DEPARTAMENTO DE INFORMÁTICA

Estudo e desenvolvimento de sistemas de geração de *back-ends* do processo de compilação

Paulo Jorge Teixeira Matos
Maio, 1999

Estudo e desenvolvimento de sistemas de geração de *back-ends* do processo de compilação

Tese submetida à Universidade do Minho para a obtenção do grau de Mestre em Informática, área de especialização em Sistemas Operativos, Comunicações por Computador e Arquitecturas de Computadores, elaborada sob a orientação do Doutor Pedro R. Henriques.

DEPARTAMENTO DE INFORMÁTICA
UNIVERSIDADE DO MINHO

Paulo Jorge Teixeira Matos

Maio, 1999

Resumo

O *back-end* de um compilador agrupa todo um conjunto de tarefas cuja implementação é intrinsecamente dependente das características do processador para o qual se pretende gerar código. A rápida evolução da industria dos processadores e microcontroladores levou esta área de desenvolvimento de software a realizar fortes investimentos na pesquisa de meios que permitissem dar uma resposta rápida e de qualidade à procura verificada.

É dentro deste contexto que surge o tema e o trabalho desenvolvido ao longo desta tese de mestrado, que pretende de alguma forma sintetizar o que já se encontra feito e propor algumas soluções, que apesar de individualmente não serem originais permitem, quando em conjunto, vislumbrar alternativas aos sistemas já concebidos e avançar um pouco mais na área de investigação dos geradores de código final e optimizadores.

O trabalho aqui descrito é extremamente abrangente para uma qualquer tese, cobrindo todas as áreas do processo de compilação a partir da análise semântica até à geração do código máquina, passando pela apresentação de modelos de compiladores, representação da informação, sistemas de análise de fluxo de controlo e de dados, alocação de registos local e global, selecção de instruções e geração de selectores, optimização de código a vários níveis, etc.

É ainda de referir que do trabalho desenvolvido resultou o *Back-End Development System*, que como o nome indica é um sistema de apoio ao desenvolvimento das tarefas de *back-end* de um compilador.

Abstract

The back-end of a compiler gathers a group of tasks, whose implementation is directly dependent on the features of the processor for which machine code is intended to be generated. The fast evolution of processors and micro-controllers industry lead this area of software development to perform strong investments in the research of means, which would give a fast and proper answer to the demand.

It is within this context that the theme and the work carried on through this thesis emerges. The aim of this work is to synthesise what has already been done and to give some solutions which, although individually not original, when put together, they allow alternatives to the pre-established systems and move on a little further in the research of generators of final code and optimisers.

This work is extremely wide-ranging, covering all areas of the compiling process, going from the semantic analyses till the generation of machine code. It also contains the presentation of models of compilers, representation of information, control and data flow analysis, local and global registers allocation, instructions selection and generation of selectors, code optimisation at several levels, etc.

It is also important to refer that from the development work emerged the Back-End Development System, which, as the name itself indicates, is a software system to support development of back-end tasks of a compiler.

Agradecimentos

Um voto de gratidão ao Doutor Pedro Henriques pela confiança, pelo tempo e pela paciência que teve de despender comigo. Julgo que, o melhor agradecimento que lhe posso mostrar é afirmar que, sem a sua ajuda, o trabalho desenvolvido ao longo desta dissertação nunca teria sido possível, pelo que a ele lhe devo a sua conclusão.

Um grande pedido de desculpa à Paula, pelo tempo que não tive para estar com ela, pelos fins de semana que não existiram, pelas férias que foram adiadas e por tudo mais, um grande obrigado.

Aos meus pais e irmãos pelo apoio que sempre disponibilizaram, pelos sacrifícios que fizeram e o encorajamento que me deram para prosseguir com os meus estudos, a eles dedico esta tese.

Pelo tempo que não tive, pela disponibilidade que não demonstrei, pelo apoio que não dei, aqui ficam as minhas desculpas a todos aqueles que fazem parte da minha família, amigos e colegas, e que merecem todo o meu respeito e admiração. A todos eles os meus mais sinceros agradecimentos e uma promessa de que tentarei de alguma forma compensar os momentos em que não estive presente.

Agradeço ainda o apoio da Junta Nacional de Investigação Científica e Tecnológica, JNICT, pela bolsa de Mestrado concedida no âmbito do Programa PRAXIS XXI.

Índice

1 INTRODUÇÃO 1

1.1 OBJECTIVOS 2
1.2 PANORAMA ACTUAL 2
1.3 ESTRUTURA DA TESE 4

2 ESTRUTURA DE UM COMPILADOR 5

2.1 FRONT-END 6
2.1.1 Análise ---- 7
2.1.2 Síntese ---- 8
2.1.3 Tratamento dos Identificadores ---- 9
2.1.4 Tratamento de Erros ---- 9
2.1.5 Optimização ---- 9
2.2 INTERFACE ENTRE FRONT-END E BACK-END 10
2.2.1 Geração do Código Intermédio ---- 17
2.3 BACK-END 23
2.3.1 Considerações sobre a Geração de Código ---- 23
2.3.2 Tratamento das Expressões da R.I. ---- 32
2.3.3 Alocação de Registos ---- 39
2.3.4 Geração do Código Máquina ---- 45

3 BEDS – SISTEMA DE APOIO AO DESENVOLVIMENTO DE *BACK-ENDS* 47

3.1 O BACK-END DEVELOPMENT SYSTEM 50
3.2 COMPILADOR COM ALOCAÇÃO LOCAL 56
3.3 COMPILADOR COM ALOCAÇÃO GLOBAL 57

4 MY INTERMEDIATE REPRESENTATION 61

4.1 ESTRUTURAÇÃO DA MIR 61
4.2 SINGLE STATIC ASSIGNMENT 62
4.3 CLASSES DA MIR 64
4.3.1 Classe Genérica ---- 64
4.3.2 Classe Expressions ---- 66
4.3.3 Classe DataTransfer ---- 67
4.3.4 Classe FlowNode ---- 74

5 OPTIMIZAÇÃO 79

5.1 ANÁLISE DO FLUXO DE CONTROLO 79
5.1.1 Árvore do fluxo de controlo ---- 81
5.1.2 Análise de intervalos ---- 81
5.1.3 Análise Estrutural ---- 83
5.2 ANÁLISE DO FLUXO DE DADOS 87
5.2.1 Listas de definições ---- 87
5.2.2 DU-Chain e UD-Chain ---- 89
5.2.3 Iterative Data Flow Analysis ---- 89
5.3 SINGLE STATIC ASSIGNMENT – IMPLEMENTAÇÃO 96
5.3.1 Dominance Frontiers ---- 96
5.3.2 Inserção das Funções $\Phi(...)$ ---- 100
5.3.3 Actualização das referências ---- 101
5.3.4 De SSA para a Forma Normal ---- 101
5.4 OPTIMIZAÇÕES INTERMÉDIAS 103
5.5 OPTIMIZAÇÕES DO CÓDIGO FINAL 107

