programa (variables globals)
- el **segment de dades dinàmic**, que proporciona l'espai demanat per les crides malloc o new que demanen reservar més memòria
- el **segment de pila**, on es guarden (entre altres) les variables locals dels procediments

Quan accedim a una adreça de memòria, podem llegir tamanys diferents: un **byte** (8 bits), un **halfword** (16 bits), un **word** (32 bits) o un **doubleword** (64 bits). Algunes instruccions de load i store necessitem que l'adreça estigui alineada, és a dir, que sigui múltiple del tamany en bytes que volem llegir. Encara que MIPS proporciona instruccions per a llegir dades no alineades (`lwl`, `lwr`, `swl` i `swr`), guardarem i accedirem a totes les dades de forma alineada.

Els modes d'adreçament permesos en les operacions de load i store són els següents:

Mode d'adreçament	Significat
`(reg)`	@ = contingut del registre
`const`	@ = valor de la constant
`const(reg)`	@ = valor de constant + contingut del registre
`label`	@ = adreça de l'etiqueta
`label±const`	@ = adreça de l'etiqueta ± valor de la constant
`label±const(reg)`	@ = adreça de l'etiqueta ± (valor de constant + contingut de registre)

5.3.2 - Convencions d'ús

Quan escrivim programes per a MIPS hem de respectar la convenció d'ús dels registres i la convenció de construcció del bloc d'activació. Això ens indica quin ús hem de donar als registres i com guardar paràmetres, variables locals, etc. a la pila.

Nom del registre	Ús	Nom del registre	Ús
$zero	constant 0	$at	reservat
$v0-$v1	avaluació d'expressions i retorn de funció	$k0-$k1	reservats
		$gp	àrea global
$a0-$a3	paràmetres de funció	$sp	punter a la pila
$t0-$t9	registre temporal ("callee-save")	$fp	punter al bloc d'activació anterior
$s0-$s6	registre salvat ("caller-save")	$ra	adreça de retorn

Els registres dedicats a guardar variables ($a0-$a3, $t0-$t9, $s0-$s6) han estat tractats al capítol 4. Només falta discutir els registres $gp, $sp, $fp i $ra, involucrats en la construcció del bloc d'activació a la pila.

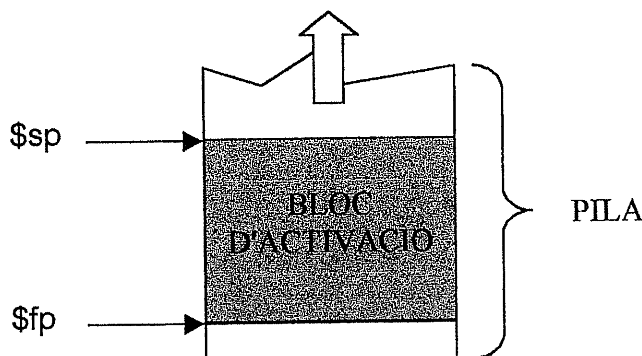


Figura 5.1 Relació entre $sp, $fp i el bloc d'activació

El registre $gp indica l'adreça de memòria d'un bloc de memòria de 64 k en el segment de dades estàtic de la memòria, on es poden guardar variables globals. El registre $sp indica la següent posició lliure a la pila, i $fp apunta a la posició de la pila que guarda el principi del bloc d'activació actual (veure Figura 5.1).

El **bloc d'activació** ("frame") és un fragment de la pila generat en el moment de produir-se una crida a una funció, i que conté informació rellevant a l'execució de la funció i el retorn a la funció que ha realitzat la crida. Les dades que hi ha en el bloc d'activació són:

- **Els paràmetres per sobre del quart**. Recordem que els quatre primers paràmetres són guardats en els registres $a0-$a3.
- **Els registres que s'han de salvar durant la funció**, tant els temporals (salvats quan es produeix una crida a funció i restaurats després de la crida) com els salvats (salvats al principi de la funció i restaurats al final de la funció). **Un dels registres que és possible que s'hagi de salvar és $ra**, que conté l'adreça de retorna a la rutina que ha realitzat la crida a aquesta funció.
- **Les variables locals de la funció que s'han de guardar a memòria**.

Existirà un bloc d'activació per cada crida a una determinada a funció, és a dir, diferents crides a una mateixa funció tindran blocs d'activació diferents. Les accions per a gestionar el bloc d'activació s'hauran de realitzar en tres moments: **quan realitzem la crida a la funció, al principi de la funció i al final de la funció**.

En el moment de realitzar la crida a una funció cal realitzar els següents passos:

- **Realitzar el pas d'arguments**, guardant els quatre primers paràmetres als registres $a0-$a3 i la resta al cim de la pila (indicat per $sp).
- **Salvar els registres temporals** que estan vius durant la crida a funció. En el nostre cas, l'assignació de registres en garantirà que no cal salvar els registres $t0-$t9, ja que si estan vius s'hauran assignat a un registre $s0-$s6. Sí que haurem de salvar els registres $a0-$a3 que utilitzem, abans de realitzar la crida a funció.
- **Executar la instrucció `jal`**, que passa el fluxe de control del programa a la nova funció i guarda l'adreça de retorn al registre $ra.

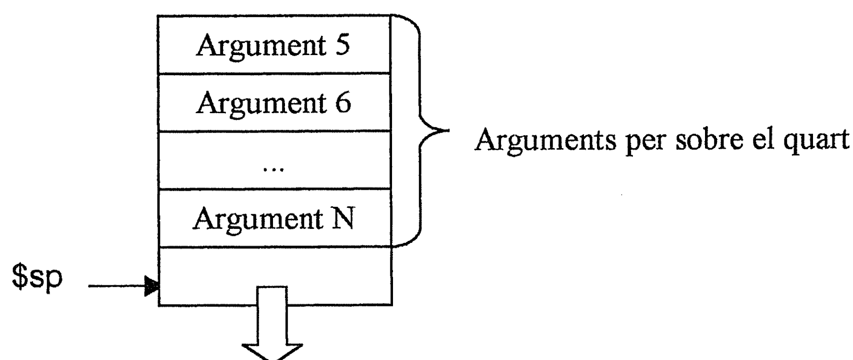


Figura 5.2 Bloc d'activació després de cridar a una funció

Abans de començar el codi de la rutina, cal realitzar els següents passos:

- **Reservar espai per al bloc d'activació** ($sp := $sp - tamany_bloc). El tamany del bloc d'activació és de 4 bytes per cada registre que haguem de salvar i per a cada variable estructurada, el tamany de cada element multiplicat pel número d'elements.

- **Salvar el registre $fp, el registre $ra i els registres salvats ($s0-$s6) que s'utilitzin**. El registre $ra només s'ha de salvar si la funció farà alguna crida a funció. Una funció que no fa cap crida s'anomena funció fulla, i no ha de salvar ni restaurar $ra.
- **Calcular el nou valor de $fp** ($fp := $sp + (tamany_bloc-4)). $fp apunta a la primera posició del bloc d'activació, que conté el registre $fp de la funció que ha realitzat la crida a aquesta funció.

Per accedir a cada element del bloc d'activació, cal còneixer el seu desplaçament dins d'aquest bloc. La primera posició és 0($fp), que conté el punter a l'inici del bloc d'activació anterior; la següent posició serà -4($fp), la següent serà -8($fp), ... Els arguments per sobre del cinquè estan guardats a 4($fp) i més endavant. La convenció no estableix un ordre determinat per a guardar els registres que s'hagin de salvar (només indica que s'han de salvar abans que les variables locals), però a la Figura 5.2 apareix l'ordre utilitzat.

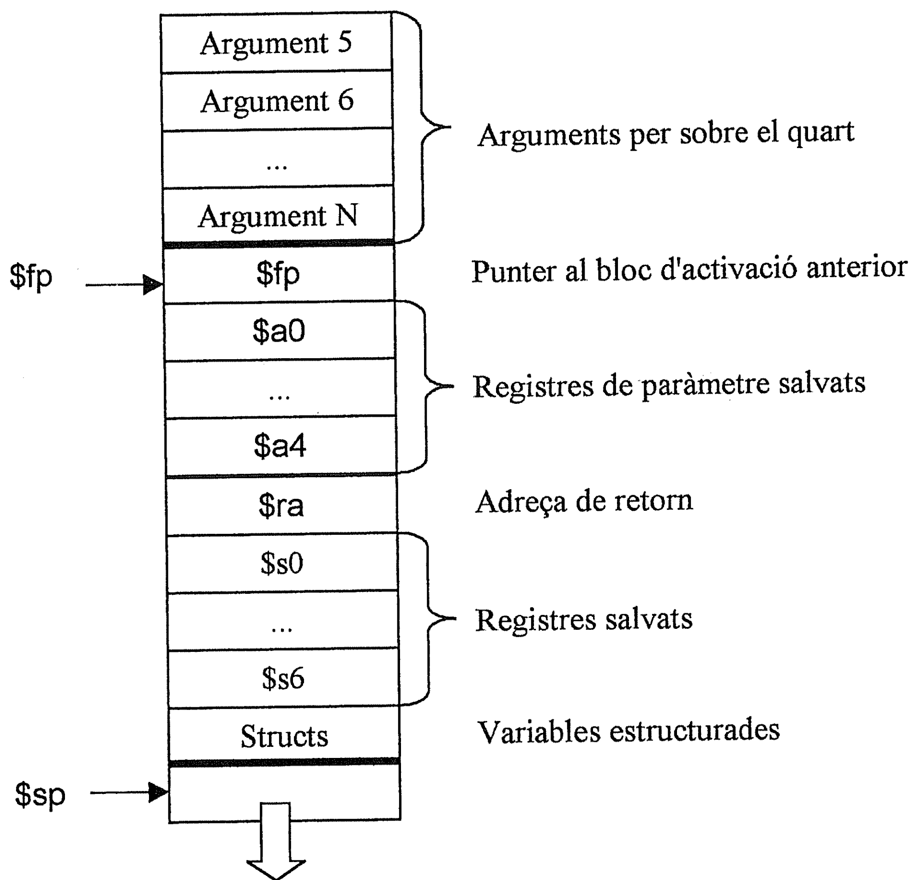


Figura 5.3 Bloc d'activació després de començar la funció

Quan acabem d'executar una funció, hem de realitzar els següents passos:

- **El valor de retorn de la funció s'ha de guardar al registre $v0**.
- **Restaurar els registres $s0-$s6, $fp i $ra** si al principi de la funció es van salvar.
- **Recuperar l'espai ocupat pel bloc d'activació** ($sp := $sp + tamany bloc).
- **Saltar a l'adreça guardada al registre $ra** per retornar a la funció que ha fet la crida.

El següent és un exemple del codi que s'ha de cridar al principi i al final d'una funció. El codi i els comentaris són els generats per **CODEGEN**.

```
fact:
   # This will be the stack frame for this function
   # Old $fp stored in 0($fp)
   # Parameters: 1
      # Par 1 (in) stored in -4($fp) uses register $a0
   # Saved registers: 4 bytes
   # Scalars: 3
      # Scalar (previous) stored in -12($fp) uses register $a1
      # Scalar (tmp) stored in -16($fp) uses register $a1
      # Scalar (res) stored in -20($fp) uses register $a0
   # Structs: 0
   # Total stack size: 24 bytes

   subu   $sp,$sp,24      # Space for stack frame
   sw     $fp,20($sp)     # Save old frame pointer
   addu   $fp,$sp,20      # Set up frame pointer
   sw     $ra,-8($fp)     # Save return address (ra)
```

(a)

```
   move   $v0,$a0
   lw     $ra,-8($fp)     # Load return address (ra)
   lw     $fp,0($fp)      # Restore frame pointer
   addu   $sp,$sp,24      # Pop stack frame
   jr     $ra             # Return from function
```

(b)

Exemple 5.4 Codi MIPS per executar
(a) al principi de la funció
(b) al final de la funció

5.3.3 - Conjunt d'instruccions

El conjunt d'instruccions de MIPS és molt complet, i aquí presento només un subconjunt. D'una mateixa instrucció pot haver-hi diverses versions, que usin un immediat (`i`), que usin un registre (v), que siguin amb signe (cap caràcter especial) o sense (u), que tinguin overflow (o) o que no en tinguin (cap caràcter especial). A la taula només indico la instrucció bàsica (per exemple, add) i les versions existents (per exemple, `iu` seria `addiu`, sumar usant un immediat sense signe).

En les operacions que produeixen un valor, si no s'indica el contrari el primer registre serà el registre destí on es guardarà el valor. Per exemple `add $a0, $a1, $a2` consulta els valors dels registres $a1 i $a2 i guarda la suma al registre $a0.

Instruccions aritmètico-lògiques		
abs	rd,rs	valor absolut
add	rd,rs,rt	suma (u,i,iu)
and	rd,rs,rt	and lògica (i)
div	rd,rs,rt	divisió (u)
mul	rd,rs,rt	producte (o,ou)
neg	rd,rs	canvi de signe amb overflow, negu sense overflow
nor	rd,rs	nor lògica
not	rd,rs	not lògica
or	rd,rs,rt	or lògica (i)
rem	rd,rs,rt	mòdul (u)
sll	rd,rt,imm	shiftar lògic a l'esquerra imm vegades, sllv ho fa amb un registre
sra	rd,rt,imm	shiftar aritmètic a la dreta imm vegades, srav ho fa amb un registre
srl	rd,rt,imm	shiftar lògic a la dreta imm vegades, srlv ho fa amb un registre
rol	rd,rs,rt	rotar a l'esquerra
ror	rd,rs,rt	rotar a l'esquerra
sub	rd,rs,rt	resta (u)
xor	rd,rs,rt	xor (i)
Instruccions de comparació		
slt	rd,rs,rt	rd és1 si rs < rt, 0 en cas contrari (u, i, iu)
seq	rd,rs,rt	igualtat
sge	rd,rs,rt	major o igual (u)
sgt	rd,rs,rt	major estricte (u)
sle	rd,rs,rt	menor o igual (u)
sne	rd,rs,rt	diferent
Instruccions de salt		
b	label	salt incondicional
bczt	label	saltar si flag zero activat
bzcf	label	saltar si flag zero del coprocessador activat
beq	rs,rt,label	saltar si iguals
bgez	rs,label	saltar si és igual a zero
bgtz	rs,label	saltar si major estricte que zero
blez	rs,label	saltar si menor o igual a zero
bltz	rs,label	saltar si menor estricte que zero
bne	rs,rt,label	saltar si diferents
beqz	rs,label	saltar si igual a zero
bge	rs,rt,label	saltar si més rs >= rt (u)
bgt	rs,rt,label	saltar si rs > rt (u)
ble	rs,rt,label	saltar si rs <= rt (u)
blt	rs,rt,label	saltar si rs < rt (u)
bnez	rs,label	saltar si diferent de zero
jal	target	saltar a una rutina guardant l'adreça de retorn a $ra
jr	rs	saltar a l'adreça guardada en el registre (retorn de funció)
Instruccions de load		
la	rd,address	carregar l'adreça (no el contingut en un registre)
lb	rd,address	carregar un byte (u)
lh	rt,address	carregar un halfword (u)

`lw rt,address`	carregar un word
`ld rt,address`	carregar un doubleword
Instruccions de store	
`sb rt,address`	guardar un byte a memòria
`sh rt,address`	guardar un halfword a memòria
`sw rt,address`	guardar un word a memòria
`sd rt,address`	guardar un doubleword a memòria
Altres instruccions	
`move rd,rs`	instrucció de còpia (rd := rs)
`syscall`	crida al sistema
`break`	acabar l'execució
`nop`	NOP (no fer res)

Aquesta llista no inclou les instruccions del coprocessador de punt flotant. Aquestes instruccions, aritmètico-lògiques en la seva majoria, són molt similars a les instruccions normals de MIPS. Les instruccions del coprocessador de punt flotant sempre contenen un punt (.) en el seu nom i acostumen a haver-hi dues versions segons la precisió: doble precisió o precisió senzilla (segons l'estàndard IEEE 754). Per a reals de doble precisió s'afegeix (`d`) al nom de la instrucció i per a reals de precisió senzilla, s'afegeix (`s`). Així, per exemple, la suma de reals de doble precisió seria **`add.d`** i la de reals de precisió senzilla seria **`add.s`**, la resta seria **`sub.d`** i **`sub.s`**, etc.

Cal tenir en compte que aquest només és el subconjunt d'instruccions de MIPS que s'utilitzarà per traduir el codi IC. [LAR98] conté el conjunt complet d'instruccions i més informació sobre les instruccions, com ara la codificació dels operands en les instruccions de codi màquina.

5.4 - Procés de generació de codi MIPS

La similitud del codi màquina de MIPS amb el code intermedi IC facilita molt el procés de generació de codi. Com que gairebé existeix una equivalència un a un entre les instruccions de MIPS i codi IC, generar codi instrucció a instrucció per a cada instrucció del codi IC pot donar bons resultats. L'optimització de finestra posterior bàsicament hauria d'evitar excessives lectures-escriptures de la memòria en les variables, fent que si una còpia d'una variable ja es troba en registre no es torni a carregar en memòria.

El mètode implementat ha estat la **generació de codi instrucció a instrucció** sense optimització de finestra posterior (donat que l'optimització de codi màquina queda fora de l'abast del projecte). El motiu d'aquesta elecció ha estat la simplicitat d'aquest mètode, generant un codi relativament bo (en el cas de la traducció de codi IC a MIPS). De totes maneres, si volguessim generar codi màquina de qualitat seria gairebé imprescindible utilitzar la generació de codi a nivell de bloc bàsic, o com a mínim afegir una bona optimització de finestra al codi maquina.

La següent és una versió de molt alt nivell de l'algorisme utilitzat per generar codi MIPS:

```
Algorisme generar codi MIPS
Entrada
    Programa escrit en codi intermedi IC
Sortida
    Programa MIPS

generar directives per a declarar les variables globals
per a cada funció f del programa
    per a cada variable o registre no temporal k utilitzat
        assignar a k una posició de memòria dins del bloc d'activació
    fper
    calcular el tamany total del bloc d'activació
    generar codi per inicialitzar el bloc d'activació al principi de la funció
    per a cada instrucció i de la funció f
        generar codi MIPS per la instrucció i
    fper
    generar codi per restaurar registres, eliminar el bloc d'activació i retornar
fper
```

Per a il.lustrar el procés de traducció, mostraré l'algorisme de traducció d'algunes classes d'instruccions IC. En la traducció de moltes classes d'instruccions, s'utilitzen les rutines forçar_registre i salvar_operand, que respectivament asseguren que un operand es troba en registre o salven un operand de registre a memòria.

```
Algorisme traduir una instrucció
Entrada
    Instrucció de codi IC: i
Sortida
    Instruccions de codi MIPS

per a cada operand o de la instrucció
    (* guardar l'operand o en un registre temporal, $v0 o $v1 *)
    forçar_registre (o,r)
fper
traduir la instrucció IC a la instrucció(ns) MIPS equivalent
si s'ha produit un resultat i el resultat està en memòria llavors
    salvar_operand()
fsi

(* En funció del tipus ($i, $r, $b, ...) de la instrucció, cal fer un sw o sb *)
(* En aquest algorisme només es presenta el cas en que les variables ocupen *)
(* word (32 bits de memòria) *)

forçar_registre (variable, reg_per_defecte)
    si variable està associada a un registre llavors
        reg:= registre (variable);
    sino
        (* Guardar la variable en el registre per defecte *)
        reg:= reg_per_defecte;
        si variable és global llavors
```

```
            generar ("lw reg, nom(variable)");
        sino
            offset:= desplaçament (variable);
            generar ("lw reg, offset($fp)");
        fsi
    fsi
    retorna reg;

salvar_operand (variable, reg_actual)
    si variable és global llavors
        generar ("sw reg, nom(variable)");
    sino
        offset := desplaçament(variable);
        generar ("sw reg,offset($sp)");
    fsi
```

Es pot veure que si la variable ja es troba en un registre, llavors les funcions forçar_registre i salvar_operand no generaran cap instrucció. Si tots els operands i el resultat estan en registre (la situació més habitual), només es generaran les instruccions MIPS equivalents a les instruccions IC. Gairebé totes les instruccions IC es poden calcular amb una sola instrucció MIPS, i per tant el ratio de traducció més usual és d'una instrucció MIPS per cada instrucció IC.

A continuació es mostren els algorismes de traducció per a algunes instruccions de codi IC. Aquests exemples no pretenen ser un llistat exhaustiu de totes les rutines de traducció, i no mostren tota la complexitat del procés de traducció, però són molt representatius del procés de traducció.

```
a := b + c   →  r1 := forçar_registre(b, $v0)
                r2 := forçar_registre(c, $v1)
                si a està en un registre llavors
                    add reg(a), r1, 2
                sino
                    add $v0, r1, r2
                    salvar_operand(a,$v0);
                fsi
```

Exemple 5.5 Traducció d'una instrucció aritmètica

```
a := b       →  si a és a memòria llavors
                    r1 := forçar_registre (b,$v0);
                    salvar_operand (variable,r1);
                sino
                    r1 := forçar_registre(b,reg(a));
                    si r1 ≠ reg (a) llavors
                        move reg(a),r1
                    fsi
                fsi
```

Exemple 5.6 Traducció d'una instrucció de còpia

```
.goto .iN           →  k := etiqueta(N);
                       b $Lk

.if a = b .got .iN  →  k := etiqueta(N);
                       r1:= forçar_registre (a,$v0);
                       r2:= forçar_registre (b,$v1);
                       beq r1,r2,$Lk
```

Exemple 5.7 Traducció de les instruccions de salt

```
.return a      →  r1 := forçar_registre(b,$v0);
                  si r1 ≠ $v0 llavors
                      move $v0,r1
                  fsi
```

Exemple 5.8 Traducció d'una instrucció de retorn

```
a:= & b        →  si a es troba a memòria llavors
                      cal_salvar := cert;
                      r1 := $v0;
                  sino
                      cal_salvar:= fals;
                      r1 := reg(a);
                  fsi
                  si b és una variable global llavors
                      la r1, nom(b)
                  sino
                      add r1,$fp, offset (b)
                  fsi
                  si cal_salvar llavors
                      salvar_operand (a,r1);
                  fsi
```

Exemple 5.9 Traducció d'una instrucció d'obtenir adreça

5.5 - Un exemple de codi MIPS

A continuació es presenten un programa IC traduit a codi MIPS. El programa IC escollit com a exemple ha estat el factorial recursiu.

```
.gstring input_string "n = "
.gstring result_string "n! = "
.gstring new_line "\n "

.function main
 .scalar address1, input_integer, result, address2

    address1 := & input_string
    print_str(address1)
    input_integer := read_int()
    print_int (input_integer)
    address1 := & new_line
    print_str(address1)
    result := fact(input_integer)
    address2 := & result_string
    print_str(address2)
    print_int(result)
    print_char('\n' $c)
    .return
.end main
```

```
.function fact
 .parameter in
 .scalar previous, tmp, res

    print_int (in)
    tmp := & new_line
    print_str (tmp)
    .if in > 1 .goto .i1
    res := 1
    .goto i3
 .i1:
    previous := in - 1
    tmp := fact(previous)
    res := in * tmp
 .i3:
    .return res

.end fact
```

Exemple 5.10 Codi IC per a calcular el factorial

Les rutines print_char i print_str estan implementades en una llibreria, juntament amb altres crides a sistema com alloc (demanar més memòria) o exit (sortir del programa). La implementació d'aquestes rutines es fa a través de la instrucció especial de MIPS `syscall`. Aquesta instrucció espera rebre com a paràmetre el número de crida a sistema a realitzar. Donat que el número de crida a sistema es fixa a la llibreria, abans de fer la crida a sistema, aquestes rutines es poden cridar com si fossin rutines normals.

El codi MIPS equivalent a aquest codi IC és el següent:

```
####### Global data #######

    .data
_input_string:     .asciiz "n = "
_result_string:    .asciiz "n! = "
_new_line:         .asciiz "\n "

    .text

### Beginning of function main ###

main:
    # This will be the stack frame for this function
    # Old $fp stored in 0($fp)
    # Parameters: 0
    # Saved registers: 8 bytes
    # Scalars: 4
       # Scalar (address1) stored in -12($fp) uses register $a0
       # Scalar (input_integer) stored in -16($fp) uses register $s0
       # Scalar (result) stored in -20($fp) uses register $s0
       # Scalar (address2) stored in -24($fp) uses register $a0
    # Structs: 0
    # Total stack size: 28 bytes

    subu   $sp,$sp,28       # Space for stack frame
```

```
    sw      $fp,24($sp)      # Save old frame pointer
    addu    $fp,$sp,24       # Set up frame pointer
    sw      $ra,-4($fp)      # Save return address (ra)
    sw      $s0,-8($fp)      # Save register $s0
$L0:
    la      $a0,_input_string
    sw      $a0,-4($sp)      # This will be parameter 1
    jal     print_str        # Function call
    jal     read_int         # Function call
    move    $s0,$v0
    move    $a0,$s0
    sw      $s0,-4($sp)      # This will be parameter 1
    jal     print_int        # Function call
    la      $a0,_new_line
    sw      $a0,-4($sp)      # This will be parameter 1
    jal     print_str        # Function call
    move    $a0,$s0
    sw      $s0,-4($sp)      # This will be parameter 1
    jal     fact             # Function call
    move    $s0,$v0
    la      $a0,_result_string
    sw      $a0,-4($sp)      # This will be parameter 1
    jal     print_str        # Function call
    move    $a0,$s0
    sw      $s0,-4($sp)      # This will be parameter 1
    jal     print_int        # Function call
    add     $a0,$0,10
    sb      $a0,-4($sp)      # This will be parameter 1
    jal     print_char       # Function call
    lw      $ra,-4($fp)      # Load return address (ra)
    lw      $s0,-8($fp)      # Load register $s0
    lw      $fp,0($fp)       # Restore frame pointer
    addu    $sp,$sp,28       # Pop stack frame
    jr      $ra              # Return from function
### End of function main ###

### Beginning of function fact ###

fact:
    # This will be the stack frame for this function
    # Old $fp stored in 0($fp)
    # Parameters: 1
        # Par 1 (in) stored in -4($fp) uses register $a0
    # Saved registers: 4 bytes
    # Scalars: 3
        # Scalar (previous) stored in -12($fp) uses register $a1
        # Scalar (tmp) stored in -16($fp) uses register $a1
        # Scalar (res) stored in -20($fp) uses register $a0
    # Structs: 0
    # Total stack size: 24 bytes

    subu    $sp,$sp,24       # Space for stack frame
    sw      $fp,20($sp)      # Save old frame pointer
    addu    $fp,$sp,20       # Set up frame pointer
    sw      $ra,-8($fp)      # Save return address (ra)
$L1:
    sw      $a0,-4($fp)      # Save parameter $a0 before call
    sw      $a0,-4($sp)      # This will be parameter 1
    jal     print_int        # Function call
    lw      $a0,-4($fp)      # Restore parameter $a0 after call
    la      $a1,_new_line
```

```
    sw      $a0,-4($fp)     # Save parameter $a0 before call
    move    $a0,$a1
    sw      $a1,-4($sp)     # This will be parameter 1
    jal     print_str       # Function call
    lw      $a0,-4($fp)     # Restore parameter $a0 after call
    bgt     $a0,1,$L2
$L3:
    add     $a0,$0,1
    b   $L4
$L2:
    sub     $a1, $a0, 1
    sw      $a0,-4($fp)     # Save parameter $a0 before call
    move    $a0,$a1
    sw      $a1,-4($sp)     # This will be parameter 1
    jal     fact            # Function call
    lw      $a0,-4($fp)     # Restore parameter $a0 after call
    move    $a1,$v0
    mul     $a0, $a0, $a1
$L4:
    move    $v0,$a0
    lw      $ra,-8($fp)     # Load return address (ra)
    lw      $fp,0($fp)      # Restore frame pointer
    addu    $sp,$sp,24      # Pop stack frame
    jr      $ra             # Return from function
### End of function fact ###
```

Exemple 5.11 Codi MIPS equivalent

5.6 - Conclusions

El procés de generació de codi màquina ha de tenir en compte moltes característiques de l'arquitectura. Afortunadament, les instruccions del codi màquina MIPS són molt similars a les instruccions del llenguatge IC, i això facilita molt el procés de traducció.

L'algorisme de generació de codi escollit és un algorisme molt senzill, donat que no es pretenia generar un codi màquina amb una eficiència màxima. Aquest algorisme podria ser millorat afegint una fase d'optimització de finestra, dependent de l'arquitectura, o generant codi màquina a nivell de bloc bàsic.

El capítol 6 mostra algunes optimitzacions dependents de l'arquitectura. Aquestes optimitzacions intenten aprofitar al màxim altres característiques de l'arquitectura (segmentació, instruccions especials de prefetch per a accelerar els accessos a memòria...).

6 Altres optimitzacions

Aquest capítol intenta donar una visió d'altres optimitzacions de codi intermedi que no han estat implementades en aquest projecte. Com ja s'ha vist al capítol 3, sempre és possible trobar noves optimitzacions de codi, i per tant, el llistat d'aquest capítol no serà complet.

Algunes de les tècniques introduides en aquest capítol difereixen una mica de les tècniques del capítol 3 perquè s'apliquen a nivell de bloc bàsic (local) o de programa (interprocedural). A més d'aquestes optimitzacions independents de l'arquitectura, també s'introdueixen breument optimitzacions dependents de l'arquitectura.

Totes les optimitzacions d'aquest capítol constitueixen el treball futur a realitzar en **CODEGEN**.

6.1 - Desenrotllat de bucles ("loop unrolling")

Una part del temps d'execució d'un bucle es consumeix realitzant salts per a tornar a començar el bucle, tornant a comprovar la condició de finalització... Aquest temps pot ser reduit utilitzant el desenrotllat de bucles. Aquesta tècnica **replica (repeteix) el codi d'un bucle un cert nombre de vegades, de manera que alguns salts i altres instruccions poden ser eliminats**. Per a poder eliminar aquestes instruccions, caldrà coneixer algunes dades sobre el bucle (com ara el número d'iteracions); com més dades es coneguin sobre el bucle en temps de compilació, més efectiu resultarà el desenrotllat del bucle.

```
      i:= 0
.i0:  .if i >= n .goto .i2
      càlcul
      i:= i+1
      .goto .i0
```

(a) Bucle inicial: realitza un cert càlcul n vegades. El volem desenrotllar 4 vegades.

```
     i:= 0
.i0: .if i >= n .goto .i2
     càlcul
     i:= i+1
     .if i >= n .goto .i2
     càlcul
     i:= i+1
     .if i >= n .goto .i2
     càlcul
     i:= i+1
     .if i >= n .goto .i2
     càlcul
     i:= i+1
     .goto .i0
```

(b) Bucle desenrotllat: el programa és més llarg, però ara s'han eliminat molts salts del tipus .goto .i0. Concretament, només s'executa un d'aquests salts cada 4 iteracions del bucle.

```
     i:= 0
.i0: .if i >= n .goto .i2
     càlcul
     i:= i+1
     càlcul
     i:= i+1
     càlcul
     i:= i+1
     càlcul
     i:= i+1
     .goto .i0
```

(c) Més optimitzacions: si el nombre d'iteracions (n) és una constant coneguda en temps de compilació, podem comprovar en temps de compilació si és múltiple de 4 o no. Si resulta que és múltiple de 4, podem eliminar les comprovacions de la condició de final del bucle, i només realitzar una comprovació cada 4 iteracions del bucle. D'aquesta manera, eliminem moltes més instruccions de salt.

```
     i:= 0
.i0: .if i >= n .goto .i2
     càlcul
     càlcul
     càlcul
     càlcul
     i:= i+4
     .goto .i0
```

(d) Encara més optimitzacions: si "càlcul" no utilitza la variable d'inducció " i ", es poden agrupar tots els increments d'aquesta variable.