6 ALOCAÇÃO DE REGISTOS 109
6.1 ALOCAÇÃO GLOBAL 109
6.1.1 Selecção dos candidatos 110
6.1.2 Construção do grafo de interferências 112
6.1.3 Coloração do grafo 117
6.1.4 Splitting de variáveis 119
6.2 ALOCAÇÃO LOCAL 121
6.2.1 Gestão dos registos 121
6.2.2 Spilling, splitting e reconversão de registos. 122
7 SELECTORES DE CÓDIGO E A SUA GERAÇÃO 125
7.1 INTRODUÇÃO AOS GERADORES DE SELECTORES 125
7.2 TREE PATTERN MATCHING 126
7.3 BOTTOM-UP REWRITE SYSTEM 128
7.3.1 Exemplo do funcionamento do BURS 129
7.4 IMPLEMENTAÇÃO DO BURS 131
7.4.1 Construção dos mecanismos do BURS 131
7.4.2 Optimizações de Proebsting 136
7.4.3 Optimizações de Chase 138
7.4.4 Optimizações de Henry 139
7.5 IBURG 143
7.5.1 Operação de labelling 143
7.5.2 Selecção das regras 145
7.6 O SELECTOR DE INSTRUÇÕES DO BEDS 146
8 UTILIZAÇÃO DO SISTEMA BEDS 147
8.1 ESTRUTURA DE UM PROJECTO 147
8.2 BEDS BACK-END GENERATOR LANGUAGE 149
8.2.1 Interface entre front-end e back-end 149
8.2.2 Estruturas para representação dos registos 151
8.2.3 Gramática do BBEGL 154
8.3 MECANISMOS PARA A GERAÇÃO DA RI 159
8.4 ORDEM DE EXECUÇÃO DAS FUNÇÕES DO BEDS 161
9 GERAÇÃO DE CÓDIGO BINÁRIO 165
9.1 DESCRIÇÃO DO NEW JERSEY MACHINE CODE TOOLKIT 165
9.2 LINGUAGEM DE ESPECIFICAÇÃO DO NJMCT 166
9.2.1 Especificação dos terminais e dos campos 167
9.2.2 Especificação dos padrões 169
9.2.3 Especificação dos constructors 170
9.2.4 Outras componentes da especificação 171
9.2.5 Especificação do assembly 172
9.3 UTILIZAÇÃO DAS FUNÇÕES DE CODIFICAÇÃO 174
9.4 RESUMO 175
10 CONCLUSÃO 177
10.1 ESTADO ACTUAL DO BEDS 177
10.2 TRABALHO FUTURO 179
BIBLIOGRAFIA 181

APÊNDICE A - GRAMÁTICA DO BBEGL A1

APÊNDICE B - EXEMPLO DE UMA ESPECIFICAÇÃO EM BBEGL B1

APÊNDICE C - ESPECIFICAÇÃO EM SLED DO μC 8051 ..... C1

APÊNDICE D - PROTÓTIPOS DAS CLASSES DO BEDS D1

APÊNDICE E - ALGORITMO DE ALOCAÇÃO POR COLORAÇÃO E1

Lista de Figuras

FIG. 2.1 – UTILIZAÇÃO DE VÁRIOS FRONT-ENDS SOBRE UM BACK-END. ---- 5
FIG. 2.2 – UTILIZAÇÃO DE UM FRONT-END SOBRE VÁRIOS BACK-ENDS. ---- 6
FIG. 2.3 – ESTRUTURA DE UM COMPILADOR. ---- 7
FIG. 2.4 – ÁRVORE DE SINTAXE DA EQ. 2.2. ---- 11
FIG. 2.5 – PROCESSAMENTO DE EXPRESSÕES POSTFIX NUMA MÁQUINA DE STACK. ---- 12
FIG. 2.6 – REPRESENTAÇÃO EM ÁRVORE E EM DAG DA EXPRESSÃO A=(B+3)(B+3). ---- 14*
FIG. 2.7 – DIAGRAMA DE FLUXO DE DADOS DO EXEMPLO 2.1. ---- 16
FIG. 2.8 – REPRESENTAÇÃO DA ÁRVORE DE SINTAXE E ÁRVORES INTERMÉDIAS DA EQ. 2.6. ---- 18
FIG. 2.9 – EXEMPLO DA ESTRUTURAÇÃO DO CÓDIGO PARA EXPRESSÕES TIPO A DA EQ. 2.8. ---- 20
FIG. 2.10 – CONVERSÃO DE UMA OPERAÇÃO DE TRÊS OPERANDOS EM DUAS DE DOIS OPERANDOS. ---- 21
FIG. 2.11 – DECOMPOSIÇÃO DA ÁRVORE BINÁRIA NUMA FLORESTA DE ÁRVORES. ---- 21
FIG. 2.12 – ESTRUTURA DAS ÁRVORES DE REPRESENTAÇÃO INTERMÉDIA PARA UMA FUNÇÃO. ---- 22
FIG. 2.13 – ESTRUTURA DO REGISTO DE ACTIVAÇÃO. ---- 25
FIG. 2.14 – ORGANIZAÇÃO DA MEMÓRIA DE EXECUÇÃO DE UM PROGRAMA, SEGUNDO AS DIFERENTES ESTRATÉGIAS DE ALOCAÇÃO. ---- 26
FIG. 2.15 – EXEMPLO DA ORGANIZAÇÃO DOS REGISTOS DE ACTIVAÇÃO NA STACK. ---- 27
FIG. 2.16 – REPRESENTAÇÃO DA ESTRUTURA DO COMPILADOR YC. ---- 33
FIG. 2.17 – REPRESENTAÇÃO DA ÁRVORE DA EQ. 2.10, COM OS RESPECTIVOS VECTORES DE CUSTO. ---- 37
FIG. 2.18 – ESTRUTURA DE UM SISTEMA DE ALOCAÇÃO DE REGISTOS. ---- 40
FIG. 3.1 – DIAGRAMA ESTRUTURAL DO BEDS ---- 51
FIG. 3.2 – MODELO DE COMPILADOR COM ALOCAÇÃO LOCAL. ---- 55
FIG. 3.3 – MODELO DE UM COMPILADOR COM ALOCAÇÃO GLOBAL. ---- 58
FIG. 4.1 – CLASSE ASSIGNMENT. ---- 69
FIG. 4.2 – CLASSES ATTRIBASSIGNMENT E ASGNEXPRESSION. ---- 70
FIG. 4.3 – CLASSES LABELASSIGNMENT E LABEL. ---- 71
FIG. 4.4 – CLASSES JUMPASSIGNMENT E JUMP. ---- 71
FIG. 4.5 – CLASSES CONDJUMPASSIGNMENT, CONDJUMP E ARGEXPRESSION. ---- 72
FIG. 4.6 – CLASSES CALLASSIGNMENT E CALL. ---- 73
FIG. 4.7 – CLASSES RETURNASSIGNMENT E RETURN. ---- 74
FIG. 4.8 – CLASSE JUMPNODE. ---- 75
FIG. 5.1 – EXEMPLO DE UM CFG E DA RESPECTIVA LISTA DE ADJACÊNCIAS. ---- 80
FIG. 5.2 – REPRESENTAÇÃO DA LISTA DE ADJACÊNCIAS DOS 3 PRIMEIROS NODOS DO GRAFO DA FIG. 5.1. ---- 80
FIG. 5.3 – REDUÇÃO DO GRAFO DA FIG. 5.1, SEGUNDO A ANÁLISE T1-T2. ---- 82
FIG. 5.4 – ÁRVORE DE CONTROLO RESULTANTE DA ANÁLISE T1-T2 PARA O GRAFO DA FIG. 5.1. ---- 83
FIG. 5.5 – EXEMPLOS DE REGIÕES ACÍCLICAS. ---- 84
FIG. 5.6 – EXEMPLOS DE REGIÕES CÍCLICAS. ---- 85
FIG. 5.7 – EXEMPLOS DE POSSÍVEIS REGIÕES IMPRÓPRIAS. ---- 85
FIG. 5.8 – ORDENAÇÃO PREORDER DO GRAFO. ---- 85
FIG. 5.9 – RESULTADO DA PRIMEIRA ITERAÇÃO DA ANÁLISE ESTRUTURAL. ---- 86
FIG. 5.10 – RESTANTES FASES DO PROCESSO DE REDUÇÃO DO GRAFO. ---- 86
FIG. 5.11 – ÁRVORE DE CONTROLO DO GRAFO DA FIG. 5.1. ---- 87
FIG. 5.12 – EXEMPLO DA REPRESENTAÇÃO MIR DA EXPRESSÃO A = A+1. ---- 88
FIG. 5.13 – ALGORITMO PARA DETERMINAR O ALCANCE DAS VARIÁVEIS. ---- 94
FIG. 5.14 – ALGORITMO PARA DETERMINAR O PERÍODO DE VIDA DAS VARIÁVEIS. ---- 95
FIG. 5.15 – ALGORITMO PARA DETERMINAR AS DF ATRAVÉS DA ÁRVORE DE DOMINADORES. ---- 98
FIG. 5.16 – ALGORITMO PARA DETERMINAR AS DF SEM A ÁRVORE DE DOMINADORES. ---- 99
FIG. 5.17 – ALGORITMO PARA INSERIR AS FUNÇÕES Φ(...). ---- 100
FIG. 5.18 – EXEMPLO DE UMA FUNÇÃO Φ(...) APÓS O PROCESSO DE ALOCAÇÃO. ---- 102
FIG. 5.19 – EXEMPLO DE COMO SE PODE RESOLVER A ALOCAÇÃO PARA AS FUNÇÕES Φ(...). ---- 102
FIG. 6.1 – REPRESENTAÇÃO ESQUEMÁTICA DA LISTA DE ADJACÊNCIAS. ---- 113
FIG. 6.2 – ESTRUTURA DA MATRIZ DE INTERFERÊNCIAS. ---- 114
FIG. 6.3 – EXEMPLO COM DUAS VARIÁVEIS SIMULTANEAMENTE VIVAS, MAS QUE APARENTEMENTE NÃO INTERFEREM UMA COM A OUTRA. ---- 115