Exemple 6.1 Desenrotllat d'un bucle

Aquesta optimització provoca un increment en l'espai ocupat pel codi del programa, malgrat que redueix el temps d'execució si s'aplica correctament. La reducció de temps produïda pel desenrotllat d'un bucle pot venir de dos fonts:

- fonamentalment, de l'**eliminació d'instruccions de salt**

- en alguns casos, de l'**acumulació dels increments de les variables d'inducció**.
- la **paral·lelització del bucle**, ja que quan desenrotllem un bucle tenim més instruccions a cada iteració i resulta més fàcil paral·lelitzar

Per altra banda, el desenrotllat de bucles pot tenir inconvenients. Si el bucle que desenrotllem és molt gran, el desenrotllat podria produir una disminució de l'eficiència. Això pot passar perquè les instruccions del programa s'emmagatzemen en una memòria petita i d'accés molt ràpid (cache) que es troba al processador. **Si desenrotllem un bucle molt gros, pot ser que el bucle original capigués sencer a la cache, però el bucle desenrotllat no**; en aquest cas, caldria accedir molts cops a la memòria per a llegir les instruccions. Com que la memòria és molt més lenta que la cache, aquest programa tindria un temps d'execució més gran que el programa original ([DAV96]).

Llavors quins bucles s'haurien de desenrotllar? Com a requisit, els bucles haurien de tenir alguna variable d'inducció, i el número d'iteracions del bucle hauria de ser conegut (en temps de compilació o en temps d'execució). Aquests bucles són bons candidats per a ser desenrotllats, però cal tenir en compte que aquesta optimització dóna bons resultats en bucles amb poques instruccions.

Es poden trobar més dades sobre aquesta optimització a [DAV96] i [APP98].

6.2 - Anàlisi d'àlies

Una aproximació possible quan es tracten dades a memòria és l'aproximació conservadora: considerar que una escriptura a memòria pot definir qualsevol posició de la memòria i que una lectura de memòria pot consultar qualsevol posició de la memòria. Aquesta aproximació conservadora dificulta l'anàlisi del fluxe de dades, perquè molts accessos a memòria només consulten una posició molt determinada (o un rang molt petit de la memòria).

L'anàlisi d'àlies intenta determinar de forma més exacta les posicions de memòria afectades per cada accés a memòria. D'aquesta manera, les optimitzacions de fluxe de dades tindran més oportunitats per a optimitzar els accessos a memòria. Això ens permetrà, per exemple, optimitzar les instruccions que treballen amb variables globals que es trobin a memòria.

Per a fer aquest anàlisi, **es divideix la memòria en un conjunt de zones lògiques anomenades àlies**. En funció de la profunditat desitjada en aquest anàlisi tindrem més o menys àlies. Per exemple, si volem un anàlisi total, podríem tenir un àlies per a cada variable a memòria; si volem un anàlisi menys complet però més ràpid, podem tenir un àlies per a cada classe de variables (un per a variables globals,un altre per als vectors d'una funció,...).

Cada objecte de la memòria té assignat un àlies, que com hem vist pot ser únic per ell o comú a totes les variables de la seva mateixa classe. **Un punter té assignat una llista d'àlies: el conjunt d'àlies als que podria estar apuntant**. La manera d'usar aquests conjunts d'àlies és conservadora:

- Si dues operacions tenen només un àlies associat, que és el mateix en les dos operacions i a més cada àlies apunta només a un sol objecte de la memòria, significa que **les dues**

operacions estan accedint a la mateixa posició de la memòria: les dues operacions es poden tractar com si treballesin amb la mateixa variable.

- Si dues operacions comparteixen algun àlies, significa que **PODRIEN estar accedint a la mateixa posició de memòria**. No podem garantir que accedeixin a la mateixa posició però tampoc podem garantir que no ho fagin.
- Si dues operacions no comparteixen cap àlies, llavors **no accedeixen a la mateixa posició de la memòria**: es poden tractar com si modifiquessim variables diferents.

```
c := &b   # àlies (c) = {b}
*c := 0   # àlies (c) = {b}
b := 0    # àlies (b) = {b}
```

(a)

```
.i0:
c := &b   # àlies (c) = {b}
.goto .i2
.i1:
c := & d  # àlies (c) = {d}
.i2:
*c := 0   # àlies (c) = {b,d}
b := 0    # àlies (b) = {b}
```

(b)

Exemple 6.2 Exemples d'anàlisi d'àlies

L'exemple 6.2 ens mostra dos possibles situacions obtingudes amb l'anàlisi d'àlies. Suposem que l'objectiu és descobrir si la instrucció *c:= 0 és codi mort i pot ser eliminada o no. En el cas (a), tenim que el punter c només pot apuntar a la variable b segons l'anàlisi d'àlies, i per tant, *c:= 0 és codi mort. En canvi, en el cas (b) el punter c pot apuntar a dues variables: b o d. Malgrat que c podria estar apuntant a b, no em podem estar segurs i considerarem que *c:= 0 no és codi mort.

Les llistes d'àlies de cada apuntador es poden calcular amb equacions recursives com es fa en tots els problemes d'anàlisi del fluxe de dades. De tota manera, hi ha algunes coses diferents:

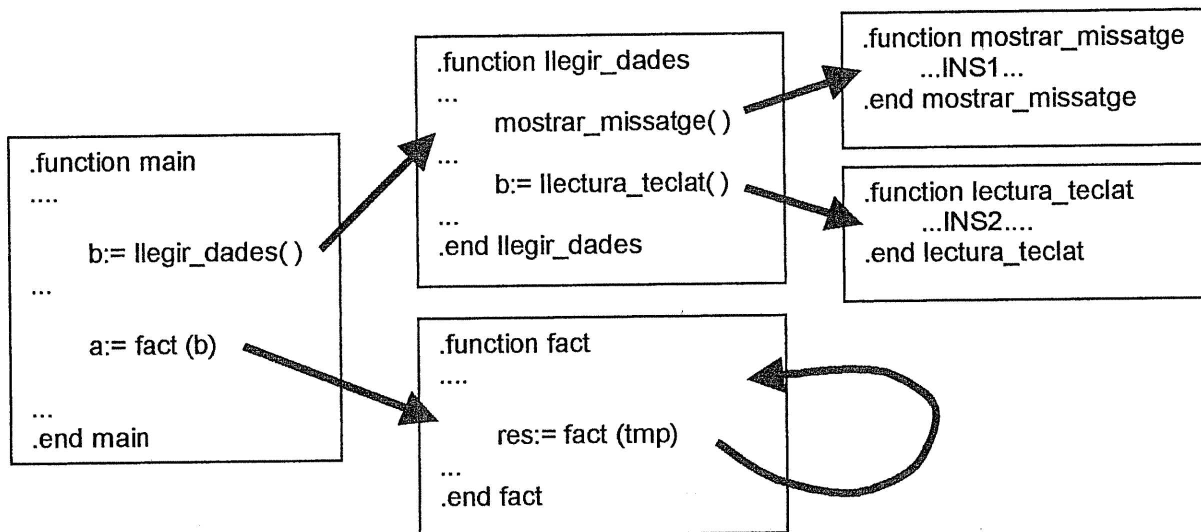
- Es poden rebre puntes com a paràmetre d'un procediment. Com es pot saber on apunten aquests punters? Hi ha dues solucions.
  - Considerar un àlies QUALSEVOL_ADREÇA, que indiqui que aquests punters poden apuntar a qualsevol posició de la memòria. Això vol dir que si comparem un àlies amb QUALSEVOL_ADREÇA, sempre considerarem que són iguals.
  - Realitzar l'anàlisi d'àlies a nivell interprocedural. Per fer això cal seguir les crides entre funcions construint un graf de crides (veure 6.3). En aquests casos habitualment és realitzat **tot** l'anàlisi del fluxe de dades a nivell interprocedural: propagació de còpies i constants entre procediments, expressions comunes entre procediments...
- Cal definir un tractament per a operacions com punter:=constant, punter:=punter+variable, punter := variable_no_punter, i altres casos similars. En aquestes situacions no és trivial decidir quins àlies hem d'assignar als punters. Es pot decidir, per exemple, que la llista d'àlies passi a ser QUALSEVOL_ADREÇA. En qualsevol cas, s'ha de mantenir una política conservadora: dos punters que puguin accedir a una mateixa adreça haurien de compartir algun àlies.

Es poden trobar més dades sobre aquesta optimització en les referències [AHO90] i [APP98].

6.3 - Optimització interprocedural

Totes les optimitzacions explicades fins ara es troben en l'àmbit d'una funció. Hi ha moltes altres tècniques que intenten aplicar optimitzacions a tot el programa com a conjunt, sense estar limitades pels límits de les funcions.

El primer pas de l'anàlisi interprocedural és la construcció d'un **graf de crides**, on cada node és una funció i hi ha un arc entre dues funcions f1 i f2 si f1 crida f2. Aquest graf permet detectar situacions com crides entre funcions, crides recursives, ... i per aquest motiu constitueix l'eina bàsica per a l'optimització interprocedural.

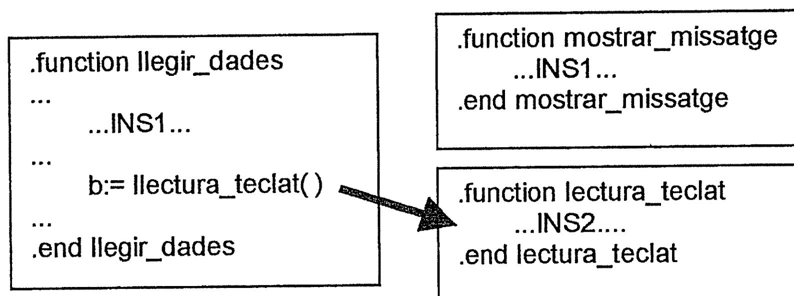


Exemple 6.3 Graf de crides d'un programa

Partint d'aquest graf de crides, es poden realitzar les següents optimitzacions:

- **Expansió de funcions ("function inlining")**

  Realitzar una crida a funció té un cert cost. A 5.3.2 hem vist que **la crida a una funció porta implícita el pas de paràmetres, la construcció del bloc d'activació al fer la crida i l'alliberament del bloc d'activació al final de la crida**. Una manera d'eliminar aquest cost consisteix en **expandir** la funció: en els llocs on es fa una crida a la funció es replica el codi de la funció eliminant el codi que realitza la crida la crida.



Exemple 6.4 Graf de crides de l'exemple 6.3 on s'ha expandit la funció "mostrar_missatge"

Aquesta optimització fa crèixer el tamany del programa, encara que millora el temps d'execució. Per tant cal tenir en compte dos factors quan volem aplicar l'expansió de funcions:

- **tamany de la rutina**: aquest procediment dón bons resultats amb funcions no recursives que tinguin un tamany comparable al codi necessari per fer la crida
- **freqüència de crida**: si aquesta rutina era cridada moltes vegades (per exemple, dins un bucle) la pèrdua d'espai pot ser compensada per la millora en l'eficiència.

El millor seria disposar d'informació ("**profiling**") sobre el temps d'execució del programa. Això permetria triar millor quines són les rutines que convé expandir. Una altra possibilitat és permetre al programador que indiqui quines són les funcions que creu que són més adequades per a expandir. Això és el que fa el llenguatge C amb la directiva `inline`.

- **Anàlisi interprocedural de fluxe de dades**

  Com ja s'ha vist al 6.2, les optimitzacions del fluxe de dades, juntament amb l'anàlisi d'àlies es poden extendre més enllà de les funcions. D'aquesta manera, es pot detectar codi mort en altres funcions, propagar constants entre procediments, ... El major avantatge d'aplicar aquest anàlisi a nivell interprocedural és la inclusió dels accessos a memòria en l'anàlisi gràcies a l'anàlisi d'àlies.

- **Altres optimitzacions**

  A partir del graf de crides hi ha més situacions que poden ser optimitzades, però ja són dependents en certa manera de l'arquitectura. Per exemple, si es detecta que una rutina és recursiva i les convencions de crides a rutines ho permeten, podem pensar a passar més paràmetres per registre per a reduir el temps d'execució. Un altre exemple és que si una funció no realitza crides a cap altra funció llavors no cal que salvi els registres salvats al principi de la funció (es poden considerar com registres temporals).

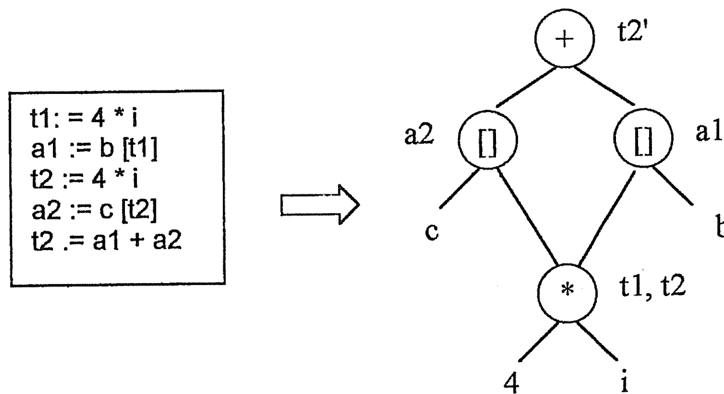
Algunes referències d'optimitzacions interprocedurals es poden trobar a [FIS88] (expansió de funcions i altres) i [AHO90] (anàlisi interprocedural del fluxe de dades).

6.4 - Optimització local

Moltes optimitzacions del fluxe de dades es poden aplicar simultàneament a les instruccions d'un bloc bàsic utilitzant una tècnica d'optimització local: **la generació de codi a partir del DAG del bloc bàsic**.

Un DAG (graf dirigit acíclic) associat a un bloc bàsic es construeix de la forma següent:

- Les fulles del graf són identificadors (constants o variables). Cada identificador està representat per una única fulla.
- Els nodes interns del graf són operacions, i tenen associats una llista d'identificadors. El seu significat és *"tots els identificadors de la llista s'obtenen aplicat l'operació associada al node als nodes predecessors en el graf"*.
- Cada cop que una instrucció calcula un nou valor per a una variable genera una nova "versió" d'aquella variable. Totes les instruccions posteriors utilitzaran la darrera versió de la variable.



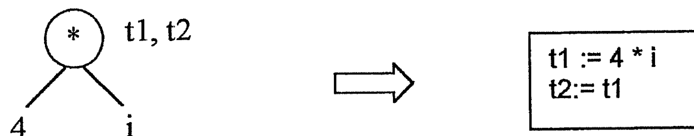
Exemple 6.4 Un bloc bàsic i el seu DAG associat

L'exemple 6.4 ens mostra el DAG associat a un bloc bàsic. Aquest DAG representa totes les expressions del bloc bàsic de manera que:

- **És clarament visible el fet que dues variables continguin la mateixa expressió.** Per exemple, t1 i t2 contenen la mateixa expressió i això es pot detectar veient que les dos variables estan associades al mateix node del DAG.
- **El DAG mostra les dependències existents en l'ordre en que es poden avaluar les instruccions**: si un node és el fill d'un altre node en el DAG, s'ha de calcular abans; si dos nodes no tenen res a veure es poden calcular en qualsevol ordre. Per exemple, el node de a1 s'ha de calcular abans que el node de t2', però els nodes a1 i a2 es poden calcular en qualsevol ordre.

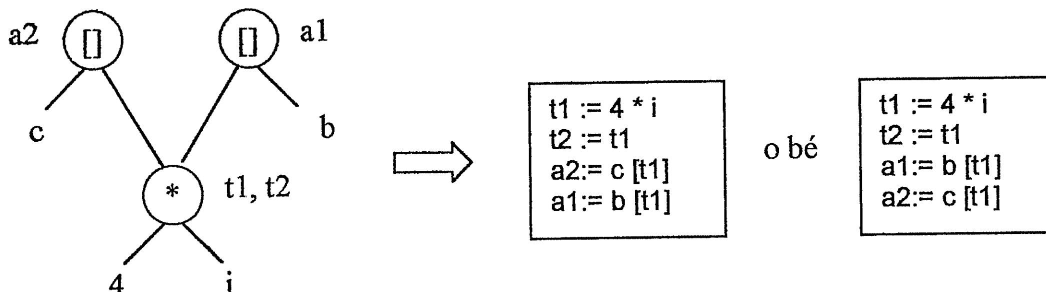
Gràcies al DAG es poden realitzar moltes optimitzacions del fluxe de dades en el bloc bàsic:

- **eliminació de subexpressions comunes**: si un node té associat més d'una variable que conté l'expressió, només s'ha de calcular l'expressió per a una de les variables i generar còpies per a la resta.



Exemple 6.5 Eliminació de subexpressions comunes en un DAG

- **intercanviar l'ordre d'avaluació d'instruccions**: Ja s'ha vist que és completament indiferent l'ordre en que es generen els nodes que no són descendents l'un de l'altre.



Exemple 6.6 Intercanviar l'ordre d'avaluació de nodes del DAG

- **eliminació de codi mort:** Només cal generar codi per a l'última versió de cada variable en el DAG. Per exemple, en l'exemple 6.4 hi ha dos versions de la variable t2: t2 i t2'. La darrera versió és t2', i per tant no cal generar el codi per a t2 que hem vist a l'exemple 6.5. Només caldria afegir al codi de l'exemple 6.6 la instrucció t2:= a2+a1 i eliminar t2:=t1.

Aquestes optimitzacions es poden realitzar a nivell global, i resulten més eficients ja que no es limiten a un sol bloc bàsic. A més, els blocs bàsics són petits en general, i per tant contenen poques instruccions. Tot això fa que el nivell d'optimització aconseguit si apliquem **només** aquestes tècniques sigui petit. De tota manera, aquesta tècnica s'acostuma a aplicar per a generar codi màquina, utilitzant al màxim la possibilitat d'intercanviar l'avaluació de les instruccions per a generar un codi màquina el més eficient possible.

Una altra possibilitat seria generar codi màquina i realitzar assignació de registre a nivell de local (de bloc bàsic). Això pot produir un codi menys eficient però es pot fer servir en un compilador senzill, o per a generar codi ràpidament si l'usuari no demana optimitzacions.

L'optimització a nivell de bloc bàsic es descriu a [APP98] i molt àmpliament a [AHO90], on detalla la generació de codi màquina a partir del DAG.

6.5 - Optimitzacions dependents de l'arquitectura

Existeix un gran nombre de transformacions que es poden realitzar (abans o després de generar codi màquina) si es tenen en compte les característiques de l'arquitectura. Aquest és un petit llistat d'aquestes transformacions, que no pretèn ser exhaustiu.

6.5.1 - Optimització de finestra

Aquesta optimització intenta cercar patrons d'instruccions en el codi màquina que es puguin substituir per patrons més eficients. El procediment per fer-ho és el següent: s'explora el programa seqüencialment, fixant l'atenció en un cert nombre d'instruccions (la finestra). Si la finestra conté algun patró dels indicats anteriorment, el substituim per un patró més eficient.

Aquests patrons intentar optimitzar diferents aspectes del codi:

- **salts**: alguns exemples de patrons es poden trobar a l'apartat 3.8.9

- **còpies**: intentar eliminar moviments innecessaris entre registres.

```
move r1, r1      →     (cap instrucció)

move r2, r1      →     move r2, r1
move r1, r2
```

- **operació d'operands constants en temps de compilació**: alguns exemples de patrons es poden trobar a l'apartat 3.8.5.

```
li   r1, const1       →      li r1, const
add r2,r1,const2             li r2, (const1+ const2)
```

- **reducció d'intensitat**: subtituir instruccions cares per altres més barates (veure 3.8.5).

```
mul r1, r2, 2^m       →      sll r1, r2, m    # Producte→ desplaçament
```

- **simplificació algebraica**: utilitzar propietats com ara l'element neutre i l'element oposat per a simplificar sumes, productes, ... (veure 3.8.5).

```
mul r1, r2, 1         →      move r1, r2
add r1, r2, 0         →      move r1, r2
```

- **accessos a memòria**: evitar carregar i salvar dades a memòria un nombre excessiu de vegades. Si salvem un valor a memòria i immediatament el tornem a restaura, podem aprofitar el valor que tenim guardat a registre i no cal que repetim l'accés a memòria.

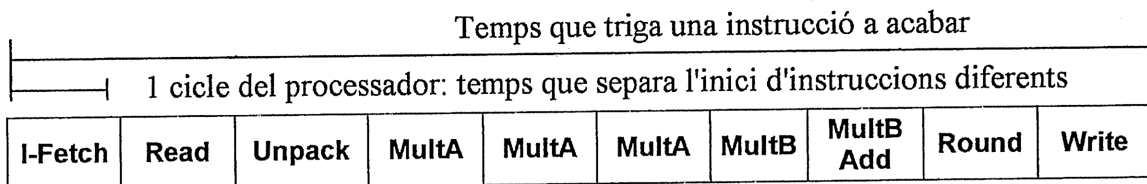
```
sw r1, const(r2)      →      sw r1, const(r2)
lw r3, const(r2)             move r3,r1
```

- **ús d'instruccions especials del codi màquina**: utilitzar instruccions com ara l'increment (sumar 1 de manera més eficient que una operació de suma), els modes d'accés d'autoincrement o autodecrement,... No tenim cap exemple en MIPS, però sí que en tenim al codi màquina del Intel 80x86.

```
ADD EAX, 1            →      INC EAX
```

6.5.2 - Optimització per a arquitectures segmentades

Molts processadors moderns utilitzen una tècnica anomenada **segmentació** per a accelerar l'execució d'instruccions. Aquesta tècnica es basa en dividir l'execució d'una instrucció en segments: un segment per a llegir la instrucció de memòria, un segment per a llegir els operands, etc. Un processador segmentat pot executar segments d'instruccions diferents al mateix temps: pot estar guardant en registre els operands d'una instrucció mentre llegeix els operands d'una altra.



(1) Lectura de la instrucció
(2) Lectura dels operands de registre
(3) Recuperar exponent i mantissa
(4), (5), (6) Iniciar multiplicació
(7),(8) Final de la multiplicació
(8) Sumar resultats
(9) Arrodoniment
(10) Escriptura dels resultats

Exemple 6.7 Segmentació per al producte en el processador MIPS R4000 [APP98]

El resultat de la segmentació és que **el processador pot començar a executar una instrucció abans d'acabar d'executar l'anterior** (veure a l'exemple 6.7 la diferència entre la durada d'una instrucció i la durada d'un cicle del processador). Això ho pot fer sempre que la instrucció següent no depengui, d'alguna manera, de la instrucció anterior. Per exemple, si una instrucció calcula el valor del registre $a0 i la següent utilitza el valor d'aquest registre, la següent no pot realitzar el càlcul fins que el valor del registre $a0 estigui disponible. Un altre problema pot ser que moltes instruccions vulguin utilitzar el tipus de component (per exemple, un multiplicador) i no hi hagi prou unitats d'aquell tipus (anomenades unitats funcionals) per a realitzar operacions. Aquests dos tipus de situacions es coneixen com dependències de dades i dependències de recursos.

I	R	U	MA	MA	MA	MB	A	RO	W		
	I	R	U	MA	MA	MA	MB	A	RO	W	
		I	R	U	MA	MA	MA	MB	A	RO	W

(a) Situació ideal: aprofitem al màxim la segmentació. La notació utilitzada és la mateixa que a l'exemple 6.7, escrivint només la inicial dels segments.

I	R	U	MA	MA	MA	MB	A	RO	W									
	I									R	U	MA	MA	MA	MB	A	RO	W

(b) Dependència de dades: la segona multiplicació utilitza com a operand el resultat de la primera instrucció. Per aquest motiu la lectura d'operands (R) de la segona multiplicació no pot començar fins que l'escriptura (W) de la primera instrucció acabi. Tots els cicles que passen són cicles perduts (ratllat).

I	R	U	MA	MA	MA	MB	A	RO	W			
	I	R	U			MA	MA	MA	MB	A	RO	W

(c) Dependència en els recursos: suposem que només hi ha un multiplicador per a realitzar el segment MA. La segona multiplicació ha d'esperar que la primera deixi lliure el multiplicador. D'aquesta manera, tornem a perdre temps.

Exemple 6.8 Efecte de les dependències en un processador segmentat

Aquestes dependències entre instruccions no permeten aprofitar la segmentació al màxim, al tenir que realitzar esperes fins que les dependències no tenen efecte (veure exemple 6.8). **Una forma d'eliminar l'efecte de les dependències es reordenant les instruccions en el programa**: si reordenem les instruccions de manera que les instruccions amb dependències entre sí estiguin molt separades, reduirem l'efecte de les esperes. Una mostra d'això es pot veure a l'exemple 6.8 (c): perdem dos cicles fent espera. Què passaria si haguèssim començat el segon producte dos cicles més tard i en aquests dos cicles haguéssim començat a executar instruccions que no tenen dependències amb cap de les multiplicacions? No caldria fer cap espera, perquè el multiplicador per al segment MA estaria lliure quan la segona multiplicació el volgués utilitzar.

El problema de trobar un ordre adequat per a executar les instruccions es coneix com **planificació** ("scheduling") **d'instruccions**. Trobar un ordre òptim és molt costós i per aquest

motiu es busquen solucions aproximades. La majoria d'algorismes de planificació fan referència als bucles, perquè són els fragments de codi més crítics per a la eficiència del programa. Aquests algorismes han de conèixer la segmentació de cada tipus d'instrucció del codi màquina i el número d'unitats funcionals que estan disponibles per a executar cada segment.

Una referència on es poden trobar més detalls sobre la segmentació i les tècniques de planificació és [BOC95]. [AHO98] i [WAN96] contenen tots dos dades sobre la segmentació i algorismes adequats per a planificació sobre bucles i bucles aniuats, respectivament.

6.5.3 - Optimització dels accessos a memòria

Un accés a memòria pot representar molts cicles de processador. Per a evitar aquests retards, en algunes arquitectures existeix un tipus d'instrucció especial, anomenat pre-cerca **(prefetch adreça)**, que **indica al processador que en el futur accedirem a una determinada posició de la memòria.** Quan rep aquesta instrucció, el processador comença a accedir a memòria per a portar aquesta posició de la memòria a la memòria cache (d'accés molt ràpid) del processador. Quan arribem a la instrucció en que consultem aquella posició de la memòria, caldrà només accedir a la cache per a consultar l'operand. No és segur que l'operand ja hagi arribat a la cache, però el temps que haurem d'esperar serà menor que si haguèssim iniciat l'accés en la instrucció actual.

És responsabilitat del compilador usar aquestes instruccions de prefetch de manera adient per a minimitzar el temps que el processador està esperant dades de la memòria. Per a prendre aquestes decisions cal conèixer com està organitzada la memòria i la cache en l'arquitectura concreta, quin és el significat concret de prefetch, i el temps que es triga a accedir a la memòria.

Es pot trobar més informació sobre aquesta optimització a [APP98].

6.6 - On s'ha arribat?

Després de veure les optimitzadors realitzades i altres optimitzacions, una pregunta lògica és: fins on s'ha arribat? Aquest apartat intenta il·lustrar les tècniques d'optimització utilitzades en compiladors d'ús comercial, per a comparar-les amb les tècniques d'optimització utilitzades per **CODEGEN**.

El compilador escollit per a fer la comparació ha estat el compilador de C de GNU, **gcc**, en la seva versió 1.12 (egcs-1.1.2). Aquest compilador suporta diferents nivells d'optimització (-O1,-O2,-O3): els dos primers (-O1, -O2) representen optimitzacions que redueixen el temps d'execució del programa sense augmentar-ne l'espai; l'últim (-O3) representa les optimitzacions que redueixen el temps d'execució però augmenten l'espai ocupat.

La següent taula indica quines optimitzacions són realitzades per **gcc** i **CODEGEN**, respectivament. Aquesta taula no inclou les optimitzacions dependents de l'arquitectura (optimitzacions de finestra, optimitzacions de salts retardats, reordenació d'instruccions per

afavori arquitectures segmentades, ...). Aquestes optimitzacions es realitzen a **gcc** però no a **CODEGEN**.

Nivell d'optimització -O1	gcc	CODEGEN
Optimització de salts condicionals ("jump threading")	X	
Nivell d'optimització -O2	**gcc**	**CODEGEN**
Optimització de salts	X	X
Eliminació de codi inabastable	X	X
Operació de constants en temps de compilació	X	X
Propagació de constants	Interprocedural	Intraprocedural
Propagació de còpies	Interprocedural	Intraprocedural
Eliminació de subexpressions comunes	Interprocedural	Intraprocedural
Extracció d'expressions invariants de bucles	X	X
Reducció d'intensitat de variables d'inducció	X	X
Eliminació de variables d'inducció	X	X
Eliminació de codi mort	X	X
Anàlisi d'àlies	Interprocedural	
Eliminar instr. de còpia en l'assignació de registres	X	X
Nivell d'optimització -O3	**gcc**	**CODEGEN**
Desenrotllat de bucles ("loop unrolling")	X	
Expansió de funcions ("function inlining")	X	
Altres optimitzacions	**gcc**	**CODEGEN**
Optimitzacions dependents de l'arquitectura	X	
Transformacions de bucles while...do en do...while	?	X
Transformació a Unica Assignació Estàtica (SSA)		X

L'únic propòsit d'aquesta taula és mostrar que les tècniques implementades en **CODEGEN** són utilitzades en el món dels compiladors comercials, i que representen un conjunt raonable de les optimitzacions implementades de forma independent de l'arquitectura. **Aquesta taula NO intenta comparar CODEGEN amb gcc.** **CODEGEN** té moltíssimes limitacions en comparació a **gcc**, que no es veuen reflexades en aquesta taula:

- El codi intermedi IC és un codi molt senzill. El codi intermedi utilitzat per **gcc** (RTL) és més complex que el codi IC, ja que permet parametritzar l'arquitectura destí (número i tipus dels registres, dependències entre instruccions i registres, excepcions que pot produir una instrucció, ...).
- Com més complexes són les instruccions d'un codi, més complex és realitzar optimitzacions sobre aquest codi. Per aquest motiu, implementar optimitzacions sobre el codi IC és una tasca més senzilla que fer-ho sobre un codi intermedi més potent com RTL.
- Pel que fa a les optimitzacions dependents de l'arquitectura, gcc no es limita a les optimitzacions intraprocedurals. L'anàlisi del fluxe de dades es realitza a nivell interprocedural, i es realitza expansió de funcions ("function inlining").
- **gcc** realitza una tasca molt important d'optimització dependent de l'arquitectura, amb algorismes de sel·lecció d'instruccions eficaços, schedulling d'instruccions, optimitzacions de finestra, ...
- A **CODEGEN** el tractament dels tipus s'ha simplificat molt. Per exemple, quan es genera codi per al llenguatge IC es suposa que un escalar és prou petit per a guardar-se en un registre, i això no té perqué ser així en tots els casos. A més, els tipus de dades en el llenguatge IC són molt senzills (int, real, char, bool, pointer). En canvi, **gcc** realitza un

tractament de tipus molt més exhaustiu i aprofita els recursos que proporciona l'arquitectura concreta per a realitzar el tractament de les instruccions de punt flotant, etc.

- Finalment, moltes altres característiques, com ara l'eficiència, la documentació (gcc segueix els estàndards GNU de documentació), la portabilitat, ... converteixen a gcc en un compilador d'ús comercial, fent impossible la comparació amb **CODEGEN**.

6.7 - Conclusions

Aquest capítol presenta algunes optimitzacions (6.1 i 6.3) que redueixen el temps d'execució a canvi d'augmentar l'espai ocupat. La utilització d'aquestes tècniques no pot ser indiscriminada, sino que s'han d'aplicar només en aquells punts on siguin més beneficioses.

Altres optimitzacions, com ara l'optimització local, l'anàlisi d'àlies i l'anàlisi interprocedural proporcionen un guany en temps d'execució sense oferir res a canvi, i haurien de ser utilitzades sempre que el temps de compilació no fos crític.

Totes aquestes optimitzacions no tenen en compte les característiques de l'arquitectura, i per tant es poden aplicar sobre el codi intermedi sense necessitat de fixar quin serà el codi màquina destí. En canvi, per a realitzar les optimitzacions dependents de l'arquitectura (6.5) necessitem conèixer detalls concrets del processador, la memòria, etc. en la màquina destí. D'aquesta manera, **les optimitzacions independents de l'arquitectura es poden reaprofitar en qualsevol compilador que utilitzi el mateix codi intermedi**. **Les optimitzacions dependents de l'arquitectura** estan més restringides i **només es poden reaprofitar si utilitzem el mateix codi intermedi i estem generant el mateix codi màquina**. Una possible forma d'evitar l'escàs reaprofitament de les optimitzacions dependents de l'arquitectura podria ser parametritzar l'arquitectura: escriure una optimització genèrica que treballés en base a una especificació de l'arquitectura. Aquesta especificació hauria de ser instanciada per a cada arquitectura concreta, però l'algorisme romandria igual ([GCC]).

Una de les conclusions que s'haurien d'extreure d'aquest capítol és: **sempre és possible aplicar una optimització de codi diferent per a millorar el rendiment dels programes**. Això és molt important, perquè deixa la porta oberta per a investigar nous tipus d'optimitzacions més eficients i més efectives.

7 Disseny del projecte

Aquest capítol mostra el disseny del compilador. En successives etapes, es mostra la descomposició del compilador optimitzador en els mòduls que realitzen cadascuna de les tasques bàsiques del compilador: traducció de codi font a codi intermedi, optimització de codi intermedi, assignació de registres i generació de codi màquina.

El capítol 8 descriu com aquest disseny ha estat implementat, quines estructures de dades s'han utilitzat pel llenguatge IC, etc.

7.1 - Arquitectura de l'entorn

La figura 7.1 presenta l'arquitectura de l'entorn de generació i optimització de codi. L'entorn constarà de dos components: el compilador frontal, **FRONTEND**, i el compilador final, **CODEGEN**.

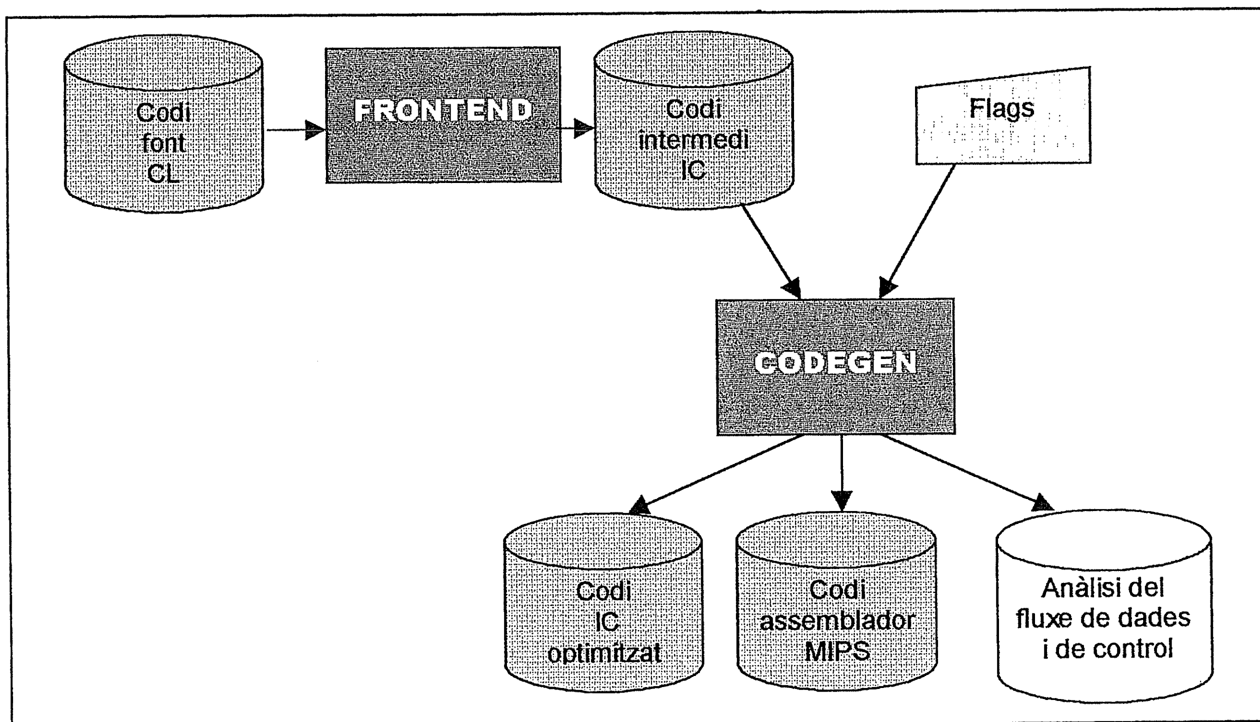


Figura 7.1 Arquitectura de l'entorn

Les dades amb les que treballa l'entorn (programes CL, IC i MIPS, dades del fluxe de dades i control) seran fitxers de text. L'objectiu que es pretenia amb la definició del llenguatge IC era que els programes en aquest llenguatge es puguessin generar manualment. De tota manera, si es volen construir programes més grans, es pot utilitzar el compilador frontal per a generar el codi intermedi equivalent.

Interfície de FRONTEND

Entrada:

⇒ Programa escrit en llenguatge CL (*Fitxer de text*)

Sortida:

⇐ Traducció del programa d'entrada a codi intermedi IC (*Fitxer de text*)

Interfície de CODEGEN

Entrada:

⇒ Programa escrit en llenguatge IC (*Fitxer de text*)

⇒ Flags de compilació que indiquen el tipus de sortida desitjada (IC,MIPS, anàlisi del fluxe de dades i de control) i les optimitzacions a realitzar (*Paràmetres en línea de comandes*)

Sortida:

⇐ Programa IC optimitzat segons els flags (*Fitxer de text*)

⇐ *Opcional*: Anàlisi del fluxe de dades i control en el programa IC (*Fitxer de text*)

⇐ *Opcional*: Codi assemblador de MIPS R2000 generat a partir del codi IC de sortida (*Fitxer de textl*)

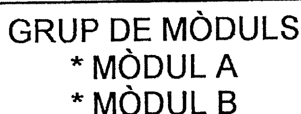
7.2- Format del disseny

La metodologia utilitzada en el disseny ha estat el *disseny estructurat*. Aquesta metodologia genera un conjunt de diagrames dels mòduls de l'aplicació (*diagrames d'estructura*). En un diagrama d'estructura, els mòduls es comuniquen a través de *crides*.

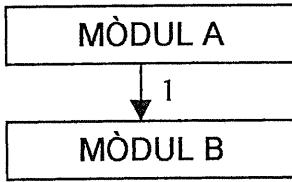
Alguns dels diagrames d'estructura són bastant complexos. Per aquest motiu, en alguns diagrames s'han utilitzat conjunts de mòduls que s'han refinat posteriorment. Els símbols utilitzats en els diagrames d'estructura han estat els següents:

Aquest és el símbol d'un mòdul de l'aplicació

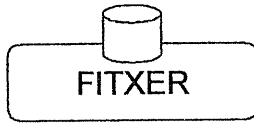
Aquest símbol representa un conjunts de mòduls, agrupats per a simplificar els diagrames d'estructura. Aquests conjunt serà refinat en successius diagrames d'estructura.

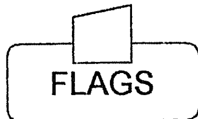


Aquest símbol representa un conjunt de mòduls, format pels mòduls llistats amb un (*). Amb aquest conjunt es pretèn simplificar el diagrama d'estructura.



Aquesta és una comunicació (crida) entre dos mòduls: el mòdul A crida el mòdul B. El número (1) correspon a les operacions cridades entre mòduls.





Dispositius que contenen dades d'entrada o de sortida. El primer representa un fitxer i el segon representa la línea de comandes.

7.3 - Compilador final

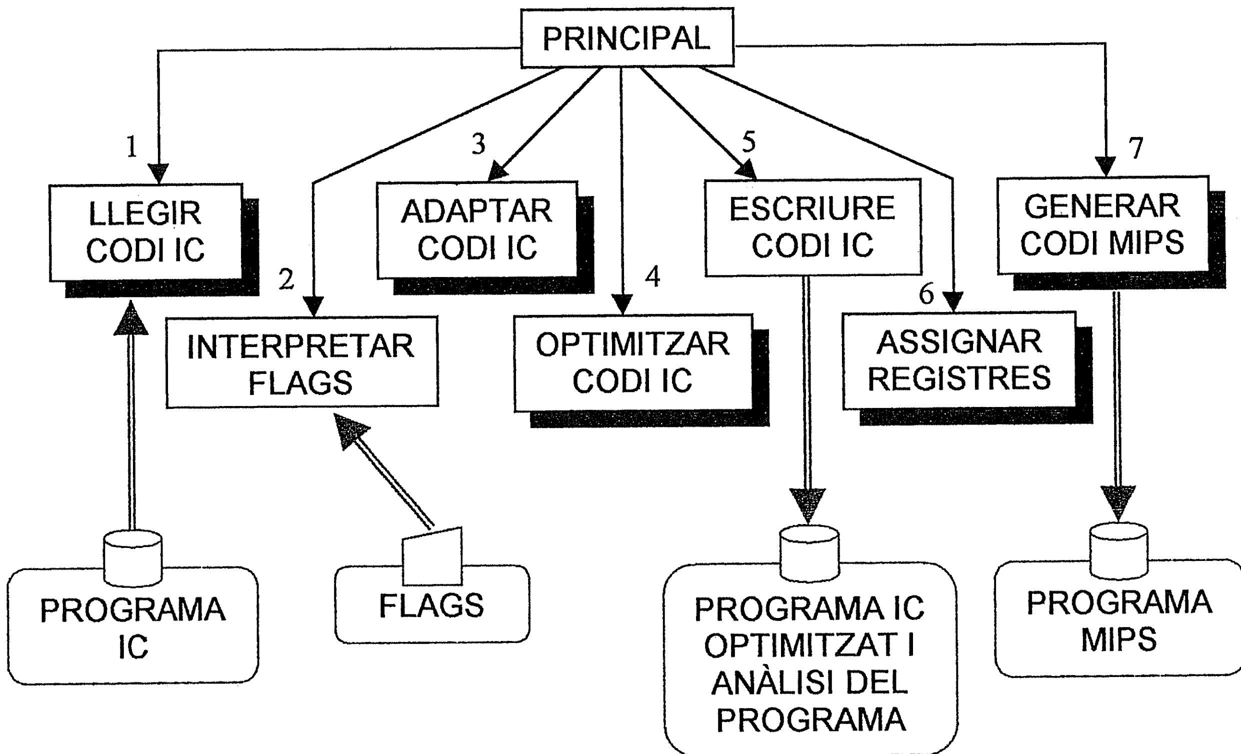


Diagrama d'estructura 7.1 Compilador final

Les tasques que ha de realitzar el compilador final (mòdul PRINCIPAL) són les següents:

- La primera tasca és carregar un programa IC en una estructura de dades en memòria. D'això s'encarrega LLEGIR IC (veure diagrama d'estructura 7.2), llegint el fitxer i carregant les instruccions del programa a memòria.
- Després d'això ha d'intepretar els flags rebuts de l'entrada (mòdul INTERPRETAR FLAG) per a saber les optimitzacions que ha de realitzar i en quin ordre, i quin tipus de sortida espera l'usuari.

- ADAPTAR CODI construeix el graf de fluxe de control (l'estructura de dades utilitzada en tot el compilador final), garanteix que cada rutina tingui un únic RETURN al final de cada funció i determina quines variables han d'estar a memòria perquè es prèn la seva adreça en una instrucció del tipus a:=&b. Després de la crida (3) l'estructura de dades en memòria està llesta per treballar-hi.
- Ara es poden aplicar les optimitzacions a OPTIMITZAR CODI IC (veure diagrama d'estructura 7.4). Un detall que s'ha de considerar és que el resultat de cada optimització continúa sent un programa IC vàlid: tota l'estona estem treballant amb un programa IC vàlid, que anem modificant. Les optimitzacions que s'apliquin en aquesta fase depenen dels flags demanats.
- Després d'optimitzar el codi IC podem escriure aquest codi, així com l'anàlisi del fluxe de dades i de control si l'usuari ho ha demanat amb els flags d'entrada. D'això s'encarrega ESCRIURE CODI IC.
- Si s'ha demanat generar codi màquina, llavors es realitza la fase ASSIGNACIÓ REGISTRES. Després d'això es fa GENERAR CODI MIPS i s'obté el programa resultant.

Un aspecte que convé destacar és que els paràmetre de totes les crides són unicament el programa IC en memòria i els flags. En tots els diagrames d'estructura posteriors, seguirà repetint-se el fet que els paràmetres en les crides a altres mòduls són un programa,una funció o una instrucció IC.

7.3.1 - Lectura del codi IC des de fitxer

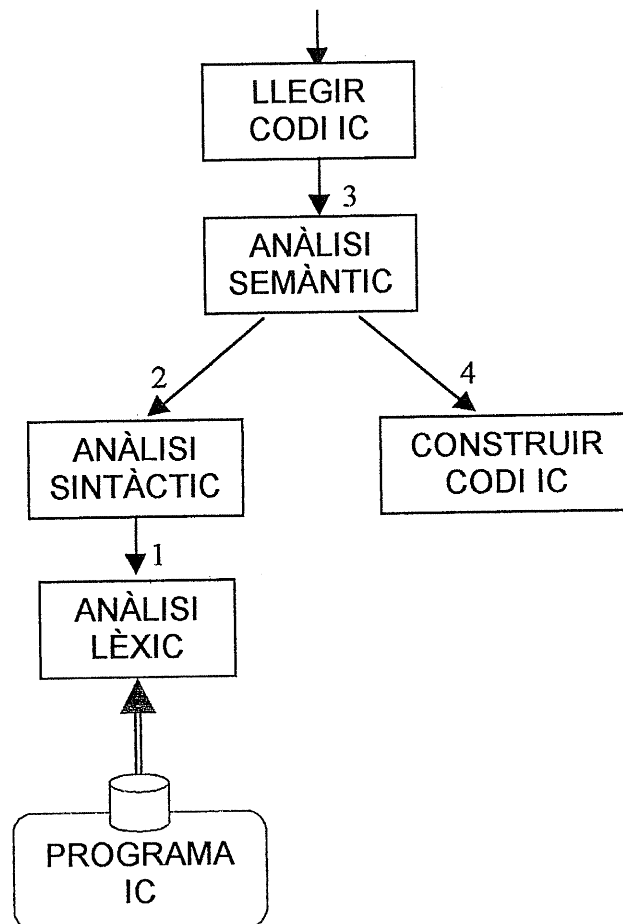


Diagrama d'estructura 7.2 Lectura de codi IC (refina diagrama 7.1)

- Per a llegir un fitxer IC, primer cal realitzar una ANÀLISI LÈXICA. Aquest anàlisi retornarà a (1) els tokens (símbols) reconeguts en l'entrada.

- L'ANÀLISI SINTÀCTICA genera a partir dels tokens rebuts de (1) un arbre sintàctic, que representa les instruccions i expressions del llenguatge IC. Aquest arbre sintàctic es passa a (2) per a la seva utilització posterior.

- Finalment, L'ANÀLISI SEMÀNTICA rep a (2) un arbre sintàctic, que intepreta per a afegir funcions, variables i instruccions IC a l'estructura de memòria. Aquesta anàlisi semàntica utilitza les rutines de CONSTRUIR CODI IC, un mòdul que conté les operacions constructores bàsiques per variables, instruccions, funcions....

- El resultat de la lectura del codi IC és un programa IC carregat en memòria. De tota manera, no està construit el graf de fluxe de control i poden haver-hi diversos RETURNs a la funció. Per poder treballar amb aquest codi IC encara cal realitzar alguns passos.

7.3.2 - Adaptar codi IC

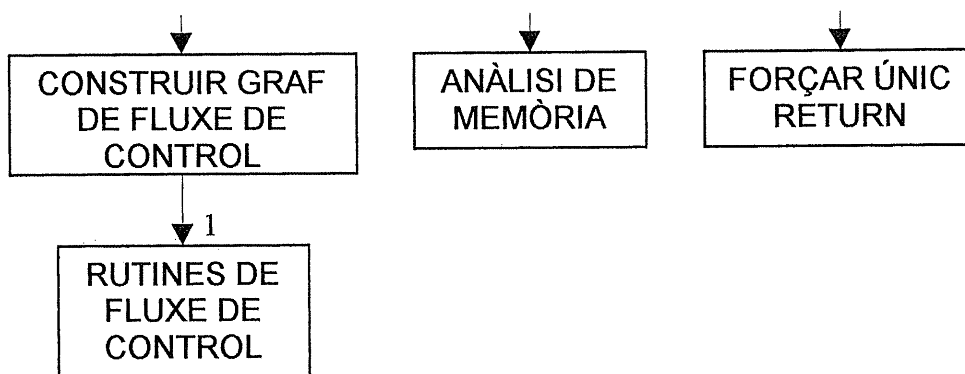


Diagrama d'estructura 7.3 Adaptar codi IC (refina diagrama 7.1)

Tots aquests mòduls reben com a entrada el programa IC, i hi realitzen transformacions necessàries per tal de poder-lo optimitzar fàcilment.

- CONSTRUIR GRAF DE FLUXE DE CONTROL parteix d'un programa que ha estat dividit en blocs bàsics (quan s'ha llegit de fitxer) i connecta aquests blocs bàsics amb els seus sucessors i predecessors. Per a fer-ho, utilitza un mòdul de RUTINES DE FLUXE DE CONTROL que conté rutines per a connectar blocs bàsics.

- ANÀLISI DE MEMÒRIA determina quines variables del programa es guardaran en memòria (p.ex: les variables b que apareguin en instruccions a:=&b s'han de guardar en memòria).

- FORÇAR ÚNIC RETURN obliga a totes les funcions a tenir una sola instrucció de retorn, al final de cada funció. Si una funció té diversos return, es tradueix cadascun d'ells en una instrucció de goto al final de la funció, on es realitza l'únic return.

7.3.3 - Optimització de codi

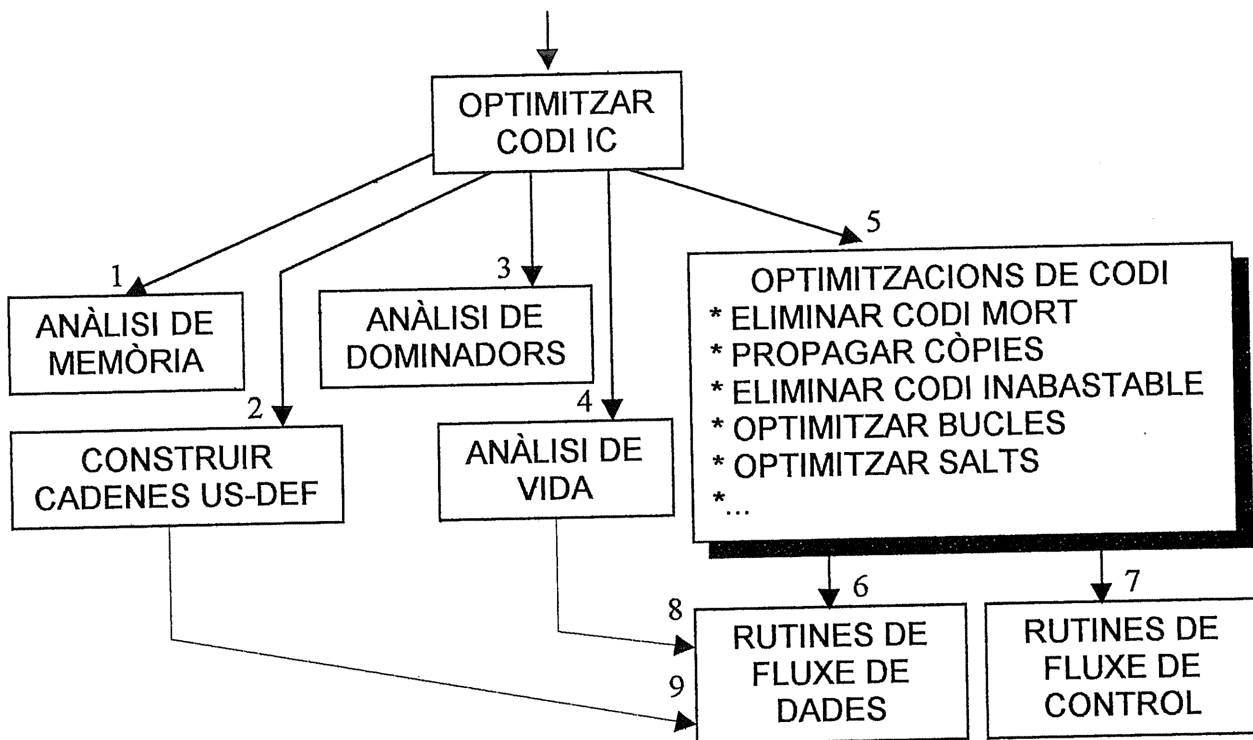


Diagrama d'estructura 7.4 Optimització de codi IC (refina diagrama 7.1)

El procés d'optimització de codi rep com a entrada un programa IC correcte, i la seva sortida és també un programa IC correcte. El tractament realitzat a aquest programa és el següent:

- El programa IC es separa en funcions en OPTIMITZAR CODI IC. La resta de mòduls del diagrama rebran com a paràmetre a (1),(2),(3),(4),(5) una única funció IC (cal recordar que totes les optimitzacions realitzades són intraprocedurals). Les optimitzacions que s'apliquen a cada funció depenen dels flags especificats per l'usuari.

- Cada optimització necessita uns anàlisis previs per a poder ser realitzada. Per tant, els mòduls que realitzen els anàlisis (ANÀLISI DE MEMÒRIA, ANÀLISI DE VIDA, CONSTRUIR CADENES US-DEF i ANÀLISI DE DOMINADORS) s'han de cridar abans de realitzar les optimitzacions. El tractament realitzat en aquests mòduls és el descrit en el capítol 3.

- Les optimitzacions (i alguns anàlisis) utilitzen algunes rutines genèriques que afecten al fluxe de dades i al fluxe de control. Aquestes rutines, per la seva utilització en gairebé totes les optimitzacions, s'agrupen en mòduls concrets: RUTINES DE FLUXE DE DADES i RUTINES DE FLUXE DE CONTROL.

- Les RUTINES DE FLUXE DE DADES reben com a paràmetre instruccions, i responen a preguntes senzilles com: aquesta variable utilitza aquesta instrucció? o quines són les variables utilitzades i definides per aquesta instrucció?.

- Les RUTINES DE FLUXE DE CONTROL serveixen per alterar el graf de fluxe de control (afegint o eliminant arestes del graf) i per a buscar camins entre instruccions (trobar un camí entre dues instruccions, trobar totes les instruccions que poden estar en algun camí entre dues instruccions,...).

7.3.3.1 - Organització de les optimitzacions: primera proposta

Un aspecte molt important a l'hora d'aplicar les optimitzacions de codi, que fins a aquest punt no s'ha discutit, és com s'han d'organitzar les optimitzacions. Fins ara, les optimitzacions han estat descrites com a tècniques per aplicar-se individualment, però no s'ha considerat en quin ordre s'han d'aplicar ni quantes vegades s'han d'aplicar.

Durant tot el capítol 3, s'ha fet molt èmfasi en el fet que realitzar alguna optimització podia produir situacions en que una altra optimització era aplicable. Per tant, no n'hi ha prou amb aplicar les optimitzacions seqüencialment i en un ordre qualsevol per a optimitzar el codi. És necessari pensar com agrupar les optimitzacions, i per això s'han de tenir en compte dos criteris (més un criteri específic del nostre projecte):

- **reduir al màxim el temps de compilació**, dins de les possibilitats
- **produir un codi el més optimitzat possible**
- intentar que les **optimitzacions** siguin **el més independents possible entre elles** per a poder afegir i treure optimitzacions sense alterar molt el codi

Els dos primers criteris són els que es tindrien en compte en qualsevol compilador, i l'últim criteri és específic del nostre projecte: recordem que es va indicar la necessitat de poder afegir o treure optimitzacions de l'entorn amb facilitat (veure requeriments del projecte a 1.3). Com que l'objectiu del projecte està orientat a la docència, els dos últims criteris tenen prioritat sobre el primer.

La majoria de textos sobre optimització de codi (entre altres [AHO90] i [APP98]) recomanen una optimització de codi que separi les optimitzacions en dos grups: **optimitzacions del fluxe de control** (salts, codi inabastable, ...) i **optimitzacions del fluxe de dades** (expressions comunes, propagació de còpies i constants,...).

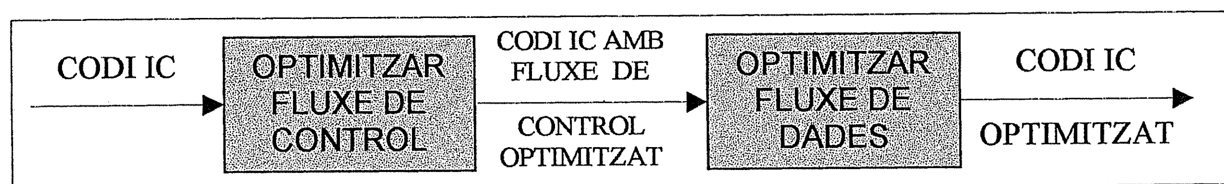


Figura 7.2 Una proposta d'organització eficient de les optimitzacions

Com s'explicava al capítol 3, les optimitzacions del fluxe de control només consideren els blocs bàsics i els salts en el programa, i les optimitzacions del fluxe de dades només consideren les expressions i les assignacions de valors a les variables. Per tant, les optimitzacions de fluxe de control i de dades són bastant independents entre sí, i aplicar les optimitzacions d'aquesta manera dóna molt bons resultats pel que fa al codi generat.

Dins de cada mòdul, **les optimitzacions s'agrupen per intentar reduir al màxim les passades que es realitzen sobre el codi** (veure Figura 7.3). D'aquesta manera, per exemple, es poden agrupar l'optimització de salts i l'eliminació de codi inabastable en una mateixa passada sobre el codi. A més cada optimització es realitza només un cop sobre el codi, pel que és important considerar l'ordre en que s'apliquen.

OPTIMITZAR_FLUXE_CONTROL (funció f)

```
construir_graf_fluxe_control( f );
anàlisi_dominadors ( f );
anàlisi_bucles ( f );

per a cada bloc bàsic b de la funció f
  si b és inabastable llavors
    eliminar_codi inabastable ( f,b );
    actualitzar_informació_dominadors (f,b);
    actualitzar_informació_bucles (f,b);
  sino
    si b és l'encapçalament d'un bucle llavors
      reescriure_bucle(f,b);
      actualitzar_informació_dominadors (f,b);
      actualitzar_informació_bucles (f,b);
    sino
      si el bloc bàsic acaba en un salt llavors
        optimització_salts (b);
        actualitzar_informació_dominadors (f,b);
        actualitzar_informació_bucles (f,b);
      fsi
    fsi
  fsi
fper
```

OPTIMITZAR_FLUXE_DADES (funció f)

```
anàlisi_vida ( f );
construir_cadenes_us_def (f);

per a cada bloc bàsic b de la funció f
  per a cada instrucció i del bloc bàsic b
    cas possibles situacions:
      cas i és codi mort:
          eliminar_codi_ mort( i );
          actualitzar les dades relatives a les
               altres optimitzacions
      cas i és una còpia de constant:
          propagar_constant ( i );
          actualitzar les dades relatives a les
               altres optimitzacions
      cas i és una còpia de variable:
          propagar_còpies ( i );
          actualitzar les dades relatives a les
               altres optimitzacions
      cas i és una expressió:
          eliminar_exp_comunes ( i );
          actualitzar les dades relatives a les
               altres optimitzacions
      ...
      en altre cas
          actualitzar les dades de totes les
               optimitzacions
    fcas
  fper
fper
```

Figura 7.3 Una possible organització de l'anàlisi del fluxe de dades i de control

Aquesta organització de les optimitzacions és **molt eficient** perquè realitza poques passades al codi. Aquestes passades es poden repetir un cert nombre de vegades per a intentar millorar el codi produit. Per exemple ([GCC]), el procés d'optimització en **gcc** (simplificat) és el següent:

- primer es realitza una optimització de salts
- després es realitza una optimització del fluxe de dades
- llavors realitza l'optimització de bucles
- repeteix l'optimització del fluxe de dades
- es repeteix l'optimització de salts.
- es passa a optimitzacions dependents de l'arquitectura; l'optimització de salts es repeteix diverses vegades

7.3.3.2 - Organització de les optimitzacions: proposta adoptada

L'organització presentada en l'apartat anterior (7.3.3.1) té alguns inconvenients que fan que no s'ajusti als objectius del projecte:

- El codi produit és de bona qualitat però és molt possible que no sigui un codi optimitzat al màxim. **L'objectiu és** reflectir les possibilitats de les optimitzacions i, per tant, **aplicar les optimitzacions tantes vegades com sigui possible.**
- Les optimitzacions estan molt relacionades entre sí, perquè cada canvi que fa una optimització ha de tenir en compte els canvis que realitzen les altres optimitzacions. Per aquest motiu, **resulta complicat afegir noves optimitzacions o treure optimitzacions existents.**

Per aquest motius, aquesta organització no ha estat l'escollida. L'organització de les optimitzacions que s'ha triat és la que proporciona *un codi optimitzat al màxim*, on totes les optimitzacions s'han aplicat tantes com vegades com ha estat possible, i *on resulta fàcil afegir/extreure optimitzacions* (veure Figura 7.4). La idea és la següent: considerem que una **ronda d'optimització** consisteix en aplicar totes les optimitzacions possibles a una funció; l'optimització de codi consistirà en **aplicar rondes d'optimització fins que la funció no pugui continuar sent optimitzada.**

```
RONDA_OPTIMITZACIÓ (funció f)

    (* Aplicar totes les optimitzacions *)
    anàlisis requerits per la optimització 1;
    f := aplicar optimització 1 a tota la funció f;
    anàlisis requerits per la optimització 2;
    f := aplicar optimització 2 a tota la funció f;
    ....
    retorna f;


OPTIMITZAR_CODI (funció f)

    f_optimitzada = f ;
    fer
        f_entrada := f_optimitzada;
        f_optimitzada := ronda_optimització (f_optimitzada);
    mentre f_optimitzada ≠ f_entrada;
    retorna f_optimitzada;
```

Figura 7.4 Organització adoptada (alt nivell)

Els avantatges d'aquesta organització són els següents:

- **El codi està optimitzat al màxim**: si una optimizació es podia aplicar, s'ha aplicat, i s'ha aplicat el màxim nombre de vegades que es podia aplicar.

- **Les optimitzacions de codi** s'apliquen a tota una funció, i **són completament independents entre sí**. Això vol dir que una optimització no depèn del que fagin les altres optimitzacions i que es pot canviar lliurement l'ordre entre elles.
- És fàcil afegir o treure optimitzacions de la ronda d'optimització, gràcies a que són independents: només cal afegir/treure la crida a la rutina corresponent.

De tota manera, aquesta organització també té inconvenvients:

- Resulta **menys eficient** que la primera proposta.
- **Cal garantir que no s'entrarà en un bucle infinit**, en algun moment, no es podran aplicar més optimitzacions i el procés d'optimització s'acabarà.

La demostració detallada de que aquesta organització no entra en bucle infinit es pot trobar en l'annex. Intuitivament, aquesta demostració es basa en el següents passos:

- Hi ha tres tipus d'optimitzacions: les que eliminen instruccions, les que reescriuen instruccions i les que afegeixen instruccions.
- Les optimitzacions que eliminen instruccions es poden aplicar, com a màxim, un nombre de vegades igual al nombre d'instruccions de la funció.
- Les instruccions que reescriuren instruccions només poden fer-ho en una direcció (no poden reescriure una instrucció i desfer el canvi en la següent ronda). Per tant, només es poden aplicar un nombre finit de vegades, menor o igual al nombre d'instruccions de la funció.
- Les instruccions que poden afegir instruccions només poden fer-ho un nombre finit de vegades. Per exemple, l'eliminació d'expressions comunes només pot afegir una instrucció de còpia per a cada expressió comuna que troba. Això ens diu que el nombre d'instruccions total que haurem de considerar serà finit (és a dir, que no podrà passar que afegim una instrucció, l'eliminem, la tornem a afegir, etc.).
- De tot el següent es dedueix que el nombre de vegades que es poden aplicar les optimitzacions està fitat. Per tant, el número de rondes d'optimització que s'aplicaran també està fitat.

7.3.3.3 - Optimitzacions concretes

A continuació es presenten els diagrames d'estructura d'optimitzacions concretes. Aquests diagrames són cridats des del mòdul principal de l'optimització de codi (mòdul OPTIMITZAR CODI del diagrama d'estructura 7.4) i reben com a paràmetre un codi IC al que se li han realitzat uns certs anàlisis.

La majoria d'aquestes optimitzacions es realitzen en un sol mòdul bàsic, però hi ha les optimitzacions de bucles han estat separades en diversos mòduls. Aquests mòduls bàsics utilitzen els mòduls amb rutines de fluxe de dades i de fluxe de control descrits a l'apartat 7.3.3.