FIG. 6.4 – EXEMPLO DE COMO É POSSÍVEL COLORIR UM GRAFO COM N=3 E TODOS OS K ≥ 3. --------- *118*
FIG. 6.5 – FASES DO PROCESSO DE COLORAÇÃO DO GRAFO. -- *118*
FIG. 7.1 – PADRÕES E MÁQUINA DE ESTADOS DO TOP-DOWN PATTERN MATCHING. ---------------------- *127*
FIG. 7.2 – EXEMPLO DE UMA ÁRVORE DE EXPRESSÕES APÓS O LABELLING.--------------------------------- *130*
FIG. 7.3 – REPRESENTAÇÃO DA TABELA DE TRANSIÇÕES DO OPERADOR ADD. ----------------------------- *132*
FIG. 7.4 – ESTRUTURA DA FUNÇÃO DE LABELLING.--- *132*
FIG. 7.5 – ALGORITMO DA FUNÇÃO PRINCIPAL, PARA GERAÇÃO DO BURS. ---------------------------------- *133*
FIG. 7.6 – ALGORITMO DE NORMALIZAÇÃO DE CUSTOS. --- *134*
FIG. 7.7 – ALGORITMO PARA DETERMINAR OS ESTADOS DO SÍMBOLOS TERMINAIS.----------------------- *134*
FIG. 7.8 – ALGORITMO PARA APLICAR AS REGRAS EM CADEIA. --- *135*
FIG. 7.9 – ALGORITMO PARA DETERMINAR OS ESTADOS DOS NÃO-TERMINAIS. ----------------------------- *136*
FIG. 7.10 – ALGORITMO DE CHAIN RULE TRIMMING. -- *138*
FIG. 7.11 – COMPACTAÇÃO DAS TABELAS, RECORRENDO A VECTORES DE INDEXAÇÃO. ----------------- *139*
FIG. 7.12 – ESQUEMA DE DECOMPOSIÇÃO DE TABELAS. --- *140*
FIG. 7.13 – ESQUEMA DA ORGANIZAÇÃO DOS MAPAS DE INDEXAÇÃO.--- *141*
FIG. 7.14 – MACROS PARA DETERMINAR O LABELLING. --- *141*
FIG. 8.1 – MODELO COM OS VÁRIOS INTERVENIENTES NO DESENVOLVIMENTO DE UM COMPILADOR. -- *148*
FIG. 8.2 – SEQUÊNCIA DAS OPERAÇÕES PARA UM BACK-END COM ALOCAÇÃO LOCAL. ------------------ *163*

Lista de Abreviaturas

BBEG – BEDS Back-End Generator
BBEGL – BEDS Back-End Generator Language
BEDS – Back-End Development System
BEG – Back-End Generator
BURG – Bottom-Up Rewrite Generator
BURS – Bottom-Up Rewrite System
CB – Código Binário
CFG – Control Flux Graph
CISC – Complex Instruction Set Computer
FSF – Free Sotware Foundation
GCB – Gerador/Geração de Código Binário
GCC – GNU C Compiler
GCI – Gerador/Geração de Código Intermédio
LCC – Light C Compiler
LHS – Left Hand Side
MDL – Machine Description Language
MIR – My Intermediate Representation
NJMCT – New Jersey Machine Code Toolkit
PCC2 – Portable C Compiler 2
PO – Peephole Optimizer
RHS – Right Hand Side
RI – Representação Intermédia
RISC – Reduced Instruction Set Computer
RTL – Register Transfer Language
SSA – Single Static Assignment

1 Introdução

Com o aparecimento dos primeiros computadores e a criação dos primeiros programas, o Homem rapidamente se depara com a necessidade de criar ferramentas que facilitem o seu diálogo com estas máquinas.

Esta necessidade advém da dificuldade que o homem tem de transmitir instruções aos computadores na sua linguagem nativa, bem como em interpretar o resultado dessas mesmas instruções.

É, como tal, nos primórdios da história da informática que aparecem os primeiros processadores de linguagens, mais concretamente os compiladores cuja função consiste em interpretar as instruções que o programador pretende transmitir ao computador, instruções essas de um nível mais legível para o homem, entenda-se programador, e de maior complexidade que a suportada pelo computador.

Antes de surgirem os primeiros compiladores, era função dos programadores suportar a malfadada tarefa de decompor as instruções a transmitir ao computador numa sequência de uns e zeros, única representação interpretável por estes.

Tal tarefa não só obrigava a ter um profundo conhecimento da arquitectura e funcionamento do processador, e das suas instruções na forma binária, como ter sempre presente um raciocínio algorítmico capaz de processar as operações de forma inequívoca permitindo assim obter programas funcionais e eficientes quer em velocidade, quer na ocupação dos recursos.

A história dos processadores de linguagens não terminou com os compiladores, muito pelo contrário fez surgir novas necessidades, bem como incentivou a procura de outras soluções para implementar o diálogo com os computadores.

Desta forma surgem os interpretadores, os depuradores (debugger), os assembladores, entre outros, generalizando a ideia e alargando o leque de aplicações.

Formalizando, dadas duas gramáticas X e Y distintas, quer nos símbolos utilizados quer na sintaxe, define-se como sendo um ***Processador de Linguagens*** um programa capaz de reconhecer expressões válidas de X e traduzi-las para expressões válidas de Y, ou/e vice versa, mantendo o seu valor semântico.

Os processadores de linguagens são normalmente construídos segundo uma determinada estrutura comum aos diversos exemplos deste tipo de programas. Aceitando alguma abstracção, é possível decompor a estrutura de um processador de

linguagens em diversas fases com funções bem definidas, a saber: análise, geração do código intermédio e geração de código final.