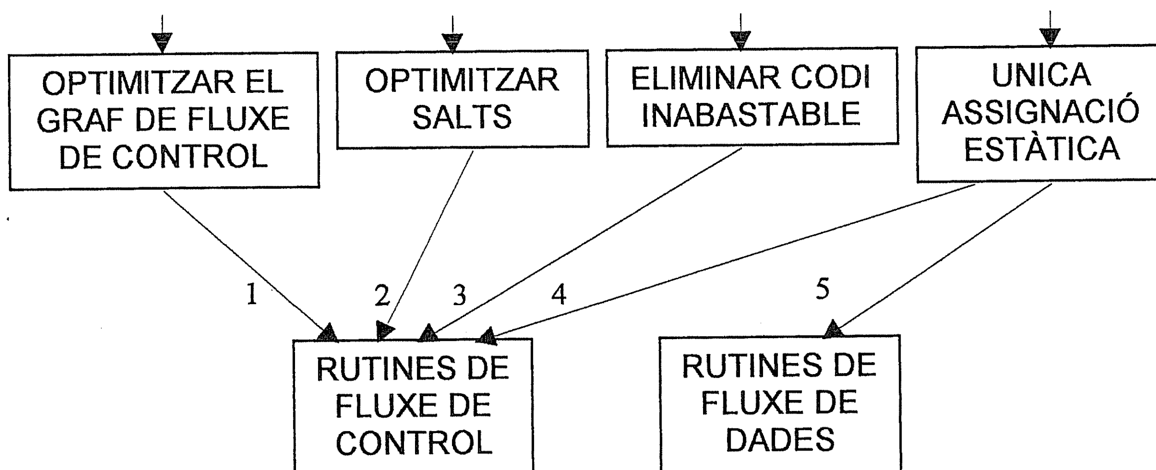


Diagrama d'estructura 7.5 Optimitzacions del fluxe de control (refina diagrama 7.4)

- Les optimitzacions del fluxe de control s'implementen en mòduls bàsics. Totes elles reben com a entrada la funció a optimitzar.

- Totes les optimitzacions del fluxe de control utilitzen les RUTINES DE FLUXE DE CONTROL (crides 1,2,3 i 4). Aquestes rutines utilitzades estan associades a la modificació del graf de fluxe de control (afegir i eliminar arestes d'aquest graf).

- L'ÚNICA ASSIGNACIÓ ESTÀTICA utilitza rutines del fluxe de dades, per accedir als operands usats i definits per una instrucció.

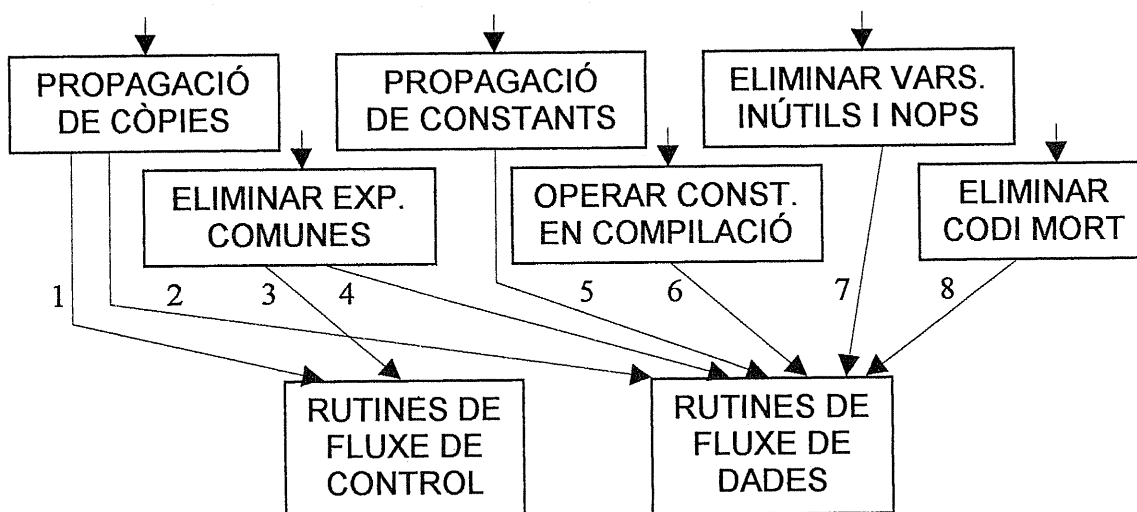


Diagrama d'estructura 7.6 Optimitzacions del fluxe de dades (refina diagrama 7.4)

- Les optimitzacions del fluxe de dades també estan implementades en un únic mòdul, i reben com a entrada la funció a optimitzar.

- Totes aquestes optimitzacions necessiten accedir als operands de les instruccions, i per això criden a RUTINES DE FLUXE DE DADES (crides 2, 4, 6 7 i 8).

- ELIMINAR EXP. COMUNES i PROPAGACIÓ CÒPIES necessiten verificar que determinades condicions es compleixen en tots els camins entre dues instruccions. Per a trobar aquests camins han de cridar al mòdul RUTINES DE FLUXE DE CONTROL (crides 1 i 3).

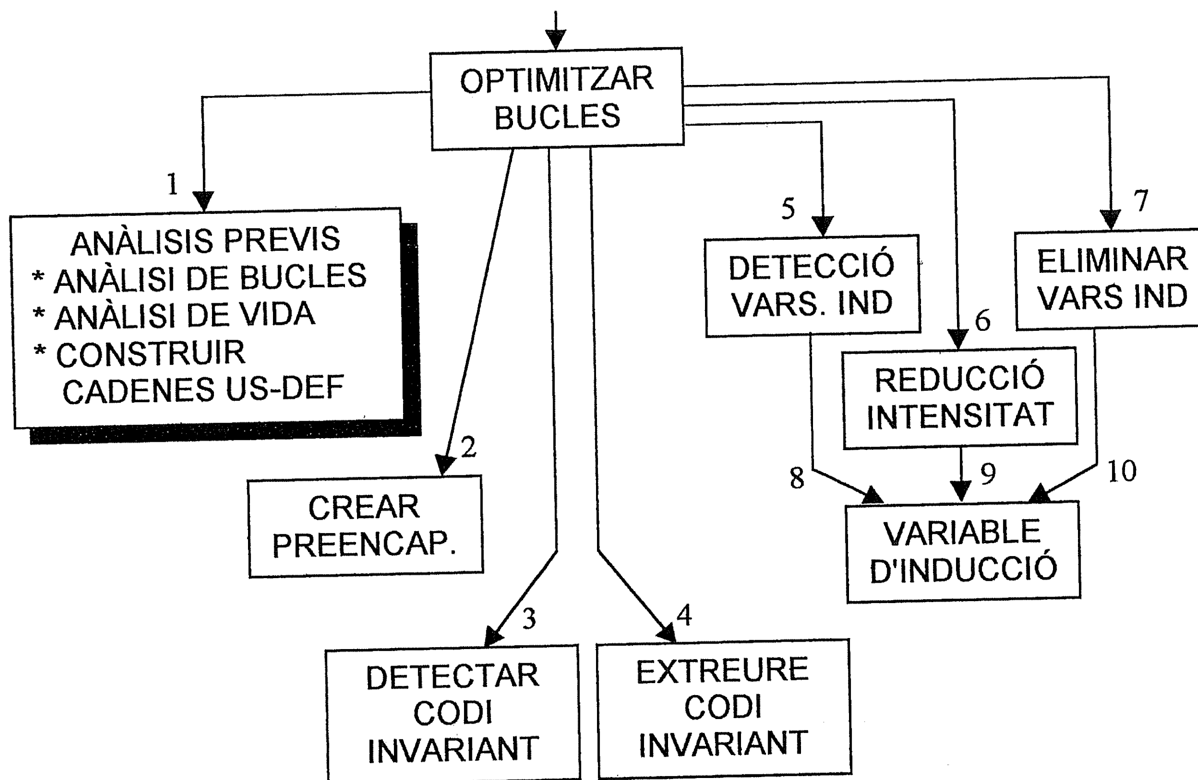


Diagrama d'estructura 7.7 Optimitzacions de bucles (refina diagrama 7.4)

Les optimitzacions de bucles estan organitzades de manera diferent a la resta d'optimitzacions.

- Abans de realitzar les optimitzacions de bucles, cal detectar els bucles (1) i construir les cadenes d'ús i definició (1) per a poder realitzar la resta d'optimitzacions. L'ANÀLISI DE BUCLES retorna una llista de tots els bucles, començant pels bucles més interns. La resta de mòduls es criden un cop per cada bucle, en l'ordre en que es troben a la llista.

- La resta de mòduls de les optimitzacions de bucles reben com a paràmetre la funció i les dades del bucle (encapçalament, preencapçalament, bucles que el formen, variables d'inducció, instruccions invariants, ...). Aquesta és la informació que es passa a (2), (3), (4), (5), (6) i (7).

- El primer que es fa es CREAR PREENCAPÇALAMENT per a poder guardar el codi invariant. Aquest codi invariant es detecta a DETECTAR CODI INVARIANT, i es fa servir per EXTREURE CODI INVARIANT i DETECTAR VARS. IND.

- Els tres mòduls dedicats a analitzar les variables d'inducció utilitzen un mòdul (VARIABLE D'INDUCCIÓ) on es defineix el tipus variable d'inducció. Aquest tipus guarda dades com la terna associada a la variable d'inducció, el tipus (bàsica o derivada) ... i proporciona operacions per a treballar amb aquestes variables.

- En aquest diagrama no es mostren les crides a RUTINES DE FLUXE DE DADES ni RUTINES DE FLUXE DE CONTROL per a intentar simplificar una mica el diagrama. Les crides RUTINES DE FLUXE DE CONTROL es farien des de CREAR PREENCAP. , per a modificar el graf de fluxe de control quan es crea el preencapçalament . Les crides a RUTINES DE FLUXE DE DADES es farien des de DETECTAR CODI INVARIANT, DETECCIO VARS. IND, REDUCCIO INTENSITAT i ELIMINAR VARS IND.

7.3.4 - Assignació de registres

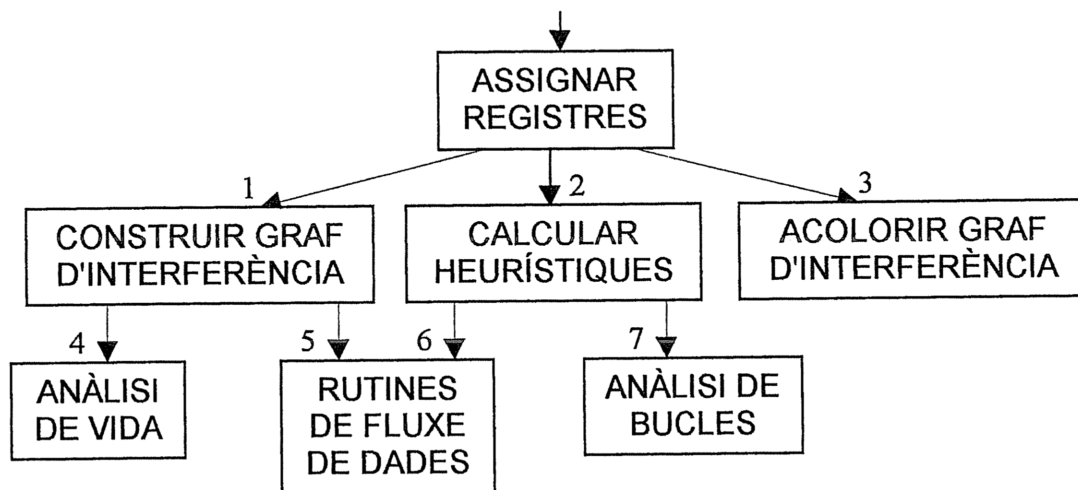


Diagrama d'estructura 7.8 Assignació de registres (refina diagrama 7.1)

- ASSIGNAR REGISTRES rep com a entrada el programa a optimitzar. Aquest programa es divideix en funcions, ja que l'assignació de registres es realitza funció a funció. Les crides a la resta de mòduls, (1), (2) i (3), reben com a paràmetre la funció sobre la que s'ha de realitzar l'assignació de registres.

- CONSTRUIR GRAF D'INTERFERÈNCIA genera el graf d'interferència associat a la funció. Per a fer-ho, necessita informació sobre els rangs de vida de les variables (4) i necessita accedir als operands de les instruccions (5).

- CALCULAR HEURÍSTIQUES calcula les heurístiques d'utilització i de còpia de les variables de la funció (veure capítol 4). Per a fer-ho, necessita saber quines instruccions estan dins d'un bucle (7), i també necessita accedir als operands de les instruccions (6).

- ACOLORIR GRAF D'INTERFERÊNCIA realitza l'algorisme d'assignació de registres descrit al capítol 4. Rep com a paràmetre (3) la funció, el seu graf d'interferència i les seves heurístiques. Aquest mòdul produeix com a resultat una assignació de registres: cada variable queda assignada a algun registre o a memòria.

7.3.5 - Generació de codi màquina

La generació de codi MIPS rep com a paràmetre un programa on s'ha realitzat l'assignació de registres, i genera com a sortida un fitxer amb el programa MIPS equivalent.

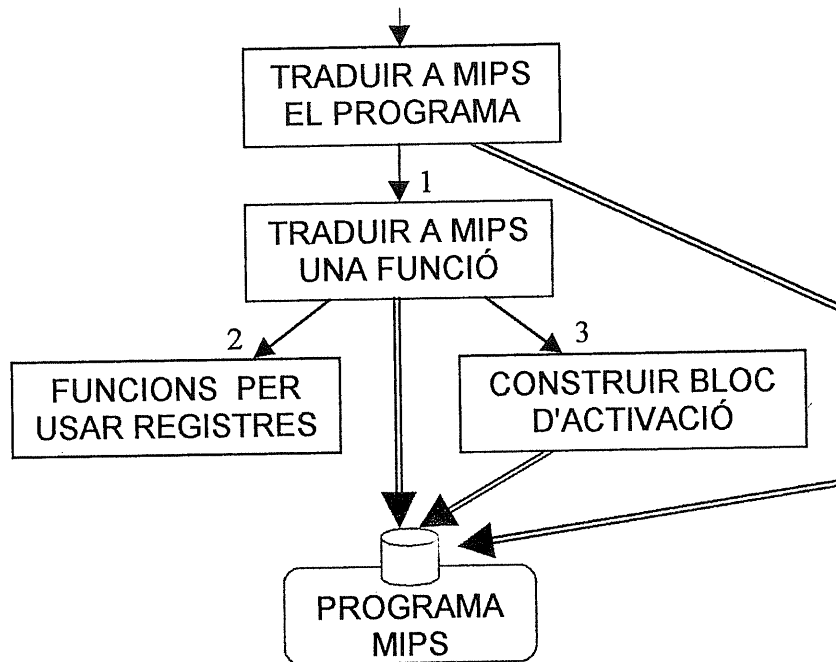


Diagrama d'estructura 7.9 Generació de codi MIPS (refina diagrama 7.1)

- TRADUIR A CODI MIPS EL PROGRAMA genera les instruccions per a definir les variables globals del programa i realitza una crida (1) per a generar codi MIPS per a cada funció.

- TRADUIR A MIPS UNA FUNCIÓ genera el codi MIPS necessari per a les instruccions d'una funció. A més de generar el codi per a les instruccions de la funció, ha de generar codi per a construir i alliberar el bloc d'activació associat a la funció, amb la crida (3).

- Finalment les operacions bàsiques amb registres (veure forçar registre i salvar registre a l'apartat 5.4) estan implementades en un mòdul a part. Aquestes rutines s'utilitzen per a generar codi per a cada instrucció concreta.

7.4 - Compilador frontal

El compilador frontal ha de llegir un fitxer amb un programa escrit en llenguatge CL i generar el fitxer amb el programa escrit en codi IC. Per a fer-ho, reutilitza alguns dels mòduls del compilador final (com ara les operacions constructores del codi IC o el mòdul que escriu el codi IC en un fitxer de text).

- El mòdul ANÀLISI LÈXIC (diferent de l'anàlisi lèxic del compilador final) llegeixel fitxer d'entrada CL i detecta els tokens (símbols) que composen el programa CL.

- L'ANÀLISI SINTÀCTIC rep els tokens del fitxer d'entrada (1), i els combina per a generar l'arbre sintàctic corresponent. Aquest arbre representa les instruccions i expressions del llenguatge CL.

- L'ANÀLISI SEMÀNTIC treballa parteix d'un arbre sintàctic (2) a partir del qual generarà el codi intermedi IC. Per a fer-ho, utilitza (4) la TAULA DE SÍMBOLS per a comprovar quins símbols han estat declarats, quin era el seu tipus associat, etc. Cadascun d'aquests símbols té associat un TIPUS DE DADES CL (enter, real, booleà, punter a un altre tipus), que s'ha d'utilitzar per a comprovar si els tipus són compatibles en una expressió.

- A més dels mòduls anteriors, l'ANÀLISI SEMÀNTIC també utilitza (7) un mòdul del compilador final, CONSTRUIR CODI IC. Aquest mòdul contenia les funcions per a crear noves instruccions IC, noves variables IC, ...

- Després de generar al codi IC, (8) escriu el codi IC al fitxer reaprofitant un mòdul del compilador final (ESCRIURE CODI IC).

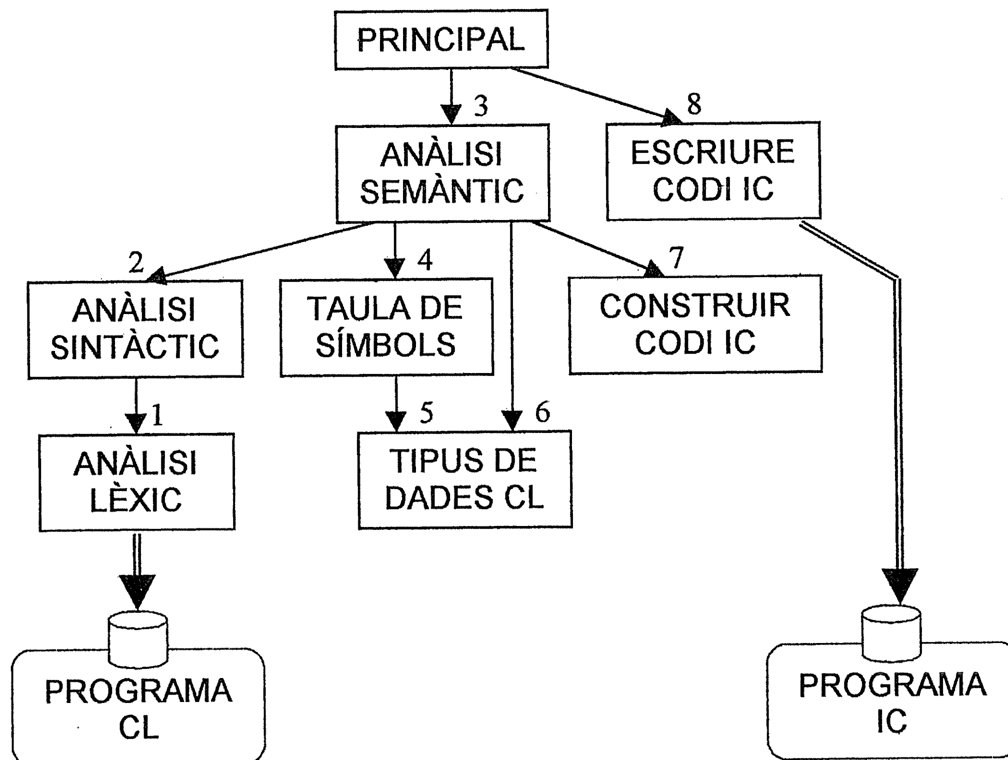


Diagrama d'estructura 7.10 Compilador frontal

7.5 - Conclusions

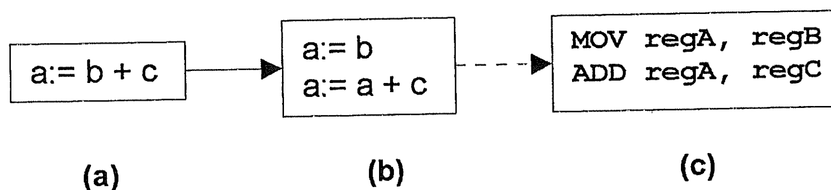
El disseny realitzat no intenta realitzar optimitzacions de codi amb una eficiència màxima, sino resultar comprensible i extensible. Per aquest motiu, aquest disseny difereix lleugerament del disseny recomanat a la literatura.

Per exemple, el disseny considera el compilador frontal i el compilador final com a programes separat, per la manera com s'ha pensat l'entorn (**CODEGEN** s'utilitza com a eina de treball majoritària i **FRONTEND** només s'utilitza ocasionalment per a traduir jocs de proves). De tota manera, amb aquest disseny és possible fusionar els dos compiladors en un sol

programa: eliminant ESCRIURE CODI IC del compilador frontal i LLEGIR CODI IC del compilador final, podem fusionar la resta de mòduls per a tenir el compilador frontal i el final en el mateix programa.

Una possible qüestió que pot sorgir és: què caldria canviar en aquest disseny si canviem el llenguatge màquina destí ? Per a respondre això, es consideraran els mòduls del diagrama d'estructura 7.1.

- El compilador frontal no es veuria afectat en absolut
- La lectura del codi IC i l'optimització de codi IC tampoc es veurien afectades, perquè són independents de l'arquitectura per a la que generem codi
- La fase ADAPTAR CODI IC potser requeriria algun mòdul concret per a adaptar el codi a la nostra arquitectura concreta. Per exemple, el codi màquina del Intel 80386 utilitza instruccions que només admeten dos operands: un ha de ser l'origen, i l'altre ha de ser alhora origen i destí. Per a generar codi per a aquesta arquitectura seria molt útil reescriure les instruccions del codi IC de manera que només utilitzessin com a molt dos operands:



(a) (b) (c)

Exemple 7.1 Un exemple de mòdul que pot ser necessari a ADAPTAR CODI IC
(a) codi IC original
(b) codi IC adaptat
(c) codi màquina generat a partir del codi IC

- En principi, l'ASSIGNACIÓ DE REGISTRES no es veuria afectada: només caldria canviar les especificacions del nombre de registres de cada tipus (paràmetres, salvats i temporals) per les especificacions de la nova arquitectura. De tota manera, si aquesta arquitectura té operacions especials que afectin registres concrets (p.ex: la multiplicació necessita tenir l'operand destí en un registre concret) caldria modificar lleugerament l'algorisme d'assignació de registres per a contemplar aquests casos particulars.
- Finalment, els mòduls de GENERAR CODI MIPS s'haurien de reescriure completament.

Una altra qüestió seria: què passa si es vol canviar el codi font, i passar del codi CL a un altre llenguatge ?

- Tot el compilador final es pot reaprofitar sense problemes
- En el compilador final, només es podrien reaprofitar CONSTRUIR CODI IC i ESCRIURE CODI IC (que de fet venen del compilador final). La resta de mòduls s'haurien de reescriure per a tractar el nou llenguatge d'alt nivell

8 Implementació

Aquest capítol descriu alguns detalls importants de la implementació dels compiladors frontals i final, com ara l'entorn de desenvolupament, convencions seguides en el codi de l'aplicació, etc.

Un aspecte molt important de la implementació, per exemple, és l'estructura de dades utilitzada per a representar el codi IC. Aquesta estructura és fonamental, perquè totes les optimitzacions hi accedeixen i la modifiquen.

8.1 - Entorn de desenvolupament

El llenguatge utilitzat en el desenvolupament de l'aplicació ha estat el llenguatge C. També s'han utilitzat les següents eines:

- **FLEX:** Generador de reconeixedors lèxics per a reconèixer cadenes de caràcters. en fitxers de text.

- **YACC:** Generador de parsers per a reconèixer una gramàtica determinada. Aquestes dues eines s'han utilitzat per a realitzar la lectura del codi IC i CL.

- **Llibreries de TADs:** Per al desenvolupament d'aquest projecte, s'han utilitzat llibreries que implementen tipus de dades senzills en C: arrays dinàmics (`array`), taules de símbols (`st`), llistes (`list`), conjunts (`set`) i grafs (`graph`).

Es pot trobar més informació sobre aquestes eines al annex A.3 (Eines utilitzades).

En aquest projecte s'han utilitzat dos entorns desenvolupament: una estació de treball Sun i un ordinador personal (PC).

Processador:	Sparc Sun Ultra-60	Intel Pentium 100 Mhz
Memòria:	-------	16 Mb
Sistema operatiu:	SunOS release 5.6	Linux RedHat 6.0
Compilador:	cc (BSD compatibility package C compiler)	gcc 1.1.2
Eines:	lex,yacc	flex , yacc

8.2 - Estructura de dades del llenguatge IC

Quan llegim un programa IC des d'un fitxer, el carreguem a memòria. La representació utilitzada és important, perquè totes les optimitzacions accedeixen i modifiquen aquesta estructura.

Els diferents elements del llenguatge IC que s'han de representar són: programa, funció, bloc bàsic, instrucció, operand i variable. A continuació s'expliquen les relacions existents entre els diferents elements.

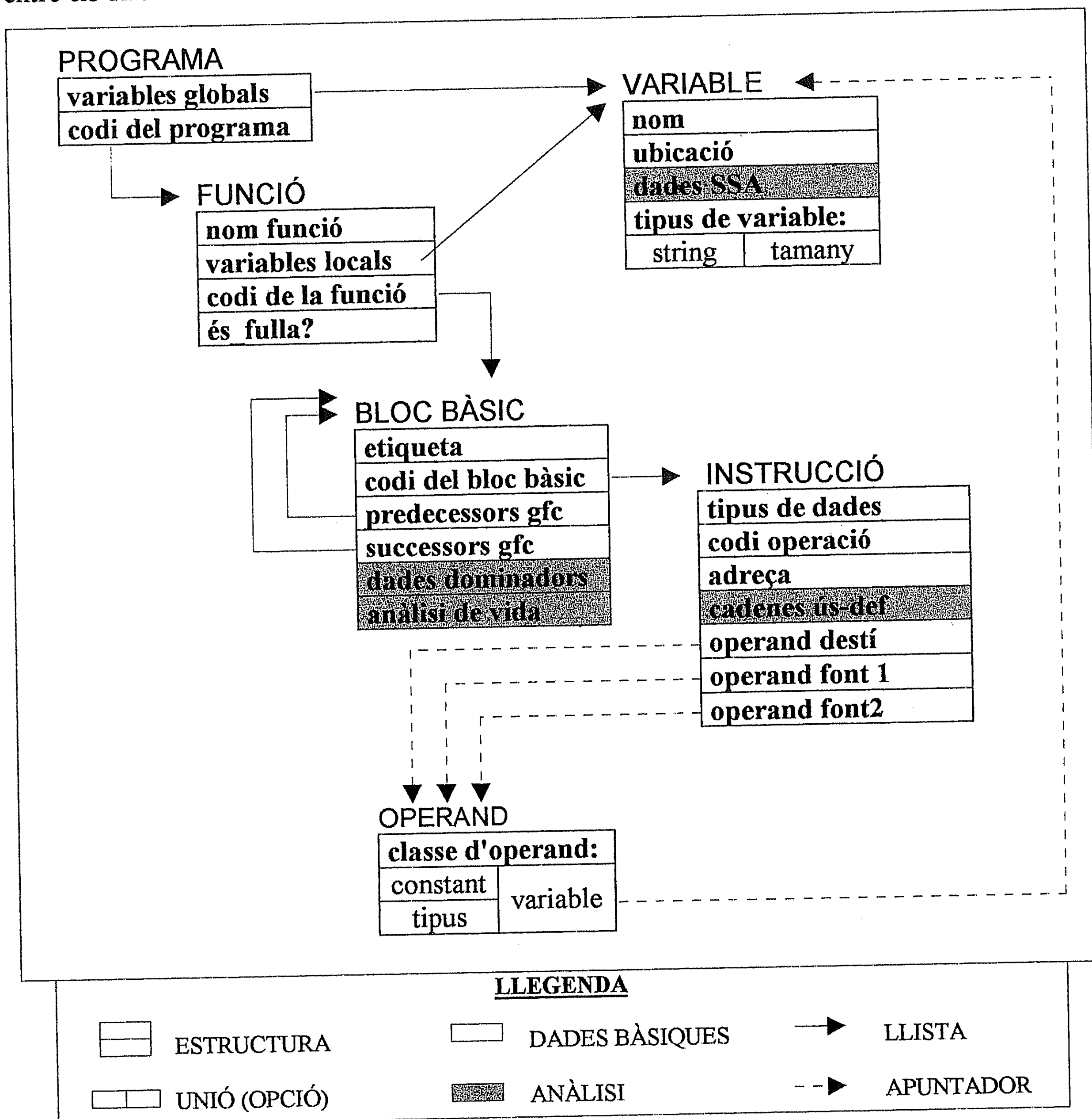


Figura 8.1 Estructura de dades per a representar el llenguatge IC

Un *programa* en llenguatge IC està format per declaracions de variables globals i declaracions de funcions. Cada *funció* té associada un nom únic, està formada per declaracions de variables globals i una llista de blocs bàsics. Un *bloc bàsic* conté una llista

d'instruccions que s'executaran sempre consecutivament. Les *instruccions* utilitzen diferents operands, dels quals poden llegir o escriure. Un *operand* pot ser una constant o una variable, local o global.

Podem distingir dos tipus d'informació en les estructures de dades (veure Figura 8.1):

- Alguns camps guarden informació bàsica, com ara el nom de la funció, o el tipus d'instrucció. Aquests camps contenen sempre algun valor.
- Altres camps guarden informació associada a anàlisis realitzats sobre el programa. Aquests camps normalment no contenen informació, però quan es realitzen els anàlisis addients s'omplen aquests camps amb les dades corresponents.

8.2.1 - Variables

```
typedef enum {CHAR, BOOLEAN, INTEGER, REAL, POINTER} IC_data_type;

typedef enum {STRING, SPACE, LOCAL} IC_variable_type;

typedef struct IC_variable_ {
  IC_variable_type type;          /* Type of the data */
  char *name;                     /* Name of the variable */
  union {
    char *data_string;            /* Only for global data strings */
    int size;                     /* Size of the data (other variables) */
  } value;
  IC_reg_assign *assign_info;     /* Register assignment (see ic_reg.h) */
  SSA_data *ssa;                  /* SSA related data (see ic_ssa.h) */
} IC_variable;
```

Totes les variables tenen un nom associat. No poden haver-hi dues variables globals amb el mateix nom o dues variables locals (dins la mateixa rutina) amb el mateix nom.

Les variables es poden classificar en tres grans grups: les variables locals, els strings globals i la resta de variables globals. El camp `type` indica el tipus de la variable actual. Per als strings globals, cal guardar la cadena de text que contenen, i per a la resta de variables, cal guardar el tamany (en bytes) que ocupa la variable. Aquesta informació es guarda al camp `value`.

El camp `assign_info` indica si la variable es troba en memòria o en registre. A part d'això indica el número de registre on es guardarà la variable (després de realitzar l'assignació de registres) i la posició de memòria on es salvarà (durant la generació de codi). El camp `ssa` conté dades que s'utilitzen només durant la transformació a Única Assignació Estàtica (veure apartat 3.8.11).

8.2.2 - Operands

```
typedef enum {GLOBAL, PARAMETER, SCALAR, STRUCT, CONSTANT,
              INSTR, FUNCTION} IC_operand_type;

typedef struct IC_operand_ *IC_operand;
```

```
struct IC_operand_ {
  IC_operand_type kind;
  IC_data_type type;          /* For constants only */
  union {
    IC_variable *var;         /* For global, scalar, struct or param */
    char  data_char;          /* For char constants */
    int   data_integer;       /* For integer constants */
    float data_real;          /* For real constants */
    char  data_boolean;       /* For boolean constants */
  } value;
};
```

Un operand pot ser o bé una constant o bé una variable, com ho indica el camp `kind`. El camp `value` conté un apuntador a la variable o el valor de la constant. El tipus adequat de la constant (que ens permet seleccionar el seu valor) s'indica al camp `type`.

8.2.3 - Instruccions

```
typedef enum {ADD, SUB, MUL, DIV, MOD, NEG,
              AND, OR, NOT, XOR,
              LSHIFT, RSHIFT, ARSHIFT,
              EQ, NEQ, GT, GEQ, LT, LEQ} IC_ARITH_LOGIC_OP;

typedef char *I_address;

typedef enum {ARITH_LOGIC, MOVE, INT2REAL, REAL2INT,
              UJUMP, CJUMP, CALL, RETURN, PARAM,
              LOAD_CONTENT, LOAD_ADDR, STORE_CONTENT, STORE_ADDR,
              ADDRESS, SRC_POINTER, DST_POINTER, NOP, ADDR } IC_itype;

typedef struct IC_instr_ {
  IC_itype kind;              /* Instruction type */
  IC_data_type type;          /* Data type of the operation */
  IC_ARITH_LOGIC_OP op;       /* Arith/logic operator */
  IC_operand dst;             /* Dst operand (right part for STORE) */
  IC_operand src1,src2;       /* Src operands */
  I_address  iaddr;           /* Instruction address (int) or function name
                              (char *) */
  use_def_info *use_def;      /* Use/Def information (see ic_usedef.h) */
} IC_instr;
```

Els primers tres camps d'una instrucció intenten fixar el tipus d'instrucció de codi intermedi. En primer lloc, `kind` ens indica quina és la instrucció que volem realitzar: còpia, aritmètica, crida a funció, salt condicional, ... Després `type` ens indica el tipus de les dades que manipula la instrucció (enter, real, caràcter, ...). Finalment `op` fixa l'operació concreta que es vol realitzar, si a `kind` hem indicat que es tracta d'una operació aritmètica.

Els següents camps ens indiquen els operands de l'operació: l'operand destí, `dst`, (que també utilitzarem per a guardar la part dreta de les instruccions de STORE) i els dos operands font, `src1` i `src2`. Algunes instruccions no utilitzen tres operands; en aquest cas, els operands que no s'estiguin utilitzant estaran apuntant a NULL.

El camp `iaddr` s'utilitza en dos tipus d'instruccions: les instruccions de salt (condicional i incondicional) i les instruccions de crida). Aquest camp conté l'etiqueta destí del salt, en el cas de la instrucció de salt, o el nom de la funció cridada, en el cas d'una crida a funció.

Finalment, el camp `use_def` conté les cadenes d'ús i definició (veure 3.7.3) per a cadascun dels operands de la instrucció. Aquestes cadenes són llistes de les instruccions que usen o defineixen els valors generats/consultats per la instrucció actual.

Pel que fa a les crides a funció cal destacar una cosa important. Malgrat que en el capítol 2 es presentava una crida a funció com una única operació, internament es representa amb diverses operacions: una sèrie d'operacions prèvies que realitzen el *pas de paràmetres* (PARAM) i una instrucció final que realitza la *crida* a funció i agafa el resultat (CALL). Aquestes instruccions de pas de paràmetre han d'estar en el mateix bloc bàsic que la instrucció de crida; un cop s'ha passat el primer paràmetre, les instruccions següents fins a la crida a funció han de ser passos de paràmetres.

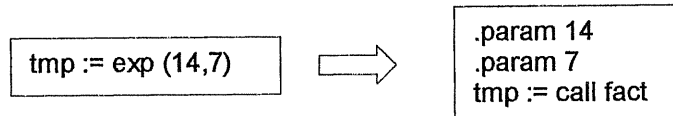


Figura 8.2 Representació interna de les crides a funció i el pas de paràmetres

8.2.4 - Bloc bàsics

```
typedef struct IC_basic_block_ {
  int label;                /* Label at the beginning of this basic block */
  IC_list *code;            /* List of instructions in this basic block */
  IC_list *next_bbs;        /* BBs that can follow this BB at run time */
  IC_list *prev_bbs;        /* BBs that can precede this BB at run time */
  IC_list *dominators;      /* BBs that dominate this bb */
  IC_list *dom_child;       /* Children of this bb in the dominator tree */
  IC_list *dom_frontier;    /* Dominance frontier of this bb */
  live_info *liveness;      /* Result of liveness analysis (see ic_live.h)*/
} IC_basic_block;
```

Tots els blocs bàsics estan identificats per una etiqueta (`label`), que ha de ser única dins de la funció. Aquesta etiqueta és la que faran servir les instruccions de salt per referir-se al bloc bàsic.

El bloc bàsic està format per una llista d'instruccions (`code`). Només poden haver-hi instruccions de salt o return al final del bloc bàsic.

Els dos camps següents enllacen aquest bloc bàsic amb els seus predecessors (`prev_bbs`) i sucessors (`next_bbs`) en el graf de fluxe de control (veure 3.6.2). Si volem eliminar un node B de la llista de predecessors del node A, cal recordar que també cal eliminar el node A de la llista de successors del node B (i el mateix per les llistes de predecessors).

La resta de camps contenen informació que és calculada per diferents anàlisis. Així, `dominators` conté la llista de dominadors del node actual, que és calculada quan s'invoca l'anàlisi de dominadors (veure 3.6.3). Quan es construeix l'arbre de dominadors (veure 3.6.4) s'obté la llista de fills en l'arbre de dominadors, `dom_child`. El camp `dom_frontier` es calcula quan s'analitza la frontera de dominació. I finalment, el camp `liveness` conté informació sobre les variables vives al principi i al final del bloc bàsic després de l'anàlisi de vida (veure 3.7.2).

8.2.5 - Funcions

```
typedef struct IC_function_code_ {
  char      *function_name;    /* Name of the function */
  int       next_bb_label;     /* Next label for a new basic block */
  IC_list   *basic_blocks;     /* List of basic blocks */
  st_table  *var_names;        /* Names of local variables */
  IC_list   *params;           /* Function parameters */
  IC_list   *scalars;          /* Scalar variables */
  IC_list   *structs;          /* Structured variables */
  int leaf_function;           /* Does this function make any function call?*/
} IC_function_code;
```

Cada funció en codi IC ha de tenir un nom de funció (`function_name`) que les identifica. No poden haver-hi dues funcions amb el mateix nom.

El codi de la funció es guarda en forma de llista de bocs bàsics (`basic_blocks`). Per a garantir que cada bloc bàsic nou que s'afegirà a la funció tingui una etiqueta nova, es manté el comptador `next_bb_label`, que s'incrementa cada cop que es crea un nou bloc bàsic en aquesta funció.

Les variables dins de la funció es guarden en tres llistes diferents: `params`, `scalars` i `structs`. Una taula de símbols (`var_names`) guarda els noms de totes les variables globals per a assegurar que no hi hagi repeticions.

Finalment, el booleà `leaf_function` ens indica si la funció és una fulla o no. Una funció és una **funció fulla** si no fa cap crida a cap funció. Això té importància per a la generació de codi, per a decidir si cal salvar l'adreça de retorn (veure a 5.3.2 la construcció del bloc d'activació).

8.2.6 - Programa

```
typedef struct IC_program_ {
  IC_list *global_data;                  /* List of global variables */
  IC_function_code *function_codes;      /* Array of IC_function_code */
  int      number_functions;             /* Size of the array */
} IC_program;
```

El programa IC és força senzill. Només conté una llista de les variables globals (`global_data`) i una taula de funcions (`function_codes`).

8.2.7 - Macros per recòrrer l'estructura

Una part dels objectius d'aquest projecte exigia que l'accés a aquesta estructura de dades fós senzill. Per aquest motiu s'han utilitzat macros del llenguatge C per a permetre realitzar recorreguts de l'estructura de dades. L'estructura d'aquestes macros és:

```
foreach_NNNNNN (contenidor, elem) {}
```

Aquesta macro serveix per a recòrrer tots els elements de la variable contenidor. Els elements es van guardant un a un a la variable `elem`; si no hi ha cap element, no s'entra mai en el foreach. Dins un foreach es poden definir instruccions com si fós un for del llenguatge C, i es pot utilitzar break i continue.

El valor de l'element després de sortir d'una macro és indefinit. Es poden inserir elements en la llista que s'està recorrent (es poden afegir instruccions), tenint en compte que es podria entrar en bucle infinit. NO es pot esborrar l'element actual del foreach, perquè es produiria un error. En cas de voler esborrar elements quan fem un recorregut usant una macro, es recomana guardar una llista dels elements que es volen esborrar i esborrar-los posteriorment; una alternativa consisteix a esborrar l'element quan ja no és l'element actual.

```
int how_many_instructions (fc)
IC_function_code *fc;
{
IC_basic_block *bb;
IC_instr *ins;
int num;

    num = 0;
    foreach_instruction_in_function (fc,bb,in) {
        num++;
    }
    return num;
}
```

Exemple 8.1 Exemple de programa que utilitza macros

Les macros definides en l'aplicació són els següents:

```
foreach_function(pr,i,f) {}
IC_program *pr;
int i;
IC_function_code *f;
```

Recorregut de totes les funcions del programa `pr`. La variable indica indica la posició de la funció actual en el vector de funcions de pr.

```
foreach_basic_block(f,bb) {}
IC_function_code *f;
IC_basic_block *bb;
```

Recorregut de tots els blocs bàsics de la funció `f`.

```
foreach_instruction(bb,ins) {}
IC_basic_block *bb;
IC_instr *ins;
```

Recorregut de totes les instruccions del bloc bàsic `bb`.

```
foreach_predecessor_bb(bb,pred) {}
IC_basic_block *bb,  *pred;
```

Recorregut de tots els predecessors del bloc `bb` en el graf de fluxe de control .

```
foreach_successor_bb(bb,succ) {}
IC_basic_block *bb,  *succ;
```

Recorregut de tots els successors del bloc `bb` en el graf de fluxe de control.

```
foreach_bb_in_df(bb,df) {}
IC_basic_block *bb, *df;
```

Recorregut de tots els blocs bàsics en la frontera de dominació de `bb`.

```
foreach_child_bb(bb,child) {}
IC_basic_block *bb, *child;
```

Recorregut de tots els fills del bloc bàsic `bb` en l'arbre de dominadors.

```
foreach_parameter(fc,par) {}
IC_function_code *fc;
IC_variable *par;
```

Recorregut de tots els paràmetres de la funció `fc`.

```
foreach_scalar(fc,sca) {}
IC_function_code *fc;
IC_variable *sca;
```

Recorregut de totes les variables escalars de la funció `fc`.

```
foreach_struct(fc,str) {}
IC_function_code *fc;
IC_variable *str;
```

Recorregut de totes les variables estructurades de la funció `fc`.

```
foreach_global_var(pr,gvar) {}
IC_program *pr;
IC_variable *gvar;
```

Recorregut de totes les variables globals del programa `pr`.

```
foreach_instruction_in_function(fc,bb,ins) {}
IC_function_code *fc;
IC_basic_block *bb;
IC_instr *ins;
```

Recorregut de totes les instruccions de la funció `fc`. La variable `ins` indica en cada moment la instrucció actual del programa, i `bb` indica el bloc bàsic d'aquesta instrucció.

8.2.8 - El tipus IC_list

L'estructura de dades del codi IC i els algorismes utilitzen massivament l'estructura de dades IC_list. Aquest tipus representa una **llista doblement encadenada** amb un conjunt d'operacions fàcils d'utilitzar i molt versàtils, i pretèn facilitar encara més l'accés a les dades.

Les operacions que permet aquest tipus de dades són les següents:

```
IC_list *list_create ()
```

Crea una llista buida.

```
void list_free (lst);
IC_list *lst;
```

Allibera una llista.

```
int list_count (lst)
IC_list *lst;
```

Retorna el nombre d'elements de la llista.

```
char *list_first_elem (lst)
char *list_last_elem (lst)
IC_list *lst;
```

Retorna el primer (últim) element de la llista (respectivament).

```
int elem_is_first (lst,elem)
int elem_is_last (lst,elem)
IC_list *lst;
char *elem;
```

Retorna 1 si `elem` és el primer (últim) element de la llista, 0 altrament (respectivament).

```
int  list_contains_elem (lst,elem)
IC_list *lst;
char *elem;
```

Retorna 1 si `elem` pertany a la llista, 0 altrament.

```
char *elem_next_elem (lst,elem)
IC_list *lst;
char *elem;
```

Retorna l'element posterior a `elem` en la llista (NULL si `elem` era l'últim).

```
char *elem_prev_elem (lst,elem)
IC_list *lst;
char *elem;
```

Retorna l'element anterior a `elem` en la llista (NULL si `elem` era el primer)

```
void list_add_begin (lst, elem)
void list_add_end (lst,elem)
IC_list *lst;
char *elem;
```

Afegeix `elem` al principi (final) de la llista (respectivament).

```
void list_add_before (lst,old,new)
void list_add_after (lst,old,new)
IC_list *lst;
char *old, *new;
```

Afegeix l'element `new` abans (després) de l'element `old` (respectivament). Es produeix un error si l'element `old` no pertany a la llista.

```
void list_insert (lst,elem)
IC_list *lst;
char *elem;
```

Afegeix l'element `elem` al final de la llista, només si no existeix ja a la llista. Si `elem` ja existeix a la llista no l'afegeix.

```
void list_remove_elem (lst,elem)
IC_list *lst;
char *elem;
```

Elimina l'element elem de la llista. Produeix un error si l'element no pertany a la llista.

```
void list_delete_elems (lst1,lst2)
IC_list *lst1, *lst2;
```

Elimina de la llista `lst1` tots els elements que hi ha a la llista `lst2`. Si algun element no de `lst2` no pertany a `lst1`, l'element s'ignora. La llista `lst2` no canvia.

```
void list_intersect (lst1,lst2)
void list_union (lst1,lst2)
IC_list *lst1, *lst2;
```

Aquestes operacions tracten la llista com si fós un conjunt, realitzant la intersecció (unió) d'elements de les dues llistes. El seu resultat és:
$lst1 := lst1 \cap lst2$ ($lst1 := lst1 \cup lst2$). La llista `lst2` no canvia.

```
void empty_list ARGS(lst)
IC_list *lst;
```

Elimina el contingut de la llista `lst`; `lst` passa a ser igual que una llista acabada de crear.

```
int  list_equal (lst1,lst2)
IC_list *lst1, *lst2;
```

Retorna 1 si les dues llistes contenen exactament els mateixos elements, 0 altrament.

```
void list_copy  (lst1,lst2)
IC_list *lst1, *lst2;
```

Copia tots els elements de la llista `lst2` a la llista `lst1`. Si la llista `lst1` contenia algun element, s'esborra abans de realitzar la còpia. La llista `lst2` no canvia.

8.3 - Fitxers de l'entorn

El codi de l'aplicació s'ha agrupat en directoris, per a facilitar l'organització del codi i la compilació. Cada optimització (de les explicades al capítol 3) es troba en un directori diferent.

Pel que fa al codi de l'aplicació, es poden considerar quatres tipus de fitxers:

- **Definició de símbols (extensió ".l") :** Especificació d'un conjunt de tokens en format LEX. Aquesta especificació serà traduida per LEX en rutines de llenguatge C que reconeixeran aquests tokens (símbols).
- **Definició d'una gramàtica (extensió ".y") :** Especificació d'un conjunt de regles de gramàtica en format YACC. Aquesta especificació serà traduida per YACC a rutines en llenguatge C, que serviran per a fer el reconeixement d'aquesta gramàtica.
- **Capçalera de C (extensió ".h") :** Fitxer de capçalera en llenguatge C. Aquests fitxers contenen macros, capcaleres de funcions o declaracions dels tipus utilitzats en l'entorn.

- **Codi C (extensió ".c")** : Fitxer de codi C. Aquests fitxers implementen rutines declarades en algun fitxer de capçalera i altres rutines internes, que no són visibles fora d'aquest fitxer (es declaren amb la paraula clau static).

Els fitxers que contenen el codi de l'aplicació són els següents:

```
Makefile
bin/
  codegen ........  Compilador final (executable)
  frontend .......  Compilador frontal (executable)

ic_cflow/
  Makefile
  ic_cflow.c .....  Construcció del graf de fluxe de control (veure 3.6.2)
  ic_cfutils.c ...  Rutines de fluxe de control (veure 7.3.3)

ic_constant/
  Makefile
  ic_constant.c .   Operació de constants en temps de compilació: instruccions
                    amb tots els operands constants (veure 3.8.5)
  ic_constant1.c    Operació de constants en temps de compilació: reducció
                    d'intensitat i instruccions amb elements neutre i absorvent
                    (veure 3.8.5)
  ic_propagate.c .  Propagació de constants (veure (3.8.4)

ic_copy/
  Makefile
  ic_copy.c ......  Propagació de còpies (veure 3.8.8)
  ic_join.c ......  Propagació de còpies (veure 3.8.8)

ic_dead/
  Makefile
  ic_dead.c ......  Eliminació de codi mort (veure 3.8.3)

ic_dom/
  Makefile
  ic_dom.c .......  Anàlisi de dominadors (veure 3.6.3), construcció de l'arbre de
                    dominadors (veure 3.6.4) i càlcul de la frontera de dominació
                    (veure 3.6.6)

ic_insert/
  Makefile
  ic_insert.c ....  Rutines per a afegir instruccions al codi (salts, còpies...), crear
                    variables temporals, ...

ic_io/
  Makefile
  ic.h ...........  Estructura de dades per al codi IC (veure 8.2)
```

```
  ic.l ........... Descripció dels tokens que formen el llenguatge IC en format
                   LEX
  ic.y ........... Descripció de les regles sintàctiques que formen la gramàtica
                   del llenguatge IC en format YACC (veure A.5)
  ic.c ........... Rutines semàntiques per a carregar el codi IC en memòria;
                   separació de les instruccions en blocs bàsics (veure 3.6.1)
  ic_macros.h .... Macros per a recòrrer l'estructura de dades (veure 8.2.7)
  ic_macros.c .... Rutines en C necessàries per a les macros.
  ic_write.c ..... Escriptura del codi IC en fitxer. Escriptura de la informació del
                   fluxe de dades i de control.
  mem_manag.c .... Rutines per a crear i alliberar memòria en l'estructura de dades
                   del codi IC (crear bloc bàsic, alliberar bloc bàsic, ...)

ic_jump/
  Makefile
  ic_jump.c ...... Optimització de salts (veure 3.8.2)

ic_list/
  Makefile
  ic_list.h ...... Definició del tipus llista utilitzat en l'estructura de dades (veure
                   8.2.8)
  ic_list.c ...... Implementació del tipus IC_list

ic_live/
  Makefile
  ic_live.h ...... Estructura de dades utilitzada per a representar la informació
                   de vida de les variables
  ic_live.c ...... Anàlisi de vida (veure 3.7.2)

ic_loop/
  Makefile
  ic_loop.h ...... Estructura de dades bucle (IC_loop) i variable d'inducció
                   (IC_ind_var)
  ic_loop_opt.c .. Optimitzacions de bucles (veure 3.8.9)
  ic_find_loop.c . Detecció de bucles (veure 3.6.5)
  ic_loop_hdr.c .. Creació d'un preencapçalament (veure 3.8.9.1)
  ic_loop_inv.c .. Detecció de codi invariant (veure 3.8.9.2)
  ic_hoisting.c .. Extracció de codi invariant del bucle (veure 3.8.9.2)
  ic_induction.c . Detecció de variables d'inducció (veure 3.8.9.3)
  ic_strength.c .. Reducció d'intensistat de variables d'inducció (veure 3.8.9.4)
  ic_rewrite.c ... Reescriptura de condicions per a eliminar variables d'inducció
                   (veure 3.8.9.5)
  ic_eliminate.c . Eliminació de variables d'inducció (veure 3.8.9.5)
  ic_ind_var.c ... Implementació del tipus de dades variable d'inducció

ic_mem/
  Makefile
  ic_mem.c ....... Anàlisi de memòria (determinar quines variables han d'estar
                   guardades en memòria)
```

```
ic_opt/
  Makefile
  ic_opt.c ....... Optimització total (per defecte)
  ic_custom_opt.c  Optimització seguint un flags determinats

ic_param/
  Makefile
  ic_param.c ..... Salvar i restaurar els paràmetres abans i després de les crides a
                   una funció.

ic_reg/
  Makefile
  ic_assign.h .... Estructura de dades assignació (assign_info), guardada a cada
                   variable (veure 8.2.1)
  ic_reg.h ....... Estructures de dades per al graf d'interferència, heurístiques i
                   assignació de registres
  ic_reg.c ....... Assignació de registres (veure capítol 4)
  ic_interfere.c . Construcció del graf d'interferència (veure 4.4.1)
  ic_igutils.c ... Rutines per accedir al graf d'interferència
  ic_heuristic.c . Càlcul d'heurístiques (veure 4.4.2)
  ic_coloring.c .. Coloració del graf d'interferència (veure 4.4.3)

ic_remove/
  Makefile
  ic_remove.c .... Eliminació de variables inútils i NOPs (veure 3.8.6)

ic_simple/
  Makefile
  ic_simple.c .... Optimització del graf de fluxe de control (veure 3.8.10)

ic_ssa/
  Makefile
  ic_ssa.h ....... Estructura de dades per a la transformació a Única Assignació
                   Estàtica o SSA
  ic_ssa.c ....... Transformació a forma SSA (veure 3.8.11)
  ic_undo_ssa.c .. Eliminar funcions φ de SSA(veure 3.8.11)

ic_subexpr/
  Makefile
  ic_operands.c .. Rutines de fluxe de dades (veure 7.3.3)
  ic_subexpr.c ... Eliminació de subexpressions comunes (veure 3.8.7)

ic_unreach/
  Makefile
  ic_unreach.c ... Eliminació de codi inabastable (veure 3.8.1)
```

```
ic_usedef/
  Makefile
  ic_usedef.h .... Estructura de dades per a les cadenes d'ús i definició
  ic_usedef.c .... Construcció de les cadenes d'ús i definició (veure 3.7.3)

include/
  *.h ............ Tots els fitxers de capçalera de l'entorn

main/
  Makefile
  flags.h ........ Definició dels flags
  flags.c ........ Tractament dels flags (línea de comandes)
  main.c  ........ Programa principal
  version.c ...... Número de versió

mips/
  Makefile
  mach-mips.h .... Especificació de caràcterístiques de l'arquitectura MIPS
                   (número i tipus dels registres, tamany ocupat cada tipus de
                   dades, ...)
  mips_gen.h ..... Rutines utilitzades per a generar codi MIPS
  mips.c ......... Generar codi MIPS per a un programa
  mips_gen.c ..... Generar codi MIPS per a una funció
  mips_frame.c ... Generar codi MIPS per a construir/destruir el bloc d'activació
                   d'una funció
  mips_regs.c .... Rutines per a salvar un registre a memòria i guardar un
                   operand a registre

pck/
  Makefile
  array.* ........ Tipus de dades "array dinàmic" (llibreria)
  graph.* ........ Tipus de dades "graf dirigit" (llibreria)
  list.* ......... Tipus de dades "llista" (llibreria)
  set.* .......... Tipus de dades "conjunt" (llibreria)
  st.* ........... Tipus de dades "taula de símbols" (llibreria)

frontend/
  Makefile
  cl.l ........... Descripció dels tokens del llenguatge CL en format LEX
  cl.y ........... Regles sintàctiques del llenguatge CL en format YACC
  semant.* ....... Rutines semàntiques associades a les regles sintàctiques
  tipos.* ........ Tipus de dades "Tipus en el llenguatge CL"
  symtab.* ....... Taula de símbols usada en la generació de codi IC a partir de
                   codi CL
```

9 Proves

Després d'implementar les optimitzacions de codi independents de l'arquitectura en CODEGEN, es desitja conèixer el grau d'efectivitat d'aquestes tècniques. Per aquest motiu, es seleccionaran un conjunt de programes que sigui representatiu de programes reals. Aquests jocs de proves seran optimitzats i el seu temps d'execució serà comparat per a comprovar la millora obtinguda.

Després d'això, es valoraran els resultats obtinguts i s'extreuran conclusions sobre l'optimització en els diferents tipus de programes.

9.1 - Organització de les proves

Un cop **CODEGEN** ha estat desenvolupat, es volen realitzar proves per a determinar l'efectivitat de les optimitzacions sobre el codi IC. Aquestes proves intentaran mesurar la reducció en temps d'execució que s'obté al aplicar les optimitzacions en una sèrie de programes realistes de diferents tipus (numèrics, recursius, ...).

Donat que els programes que es volen provar són realistes, i per tant grans, s'escriuran en llenguatge d'alt nivell (CL) i es traduiran a codi IC mitjançant **FRONTEND**. Finalment, s'optimitzaran mitjançant **CODEGEN** i es traduiran a codi màquina MIPS. El codi MIPS resultant s'executarà en un simulador de l'arquitectura MIPS (SPIM).

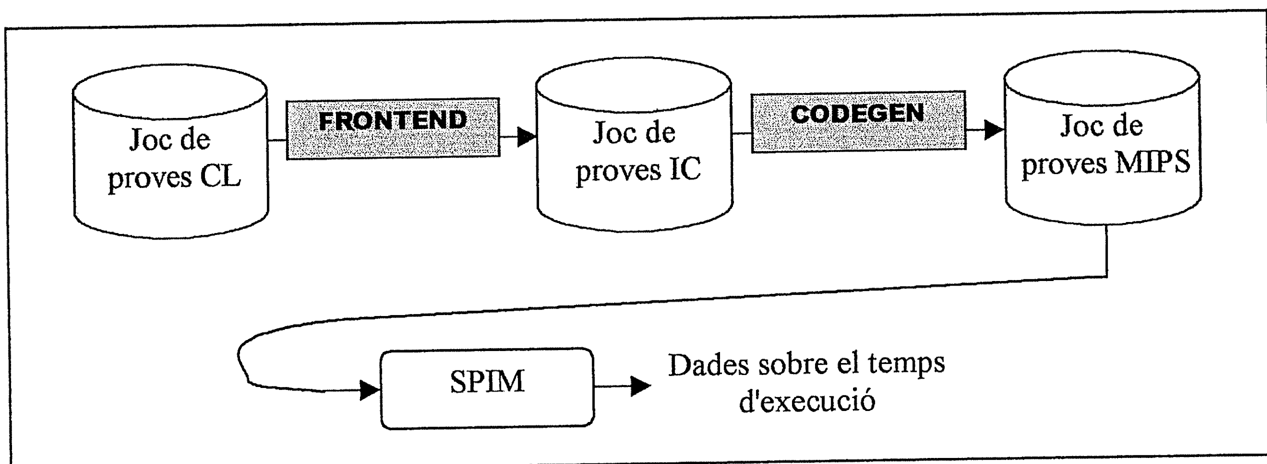


Figura 9.1 Estructura de les proves

Donat que SPIM és un simulador de MIPS i no podrem obtenir dades directes sobre el temps d'execució en un processador MIPS, utilitzarem una herurística: en aquestes proves, es mesurarà el número d'instruccions de codi màquina MIPS executades. Per tant, l'objectiu serà **comprovar que, després d'una optimització, el programa executa menys instruccions per a realitzar el mateix càlcul**. Evidentment, això és només una heurística, perquè hi ha instruccions més cares que altres: un producte no tindrà el mateix cost que una suma, un accés a memòria té un cost més alt que una operació amb registres...

Per a cada joc de proves s'obtindran dos resultats:

- **Número d'instruccions de codi màquina necessàries per a executar el programa sense optimitzar**: per a fer això s'executarà en el simulador SPIM el codi màquina generat per **CODEGEN** sense aplicar cap optimització, i es comptaran el nombre d'instruccions executades.
- **Número d'instruccions de codi màquina necessàries per a optimitzar el programa després d'una optimització concreta**: per a fer això s'executarà en el simulador SPIM el codi màquina generat per **CODEGEN** al nivell màxim d'optimització i es comptaran el nombre d'instruccions executades.

S'espera que entre els dos números hi hagi una diferència significativa, que reflexi l'impacte de les optimitzacions. També s'espera que l'impacte de les optimitzacions sigui diferent en funció del tipus de programa que s'està optimitzant.