O passo seguinte na evolução desta área da informática consistiu em elevar o nível de concepção, desenvolvendo programas capazes de eles próprios gerarem automaticamente rotinas para a execução de algumas das fases típicas de um processador de linguagens. Este tipo de programas passou-se a designar por ***Gerador de Processadores de Linguagens***.

Muitas destas tarefas foram alvo de intensos estudos, atingindo actualmente níveis de automatização muito satisfatórios, permitindo a sua geração de modo eficiente e rápido, através de uma simples especificação das suas características.

Infelizmente a automatização das diversas fases não foi uniforme, talvez pelas dificuldades encontradas, ou por dependerem de factores externos de grande variabilidade.

1.1 *Objectivos*

É pelas dificuldades encontradas na implementação (e sua automatização) de determinadas tarefas de um processador de linguagens que surge a proposta para o tema desta dissertação.

Focando a nossa atenção apenas nos processadores de linguagens cuja finalidade é converter expressões de uma linguagem (gerada por determinada gramática), em código *assembly*, ou em código máquina – ditos compiladores, é possível classificar as suas diversas fases, em dois grandes grupos: um, designado por *front-end*, que é responsável por conhecer as características da gramática, permitindo uma correcta compreensão das suas expressões; e outro grupo, designado por *back-end*, que é responsável por conhecer as características do computador, ou mais concretamente do processador e do seu conjunto de instruções, para se poder proceder à tradução.

O *back-end* é a área sobre a qual se desenvolve o tema desta tese de mestrado, em que os principais objectivos passam por caracterizar clara e exaustivamente as tarefas englobadas no *back-end* e pelo estudo das possibilidades de automatizar o desenvolvimento dos módulos que executam essas tarefas.

1.2 *Panorama actual*

Os geradores de *back-ends* são matéria de estudo desde há vários anos. No entanto os progressos têm sido lentos e divergentes, e muitas vezes sigilosamente guardados por motivos comerciais.

As primeiras tentativas para se desenvolverem geradores de *back-ends*, datam de 1977, com os trabalhos de doutoramento de Glanville[Glan77] e Fraser[Fras77]. Em 1982, Hoffmann e O'Donnell [HO82] desenvolvem um algoritmo de reconhecimento de padrões de estruturas em árvore, que será mais tarde aplicado por Graham e Glanville [HKC84, GHAMP84] na concepção do primeiro gerador de geradores de código, com algum impacto científico. Foi com base neste trabalho, que diversos investigadores desenvolveram muitos dos actuais geradores de *back-ends*.

É no entanto no fim da década de 80 e início da década de 90, que se publica o maior número de artigos científicos, com origem em diversos grupos de investigação. Destaca-

se o grupo do Departamento de Ciências da Computação da Universidade de Berkeley, na Califórnia USA, com nomes sonantes como o R.S. Glanville que desenvolveu em conjunto com S. Graham, o *Graham-Glanville Code Generator*; ou R.R. Henry que desenvolveu o *CODEGEN* e contribuiu com um vasto conjunto de optimizações para este tipo de geradores; ou ainda E. Pelegri-Llopart responsável por desenvolver a teoria de suporte para um dos sistemas de maior sucesso nesta área, o *Bottom-Up Rewrite System* - BURS. No entanto, qualquer um destes sistemas apenas serve como solução parcial, não abrangendo todas as fases necessárias para a geração de um *back-end*.

Actualmente, a continuação do trabalho desenvolvido pelo grupo de Berkeley, faz-se acompanhar por C. Fraser, D. Hanson e T. Proebsting, este último responsável por desenvolver o *Bottom-Up Rewrite Generator* – BURG, que é uma implementação prática da teoria do BURS, apresentada por Pelegri-Llopart. Fraser e Hanson são actualmente dois dos investigadores, que mais contribuem para o progresso desta área, sendo responsáveis pelo desenvolvimento de uma solução integrada, o sistema LCC, composta por um *front-end* para ANSI C, com a respectiva definição da interface entre *front* e *back-end*, e por um gerador semelhante ao desenvolvido por Proebsting, o qual é complementado por um conjunto de rotinas com a capacidade de realizar as restantes fases do *back-end* [1].

Pelo lado comercial a evolução desta área era acompanhada pelos laboratórios Bell AT&T, do qual constavam investigadores de renome como A.V. Aho, M. Ganapathi e S.W. Tjiang. Este grupo foi responsável pelo desenvolvimento do TWIG, um gerador de *back-ends*, que também teve por base o gerador de código Graham-Glanville, mas que segue uma filosofia diferente do modelo de Pelegri-Llopart. Infelizmente, e tanto quanto consegui apurar, este grupo não deu continuidade ao trabalho desenvolvido. No entanto, existem diversas experiências de outros grupos de investigação que utilizam como suporte o TWIG.

Do lado europeu, a área é acompanhada por um grupo da Faculdade de Ciências da Universidade de Karlsruhe, Alemanha, encabeçado por Helmut Emmelmann, autor de uma das mais bem sucedidas, se não mesmo a melhor, solução para geradores de *back-ends*, o Back-End Generator - BEG. Em filosofia o sistema é bastante semelhante ao TWIG, no entanto permite com uma só aplicação gerar várias fases do *back-end*.

O BEG possui ainda a vantagem de estar integrado num pacote de *software*, o *Compiler Tool Box*, composto por várias aplicações que permitem desenvolver todas as fases de um processador de linguagens, quer sejam do *front* ou *back-end* [GE90].

Actualmente este é dos grupos mais activos, tendo mesmo desenvolvido alguns exemplos práticos, perfeitamente funcionais, provando que este tipo de ferramentas não são uma utopia. Infelizmente são poucas as publicações realizadas, e as que existem são normalmente reservadas e em alemão. Para além disso o BEG é uma aplicação comercial pelo que se colocam algumas restrições na divulgação da sua estrutura e do seu código fonte. Talvez por este motivo o trabalho desenvolvido por C. Fraser e D. Hanson, tem cada vez mais adeptos sendo de todos o mais activo e prometedor.

Seguindo uma filosofia diferente para a geração de código, encontra-se o GNU CC ou GCC[2] da FSF[3]. Ao contrário dos exemplos anteriores, o GCC não possui um suporte próprio para a geração automática de *back-ends* mas, no entanto, é dotado de meios que facilitam a adaptação a novas arquitecturas. A ideia consiste em utilizar um gerador de código, cujas características podem ser modeladas através da descrição da arquitectura

[1] Todo o pacote de *software* do LCC é *freeware*.
[2] GCC – GNU C Compiler.
[3] FSF – Free Sotware Foundation, Inc.

da máquina, de forma a modificar a geração do código final. Essa descrição é feita numa linguagem especialmente desenvolvida para tal, a MDL - *Machine Description Language*, descrição essa que é utilizada na compilação do código do gerador.

O GCC utiliza um vasto conjunto de optimizações, o que permite gerar código de qualidade superior aos modelos anteriores. Tem no entanto a desvantagem de ser muito maior e levar mais tempo a compilar e a ser compilado.

Apesar do GCC ser uma solução específica de um gerador de código e não um sistema de geração de processadores de linguagens, não deixa de ser uma das melhores soluções para um compilador portável. Infelizmente, por falta de tempo, não foi possível um estudo mais detalhado deste sistema. No entanto as ideias base, utilizadas na sua concepção estiveram sempre presentes, uma vez que o trabalho que esteve na origem do GCC, serviu também de base a outros trabalhos, tal como o *RTL System* [JML91, MRS90], desenvolvido no Departamento de Ciências da Computação, da Universidade de Illinois, Urbana-Champaign, USA, o qual foi uma das principais referências para o desenvolvimento desta tese.