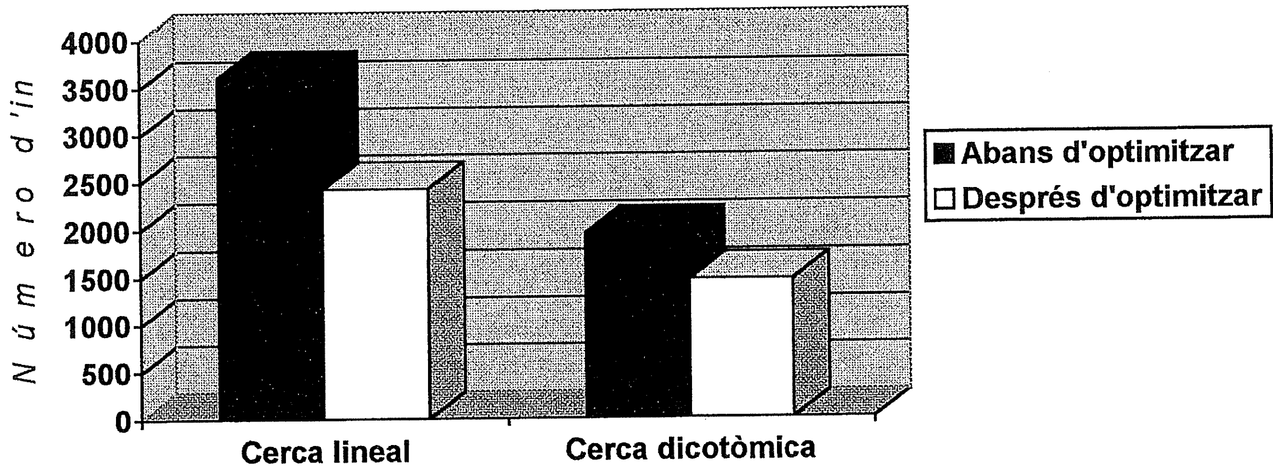
9.2 - Jocs de proves

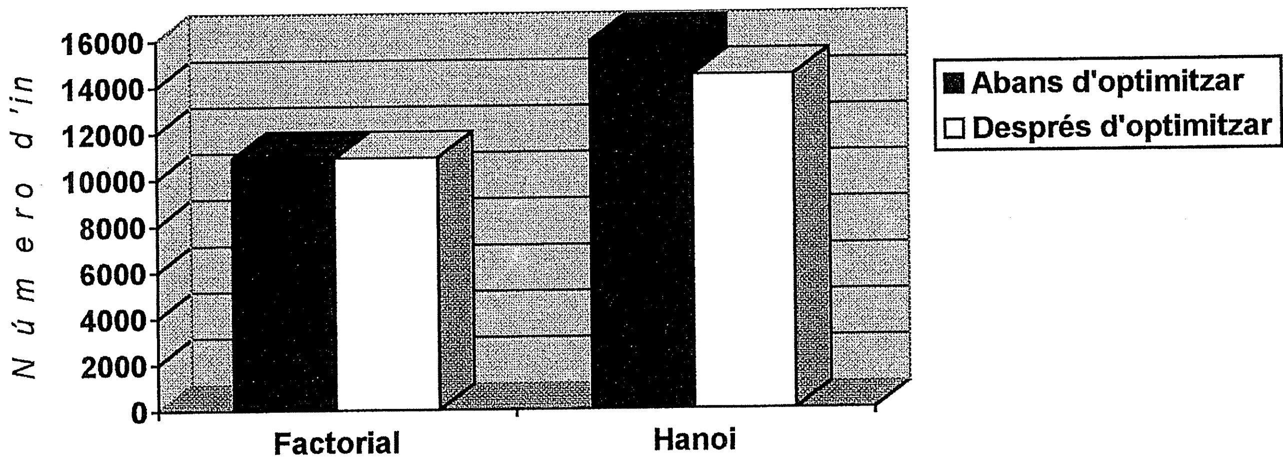
Els programes utilitzats com a jocs de prova intenten reflectir diferents tipus de programes reals, per tal de veure l'actuació de les optimitzacions de codi sobre cada tipus de programa. Els programes sel.leccionats han estat:

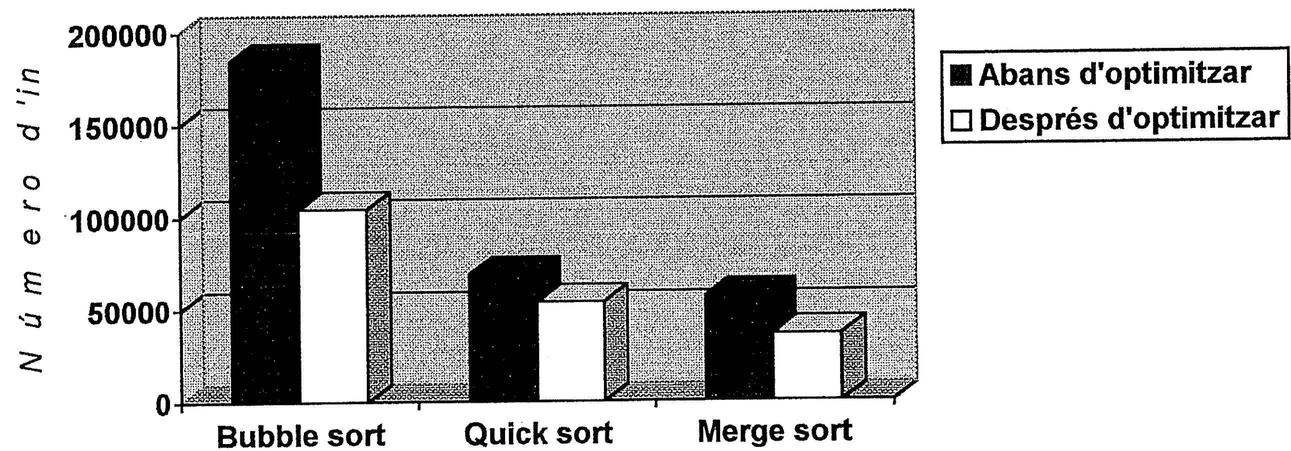
CERQUES	**Lineal**	Cerca lineal d'un element en un vector de 100 elements
	Dicotòmica	Cerca dicotòmica d'un element en un vector de 100 elements
RECURSIU	**Factorial**	Factorial recursiu
	Hanoi	Solució al problema de les torres de Hanoi
ORDENACIÓ	**Bubblesort**	Ordenació d'un vector pel mètode de la bombolla d'un vector de 100 elements
	Quicksort	Ordenarció per classificació d'un vector de 100 elements
	Mergesort	Ordenació per fusió d'un vector de 100 elements
NUMÈRIC	**Suma**	Suma de matrius de 50 x 50 elements
	Producte	Producte de matrius de 10 x 10 elements
	Sedàs d'Eratòstenes	Algorisme per a calcular els nombres primers més petits de 100

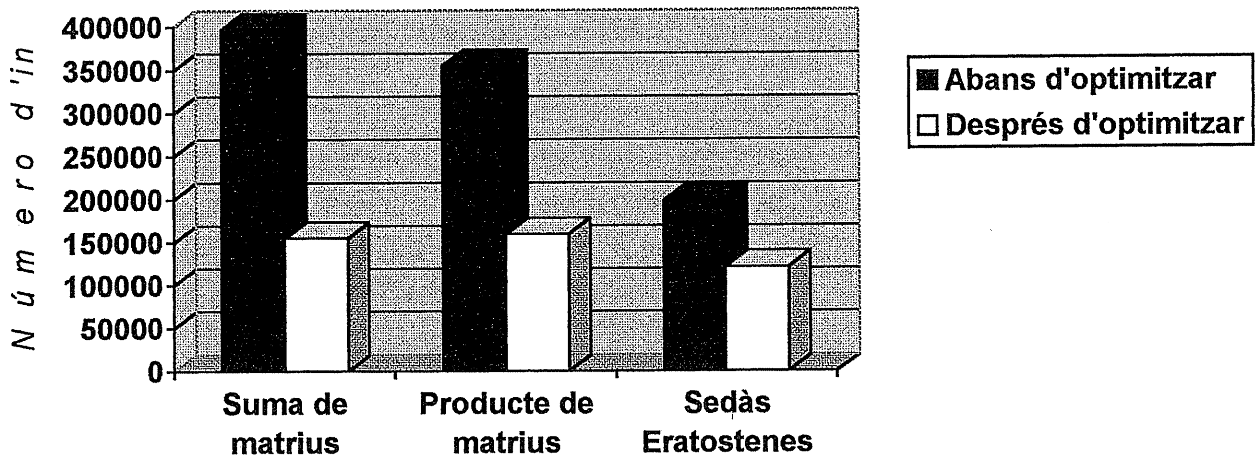
9.3 - Resultats de les proves

Els resultats de les proves es poden veure en les gràfiques següents:

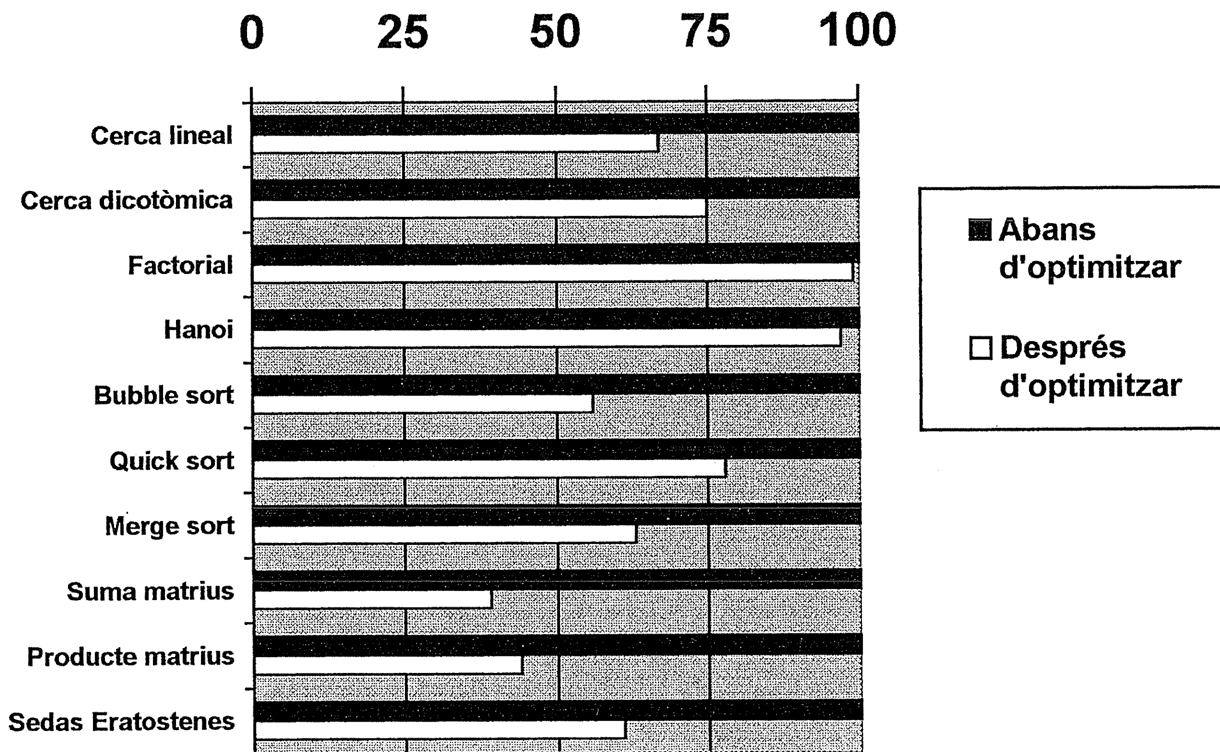








El que es vol comprovar és la **relació** entre el número d'instruccions abans i després de realitzar les optimitzacions de codi (no interessa la quantitat concreta, que depèn del problema). La gràfica següent expressa aquesta relació en forma de percentatge: es considera el temps d'execució del programa original com a 100% i després es representa el percentatge respecte al temps d'execució original que triga el programa optimitzat. Per exemple, la cerca dicotòmica executa un 75% de les instruccions que executava abans d'optimitzar-se; el 25% restant s'ha aconseguit eliminar gràcies a les optimitzacions de codi.



9.4 - Anàlisi dels resultats

Els resultats de les proves indiquen que els programes optimitzats executen d'un 39% a un 99% de les instruccions originals. Per tant, la primera conclusió és que **les optimitzacions realment són efectives**, perquè hi ha una diferència significativa entre un programa sense optimitzar i un programa optimitzat. **El millor dels resultats elimina un 61% de les instruccions del programa original.**

9.4.1 - Resultats en funció del tipus de programa

Hi ha grans diferències en els nivells d'optimització obtinguts en els diferents programes, i per tant, hi ha més programes més susceptibles a ser optimitzats que altres. Més concretament, el grau d'optimització sembla estar relacionat amb una característica del programa: si es calcula utilitzant bucles o utilitzant crides recursives.

RESULTATS		ITERATIU - RECURSIU ?
Cerca lineal	67 %	Iteratiu (1 bucle)
Cerca dicotòmica	75 %	Iteratiu (1 bucle)
Factorial	99 %	Recursiu
Hanoi	97 %	Recursiu
Bubblesort	56 %	Iteratiu (2 bucles imbricats)
Quicksort	78 %	Iteratiu (1 bucle) + Recursiu
Mergesort	63 %	Iteratiu (1 bucle) + Recursiu
Suma de matrius	39 %	Iteratiu (2 bucles imbricats)
Producte de matrius	44 %	Iteratiu (3 bucles imbricats)
Sedàs d'Eratòstenes	61 %	Iteratiu (2 bucles imbricats)

Els programes iteratius (que tenen bucles) són optimitzats en major grau que els programes recursius. Hi ha tres motius que poden explicar això:

- ***Els programes recursius acostumen a ser més senzills que els programes iteratius.*** Per tant, contenen menys càlculs, expressions, salts, etc. i hi ha menys possibilitats de realitzar optimitzacions. Aquest és el cas de Factorial i Hanoi, programes molt senzills.

- Una part important de les instruccions que s'han d'executar en un programa recursiu estan relacionades amb ***el pas de paràmetres, la construcció del bloc d'activació i salvar/restaurar registres***. No es poden fer moltes optimitzacions sobre aquestes instruccions, excepte realitzar una bona assignació de registres. A **CODEGEN** es realitza aquesta assignació de registres per a reduir aquestes instruccions, però és la mateixa amb i sense optimitzacions (per aquest motiu, no es veu la diferència en el temps d'execució).

- *Existeixen optimitzacions concretes per a bucles* (extracció de codi invariant, reducció d'intensitat i eliminació de variables d'inducció) *però no existeixen optimitzacions*

concretes per a funcions recursives. Per aquest motiu, s'optimitzarà més un bucle que un programa recursiu.

Dins dels programes iteratius, també hi ha diversos graus d'optimització. **Com més nivells d'imbricació hi ha en els bucles d'un programa, més es pot optimitzar**. De tota manera, això no es compleix per a la Suma de matrius i el Producte de matrius: la suma de matrius es pot calcular amb dos bucles aniuats i triga un 39% de les instruccions originals; el producte necessita tres bucles aniuats i triga un 44% de les instruccions originals. Hi ha diversos motius per explicar aquestes excepcions:

- Moltes de les optimitzacions de bucle fan referència a les variables d'inducció. *Hi ha bucles que no tenen variables d'inducció* (segons la definició de variable d'inducció de l'apartat 3.8.9.3), i per tant no poden ser optimitzats com els bucles amb variable d'inducció. Per exemple, la cerca lineal té variables d'inducció, mentre que la cerca dicotòmica no en té. Per això, els resultats de la cerca lineal són millors (67 %) que els resultats de la cerca dicotòmica (75 %).

- Per a poder eliminar una variable d'inducció s'han de complir una sèrie de propietats (veure l'apartat 3.8.9.5). *En els bucles en que no es compleixen aquestes propietats es poden eliminar variables d'inducció*. Això fa que l'optimització de codi no pugui ser tan efectiva. Això és el que passa en els casos Quicksort (78 %) i Producte de matrius (44 %): els bucles no són "senzills" i no es poden eliminar variables d'inducció.

9.4.2 - Resultats en funció del tipus d'optimització

Les **optimitzacions de bucles** han tingut un paper molt important en els resultats, permetent els bons resultats obtinguts en la majoria de programes recursius. El més important d'aquestes optimitzacions és, si s'aconsegueix eliminar una instrucció, aquesta millora es veu reflectida a cada volta del bucle.

Les **optimitzacions del fluxe de dades** també han estat molt importants. Bàsicament, les optimitzacions que s'han realitzat provenien dels accessos a bucles, i d'expressions generades per les optimitzacions de variables d'inducció en els bucles. La conclusió que es pot extreure és que aquestes optimitzacions són molt importants, i encara ho són més si el llenguatge font té tipus de més alt nivell (com ara estructures), perquè el càlcul d'adreces en aquests tipus genera moltes expressions que poden ser optimitzades (operació de constants, propagació de constants, eliminació de subexpressions comunes, ...).

Les **optimitzacions de fluxe de control** (optimització de salts, eliminació de codi inabastable) no han realitzat moltes optimitzacions, però això podria ser degut a que el llenguatge CL té estructures de control molt senzilles (IF, WHILE). En canvi, la reescriptura de bucles WHILE..DO com a DO..WHILE s'ha aplicat moltes vegades, perquè els bucles en el llenguatge CL només es poden escriure com a WHILE..DO.

Malgrat que no es reflecteix en les proves, cal destacar que **l'assignació de registres** té un impacte importantíssim en el número d'instruccions d'un programa. Una bona assignació de registres elimina moltes operacions de restaurar/salvar registre a memòria, i pot reduir molt el tamany del bloc d'activació de la pila.

9.5 - Conclusions finals

Dels resultats obtinguts a les proves en destaquen els següents:

Sobre els resultats:

- **Efectivitat:** Les optimitzacions són realment efectives. El millor dels resultats elimina un 61% de les instruccions del programa original.

Sobre els programes:

- **Iteratiu més que recursiu:** Els programes iteratius s'optimitzen més que els programes recursius.
- **Nivell d'imbricació:** De tots els programes iteratius, s'optimitzen més els programes amb més bucles imbricats.
- **Variables d'inducció:** De tots els bucles, s'optimitzen més aquells que tenen variables d'inducció. I dins d'aquests, s'optimitzen més aquells en que es pot eliminar alguna de les variables d'inducció.

Sobre les optimitzacions:

- **Bucles:** Les optimitzacions de bucles proporcionen molt bons resultats en general.
- **Dades:** Les optimitzacions de dades són molt efectives, sobretot en l'accés a bucles. Com més tipus de dades complexes hi hagi (estructures, taules) més adreces s'hauran de calcular en el codi intermedi i més optimitzacions es podran realitzar.
- **Control:** Les optimitzacions de fluxe de control no han resultat tan efectives com es pensava inicialment. Una causa possible és la senzillesa de les estructures del llenguatge CL (IF,WHILE).
- **Assignació de registres:** L'assignació de registres és molt important, sobretot quan es parla de programes que realitzen moltes crides a funció.

D'aquest projecte i aquests resultats es poden extreure algunes conclusions aplicades al món dels compiladors d'ús comercial.

Els llenguatges utilitzats en la realitat (C, JAVA, ...) tenen construccions molt més complexes que el llenguatge CL, tant des del punt de vista dels tipus (estructures, matrius de bits, ...) com del les construccions del llenguatge (while, for, switch, if, while-do, do-while, break, goto, continue...). Això significa que quan es tradueix aquest llenguatge a codi intermedi hi ha més possibilitats de realitzar optimitzacions. A més, aplicant optimitzacions dependents de l'arquitectura el codi pot millorar encara més del que s'ha aconseguit en aquestes proves. Aquest és un avantatge molt gran des del punt de vista dels usuaris del compilador: sense modificar el seu programa, pot obtenir un fitxer executable més ràpid i que ocuparà menys espai.

10 Planificació i cost

Aquest capítol descriu dos aspectes: la planificació de treball del projecte i els anàlisis de costos.

La planificació del treball mostra la planificació del projecte en forma de diagrames de Gantt Per una banda es presenten les tasques a realitzar en el projecte i les planificacions d'aquestes tasques. Per altra banda, es presenta la duració real de les tasques del projecte i els motius d'aquestes desviacions.

Posteriorment, a partir de les dades sobre les hores de treball del projecte, es determina el cost econòmic total del projecte.

10.1 - Planificació del projecte

Aquest projecte va començar el juliol del 1.999 i ha finalitzat l'abril del 2.000, durant el seu desenvolupament 10 mesos. La dedicació a aquest projecte ha estat parcial en els periodes de docència, perquè durant aquest periode he estat cursant les últimes assignatures de la carrera. En els periodes de vacances, la dedicacio ha estat a temps complet.

Les tasques que s'han realitzat en el projecte han estat les següents:

1. **Definició d'un codi intermedi**
   - Estudi de les optimitzacions independents de l'arquitectura
   - Estudi dels possibles codis intermedis
   - Definició d'un llenguatge de codi intermedi (IC)

2. **Desenvolupament d'un compilador final**
   - Estudi de l'arquitectura MIPS R2000
   - Estudi de les tècniques d'assignació de registres i generació de codi
   - Disseny del compilador final
   - Implementació del compilador final
   - Proves del compilador final per eliminar errors

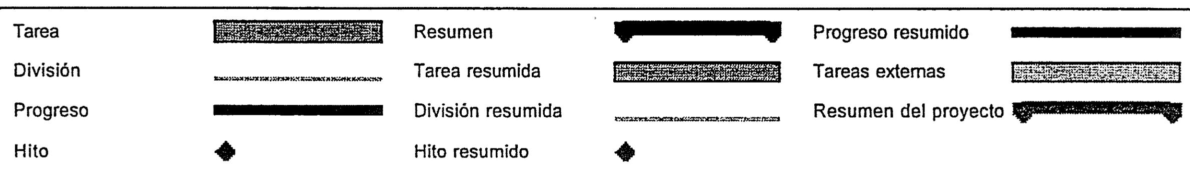
3. **Desenvolupament de les optimitzacions de codi**
   - Selecció de les optimitzacions de codi a realitzar en el projecte i els algorismes utilitzats
   - Implementació de les optimitzacions

Id	Nombre de tarea	Duración
1	**Definició d'un codi intermedi**	**10 días**
2	Estudi de les optimitzacions	50 horas
3	Estudi del codi intermedi	10 horas
4	Definició del codi intermedi	20 horas
5	**Desenvolupament d'un compilador final**	**30 días**
6	Estudi de l'arquitectura MIPS	30 horas
7	Estudi de assig registres i gen codi	25 horas
8	Disseny del compilador final	30 horas
9	Implementació del compilador final	120 horas
10	Prova del compilador final	35 horas
11	**Desenvolupament de les optimitzacions**	**36,88 días**
12	Selecció optimitzacions i algorismes	35 horas
13	Implementació optimitzacions	220 horas
14	Prova de les optimitzacions	40 horas
15	**Proves de les optimitzacions**	**13,13 días**
16	Disseny del compilador frontal	10 horas
17	Implementació del compilador frontal	40 horas
18	Prova del compilador frontal	10 horas
19	Construcció dels jocs de proves	15 horas
20	Mesurar els resultats de les proves	10 horas
21	Anàlisi dels resultats de les proves	20 horas
22	**Redacció de la memòria**	**18,75 días**
23	Redacció de la memòria del projecte	150 horas

julio | agosto | septiembre | octubre

28/06 | 5/07 | 12/07 | 19/07 | 26/07 | 2/08 | 9/08 | 16/08 | 23/08 | 30/08 | 6/09 | 13/09 | 20/09 | 27/09 | 4/10

Entorn per optimització i generació de codi
Planificació



Id	Nombre de tarea	Duración
1	**Definició d'un codi intermedi**	**10 días**
2	Estudi de les optimitzacions	50 horas
3	Estudi del codi intermedi	10 horas
4	Definició del codi intermedi	20 horas
5	**Desenvolupament d'un compilador final**	**30 días**
6	Estudi de l'arquitectura MIPS	30 horas
7	Estudi de assig registres i gen codi	25 horas
8	Disseny del compilador final	30 horas
9	Implementació del compilador final	120 horas
10	Prova del compilador final	35 horas
11	**Desenvolupament de les optimitzacions**	**36,88 días**
12	Selecció optimitzacions i algorismes	35 horas
13	Implementació optimitzacions	220 horas
14	Prova de les optimitzacions	40 horas
15	**Proves de les optimitzacions**	**13,13 días**
16	Disseny del compilador frontal	10 horas
17	Implementació del compilador frontal	40 horas
18	Prova del compilador frontal	10 horas
19	Construcció dels jocs de proves	15 horas
20	Mesurar els resultats de les proves	10 horas
21	Anàlisi dels resultats de les proves	20 horas
22	**Redacció de la memòria**	**18,75 días**
23	Redacció de la memòria del projecte	150 horas

Entorn per optimització i generació de codi
Planificació

Id	Nombre de tarea	Duración
1	**Definició d'un codi intermedi**	**10,25 días**
2	Estudi de les optimitzacions	50 horas
3	Estudi del codi intermedi	10 horas
4	Definició del codi intermedi	22 horas
5	**Desenvolupament d'un compilador final**	**33,5 días**
6	Estudi de l'arquitectura MIPS	30 horas
7	Estudi de assig registres i gen codi	25 horas
8	Disseny del compilador final	33 horas
9	Implementació del compilador final	137 horas
10	Prova del compilador final	45 horas
11	**Desenvolupament de les optimitzacions**	**41,88 días**
12	Selecció optimitzacions i algorismes	35 horas
13	Implementació optimitzacions	252 horas
14	Prova de les optimitzacions	48 horas
15	**Proves de les optimitzacions**	**18,13 días**
16	Disseny del compilador frontal	10 horas
17	Implementació del compilador frontal	56 horas
18	Prova del compilador frontal	11 horas
19	Construcció dels jocs de proves	15 horas
20	Mesurar els resultats de les proves	35 horas
21	Anàlisi dels resultats de les proves	18 horas
22	**Redacció de la memòria**	**22,5 días**
23	Redacció de la memòria del projecte	180 horas

julio | agosto | septiembre | octubre | noviembr

28/06 | 5/07 | 12/07 | 19/07 | 26/07 | 2/08 | 9/08 | 16/08 | 23/08 | 30/08 | 6/09 | 13/09 | 20/09 | 27/09 | 4/10 | 11/10 | 18/10 | 25/10 | 1/11

100% (tasques 1–10 i 12)

Entorn per a generació i optimització de codi
Seguiment del projecte

Tarea crítica	Hito de línea de base	División resumida
División crítica	Hito	Progreso de tarea resumido
Progreso de tarea crítica	Progreso del resumen	Línea de base resumida
Tarea	Resumen	Hito de línea de base resumida
División	Tarea crítica resumida	Hito resumido
Progreso de tarea	División crítica resumida	Tareas externas
Línea de base	Progreso crítico resumido	Resumen del proyecto
División prevista	Tarea resumida	

Id	Nombre de tarea	Duración
1	**Definició d'un codi intermedi**	**10,25 días**
2	Estudi de les optimitzacions	50 horas
3	Estudi del codi intermedi	10 horas
4	Definició del codi intermedi	22 horas
5	**Desenvolupament d'un compilador final**	**33,5 días**
6	Estudi de l'arquitectura MIPS	30 horas
7	Estudi de assig registres i gen codi	25 horas
8	Disseny del compilador final	33 horas
9	Implementació del compilador final	137 horas
10	Prova del compilador final	45 horas
11	**Desenvolupament de les optimitzacions**	**41,88 días**
12	Selecció optimitzacions i algorismes	35 horas
13	Implementació optimitzacions	252 horas
14	Prova de les optimitzacions	48 horas
15	**Proves de les optimitzacions**	**18,13 días**
16	Disseny del compilador frontal	10 horas
17	Implementació del compilador frontal	56 horas
18	Prova del compilador frontal	11 horas
19	Construcció dels jocs de proves	15 horas
20	Mesurar els resultats de les proves	35 horas
21	Anàlisi dels resultats de les proves	18 horas
22	**Redacció de la memòria**	**22,5 días**
23	Redacció de la memòria del projecte	180 horas

Entorn per a generació i optimització de codi
Seguiment del projecte

Tarea crítica
División crítica
Progreso de tarea crítica
Tarea
División
Progreso de tarea
Línea de base
División prevista
Hito de línea de base
Hito
Progreso del resumen
Resumen
Tarea crítica resumida
División crítica resumida
Progreso crítico resumido
Tarea resumida
División resumida
Progreso de tarea resumido
Línea de base resumida
Hito de línea de base resumida
Hito resumido
Tareas externas
Resumen del proyecto

10.2 - Durada real del projecte

A les pàgines anteriors es troben els diagrames de Gantt de seguiment del projecte, que mostren la durada real de les etapes del projecte i la desviació respecte a la previsió.

La durada real de les tasques del projecte ha estat la següent:

Tasca	Perfil	Data inici	Data fi	H/dia	Total hores
Estudi de les optimitzacions	A	1 juliol	9 juliol	8 h/dia	50 hores
Estudi del codi intermedi	A	9 juliol	12 juliol	8 h/dia	10 hores
Definició del codi intermedi IC	A	12 juliol	15 juliol	8 h/dia	22 hores
Estudi de l'arquitectura MIPS	A	15 juliol	20 juliol	8 h/dia	30 hores
Estudi assig. registres i gen. codi	A	20 juliol	23 juliol	8 h/dia	25 hores
Disseny del compilador final	A	23 juliol	29 juliol	8 h/dia	33 hores
Implementació del compilador final	P	30 juliol	10 setembre	8 h/dia	137 hores
Prova del compilador final	P	13 setembre	22 setembre	5 h/dia	45 hores
Selecció optimitzacions i algorismes	A	22 setembre	28 setembre	4 h/dia	35 hores
Implementació optimitzacions	P	28 setembre	3 desembre	4 h/dia	252 hores
Prova de les optimitzacions	P	3 desembre	16 desembre	3 h/dia	48 hores
Disseny del compilador frontal	A	16 desembre	20 desembre	3 h/dia	10 hores
Implementació del compilador frontal	P	20 desembre	4 gener	5 h/dia	56 hores
Prova del compilador frontal	P	5 gener	7 gener	5 h/dia	11 hores
Construcció dels jocs de proves	P	7 gener	12 gener	5 h/dia	15 hores
Mesurar els resultats de les proves	P	12 gener	21 gener	5 h/dia	35 hores
Anàlisi dels resultats de les proves	A	21 gener	26 gener	5 h/dia	18 hores
Redacció de la memòria del projecte	A	26 gener	17 març	5 h/dia	180 hores

TOTALS	Total hores
Definició d'un codi intermedi	82 hores
Desenvolupament d'un compilador final	270 hores
Desenvolupament de les optimitzacions de codi	335 hores
Proves de les optimitzacions	145 hores
Redacció de la memòria del projecte	180 hores
TOTAL DEL PROJECTE	**1.012 hores**

10.3 - Motius de les desviacions

Hi ha diversos motius que han provocat una desviació respecte les previsions inicials:

- **Manca d'experiència en planificació de projectes**

El motiu més important de la desviació és la manca d'experiència en planificar projectes. L'experiència en la planificació és molt important, i ajuda a avaluar de forma molt més exacta la duració de les diferents tasques. Per exemple, la duració prevista de les tasques de

codificació (implementació) ha estat inexacta. La duració real ha superat àmpliament la previsió.

- **Dedicació a les assignatures més important del que s'havia previst**

He compaginat el projecte amb les últimes assignatures de la carrera, i la dedicació que han requerit aquestes assignatures ha estat superior al que havia previst en la planificació. Això ha fet disminuir la dedicació al projecte, enderrerint les dates de realització de les diferents tasques a partir del moment en que van començar les classes. D'aquesta manera, encara que la durada d'algunes tasques s'ha ajustat a la previsió, han durat un major nombre de dies.

- **Dificultat de comprovar la correctesa**

Per a comprovar la correctesa dels diferents mòduls, s'ha de provar amb el mòdul amb una certa entrada i comprovar que el resultat s'adapta al que havíem previst. El problema és que algun dels mòduls realitzen tasques complexes (assignar registres, optimitzar bucles, ...) i per tant, calcular la sortida esperada manualment ha estat costós. Per exemple, per a provar l'assignació de registres cal realitzar l'anàlisi de vida, construir el graf d'interferència, calcular les heurístiques de totes les variables i aplicar l'algorisme de coloració de registres; fins i tot en un programa senzill, fer això és costós.

Per altra banda, el resultat de la generació de codi màquina és un programa escrit en assemblador de MIPS. En cas de detectar algun error en el codi MIPS de sortida, calia detectar on es produia aquest error. Això significa depurar un programa escrit en llenguatge assemblador, un procés molt lent.

Aquests dos factors han fet que el temps dedicat a les tasques de prova hagi estat superior al que s'havia previst inicialment.

10.4 - Anàlisi de costos

El cost econòmic del projecte ha estat calculat a partir de les hores de treball dedicades al projecte i el perfil necessari per a la seva realització (encara que en realitat les tasques només hagin estat realitzades per una persona).

Els costos associats a cada perfil són els següents:

- **Analista:** Persona que realitza tasques de disseny i documentació. El seu sou s'ha estimat en 5.000 ptes/hora.
- **Programador:** Persona que realitza tasques de codificació i proves d'un sistema. Té un sou estimat en 2.500 ptes/hora.

El treball dedicat al projecte ha estat de 1012 hores, de les quals 413 hores corresponen a un analista i 599 corresponen a un programador.

Cost de 413 hores de treball d'analista:	413 hores x 5.000 ptes/hora = 2.065.000 ptes
Cost de 599 hores de treball de programador:	599 hores x 2.500 ptes/hora = 1.497.500 ptes
Cost total del projecte:	**3.562.500 ptes**

No s'ha comptabilitat en aquest anàlisi el cost dels recursos utilitzats en aquest projecte que han estat cedits per la Universitat Politècnica de Catalunya.

Annexos

En aquesta secció s'han inclós tots aquells continguts que, per la seva extensió, no es podien incloure en cap dels apartats anteriors.

Els diferents annexos contenen: els algorismes utilitzats per a realitzar l'optimització de codi; la demostració de que el procés d'optimització acaba; alguns comentaris sobre les eines utilitzades en el projecte; les gramàtiques dels llenguatges CL i IC; i un petit manual d'usuari de **CODEGEN**.

A.1 - Algorismes d'optimització

En el capítol 3 s'han presentat els diferents anàlisis i optimitzacions que es poden realitzar sobre el codi intermedi. A continuació es presenta, per a cada anàlisi i optimització, l'algorisme en alt nivell que s'ha utilitzat per a realitzar l'optimització. Abans de presentar aquest algorismes, es recorda com s'organitzen les diferents optimitzacions i anàlisis.

```
RONDA_OPTIMITZACIÓ (funció f)

    (* Aplicar totes les optimitzacions *)
    anàlisis requerits per la optimització 1;
    f := aplicar optimització 1 a tota la funció f;
    anàlisis requerits per la optimització 2;
    f := aplicar optimització 2 a tota la funció f;
    ....
    retorna f;


OPTIMITZAR_CODI (funció f)

    f_optimitzada = f ;
    fer
        f_entrada := f_optimitzada;
        f_optimitzada := ronda_optimització (f_optimitzada);
    mentre f_optimitzada ≠ f_entrada;
    retorna f_optimitzada;
```

```
Algorisme detectar blocs bàsics
Entrada
    Llista d'instruccions de la funció:  f
Sortida
    Llista de blocs bàsics que contenen les instruccions de la funció: blocs

blocs := llista_buida ();
b := crear_bloc_bàsic ();
per a cada instrucció i de la funció f
    afegir_instrucció (b, i);
    si i és un salt o i és un RETURN o i és destí d'un salt llavors
        (* Després de la instrucció "i" comença un nou bloc bàsic *)
        afegir_bloc (blocs,b);
        b:= crear_bloc_bàsic ();
    fsi
fper
afegir_bloc (blocs,b);
retorna blocs
```

```
Algorisme construir graf de fluxe de control
Entrada
    Llista de blocs bàsics de la funció: blocs
Sortida
    Llista de blocs bàsics, on cada bloc té associat una llista de predecessors i
    successors

bloc_inicial := crear_bloc_bàsic ();
afegir_bloc_al_principi (blocs, bloc_inicial);
ts := crear_taula_símbols ();
per a cada bloc bàsic b de la llista blocs
    afegir_taula_símbols (ts, etiqueta (b), b);
    predecessors (b) := llista_buida ();
    sucessors (b) := llista_buida ();
fper
per a cada bloc bàsic b de la llista blocs
    i := última_instrucció (b);
    si  (b no és l'últim node de blocs) i
        (b no conté instruccions o
        i és un salt incondicional o i és un RETURN) llavors
         següent := següent_bloc (blocs, b);
         afegir_aresta (b, següent);
    fsi
    si (b conté instruccions i i és un salt) llavors
         et := etiqueta_destí_salt (i);
         destí := consultar_taula_símbols (ts,et);
         afegir_aresta (b,destí):
    fsi
fper

afegir_aresta (bloc_origen, bloc_destí)
    si bloc_destí ∉ sucessors (bloc_origen) llavors
        afegir (sucessors (bloc_origen), bloc_destí)
        afegir (predecessors (bloc_destí), bloc_origen)
    fsi
```

```
Algorisme calcular dominadors
Entrada
    Graf de control de fluxe de la funció: cfg
Sortida
    Graf de control de fluxe amb informació dels dominadors de cada node: cfg

per a cada bloc bàsic b del graf cfg
    b.dominadors := llista_buida();
    copiar_llista_blocs (b.dominadors, cfg);
fper
primer := bloc_bàsic_inicial (cfg);
primer.dominadors := {primer};
fer
    canvis:= 0;
    per a cada bloc bàsic b del graf cfg
        tmp := llista_buida();
        copiar_llista_blocs (tmp, cfg);
        per a cada predecessor p del node b
            tmp := tmp ∩ p.dominadors;
        fper
        tmp := tmp ∪ {b};
        si b.dominadors ≠ tmp llavors
            canvis:= canvis +1;
        fsi
        b.dominadors:= tmp;
    fper
mentre canvis ≠ 0
retorna cfg
```

```
Algorisme construir arbre de dominadors
Entrada
     Graf de control de fluxe amb informació de dominadors: cfg
Sortida
     Arbre de dominadors, en forma d'apuntadors als pares i als fills

per a cada bloc bàsic b del graf cfg
     b.pare := NULL;
     b.fills := llista_buida();
fper
per a cada bloc bàsic b del graf cfg
     pare_trobat := fals
     per a cada bloc bàsic d de la llista de dominadors de b
     i mentre no pare_trobat fer
          domina := fals
          per a cada bloc bàsic e de la llista de dominadors de b
          i mentre no domina  fer
               si d ≠ e i e ≠ b i d domina e llavors
                    domina := cert;
               fsi
          fper
          si domina = fals llavors
               b.pare := d;
               afegir_fill (d.fills,b);
               pare_trobat := cert;
          fsi
     fper
fper
```

```
    Algorisme trobar bucles
    Entrada
        Graf de fluxe de control amb informació de dominadors: cfg
    Sortida
        Llista de bucles: bucles

    bucles := llista_buida ();
    per a cada bloc bàsic b del graf cfg
        per a cada successor s del bloc bàsic b
            si s DOM b llavors
                (* Aresta de retrocés: s és l'encapçalament del bucle *)
                bucle:= construir_bucle (cfg,b, s)
                afegir_bucle (bucles,bucle);
            fsi
        fper
    fper
    retorna bucles

construir_bucles (cfg, n, encapçalament)
    bucle := crear_bucle ();
    bucle.encapçalament := encapçalament;
    bucle.nodes := llista_buida();
    afegir_node (bucle, encapçalament);
    afegir_node (bucle, n);
    si (encapçalament ≠ n) llavors
        (* Bucle de més d'un node *)
        per a cada predecessor p del bloc bàsic n
            examinar_node (p, bucle);
        fper
    retorna bucle;

examinar_node (n, bucle)
    si (bucle.encapçalament DOM n) i (n ∉ bucle) llavors
        afegir_node (bucle, n);
        per a cada predecessor p del bloc bàsic n
            examinar_node (p,bucle)
        fper
    fsi
```

```
Algorisme fusionar i ordenar bucles
Entrada
    Llista de bucles: bucles
Sortida
    Llista de bucles ordenada parcialment, segons el criteri "els bucles imbricats
    abans que els bucles externs": arbre_bucles

arbre_bucles := llista_buida ();
per a cada bucle b de la llista bucles
    altre := trobar_encapçalament (arbre_bucles, b.encapçalament)
    si altre = NULL llavors
        (* Cap bucle comparteix l'encapçalament de b fins al moment *)
        inserir_ordenat_bucle (arbre_bucles,b);
    sino
        (* b i altre comparteixen encapçalament : cal fusionar-los *)
        altre.nodes := altre.nodes ∪ b.nodes;
    fsi
fper
retorna arbre_bucles

trobar_encapçalament (arbre_bucles,encapçalament)
    per a cada bucle b de la llista arbre_bucles
        si b.encapçalament = encapçalament llavors
            retorna b
        fsi
    fper
    retorna NULL

inserir_ordenat_bucle (arbre_bucles, b)
    últim_dominat := NULL;
    per a cada bucle c de la llista arbre_bucles
        si b.encapçalament DOM c.encapçalament llavors
            (* Bucle c imbricat en b: b va després de b *)
            ultim_dominat := c;
        fsi
    fper
    si ultim_dominat = NULL llavors
        inserir_principi (arbre_bucles,b);
    sino
        inserir_després (arbre_bucles, últim_dominat,b);
    fsi
```

```
Algorisme calcular frontera de dominació
Entrada
    Graf de fluxe de control amb informació de dominadors i l'arbre de
    dominadors: cfg
Sortida
    Graf de fluxe amb informació de la frontera de dominació: cfg

per a cada bloc bàsic b del graf cfg
    b.frontera := llista_buida();
fper
primer := bloc_bàsic_inicial (cfg);
construir_frontera_dominació (primer);

construir_frontera_dominació (n)
    df := llista_buida ();
    per a cada successor s del bloc bàsic n
        si s ∉ n.fills llavors
            (* s ∈ DFlocal [n] per la definició de DFlocal [n] *)
            df:= df ∪ s;
        fsi
    fper
    per a cada fill f del bloc bàsic n
        df_up := construir_frontera_dominació (f);
        per a cada node d de la llista df_up
            si d no domina estrictament n llavors
                (* d ∈ DFup [c] segons la definició de DFup [c] *)
                df:= df ∪ d;
            fsi
        fper
    fper
    n.frontera := df;
    retorna df
```

```
Algorisme anàlisi de vida
Entrada
    Graf de fluxe de control: cfg
Sortida
    Graf de fluxe de control amb informació de les variables vives al inici (b.in) i
    al final (b.out) de cada bloc bàsic

per a cada bloc bàsic b del graf cfg
    b.in:= llista_buida ();
    b.out:= llista_buida ();
    calcular_use_def (b);
fper
fer
    hi_ha_canvis := fals;
    per a cada bloc bàsic b del graf cfg
        in_anterior := b.in;
        out_anterior := b.out;
        b.in := b.use ∪ (b.out - b.def);
        b.out := llista_buida();
        per a cada successor s del bloc bàsic b
            b.out := b.out ∪ s.in;
        fper
        si (b.in ≠ in_anterior) o (b.out ≠ out_anterior) llavors
            hi_ha_canvis := cert;
        fsi
    fper
mentre (hi_ha_canvis);

calcular_use_def (b)
    b.use := llista_buida ();
    b.def := llista_buida ();
    per a cada instrucció i del bloc bàsic b
        b.use := b.use ∪ (variables_usades(i) - b.def);
        b.def := b.def ∪ variables_definides(i);
    fper

variables_usades(ins)
variables_definides(ins)
    Conjunts de variables usades i definides per cada instrucció; si la instrucció
    usa/defineix la memòria, contenen totes les variables que es troben a memòria.
```

```
Algorisme construir cadenes d'ús i definició
Entrada
    Graf de fluxe de control : cfg
Sortida
    Graf de fluxe de control, amb informació sobre els usos (i.use) i definicions
    (i.def) associada a cada instrucció: cfg

per a cada bloc bàsic b del graf cfg
    per a cada instrucció i del bloc bàsic b
        i.use := llista_buida();
        i.def := llista_buida();
        per a cada variable v usada a i
            buscar_definicions (b,i,v,MATEIX_BLOC)
        fper
        per a cada variable v definida a i
            buscar_usos (b,i,v,MATEIX_BLOC)
        fper
    fper
fper

buscar_definicions (b,ins,v,on)
    si on = MATEIX_BLOC llavors
        (* b és el bloc on es troba la instrucció ins *)
        examinar:= instruccions anteriors a ins en el bloc bàsic b;
    sino
        (* b és un antecessor del bloc on es troba ins *)
        examinar:= totes les instruccions del bloc bàsic b;
    fsi
    d:= ultima definició de la variable v en la llista examinar;
    si d existeix llavors
        (* Hi ha una definició en el bloc bàsic: aturem la cerca *)
        afegir_instrucció (ins.def, d);
    sino
        (* Buscar als blocs bàsics anteriors *)
        per a cada predecessor p del bloc bàsic b
            si p no ha estat visitat per a aquesta variable v llavors
                buscar_definicions(p,ins,v,ANTECESSOR);
            fsi
        fper
    fsi

buscar_usos (b,ins,v,on)
    si on = MATEIX_BLOC llavors
        (*b és el bloc on es troba la instrucció ins *)
        examinar:= instruccions posteriors a ins en el bloc bàsic b;
    sino
        (* b és un descendent del bloc on es troba ins *)
        examinar:= totes les instruccions del bloc bàsic b;
    fsi
```

```
Algorisme optimització de salts
Entrada
    Graf de fluxe de control: cfg
Sortida
    Graf de fluxe de control amb salts optimitzats: cfg

per a cada bloc bàsic b del graf cfg
    ins := ultima_instrucció (b);
    si ins existeix i ins és un salt llavors
        substituir_patro (cfg,b,ins)
    fsi
fper

substituir_patro (cfg,b,ins)
    Aquesta rutina busca els quatre patrons indicats anteriorment i, en cas de
    trobar-los, realitza la substitució pel patró més eficient (actualitzant les llistes
    de predecessors i successors del graf de fluxe de control).
```

```
Algorisme eliminar codi mort
Entrada
    Graf de fluxe de control amb cadenes d'ús i definició calculades: cfg
Sortida
    Graf de fluxe de control optimitzat: cfg

codi_mort := llista_buida ();
per a cada instrucció ins del graf cfg
    si es_buit( ins.use ) i no efectes_laterals (ins) llavors
        afegir_instrucció (codi_mort, ins);
    fsi
fper
per a cada instrucció ins de la llista codi_mort
    per a cada variable var usada a ins
        per a cada instrucció j de la llista de definicions ins.var.def
            (* j defineix una variable usada a ins: eliminem ins de la llista *)
            (* d'usos de la instrucció j i comprovem si j passa a ser codi mort *)
            eliminar_instrucció (j.use, ins);
            si es_buit (j.use) i no efectes_laterals (j) llavors
                afegir_instrucció (codi_mort,ins);
            fsi
        fper
    fper
    eliminar_instrucció (codi_mort, ins);
    eliminar_instrucció (cfg, ins);
fper
```

es_buit (llista_usos)

Aquesta funció retorna cert si la instrucció defineix algun operand i té una llista d'usos buida. Si la instrucció no defineix cap operand o té una llista d'usos buida, retorna fals.

```
Algorisme propagació de constants
Entrada
    Graf de fluxe de control amb cadenes d'ús i definició: cfg
Sortida
    Graf de fluxe de control optimitzat: cfg

per a cada instrucció ins del graf cfg
    si ins és de la forma var1:= constant llavors
        per a cada instrucció j de la llista d'usos ins.use
            si la única definició de var que arriba a j és ins i
               j no és de la forma var2: = &var1 llavors
                 substituir_operand (j, var1, constant);
            fsi
        fper
    fsi
fper
```

```
Algorisme operació de constants en temps de compilació
Entrada
    Graf de fluxe de control: cfg
Sortida
    Graf de fluxe de control optimitzat: cfg

per a cada instrucció ins del graf cfg
    si ins és una instrucció de conversió de tipus llavors
        si l'operand de ins és constant llavors
            calcular_resultat (ins);
        fsi
    sino si ins és una instrucció aritmètica llavors
        si tots els operands són constants llavors
            calcular_resultat (ins);
        sino si un operand és el neutre o l'absorvent llavors
            calcular_resultat (ins);
        sino si ins és un producte i un operand és 2^m llavors
            substituir_producte_lshift (ins);
        fsi
    fsi
fper
```

```
Algorisme eliminar variables inutils i NOPs
Entrada
    Funció, formada per un graf de fluxe de control (fc.cfg) i les llistes de
    variables escalars i estructurades (fc.scalars, fc.structs): fc
Sortida
    Funció optimitzada, sense NOPs ni variables inútils: fc

usades := llista_buida ();
per a cada instrucció ins del graf fc.cfg
    si ins és un NOP llavors
        eliminar_instrucció (fc.cfg, ins);
    sino
        usades:= usades ∪ variables_usades (ins) ∪ variables_definides (ins);
    fsi
fper
fc.scalars := fc.scalars ∩ usades;
fc.structs:= fc.structs ∩ usades;
```

```
Algorisme eliminar subexpressions comunes
Entrada
    Graf de fluxe de control: cfg
Sortida
    Graf de fluxe de control optimitzat: cfg

per a cada instrucció ins del graf cfg
    si ins conté una expressió llavors
        escriure_forma_canonica (ins);
    fsi
fper
per a cada bloc bàsic b del graf cfg
    per a cada instrucció ins del bloc bàsic b
    si ins conté una expressió llavors
            buscar_expressions_comunes (b,ins);
        fsi
    fper
fper

buscar_expressions_comunes (bloc,ins)
    previes: = definicions_previes (bloc,ins,MATEIX_BLOC);
    si previes ≠ ∅ llavors
        tmp:= nova_variable ();
        per a cada instruccio j de la llista previes
            var:=  j.destí;
            j.destí:= tmp;                  (* var:= expressio → tmp := expressio *)
            afegir_còpia (bloc,j,tmp,var);  (* var:= tmp *)
        fper
        reescriure_ins (ins,tmp);           (* desti:= expressio → desti := tmp *)
    fsi

definicions_previes (bloc,ins,on)
    trobat:= fals;
    exp_matada := fals;
    def_previes := llista_buida ();
    si on = MATEIX_BLOC llavors
        (* b és el bloc on es calcula la expressió, a la instrucció ins *)
        ant:= instrucció_anterior (bloc,ins);
    sino
        (* b és un antecessor del bloc on es calcula ins*)
        ant:= ultima_instrucció (bloc);
    fsi
    mentre (no trobat) i (ant existeixi) i (no exp_matada) fer
        si mata_expressió (ins,ant) llavors
            exp_matada := cert;
        sino si mateixa_expressio (ins,ant) i (ant ≠ ins) llavors
            trobat:= cert;
        sino ant = instrucció_anterior(bloc,ant);
        fsi
```

```
fmentre
si trobat llavors
     afegir_instruccio (def_previes, ant);
sino
     per a cada precedessor p del bloc bàsic bloc
          si p no ha estat visitat per a aquesta expressió llavors
               previes := definicions_previes(p,ins,ANTECESSOR);
               si previes = ∅ llavors (* L'expressió no està disponible en un camí *)
                    retorna ∅;         (* No podem eliminar expressions comunes *)
               fsi
               def_previes := def_previes ∪ previes;
          fsi
     fper
fsi
retorna (def_previes);
```

```
Algorisme propagació de còpies
Entrada
    Graf de fluxe de control amb cadenes d'ús i definició: cfg
Sortida
    Graf de fluxe de control optimitzat: cfg

per a cada bloc bàsic b del graf cfg
    per a cada instrucció ins del bloc bàsic b
        si ins és una instrucció de còpia llavors
            si ins és de tipus a:= a llavors
                eliminar_instrucció (b,ins);
            sino
                propagar_còpia (b,ins);
            fsi
        fsi
    fper
fper

propagar_còpia (b,ins)
    per a cada instrucció j de la llista d'usos ins.use
        (* ins és una instrucció de tipus a:= b, j és la instrucció que usa ins *)
        si ins és la única definició de la llista de definicions de j
          i ins no és de tipus e:=&a i b no és definida en el camí de j a ins llavors
            substituir (j, b,a );          (* c:= a ⊕ d → c:= b ⊕ d *)
        fsi
    fper
```

```
Algorisme crear preencapçalament
Entrada
    Bucle: bucle
    Graf de fluxe de control: cfg
Sortida
    Bucle amb preencapçalament afegit: bucle

preencapçalament := crear_bloc_bàsic ();
encapçalament := bucle.encapçalament;
afegir_node (cfg, encapçalament);
per a cada predecessor p del bloc bàsic encapçalament
    si p ∉ bucle.nodes llavors
        eliminar_aresta (p, encapçalament);
        afegir_aresta (p, preencapçalament);
        ins := ultima_instruccio (p);
        si ins existeix i ins és un salt llavors
            (* Canviar l'etiqueta de destí del salt *)
            canviar_etiqueta (ins, encapçalament.etiq, preencapçalament.etiq);
        fsi
    fsi
fper
afegir_aresta (preencapçalament, encapçalament);
recalcular_dominadors (cfg, preencapçalament, encapçalament);
bucle.preencapçalament := preencapçalament;

recalcular_dominadors (cfg, preencapçalament, encapçalament)
    (* El preencapçalament domina tots els nodes del bucle i l'encapçalament *)
    preencapçalament.dominadors:= (encapçalament.dominadors - encapçalament)
                                        ∪ preencapçalament;
    per a cada bloc bàsic b del graf cfg
        si encapçalament ∈ b.dominadors llavors
            b.dominadors := b.dominadors ∪ preencapçalament;
        fsi
    fper
    (* El pare de encapçalament passa a ser preencapçalament *)
    preencapçalament.fills := llista_buida();
    afegir_node (preencapçalament.fills,encapçalament);
    (* L'anterior pare d'encapçalament passa a ser el pare de preencapçalament *)
    pare := pare (cfg,encapçalament);
    eliminar_node (pare.fills, encapçalament);
    afegir_node (pare.fills, preencapçalament);
```

```
Algorisme trobar instruccions invariants
Entrada
      Bucle amb cadenes d'ús i definició: bucle
 Sortida
      Llista d'instruccions invariants: invariants

 bucle.invariants:= llista_buida ();
 candidats:= llista_buida ();
 per a cada instrucció ins del bucle bucle
      si és_invariant (bucle,ins) llavors
           afegir_instrucció (bucle.invariants, ins);
      sino
           afegir_instrucció (candidats, ins);
      fsi
 fper
 per a cada instrucció ins de la llista candidats
      si és_ invariant (bucle, ins) llavors
           afegir_instrucció (bucle.invariants, ins);
           candidats:= (candidats - ins) ∪ (usos(ins) - candidats - bucle.invariants);
      fsi
 fper
 retorna bucle.invariants

és_invariant (bucle, ins)
 si instrucció_defineix_memòria (ins) o ins no defineix cap operand o
    algun operand de ins a memòria llavors
      invariant := fals;
 sino
      per a cada operand op de la instrucció ins
           invariant :=  "op" és constant o
                 "op" només té una definició d que és dins el bucle i és invariant o
                 totes les definicions de "op" són fora del bucle;
      fper
 fsi
 retorna invariant
```

```
    Algorisme extracció d'expressions invariants
    Entrada
        Bucle amb cadenes d'ús i definició calculades: bucle
    Sortida
        Bucle optimitzat: bucle

    per a cada instrucció ins de la llista bucle.invariants
        si pot_ser_extreta (bucle, ins) llavors
            extreure_instrucció (bucle,ins);
        sino
            extreure_avaluació (bucle,ins);
        fsi
    fper

pot_ser_extreta (bucle, ins)
    retorna  condició0 (bucle,ins) i condició2 (bucle,ins) i condició3 (bucle,ins) i
             condició4 (bucle, ins);

extreure_instrucció (bucle, ins)
    bb := bloc_bàsic (ins);
    eliminar_instrucció (bb, ins);
    afegir_instrucció (bucle.preencapçalament,ins);

extreure_avaluació (bucle, ins);
    si condició0 (bucle,ins) i condició1 (bucle,ins) i
       no ins és una instrucció de còpia llavors
        tmp:= nova_variable ();
        avaluació := copiar_instrucció (ins);
        avaluació.destí := tmp;  (* a:= b ⊕ c → tmp:= b ⊕ c *)
        afegir_instrucció (bucle.preencapçalament, avaluació);
        ins.font1 := tmp;
        ins.tipus := CÒPIA;       (*a:= b ⊕ c → a:= tmp *)
    fsi

condicióN (bucle,ins)
    Comprovar la condició N per a la extracció d'instruccions invariants
```

```
Algorisme detectar variables d'inducció
Entrada
    Funció: fc
    Bucle amb la llista d'instruccions invariants i l'anàlisi de vida: bucle
Sortida
    Bucle amb un conjunt de ternes de variables d'inducció: bucle

candidats := llista de variables que no estan a memòria i són definides en
             algun bloc bàsic del bucle (aquestes poden ser vars. d'inducció);
buscar_variables_bàsiques (bucle,candidats);
si candidats ≠ ∅ llavors
    buscar_variables_derivades (bucle,candidats);
fsi

buscar_variables_bàsiques (bucle, candidats)
    basiques := copiar_llista (candidats);
    per a cada instrucció ins del bucle bucle
        si ins no és una instrucció de tipus "var:= var ± invariant" llavors
            basiques:= basiques - variable_destí (ins);
        fsi
    fper
    per a cada variable v de la llista basiques
        terna (v) = (v,1,0);
        definició (v) = NULL;
    fper
    candidats := candidats - bàsiques;
    bucle.inducció := basiques;

buscar_variables_derivades (bucle, candidats)
    derivades := llista_buida();
    fer
        per a cada instrucció ins del bucle bucle
            si ins és una instrucció del tipus "var := ind ⊗ invariant" i
               ind és una variable d'inducció bàsica o derivada i es
               verifiquen totes les condicions de la definició de var. derivada llavors
                (* var es una variable derivada *)
                derivades := derivades ∪ var;
                candidats := candidats - var;
                terna (var) := calcular_terna (ind,ins);
                definició (var) := ins;
            sino si ins és del tipus "var := candidat ⊗ invariant" llavors
                (* Esperem a veure si candidat és variable d'inducció o no *)
            sino
                candidats := candidats - variable_destí (ins);
            fsi
        fper
    mentre (afegim noves variables a "derivades" a cada volta);
    bucle.inducció := bucle.inducció ∪ derivades;
```

```
Algorisme reducció d'intensitat de variables d'inducció
Entrada
    Bucle amb cadenes d'ús i definició i el conjunt de variables d'inducció: bucle
Sortida
    Bucle optimitzat: bucle

per a cada variable d'inducció bàsica bas del conjunt bucle.inducció
    família:= conjunt de variables derivades que tenen "bas" com a
              variable bàsica;
    per a cada variable v de la llista família
        nova_var := variable_intensitat_reduida (bucle,v);
        per a cada variable w de la llista família amb terna(v) = terna(w)  (v incl.)
            família:= família - w;
            canviar la instrucció definició(w), que calcula "w:= ind * invariant",
             per  la instrucció "w:= nova_var";
        fper
    fper
fper

variable_intensitat_reduida (bucle,var)
    Sigui terna(var) de la forma (bas,factor,desp) (* Notació *)
    nova_var := nova_variable();
    (* Inicialitzar la variable "nova_var" en el preencapçalament *)
     Afegir al final del preencapçalament les instruccions:
        "nova_var := bas * factor"
        "nova_var := nova_var + desp"
    per a cada instrucció ins del bucle bucle
        si ins defineix la variable bàsica bas llavors
            (* Recalcular el valor de "nova_var" després de canviar la var. bàsica *)
            Sigui la instrucció ins de la forma "bas := bas ⊕ inv "; (* Notació *)
            si inv i factor són constants llavors
                const := inv*factor;
                Afegir després de la instrucció ins:
                    "nova_var := nova_var + const"
            sino
                Afegir després de la instrucció ins:
                    "tmp := factor * inv"
                    "nova_var := nova_var + tmp"
            fsi
        fsi
    fper
    retorna nova_var
```

```
Algorisme eliminar variables d'inducció
Entrada
    Bucle amb informació de les variables d'inducció i càlculs invariants: bucle
Sortida
    Bucle optimitzat: bucle

INS:= variables_inservibles (bucle);
GINS:= variables_gairebé_inservibles (bucle);
ALTRES:= bucle.inducció - INS - GINS;
reescriure_comparacions (bucle, GINS, ALTRES);
per a cada variable v de la llista GINS
    si s'han reescrit totes les comparacions que usen v llavors
        eliminar_variable (GINS,v);
        afegir_variable (INS,v);
    fsi
fper
eliminar_variables_inútils (bucle, INS);

reescriure_comparacions (bucle, GINS, ALTRES)
    per a cada instrucció i del bucle bucle
        si i és un salt condicional i i utilitza una variable de GINS llavors
            reescriure_una_comparació (bucle, GINS, ALTRES, i);
        fsi
    fper

reescriure_una_comparació (bucle, GINS, ALTRES, i)
    si i conté dues variables d'inducció A i B llavors
        C:= variable tal que C∈ ALTRES i terna (C) = terna (A);
        D:= variable tal que D∈ ALTRES i terna (D) = terna (B);
        si C i D existeixen llavors
            substituir A per C i B per D a la instrucció i;
        fsi
    sino
        B:= l'operand de i que no és una variable d'inducció;
        A:= la variable d'inducció a la instrucció i;
        si B és constant o B és definit fora del bucle llavors
            C:= variable tal que C∈ ALTRES i terna (C) = terna (A);
            si C existeix llavors
                substituir la variable A per C a la instrucció i;
            sino
                C:= variable tal que C∈ ALTRES i terna (C).var_basica = A;
                si C existeix llavors
                    tmp1:= nova_variable ();
                    tmp2:= nova_variable ();
                    càlcul := PRODUCTE (tmp1,terna(C).op1, B);
                    afegir_instrucció (bucle.preencapçalament, càlcul);
                    càlcul := SUMA (tmp2,tmp2,terna(C).op2);
                    afegir_instrucció (bucle.preencapçalament, càlcul);
                    substituir A per C i B per tmp2 a la instrucció i;
```

```
    Algorisme reescriptura de bucles while...do com do...while
    Entrada
        Llista de bucles: bucles
        Graf de fluxe de control: cfg
    Sortida
        Graf de fluxe de control optimitzat: cfg

    per a cada bucle b de la llista bucles
        si ( |b.blocs| > 1) i b.encapçalament és una sortida del bucle b i
          predecessor (b.encapçalament) ∩ b.blocs = ∅ llavors
            reescriure_bucle (cfg,b);
        fsi
    fper

reescriure_bucle (cfb,b)
    sortida := successor de b.encapçalament que no està dins el bucle;
    (* Escollir una aresta de retrocés *)
    retrocés:= bloc_anterior (cfg, sortida);
    si retrocés no conté una aresta de retrocés
      o retrocés no acaba en salt incondicional llavors
        retrocés := node del bucle que conté una aresta de retrocés i que a més
                    acaba en salt incondicional;
    fsi
    si retrocés existeix llavors
        t := crear un nou bloc bàsic anterior a b.encapçalament i que acaba en el salt
        incondicional a b.encapçalament;
        afegir_bloc (cfg, t);
        per a cada bloc bàsic p de la llista predecessors(b.encapçalament)
            eliminar_aresta (p,b.encapçalament);
            afegir_aresta (p,t);
        fper
        (* b.encapçalament té dos successors: primer, dins el bucle, *)
        (* i sortida, fora del bucle *)
        moure el bloc bàsic b.encapçalament després del bloc retrocés;
        afegir_aresta (retrocés, b.encapçalament);
        s:= crear un nou bloc bàsic després de b.encapçalament que salti a sortida;
        afegir_aresta (b.encapçalament, s);
        afegir_aresta (s, sortida);
        eliminar_aresta (b.encapçalament, sortida);
        reescriure la condició de salt de b.encapçalament, de manera que la nova
        condició sigui la condició contrària a l'anterior i que salti cap a primer.etiq en
        comptes de saltar a sortida.etiq;
    fsi
```

```
    Algorisme conversió a SSA
    Entrada
        Graf de fluxe de control amb informació de dominadors, arbre de
        dominadors i fronteres de dominació: cfg
    Sortida
        Graf de fluxe en forma SSA: cfg

    primer := bloc_bàsic_inicial (cfg);
    afegir_instruccions_φ (cfg)
    renombrar_variables (cfg, primer);

afegir_instruccions_φ (cfg)
    per a cada variable var paràmetre o escalar
        var.llocs_definició := ∅;
    fper
    per a cada bloc bàsic b del graf cfg
        per a cada instrucció ins del bloc bàsic b
            si ins defineix una variable w llavors
                afegir_bloc (w.llocs_definició, b);
            fsi
        fper
    fper
    per a cada variable var paràmetre o escalar
        instruccions_φ_variable (cfg, var);
    fper

instruccions_φ_variable (cfg, var);
    candidats:= copiar (var.llocs_definició);
    per a cada bloc bàsic b de la llista candidats
        per a cada bloc bàsic df de la llista frontera_dominació (b)
            si df no conté una instrucció φ per a aquesta variable llavors
                afegir_una_φ (cfg,df,var);
                si df ∉ var.llocs_definició llavors
                    afegir_bloc (candidats, df);
                fsi
            fsi
        fper
    fper

afegir_una_φ (cfg,b,var)
    n:= número de predecessor del bloc bàsic b;
    ins:= instrucció "var:= φ (var, ..., var)" amb n arguments;
    afegir_instrucció (b,ins) al principi;

renombrar_variables (cfg, actual)
    (* Cada variable té associada la següent informació: una pila de versions i *)
    (* un número de l'última versió creada *)
    defs .= ∅;
```

Esquema de la demostració

A continuació es presenten les definicions i lemes utilitzats en la demostració. Les pàgines següents mostren en detall les demostracions d'aquests lemes i corolaris.

Definició 1: *Aplicar* una optimització a una funció consisteix en executar l'algorisme d'optimització sobre aquella funció. Si aplicar l'optimització produeix canvis a la funció, direm que l'optimització era *aplicable*.

Definició 2: Les optimitzacions es poden agrupar en tres grups: les que *eliminen instruccions* de la funció; les que (a part d'eliminar instruccions) *reescriuen* la funció; i les que (a part de reescriure instruccions) *afegeixen instruccions* en el codi.