1.3 *Estrutura da tese*

Nesta tese começa-se por situar o tema do trabalho, descrevendo o processo de compilação, em especial as tarefas do *back-end*, focando os problemas associados à sua implementação e propondo algumas soluções típicas, mas mantendo sempre em mente a portabilidade do compilador, que se traduz na maior, ou menor, facilidade que há em o adaptar a novas arquitecturas.

O capítulo seguinte apresenta, de forma sucinta, o modelo e o funcionamento do sistema de apoio ao desenvolvimento de compiladores, o BEDS – *Back-End Development System*, que resultou do trabalho desenvolvido na preparação desta dissertação.

O quarto capítulo apresenta os princípios que orientaram a concepção e o desenvolvimento da representação intermédia utilizada no BEDS, a MIR – *My Intermediate Representation*, bem como as infra-estruturas de suporte implementadas.

O quinto capítulo apresenta os sistemas de apoio fornecidos como parte integrante do BEDS para suporte à optimização de código. Descreve ainda algumas rotinas de optimização e respectivas implementações.

O sexto capítulo descreve o sistema de alocação local e global de registos seguido pelo BEDS, este último desenvolvido com base no algoritmo de alocação proposto por Chaitin [CACCHM81, Chait82].

O sétimo capítulo apresenta alguns dos geradores de código que têm por base o *Bottom-Up Rewrite System* (BURS) e faz referência à solução utilizada no BEDS.

O oitavo capítulo descreve a gramática que permite especificar as características dos processadores, e explica como gerar as expressões da representação intermédia e como utilizar os diversos módulos que compõem o BEDS.

O nono capítulo apresenta o New Jersey Machine Code Toolkit [RF94, RF95, RF95a, RF96], como uma possível solução a integrar ou a utilizar conjuntamente com o BEDS para a geração directa de código máquina.

Na conclusão faz-se um breve resumo do trabalho desenvolvido ao longo da tese e deixa-se algumas propostas para projectos futuros.

2 Estrutura de um Compilador

Este capítulo começa com uma breve referência às fases do *front-end* de um compilador, e à interface entre este e o *back-end*. Depois descrevem-se as diversas etapas que compõem este último, apresentando as dificuldades em transpor o seu desenvolvimento para um gerador de geradores.

No entanto antes de se avançar para a descrição da estrutura de um compilador, é necessário perceber o porquê da sua divisão em *front* e *back-end*. Sob o ponto de vista da concepção, uma vez desenvolvido um processador de linguagens para uma determinada máquina segundo esta estrutura, é possível reutilizar o *back-end* na implementação de novos processadores de linguagens para essa máquina.

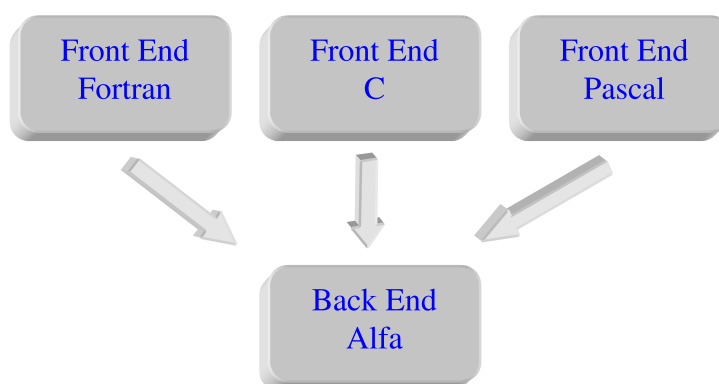


Fig. 2.1 – Utilização de vários *front-ends* sobre um *back-end*.

Por exemplo, uma vez desenvolvido um compilador de linguagem C para o processador Alfa, facilmente se pode reutilizar o *back-end*, na implementação de compiladores para outras linguagens fonte, como Fortran ou Pascal, bastando para tal desenvolver os respectivos *front-ends*, respeitando a mesma representação intermédia.

A concepção encontra-se, também, facilitada para a situação em que se pretende modificar um compilador para gerar código para uma outra máquina, uma vez que nesta situação basta desenvolver o *back-end* para a nova arquitectura e reutilizar o *front-end*.

Convém no entanto realçar que nem sempre é fácil detectar num processador de linguagens, onde é que termina o *front-end* e onde se inicia o *back-end*, o que pode comprometer a reutilização de ambas as partes. O que se consegue evitar definindo correctamente o papel de cada uma das fases intervenientes do processo de compilação e criando um nível intermédio de representação, do código a compilar e de toda a informação a este necessária, que possa ser partilhado por ambas as partes.

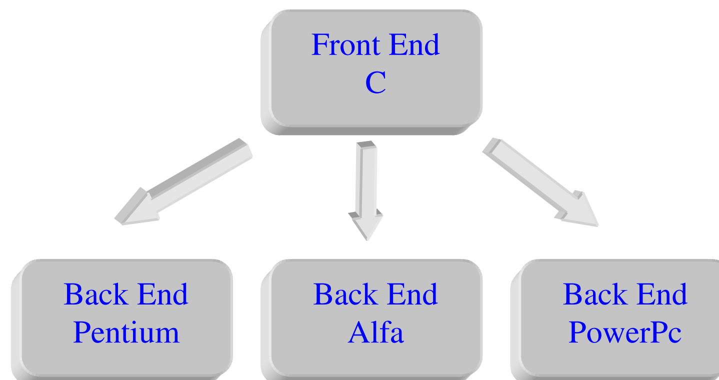


Fig. 2.2 – Utilização de um *front-end* sobre vários *back-ends*.

2.1 *Front-End*

O processo de transformar um texto escrito numa qualquer linguagem fonte, em código capaz de funcionar sobre numa determinada máquina, é extremamente complexo para que possa ser analisado de uma forma global. É por esta razão, que a tarefa de um compilador é normalmente vista como uma sequência de fases, cada uma responsável por realizar um conjunto limitado de funções. Dessas fases, umas são totalmente independentes da máquina objecto, estando apenas relacionadas com as características da linguagem fonte e outras dependem essencialmente das características da linguagem final, o que normalmente está relacionado com as características do processador. Por essa razão, o conjunto total de fases em que se decompõe o processo de compilação, continua a ser organizado em dois grandes blocos: o *front-end* (descrito nesta secção) e o *back-end* (descrito na secção 2.3).

O modelo que se apresenta na Fig. 2.3 mostra as fases principais da compilação. Não é no entanto obrigatório que todas elas façam parte de um compilador. Por vezes é possível reunir funções de fases distintas numa única, bem como decompor algumas das fases representadas em diversas sub-fases, permitindo diferenciar com maior detalhe as suas funções.

2.1.1 Análise

Analisa Léxica

Sempre que o código fonte se encontra representado sob a forma de um texto, é necessário proceder ao reconhecimento dos símbolos que o compõem. É exactamente esta a função da primeira fase, a do analisador léxico ou *scanner*, que considera o texto fonte como uma *stream* de caracteres, a qual é processada da esquerda para a direita, procedendo à eliminação de caracteres nulos, tal como espaços e caracteres do tabulador, bem como de comentários, agrupando os restantes de forma a obter símbolos, designados por *tokens*.

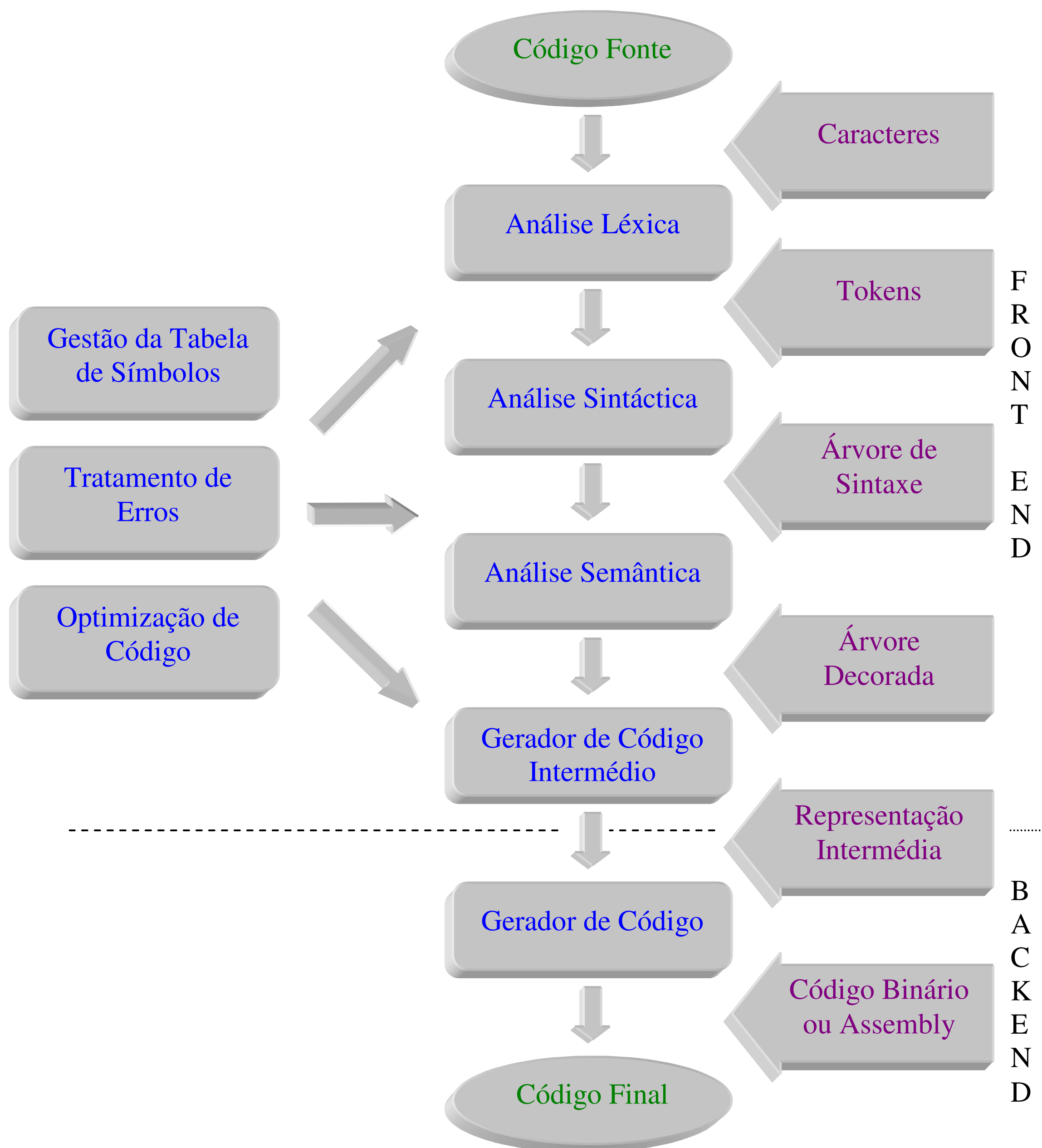


Fig. 2.3 – Estrutura de um compilador.

Análise Sintáctica

Na fase seguinte, o analisador sintáctico ou *parser*, requisita os *tokens* ao analisador léxico agrupando-os sob a forma de uma árvore de sintaxe, de acordo com a gramática da linguagem em causa. Essa árvore, que traduz a estrutura do texto fonte, indica quais as regras de derivação da gramática, que foram usadas na construção da frase, contribuindo assim para se conhecer o seu significado.

Análise Semântica

Na terceira fase, encontra-se o analisador semântico. A cada símbolo detectado no código fonte está associado um conjunto de atributos, que podem representar diversos tipos de informação e cujo objectivo é fornecer os conhecimentos necessários à determinação do significado exacto desse texto fonte (o que é determinante para se fazer a geração de código).

Conhecendo o conjunto de regras que permitem determinar o valor dos atributos e tomando em consideração o valor associado aos *tokens*, o analisador semântico procede ao cálculo do significado exacto de cada símbolo gramatical.

O processo consiste em pegar na árvore de sintaxe abstracta e decorá-la com os atributos, calculando o seu valor e testando se estes cumprem às condições de contexto.

2.1.2 Síntese

No fim da análise semântica, a informação obtida pelo *front-end* consiste num conjunto de árvores decoradas e numa tabela de símbolos. Coloca-se a questão de saber se este é o formato mais correcto para a representação intermédia, ou seja, para a representação interna do significado do texto fonte.

O ideal seria obter uma representação intermédia possível de ser utilizada para todos os *front-ends* e *back-ends*.

As árvores semânticas são raramente utilizadas como representação intermédia porque são muito dependentes da linguagem fonte e não contêm, normalmente, toda a informação necessária ao *back-end*.

É por esta razão que surge uma fase, na estrutura do compilador, destinada a transformar a informação que se encontra sobre o formato de árvores decoradas, em expressões de uma representação intermédia. Tal não só facilita a reutilização do *front* e *back-end*, como garante um nível de representação da informação independente das características destes dois níveis. Permite-se, assim, desenvolver toda uma série de rotinas para o tratamento de código que, por funcionarem sobre a representação intermédia, são completamente independentes da linguagem fonte e linguagem final, garantindo a sua total reutilização noutros compiladores. É o caso da optimização de código que normalmente se processa a este nível de representação.

Esta fase de síntese do significado é geralmente designada por geração de código intermédio e será descrita adiante na secção 2.2.1.

2.1.3 Tratamento dos Identificadores

A gestão da tabela de símbolos, o tratamento de erros e a optimização do código são fases que se encontram representadas à parte, visto poderem ocorrer em conjunto com outras fases ou por poderem ser intercaladas em diversas posições da sequência de compilação.

A tabela de símbolos é uma estrutura presente nas diversas fases do compilador, cuja função é armazenar a informação sobre cada identificador que surge no programa fonte. Tipicamente é uma estrutura de procura em que as chaves são os identificadores, às quais a tabela associa um conjunto de informação proveniente das diversas fases. É no entanto da responsabilidade do analisador léxico colectar a informação necessária para criar os elementos da estrutura de dados. Este sempre que detecta um *token* testa se o seu lexema (*string* que o representa) já se encontra na tabela de símbolos. Caso não exista, cria-se uma nova entrada na tabela. Nada impede que outras fases do processo, para além da análise lexical, acrescentem novos símbolos à tabela de identificadores.

Por questões de eficiência, é vulgar que cada *token* possua pelo menos um atributo, que funciona como apontador para a entrada correspondente da tabela de símbolos.

2.1.4 Tratamento de Erros

À semelhança da gestão da tabela de símbolos, o tratamento de erros também envolve praticamente todas as fases do compilador. Por exemplo, o analisador léxico é responsável por identificar a posição de cada uma das ocorrências dos diversos *tokens*, para que em caso de erro seja possível identificar a sua origem (posição) dentro do texto fonte. Para além disso, o analisador léxico deve detectar erros ao nível dos caracteres que pertençam à linguagem ou caracteres que apareçam inseridos em grupos errados (por exemplo pontos de exclamações no meio de palavras).

Já o analisador sintáctico, deve detectar violações das regras gramaticais, as quais definem as expressões válidas da gramática. Por vezes estas violações têm origem na análise lexical, mas normalmente só podem ser detectadas na análise sintáctica, é o caso de palavras reservadas ou operadores que se encontram mal escritos.