- *les que només eliminen instruccions*: eliminar codi mort, eliminar codi inabastable, eliminar variables inútils i NOPs.
- *les que reescriuen la funció*: optimització de salts, operació de constants en temps de compilació, propagació de constants, propagació de còpies, eliminar variables d'inducció, reescriptura de bucles.
- *les que poden afegir instruccions*: són l'eliminació de subexpressions comunes (que pot afegir instruccions de còpia) i la reducció d'intensitat en variables d'inducció (que pot afegir expressions per a eliminar variables d'inducció derivades)

Lema 1: *Una optimització (considerada individualment) deixa de ser aplicable després d'un nombre finit d'aplicacions successives.*

Corolari del lema 1: *Una optimització mai podrà entrar en bucle infinit per ella mateixa. Només la interacció entre diverses optimitzacions podria provocar un bucle infinit.*

Lema 2: *Les optimitzacions que poden eliminar instruccions només poden entrar en bucle infinit si alguna altra optimització està afegint un número infinit d'instruccions a la funció.*

Lema 3: *Les optimitzacions que afegeixen instruccions són aplicables un nombre finit K de vegades, que es pot determinar a priori i no depèn de les altres optimitzacions. Per tant, aquestes optimitzacions només poden afegir un nombre finit d'instruccions.*

Corolari dels lemes 2 i 3: *Només poden entrar en bucle infinit les optimitzacions que reescriuen la funció.*

Lema 4: *Les optimitzacions que reescriuen la funció no poden entrar en bucle infinit.*

Corolari dels lemes 1,2,3 i 4: ***Les optimitzacions només es poden aplicar un nombre finit de vegades i, per tant, l'organització escollida per a les optimitzacions acabarà i no entrarà mai en bucle infinit.***

Demostració

Lema 1: *Una optimització (considerada individualment) deixa de ser aplicable després d'un nombre finit d'aplicacions successives.*

Demostració: Intentarem demostrar que les optimitzacions són exhaustives, és a dir, que realitzen tota la feina possible quan són aplicades i que per tant, dues aplicacions successives donen el mateix resultat que una única aplicació. Per fer-ho, considerarem les classes d'optimitzacions diferenciades a la definició 2:

1. Quan s'aplica una optimització que només elimina instruccions, s'elimina tot el codi possible associat a aquesta optimització (tot el codi mort, tot el codi inabastable o totes les instruccions NOP). Per tant, si després d'aplicar una d'aquestes optimitzacions la tornem a aplicar, no podrà eliminar més instruccions.
2. Per a les optimitzacions que reeescriuen instruccions, haurem fer l'anàlisi cas a cas:
   - operació de constants en temps de compilació: després de reescriure les expressions que contenen constants, aquestes expressions desapareixen, i per tant, aquesta optimització deixa de ser aplicable.
   - propagació de còpies: si **garantim** que una còpia (a:=a) s'eliminarà i no serà propagada, aquesta optimització també és exhaustiva i un aplicada deixa de ser aplicable.
   - propagació de constants: considerem les instruccions a:=const; b:=a. Després de fer la propagació de constants, el resultat seria b:=const. i per tant, la propagació de constants tornaria a ser aplicable. Però només es pot aplicar un nombre finit de vegades, perquè cada aplicació redueix el nombre d'instruccions del programa que operen amb variables.
   - extracció de codi invariant: un cop el codi ha estat extret de tots els bucles que es podia extreure, aquesta optimització deixa de ser aplicable.
   - optimització de salts: cada aplicació d'aquesta optimització o bé elimina salts, o bé escriu salts condicionals com a salts incondicionals, o bé canvia l'etiqueta de destí d'un salt. Els dos primers canvis no poden portar a bucle infinit perquè el nombre de salts, condicionals o no, és finit. De tota manera, el canvi d'etiqueta podria provocar bucle infinit i cal **garantir** que quan canviem l'etiqueta de destí d'un salt no desfarem el canvi realitzat en una aplicació posterior de l'optimització de salts (p.ex: que no farem .goto .i1 → .goto .i2 → .goto .i1).
   - eliminar variables d'inducció: després de reescriure les condicions d'acabament dels bucles per a eliminar variables d'inducció i eliminar aquestes variables d'inducció, aquesta optimització no pot realitzar més tasques.
   - reescriptura de bucles: un cop hem reescrit els bucles while-do com a do-while ja no podem reescriure més bucles i aquesta optimització deixa de ser aplicable.
3. Per a les optimitzacions que afegeixen instruccions tenim:
   - eliminació de subexpressions comunes: quan eliminem expressions comunes en una funció afegim instruccions de còpia a la funció, però les instruccions de còpia no es consideren expressions, i per tant, no provoquen que que l'eliminació de subexpressions comunes torni a ser aplicable.
   - reducció d'intensitat de variables d'inducció: quan fem la reducció d'intensitat eliminem variables derivades d'inducció dels bucles afegint variables bàsiques d'inducció.

Corolari del lema 1: *Una optimització mai podrà entrar en bucle infinit per ella mateixa. Només la interacció entre diverses optimitzacions podria provocar un bucle infinit.*

Lema 2: *Les optimitzacions que poden eliminar instruccions només poden entrar en bucle infinit si alguna altra optimització està afegint un número infinit d'instruccions a la funció.*

Demostració: Cada cop que s'aplica una optimització que elimina instruccions es redueix el número d'instruccions de la funció. Per tant, una optimització que elimina instruccions només es pot executar un nombre infinit de vegades si la funció té un nombre infinit d'instruccions. Això només és possible si una optimització que afegeix instruccions s'aplica un nombre infinit de vegades.

Lema 3: *Les optimitzacions que afegeixen instruccions són aplicables un nombre finit de vegades que es pot determinar a priori i no depèn de les altres optimitzacions. Per tant, aquestes optimitzacions només poden afegir un nombre finit d'instruccions.*

Demostració: Hi ha dues optimitzacions que afegeixen instruccions:

- reducció de variables d'inducció: Només afegirà un nombre finit d'expressions per a cada variable d'inducció derivada que existeixi en la funció (2 instruccions per a inicialitzar la variable i dues instruccions a cada avaluació de la variable d'inducció bàsica associada). Cap optimització pot crear variables d'inducció, i per tant, aquesta optimització afegirà un nombre d'instruccions finit i determinable a priori.
- eliminació de subexpressions comunes: El nombre d'expressions en una funció és finit i conegut a priori (nº d'expressions la funció programa original + nº d'expressions que afegirà la reducció de variables d'inducció). Aquesta optimització afegirà com a màxim una instrucció de còpia per a cada expressió a eliminar, i per tant, afegirà un nombre d'instruccions finit i determinable a priori.

Corolari dels lemes 2 i 3: *Només poden entrar en bucle infinit les optimitzacions que reescriuen la funció.*

Demostració: Les optimitzacions que eliminen instruccions no poden entrar en bucle infinit, pels lemes 2 i 3. A més, pel lema 3, les optimitzacions que afegeixen instruccions tampoc poden entrar en bucle infinit, ja que són aplicables només un nombre finit de vegades.

Lema 4: *Les optimitzacions que reescriuen la funció no poden entrar en bucle infinit.*

Demostració: Les optimitzacions que reescriuren el codi només podrien entrar en bucle si existís la possibilitat que la transformació realitzada per una optimització fos "desfeta" per una altra optimització, recuperant la funció inicial. Demostrarem que això no és possible.

Algunes optimitzacions que reescriuen el codi només són aplicables un nombre finit de vegades, independentment de les altres optimitzacions:

- reescriptura de bucles: un cop els bucles està reescrit com a while-do, aquesta optimització deixa de ser aplicable. Cap altra optimització pot desfer aquest canvi.

- eliminar variables d'inducció: un cop s'han reescrit les comparacions dels bucles per a eliminar variables d'inducció i aquestes variables han estat eliminades, aquesta optimització deixa de ser aplicable, perquè no hi ha més variables d'inducció a eliminar.
- extracció de codi invariant: quan tot el codi invariant s'ha extret de tots els bucles possibles, aquesta optimització no es pot tornar a aplicar, fagin el que fagin les altres optimitzacions.

Després de demostrar això, només ens falta demostrar que l'optimització de salts, l'operació de constants en temps de compilació i la propagació de constants s'apliquen un nombre finit de vegades. Evidentment, l'optimització de salts no té cap relació amb les altres dues optimitzacions, i per tant no pot produir un bucle infinit amb elles. Per tant, només cal demostrar que l'operació de constants en temps de compilació i la propagació de constants només es poden aplicar un nombre finit de vegades. Podem veure-ho des del punt de vista següent: l'operació de constants en temps de compilació transforma expressions en instruccions de còpia; per tant, la propagació de constants no farà entrar en bucle l'operació de constants, perquè no pot desfer el canvi realitzat per l'operació de constants, i la propagació de constants no entra en bucle per sí mateixa (lema 1) .

Corolari dels lemes 1,2,3 i 4: ***Les optimitzacions només es poden aplicar un nombre finit de vegades i, per tant, l'organització escollida per a les optimitzacions acabarà i no entrarà mai en bucle infinit.***

Demostració: El corolari dels lemes 2 i 3 indica que només poden entrar en bucle infinit les optimitzacions que reescriuen la funció. El lema 4 demostra que aquestes optimitzacions no poden entrar en bucle per interaccions entre elles i per tant, el procés d'optimització acabarà.

Conclusions

Si volem afegir una nova optimització a les rondes d'optimització i demostrar que el bucle d'optimització no pot entrar en bucle infinit, hem de seguir els següents passos:

1. Demostrar que l'optimització (individualment) deixa de ser aplicable després d'un nombre finit d'aplicacions successives. És a dir, cal demostrar que l'optimització no pot entrar en bucle infinit amb sí mateixa (Lema 1)
2. Classificar l'optimització en funció del tipus:
   - elimina instruccions? En aquest cas, no cal fer més demostracions.
   - reescriu la funció? En aquest cas, cal demostrar que no pot entrar en bucle amb la resta d'optimitzacions que reescriuen la funció (Lema 4)
   - afegeix instruccions? En aquest cas, cal demostrar que només pot afegir un nombre finit d'instruccions (Lema 3)
3. Si hem pogut demostrar els casos anteriors, podem afegir l'optimització al bucle. En cas contrari, no podem garantir que l'optimització fagi entrar el bucle d'optimització en un bucle infinit: hauríem d'afegir l'optimització abans o després del bucle, fent que s'executi una sola vegada.

A.3 - Eines utilitzades

En el desenvolupament del projecte, s'han utilitzat les següents eines:

- **YACC**

YACC és un eina per a generar parsers a partir d'una especificació de la gramàtica. La gramàtica està formada per una sèrie de regles, que indiquen com s'han de combinar els símbols del llenguatge. A partir d'aquestes regles, YACC genera rutines en llenguatge C que reconeixen les entrades que segueixen el format indicat per la gramàtica. Aquestes rutines es poden cridar des d'un altre programa, i s'utilitzen per a llegir els fitxers de codi IC o CL.

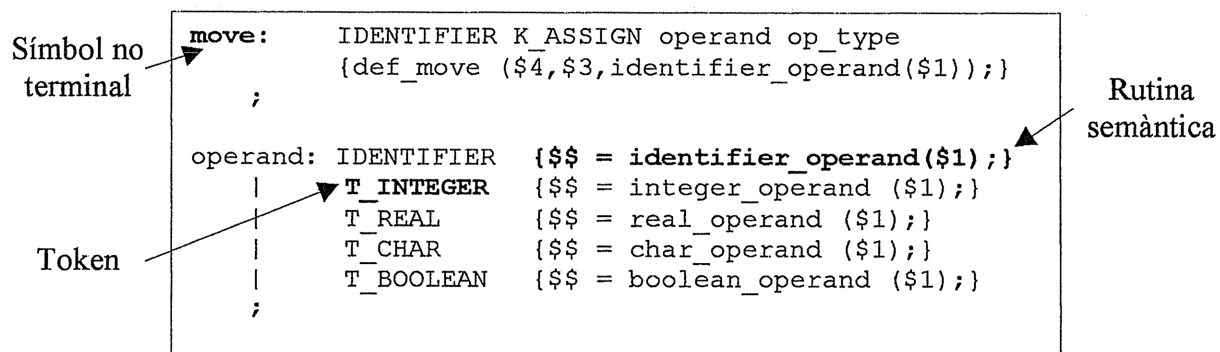
La gramàtica d'un llenguatge està formada per un conjunt de regles sintàctiques, de les quals se'n distingeix una, la *regla inicial.* També es defineix un conjunt de símbols acceptats en el llenguatge. Hi ha dos tipus de símbols: *terminals* (que apareixen el text), i *no terminals*, que no apareixen en el text i estan associats a una regla sintàctica. Cada símbol pot tenir associats uns *atributs*, valors definits per l'usuari i que es van calculant en les diferents regles.

Una regla està formada per dues parts: la part esquerra, que conté un *símbol no terminal*, i la part dreta, que conté una sèrie d'opcions. Aquesta construcció significa "*si es detecta alguna de les construccions de l'esquerra, es pot substituir pel símbol no terminal de la dreta*" . Cadascuna de les opcions de la part dreta pot contenir:

- **símbols no terminals:** indiquen que s'ha d'aplicar la regla associada a aquest símbol no terminal en aquest punt

- **símbols terminals (tokens):** indiquen que en el text s'ha de trobar el símbol indicat. Un símbol terminal no té perquè ser un sol caràcter; pot ser una cadena més o menys complicada de caràcters. L'especificació dels tokens no es fa en el fitxer YACC, sino que es fa en un fitxer LEX.

- **rutines semàntiques:** indiquen l'acció a realitzar en el cas que s'arribi a aquest punt de la regla. Estan escrites en llenguatge C, i són utilitzades per a calcular/consultar els valors dels atributs dels símbols de la regla. Els atributs es referencien amb el símbol $0 (valor del símbol no terminal quan s'entra a la regla), $1 (valor del primer símbol), $2 (valor del segon símbol,...), $$ (valor del símbol no terminal que sortim de la regla).



Exemple A.1 Regles en format YACC (operand IC i instrucció de còpia IC)

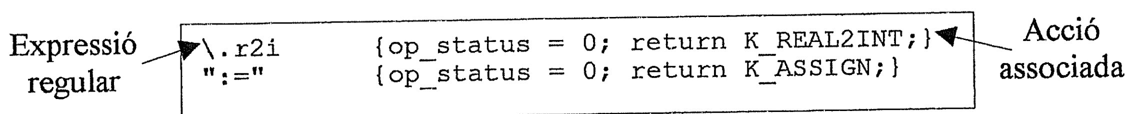
En el projecte s'han definit dues gramàtiques per a YACC: una per al llenguatge CL i una altra per al llenguatge IC. Es poden veure aquestes dues gramàtiques en els següents annexos, en un format proper a YACC.

Es pot trobar més informació sobre YACC a [LEV97]. A més, YACC acostuma a distribuir-se en tots els sistemes UNIX i LINUX, i en aquests sistemes operatius es pot consultar el manual per a trobar informació sobre l'eina (`man yacc`).

- **LEX**

LEX és una eina molt relacionada amb YACC, que permet generar analitzadors lèxics. Un analitzador lèxic conté una descripció de símbols terminals (tokens), i analitza un text buscant aquests símbols. YACC utilitza un analitzador lèxic generat per LEX cada cop que necessita consultar un símbol terminal de l'entrada.

Un fitxer en format LEX està format per una llista de parelles: expressió regular que descriu el token, i acció a realitzar en el cas que el token es trobi. Normalment l'acció està encaminada a indicar a l'analitzador sintàctic el token trobat (return NUM_TOKEN).



Exemple A.2 Expressions regulars en LEXC i regles associades

LEX genera un conjunt de rutines C a partir d'aquestes descripcions. Aquesta rutines permeten examinar un text d'entrada i buscar els símbola que s'hi troben.

La majoria de símbols són senzillament una cadena fixa de text, però hi ha símbols que requereixen expressions regulars més complexes. Les expressions regulars més complicades són les fan referència als identificadors i a les constants. A les pàgines següents es pot trobar una petita descripció dels símbols terminals dels llenguatges CL i IC.

Tal com passava en el cas de YACC, es pot trobar més informació sobre aquesta eina a [LEV97] i a les pàgines de manual dels sistemes UNIX i LINUX (`man lex, man flex`).

- **Llibreries d'estructures de dades**

Per a implementar el compilador final s'han utilitzat llibreries que defineixen l'estructura de dades i les operacions per als tipus de dades bàsics. Les llibreries utilitzades han estat les següents:

`list, array,graph, st` ... **Autor: David Harrison**
Universitat de California, Berkeley

`set` .... **Autor: Jordi Cortadella**
Universitat Politècnica de Catalunya

- **Llista (`list`):** La llibreria "list" implementa una llista doblement encadenada, que ofereix les operacions bàsiques sobre llistes: accedir al primer element, accedir a l'últim element, accedir al següent element, inserir un element, esborrar un element, realitzar recorreguts sobre una llista. Aquest no és el tipus utilitzat en les estructures de dades del codi IC (veure 8.2), per la complexitat d'utilització; es va dissenyar un nou tipus (IC_list, veure 8.2.8), que no és tan potent però és més senzill d'utilitzar.

- **Array dinàmic (`array`):** Aquest tipus té les mateix operacions que un vector clàssic, dels que es poden definir en el llenguatge C, però permet extensions (es pot fer crèixer el seu tamany). Les operacions són les mateixes que pot tenir un vector, a més de l'extensió: accedir a una posició del array, modificar una posició de l'array, extendre l'array, etc.

- **Taula de símbols (`st`):** Aquesta llibreria implementa taules de hash. Permet definir la funció de hash (dispersió) que s'utilitzarà, afegir elements, consultar elements, recòrrer tots els elements de la taula, etc.

- **Graf dirigit (`graph`):** Proporciona operacions per afegir vèrtex i arestes, eliminar-los, recòrrer les arestes que entren o surten d'un vèrtex o realitzar recorreguts de tots els vèrtexs o arestes del graf.

- **Conjunt (`set`):** Representa un conjunt d'enters entre 1 i un cert N i ofereix operacions bàsiques sobre conjunts: inserció, eliminació, veure si un element pertany a un conjunt, intersecció de conjunts, unió de conjunts, etc.

Es pot trobar més informació sobre aquestes llibreries, els tipus de dades i les seves operacions als fitxers `list.doc`, `array.doc`, `st.doc`, `graph.doc` i `set.doc`, al directori `docs` de la distribució del projecte. Els fitxers de capçalera i el codi de les llibreries es poden trobar al directori `pck`.

A.4 - Gramàtica del llenguatge IC

A continuació es presenta la gramàtica del llenguatge IC, que proporciona l'estructura de les instruccions i declaracions del llenguatge IC . D'aquesta manera es pretèn completar la descripció d'aquest llenguatge realitzada al capítol 2.

En una gramàtica hi ha dos tipus de símbols: els símbols terminals o 'tokens', que es corresponen a símbols dels programa; i els símbols no terminals, que es corresponen a regles sintàctiques que s'han d'aplicar. En aquesta gramàtica, els símbols no terminals estan representats com a `<símbol_no_terminal>` , i els "tokens" estan representats com a **'text del token'** o **TOKEN**.

Els tokens que apareixen en majúscules indiquen que hi ha més d'una possibilitat per a aquest token concret. La següent taula mostra els possibles valors de cadascun d'aquests tokens.

TOKENS DEL LLENGUATGE IC	
IDENTIFIER	cadena de caràcters que comença per una lletra i només conté dígits (0..9), lletres (a..z, A..Z) o el caràcter "underscore" (_)
NEWLINE	salt de línia
ILABEL	**'.iX'** on X és una seqüència no buida de dígits (0..9)
TYPE	els tipus permesos són **$i** (enter), **$r** (real), **$c** (caràcter), **$b** (booleà) i **$p** (punter)
OPERATION	les operacions permeses són +,-,*,/,%,&,|,^,!,<<,>>,/>,=,!=,>,>=,<,<=
STRING_CONSTANT	**"This is a string constant\n", ...**
INTEGER_CONSTANT	**23, 14, -2,...**
BOOLEAN_CONSTANT	**'.true','.false'**
REAL_CONSTANT	**2.5, .3, -5.6e4, .3E-6, ...**
CHAR_CONSTANT	**'a', '\n','\023', ...**

Les regles de la gramàtica del llenguatge IC són les següents:

```
<code> ::=
    | /* empty */
    | <code> <line_of_code>
    ;

<line_of_code> ::=
    |                        NEWLINE
    | <data_decl>            NEWLINE
    | <function_decl>        NEWLINE
    | <end_function_decl>    NEWLINE
    | <instr_decl>           NEWLINE
    ;

<data_decl> ::=  /* Declaration of a new variable */
    | '.gstring'   <string_list>
    | '.gscalar'   <id_list>
```

```
    | '.scalar'    <id_list>
    | '.parameter' <id_list>
    | '.gstruct'   <struct_list>
    | '.struct'    <struct_list>
    ;

<string_list> ::= /* List of string variables to be declarated */
    | <string_item>
    | <string_list> ',' <string_item>
    ;

<string_item> ::= /* Declaration of a string variable: id + value */
    | IDENTIFIER STRING_CONSTANT
    ;

<id_list> ::= /* List of scalars or parameters to be declared */
    | IDENTIFIER
    | <id_list> ',' IDENTIFIER
    ;

<struct_list> ::= /* List of structs to be declared */
    | <struct_item>
    | <struct_list> ',' <struct_item>
    ;

<struct_item>::= /* Declaration of a structured variable:id + size*/
    | IDENTIFIER '[' INTEGER_CONSTANT ']'
    ;

<function_decl> ::= /* Declaration of a new function */
    | '.function' IDENTIFIER
    ;

<end_function_decl> ::= /* End of function */
    | '.end' IDENTIFIER
    ;

<instr_decl> ::= /* Instruction with an optional label */
    | <instr_label> <instruction>
    ;

<instr_label> ::= /* Instruction label */
    | /* empty */
    | ILABEL ':'
    ;

<instruction> ::= /* All types of instruction */
    | /* empty */
    | <arit_log>
    | <move>
    | <int2real>
    | <real2int>
    | <jump>
    | <call>
    | <return>
    | <load>
```

A.5 - Gramàtica del llenguatge CL

A continuació es presenta la gramàtica del llenguatge CL, que proporciona l'estructura de les expressions i les instruccions del llenguatge IC. L'annex anterior ha presentat la gramàtica del llenguatge intermedi IC, seguint el mateix format.

Els tokens del llenguatge CL apareixen en negreta. La majoria de tokens, com 'IF' o 'PROGRAM' només són cadenes de text. Altres tokens (els que no van entre cometes senzilles (')) poden representar diverses cadenes de text:

TOKENS DEL LLENGUATGE CL	
IDENT	cadena de caràcters que comença per una lletra i només conté dígits (0..9), lletres (a..z, A..Z) o el caràcter "underscore" (_)
STRING	`"This is a string constant\n", ...`
INT_CONST	`23, 14, -2,...`
REAL_CONST	`2.5, .3, -5.6e4, .3E-6, ...`

Les regles de la gramàtica del llenguatge CL són les següents:

```
<programa> ::= /* Declaration of a CL program */
    | 'PROGRAM'
      <dec_variables>
      <l_dec_procs>
      <l_instrucciones>
      'ENDPROGRAM'
    ;

<dec_variables> ::= /* Declaration of variables */
    | /* empty */
    | 'VARS' <l_dec_variables> 'ENDVARS'
    ;

<l_dec_variables> ::= /* Non-empty list of variables */
    | <l_dec_variables> <dec_var>
    | <dec_var>
    ;

<dec_var> ::= /* CL variable */
    | IDENT <tipo>
    ;

<l_dec_procs> ::= /* Declaration of procedures and functions */
    | /* empty */
    | <l_dec_procs> <dec_proc>
    | <l_dec_procs> <dec_func>
    ;

<dec_proc> ::= /* Declaration of a procedure */
    | 'PROCEDURE' IDENT <pars_formales>
      <dec_variables>
```

```
      <l_dec_procs>
      <l_instrucciones>
      'ENDPROCEDURE'
    ;

<dec_func> ::= /* Declaration of a function */
    | 'FUNCTION' IDENT <pars_formales> 'RETURN' <tipo>
      <dec_variables>
      <l_dec_procs>
      <l_instrucciones>
      'RETURN' <expresion>
      'ENDFUNCTION'
    ;

<pars_formales> ::= /* Declaration of parameters */
    | /* empty */
    | '(' <l_pars_formales>  ')'
    ;

<l_pars_formales> ::= /* Non-empty parameter list */
    | <par_formal> ',' <l_pars_formales>
    | <par_formal>
    ;

<par_formal> ::= /* Parameter (by reference or by value) */
    | 'REF' IDENT <tipo>
    | 'VAL' IDENT <tipo>
    ;

<tipo> ::= /* CL data types */
    | 'INT'
    | 'FLOAT'
    | 'BOOL'
    | 'POINTER' 'TO' <tipo>
    | 'ARRAY' '[' INTCONST ',' INTCONST ']' 'OF' <tipo>
    ;

<l_instrucciones> ::= /* Instruction list */
    | /* empty */
    | <l_instrucciones> <instruccion>
    ;

<final_if> ::= /* End of a IF statement */
    | 'ENDIF'
    | 'ELSE' <l_instrucciones> 'ENDIF'
    ;

<instruccion> ::= /* CL instructions */
    | IDENT ':=' <expresion>
    | <acceso_array> ':=' <expresion>
    | 'IF' <expresion> 'THEN' <l_instrucciones> <final_if>
    | 'WHILE' <expresion> 'DO' <l_instrucciones> 'ENDWHILE'
    | 'READ'   '(' IDENT ')'
    | 'WRITE'  '(' <expresion> ')'
    | 'WRITE'  '(' STRING   ')'
    | 'NEW'    '(' IDENT ')'
```

```
    | 'FREE'  '(' IDENT ')'
    | IDENT <pars_reales> /* Procedure call */
    ;

<pars_reales> ::= /* Parameters in a procedure/function call */
    | /* empty */
    | '(' <l_pars_reales> ')'
    ;

<l_pars_reales> ::= /* Non-empty parameter list */
    | <par_real> ',' <l_pars_reales>
    | <par_real>
    ;

<par_real> ::= /* Parameter */
    | IDENT
    | <expresion_no_ident>
    ;

<expresion> ::= /* CL expression */
    | IDENT
    | <expresion_no_ident>
    ;

<expresion_no_ident> ::= /* CL expression (but not identifier) */
    | INTCONST
    | REALCONST
    | 'FALSE'
    | 'TRUE'
    | 'NUL'
    | IDENT <pars_reales> /* Function call */
    | <acceso_array>
    | <expresion>  '='   <expresion>
    | <expresion>  '>'   <expresion>
    | <expresion>  '<'   <expresion>
    | <expresion> 'OR'   <expresion>
    | <expresion> 'AND'  <expresion>
    | <expresion>  '+'   <expresion>
    | <expresion>  '-'   <expresion>
    | <expresion>  '*'   <expresion>
    | <expresion>  '/'   <expresion>
    | <expresion> 'DIV'  <expresion>
    | <expresion> 'MOD'  <expresion>
    | '(' <expresion> ')'
    | '-'    <expresion>
    | 'NOT' <expresion>
    | '*'    <expresion>
    | '&' IDENT
    ;

<acceso_array> ::= /* Array access */
    | IDENT '[' <expresion> ']'
    | <acceso_array> '[' <expresion> ']'
    ;
```

A.6 - Manual d'usuari de CODEGEN

CODEGEN és el compilador final desenvolupat en aquest projecte. Partint d'un programa escrit en llenguatge IC, dóna com a sortida el programa IC optimitzat, informació sobre el fluxe de dades i de control en aquest programa i/o el codi màquina MIPS generat. La seva sintaxi és la següent:

```
codegen [flags] [fitxer_entrada] [fitxer_sortida]
```

1. Opcions per defecte

Si no s'indica cap flag ni nom de fitxer, CODEGEN produeix el següent resultat:

- Llegeix un programa IC de l'entrada estàndard
- Aplica sobre el programa IC d'entrada totes les optimitzacions possibles
- Escriu el programa IC optimitzat a l'entrada estàndard

2. Indicar l'entrada i la sortida

Si s'indica un nom de fitxer, el compilador llegirà el programa d'entrada d'aquest fitxer; si s'indiquen dos noms de fitxer, el compilador llegirà el programa d'entrada del primer fitxer i guardarà el resultat en el segon.

Per a generar codi màquina MIPS o informació del fluxe de dades s'han d'utilitzar els flags:

`-M fitxer`	Genera codi màquina i guarda el resultat en el fitxer indicat
`-v`	Guarda informació sobre el fluxe de dades i de control en el mateix fitxer en que es guarda el programa IC optimizat, en forma de comentaris (#comentari). Aquesta informació inclou l'anàlisi de vida, l'anàlisi de dominadors, l'anàlisi de bucles, el graf d'interferència, ...

Per exemple,

```
codegen bubble_sort.ic
```

Llegeix el codi IC del fitxer bubble_sort.ic, l'optimitza i escriu el programa optimitzat en la sortida estàndard.

```
codegen -v quick_sort.ic quick_sort_opt.ic
```

Llegeix el codi IC del fitxer quick_sort.ic, l'optimitza i guarda el resultat al fitxer quick_sort_opt.ic. A més del codi optimitzat, aquest fitxer també conté informació del fluxe de dades i de control del programa IC.

```
codegen merge_sort.ic merge_sort_opt.ic -M msort.s
```

Llegeix el codi IC del fitxer merge_sort.ic, guarda el codi IC optimitzat al fitxer merge_sort_opt.ic i finalment, el codi màquina MIPS generat al fitxer msort.s.

3. Flags sobre les optimitzacions

Per defecte, s'intenten aplicar totes les optimitzacions possibles. Això significa que s'apliquen totes les optimitzacions mentre alguna optimització produeixi canvis en el programa. Es poden aplicar unes optimitzacions concretes amb el flag:

`-O opt_string`	Optimitza el codi segons les optimitzacions indicades a la cadena opt_string.

El format d'aquesta cadena és:

`( opt_string )`	Defineix un bloc: les optimitzacions dins d'aquest bloc es realitzaran tantes vegades com sigui possible (és a dir, tantes vegades com produeixin canvis en el programa). Es poden definir blocs dins d'altres blocs.
`opt_char`	Caràcter que identifica una optimització concreta. Si aquesta optimtizació està fora d'un bloc, només s'aplicarà una vegada. Els caràcters concrets que es poden utilitzar són els següents.

`j`	Optimització de salts
`b`	Optimització del graf de fluxe de control
`u`	Eliminar codi inabastable
`d`	Eliminar codi mort
`f`	Operació de constants en temps de compilació
`c`	Propagació de constants
`p`	Propagació de còpies
`e`	Eliminació de subexpressions comunes
`l`	Optimitzacions de bucles
`s`	Transformació a Única Assignació Estàtica (SSA)
`r`	Eliminar NOPs i variables inútils

Per exemple,

```
codegen matrix_add.ic
codegen -O (jbdfcpulr)s matrix_add.ic
```

Aquestes dues comandes realitzen les mateixes optimitzacions.

```
codegen -O e merge_sort.ic
```

Aquesta comanda realitza l'eliminació de subexpressions comunes una sola vegada. Això permet observar els canvis produits per aquesta optimització.

```
codegen -O (fc)ur merge_sort.ic
```

Aquesta comanda realitza operació de constants en temps de compilació i propagació de constants tantes vegades com sigui possible. Després d'això realitza l'eliminació de codi mort i l'eliminació de NOPs i variables inútils una sola vegada.

4. Altres flags

Altres flags que es poden indicar a **CODEGEN** són:

`-h`	Mostra un missatge d'ajuda similar a aquest manual d'usuari
`-s`	Incloure en el codi màquina MIPS (si es genera) una llibreria amb les crides a sistema més habituals (print_int, print_str,exit, malloc,...)

11 Bibliografia

Compiladors

[APP98] A. W. APPEL
"Modern compiler implementation in C"
Cambridge University Press, 1998

[AHO90] A. AHO, R. SETHI, J. D. ULLMAN
"Compiladores. Principios, técnicas y herramientas"
Addison-Wesley Iberoamericana, 1990

[BAN84] J. P BANÂTRE, P. BRANQUART ET AL
"Methods and tools for compiler construction"
Cambridge University Press, 1984

[AHO78] A. AHO, J. D. ULLMAN
"Principles of Compiler Design"
Addison-Wesley, 1978

[GRI84] D. GRIES
"Compiler construction"
Springer-Verlag, 1984

[FIS88] C. N. FISCHER, R. J. LEBLANC Jr
"Crafting a compiler"
Benjamin-Cummings, 1988

[GCC] *"GNU C and C++ compiler information file (egcs-1.1.2)"*
Distribució de Linux RedHat 6.0

[CHOI96] J.D. CHOI, V. SARKAR, E. SCHONBERG
"Incremental computation of Static Single Assignment Form"
6th International Conference on Compiler Construction, CC ' 96
Proceedings, Pags. 223- 237, 1996

[CHOW96] F. CHOW ET AL
"Effective Representation of Aliases and Indirect Memory Operations in SSA Form"
6th International Conference on Compiler Construction, CC ' 96
Proceedings, Pags. 253- 267, 1996

[DAV96] J.W. DAVIDSON, S. JINTURKAR
"Aggressive Loop Unrolling in a Retargetable, Optimizing compiler"
6th International Conference on Compiler Construction, CC ' 96
Proceedings, 1996

[WAN96] J. WANG, G. R. GAO
"Pipelinig-Dovetailing: A Transformation to Enhance Software Pipelining"
6th International Conference on Compiler Construction, CC ' 96
Proceedings, 1996

[BOC95] G. BÖCKLE
"Exploitation of Fine-Grain Parallelism"
Springer, 1995

Jocs de proves

[BAL94] J.L. BALCÁZAR
"Programación metódica"
McGraw-Hill/Interamericana de España, 1994

[FRA95] X. FRANCH
"Estructures de dades. Especificació, disseny i implementació"
Edicions UPC,1995

Eines i llenguatges

[KER91] B. W. KERNIGHAN, D. M. RITCHIE
"El lenguaje de programación C"
Prentice-Hall Hispanoamericana, 1991

[LAR98] J. R. LARUS
"Assemblers, Linkers, and the SPIM Simulator"
Morgan Kauffmann Publishers, 1998

[LEV97] J.LEVY, D. GUIJARRO, J.L. BALCÁZAR
"Manual de LEX i YACC"
CPET, 1997