O analisador semântico também é responsável pelo tratamento de erros, detectando por exemplo, situações de incompatibilidade entre o tipo dos operandos e operadores, ou de uma forma geral, expressões que se encontrem sintacticamente correctas, mas cujo significado não é válido.

Em qualquer caso um bom compilador deve ser capaz de lidar com a situação de erro, de forma a continuar o processo de compilação após corrigir o erro ou recuperar dessa situação e ser capaz de detectar erros subsequentes. Deve ainda sinalizar tais anomalias ao utilizador da forma mais completa e explicável possível.

2.1.5 Optimização

A fase de optimização serve para melhorar a qualidade do código gerado, através da transformação da sua representação, permitindo obter maior velocidade de execução ou reduzir o tamanho do código final. No entanto a optimização não pode modificar o valor

semântico das expressões e não deve optimizar determinadas características do código à custa de uma degradação significativa de outras características.

Ao longo do processo de compilação, podem existir várias fases de optimização, ou até nenhuma, as quais podem ser classificadas em relação às características da máquina, como independentes, e neste caso fazem parte das fases do *front-end* ou do nível intermédio de representação, ou então como dependentes, fazendo parte do *back-end*. Ambas as formas serão abordadas no capítulo 5.

As optimizações independentes da máquina dividem-se ainda em dois tipos distintos, as optimizações locais, que se processam dentro do contexto de um bloco simples de código com estruturas de controlo muito simples, e as optimizações globais, as quais se processam sobre grandes blocos de código, onde se admite qualquer tipo de estrutura de controlo.

Algumas das optimizações com maior utilização, devido aos resultados que permitem obter, são: simplificação algébrica, eliminação de sub-expressões comuns, eliminação de código morto, eliminação de propagação de cópias, remoção de expressões constantes de dentro de ciclos, etc.

Algumas destas optimizações são facilmente implementáveis, mas normalmente tal não acontece, obrigando mesmo a manter ou a criar mecanismos de análise, que forneçam a informação necessária à sua realização.

Alguns desses mecanismos, são implementados recorrendo à análise do fluxo de controlo, à análise do fluxo dos dados e à análise de dependência entre dados. Algumas destas formas de análise serão descritas no capítulo 5.

2.2 Interface entre *Front-End e Back-End*

Como se disse atrás, a escolha duma boa representação intermédia para exprimir o significado do texto fonte (reconhecido na fase de análise) é fundamental para a implementação dum compilador.

Assim, a representação intermédia deve ser facilmente adaptável, quer ao *front-end* ou ao *back-end*; deve permitir representar qualquer tipo de estrutura que possa surgir no código fonte, quer esta seja de dados ou de controlo; e ser facilmente manipulável pelos procedimentos que se executam a este nível de representação.

A notação *postfix*, *three-address code*, árvores binárias, grafos e a linguagem de transferência de registos (RTL) são algumas das formas utilizadas para a representação intermédia.

Notação *Postfix*

A representação em notação *postfix* obtém-se a partir da "linearização" da árvore de sintaxe, convertendo-a numa representação recursiva do tipo:

$$opd_{esqn}\ \ opd_{dirn}\ \ op_n \qquad \text{Eq. 2.1}$$

Em que opd_{esq}, opd_{dir} e op_n representam respectivamente o operando esquerdo e direito, e o operador do nodo *n* da árvore de sintaxe. Por exemplo, a expressão da Eq. 2.2, cuja árvore de sintaxe se encontra representada na Fig. 2.4, em notação *postfix*, resulta na Eq. 2.3.

a = (b + 3)*c Eq. 2.2

a b 3 + c * = Eq. 2.3

Este tipo de representação é ideal para o caso dos interpretadores, em que o código intermédio é executado por máquinas virtuais baseadas em *stack*; o *front-end* vai passando os elementos das expressões em *postfix*, da esquerda para a direita ao interpretador, os quais são inseridos na *stack*, através de instruções de *push*. No entanto, sempre que se detecta um operador (op_n), este ao invés de ser inserido na *stack*, é executado de imediato. A utilização da notação *postfix* garante que os operandos necessários à execução de op_n se encontram no topo da *stack*, bastando para tal retira-los através de simples operações de *pop*.

Após a execução de op_n, o resultado é inserido na *stack*, servindo de operando para a instrução seguinte. O processo repete-se até se executar a última operação da expressão.

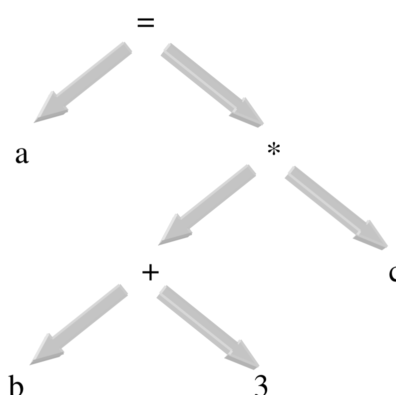


Fig. 2.4 – Árvore de sintaxe da Eq. 2.2.

O tratamento da Eq. 2.3, encontra-se representado na Fig. 2.5. A *stack* é carregada com *a*, *b* e *3*; quando se detecta o operador +, do tipo binário retiram-se os dois primeiros elementos da *stack*, *3* e *b*, que são utilizados como operandos; o resultado da operação é colocado na *stack*, onde ainda se encontra *a*; volta-se a inserir mais elementos na *stack*, neste caso *c*, até se encontrar o próximo operador, ou seja *, o qual retira da *stack* *c* e o resultado de *b+3*; realizada a operação, insere-se o resultado na *stack*; O processo repete-se até ao fim da expressão.

Este tipo de representação intermédia deve conter para além das instruções aritméticas e lógicas, do *push* e do *pop*, algumas instruções para acesso à memória e para a implementação de estruturas de controlo, tais como:

```
rvalue v      //Realiza o push do valor da variável v.
lvalue v      //Realiza o push do endereço da variável v.
copy          //Realiza o push do valor do topo da stack.
label l       //Identifica a posição da label l.
goto l        //Salto para a expressão com a label l.
gofalse l     //Salto condicional para label l,
              // se o valor removido do topo
              // da stack é falso.
```

```
gotrue l        //Salto condicional para a label l,
                //se o valor removido do topo
                // da stack é verdadeiro.
```

A grande desvantagem deste tipo de representação encontra-se na dificuldade que existe em manusear as expressões, nomeadamente reordená-las, o que é fundamental para fases como a optimização.

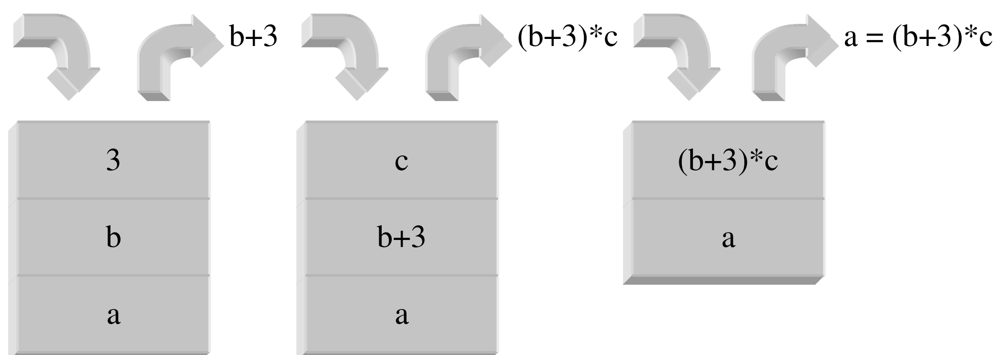


Fig. 2.5 – Processamento de expressões postfix numa máquina de *stack*.

Three Address Code

Outra forma de representação intermédia é o *Three Address Code*, em que cada expressão é decomposta numa sequência de várias sub-expressões, cada uma com a forma da Eq. 2.4.

$$x := y \text{ op } z \qquad \text{Eq. 2.4}$$

Onde *op* representa um qualquer operador aceite na representação intermédia, e *x*, *y* e *z* são *identificadores* ou *constantes*, representando o endereço de *variáveis* do programa, variáveis temporárias (ex: registos) ou valores constantes.

Neste tipo de representação, uma expressão como a da Eq. 2.2 é reescrita na seguinte sequência de sub-expressões:

```
t1 := b + 3
a := t1 * c
```

As quais são obtidas a partir da "linearização" da árvore de sintaxe, recorrendo à utilização de variáveis temporárias para guardar os resultados dos nodos interiores da árvore, como é o caso de *t1*.

Trata-se de uma representação mais legível e fácil de manusear que a anterior, e mais próxima da representação em *assembly*. A manutenção das estruturas de controlo é mais simples, implementando as *labels* através dos índices do *array* que contém as sub-expressões.

Para além das instruções do tipo da Eq. 2.4, o *Three Address Code* possui instruções de cópia, operações unárias, chamadas de procedimentos, saltos incondicionais e condicionais, variáveis indexadas, atribuição de endereços e de apontadores.

```
x := op y                    //Operador unário.
x := y                       //Atribuição de y a x.
goto l                       //Salto incondicional para label l.
if x cond y goto label l     //Salto condicional para label l,
                             // se (x cond y) é verdadeiro.

param x1, x2, ..., xn        //Chamada do procedimento p,
call p, n                    //com n parâmetros x1, x2, ..., xn
...
return y                     // e a devolver y.
x := y[i]                    //Atribuição a x, do valor da
                             // posição de memória i, para lá de y.
x[i] := y                    //Atribuição de y, à posição de
                             // memória i, para lá de x.
x := &y                      //Atribuição a x do endereço de y.
x := *y                      //Atribuição a x do valor apontado por y.
*x := y                      //Atribuição de y à posição apontada por x.
```

Árvores e DAG's

A representação em forma de árvore binária, passa por considerar que cada nodo interno da árvore representa um operador, cujos operandos resultam das operações dos nodos descendentes.

A árvore de representação intermédia possui uma estrutura semelhante à da árvore de sintaxe, diferindo no entanto no conteúdo, o qual traduz o conhecimento obtido pela análise semântica e a pela própria sintaxe da representação intermédia.

Neste caso a expressão da Eq. 2.2, corresponde à árvore binária da Eq. 2.5.

=(a, *(+(b, 3), c)) Eq. 2.5

À semelhança das representações anteriores, as árvores binárias também utilizam variáveis temporárias e permitem a utilização de algoritmos de pesquisa e substituição, simples e eficientes, com a vantagem de possuir uma representação mais compacta e mais fácil de obter.

Os DAG – Direct Acyclic Graph, são bastante semelhantes às árvores mas compactam ainda mais a informação, uma vez que permitem a reutilização de sub-expressões, como se demonstra na Fig. 2.6. A sua grande desvantagem está na utilização de algoritmos de pesquisa e substituição mais complexos e lentos.

As árvores binárias são assim uma das formas de representação com maior utilização na ligação entre *front* e *back-end*. A razão não se prende apenas pelas vantagens apresentadas, mas também porque muitos dos algoritmos utilizados para desenvolver geradores de geradores de código, trabalham sobre árvores, como será demonstrado no capítulo 7.

Não é no entanto de desprezar o *Three Address Code*, que tem a vantagem de permitir representar determinados aspectos que nas árvores ficam camuflados.

A selecção entre as duas formas de representação, depende do nível de proximidade que se pretende obter entre a representação intermédia e o código final, e da relação

entre a qualidade do processo de selecção das instruções e os processos de optimização a implementar, como se poderá ver em capítulos posteriores.

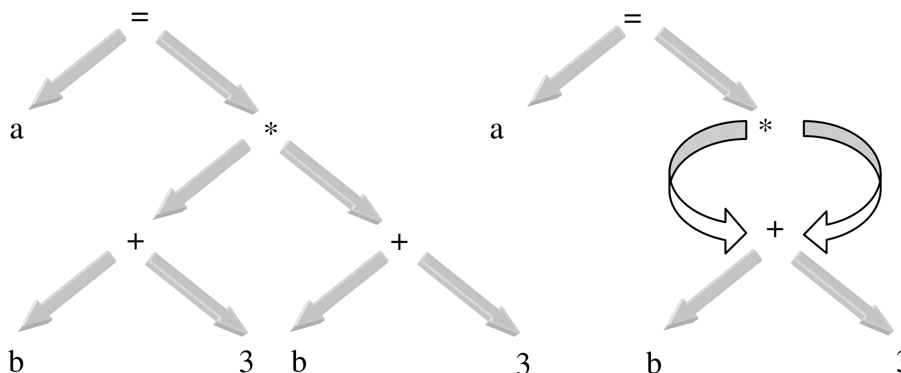


Fig. 2.6 – Representação em árvore e em DAG da expressão a=(b+3)*(b+3).

Linguagem de Transferência de Registos

A Linguagem de Transferência de Registos, ou RTL, é uma forma de representação que permite que com um conjunto de operações muito básicas e muito próximas de uma linguagem do tipo *assembly*, se descrevam diversos níveis de representação da informação. À primeira vista é muito semelhante ao *Three Address Code*, no entanto as operações são essencialmente implementadas sobre registos.

A RTL passou a ter alguma relevância quando apareceu como forma de representação intermédia do Peephole Optimizer (PO), desenvolvido por Christopher Fraser e Jack Davidson [DF80, DF84a, DF84b]. Sendo desde então uma referência, não tanto pelas suas características descritivas, uma vez que estas são muito semelhantes ao *Three Address Code*, mas mais pelo contexto em que foi utilizada. Na realidade o que se tornou uma referência foi o modelo proposto para o PO e todo o conjunto de conceitos a este associado, de onde se destaca a implementação das rotinas de optimização de forma independente das características do processador e apresentação de uma solução, para que, a partir de uma descrição dessas mesmas características fosse possível transformar as expressões da representação intermédia em código máquina.

A Linguagem de Transferência de Registos foi desenvolvida como fazendo parte de um sistema (RTLS) de *software* [JML91] para desenvolvimento de optimizadores de código. É composto pela linguagem de representação e manipulação dos dados (RTL), e por um vasto número de rotinas, pré-definidas, para conversão das expressões RTL em código *assembly* ou código máquina, bem como para optimização de código.

No desenvolvimento da RTL, esteve sempre presente a ideia de criar uma forma de representação completamente independente das características do *front* e *back-end*, de modo a generalizar a aplicação dos algoritmos de optimização. Pela experiência, concluiu-se que a RTL deveria ser tanto quanto possível uma espécie de intersecção dos conjuntos de instruções das diversas máquinas conhecidas, de tal forma que qualquer operação descrita fosse facilmente reescrita para uma qualquer máquina. Ideia esta que foi herdada do PO e utilizada em outros sistemas, como é o caso do GCC.

Mais ainda, a representação não deve depender das limitações físicas de cada máquina, por exemplo a quantidade de registos possíveis de utilizar em RTL é infinita,

de forma a evitar os problemas de alocação de registos, que dependem essencialmente das características das máquinas.

As operações utilizadas pela RTL são extremamente simples, mas a partir das quais é possível construir novas formas de representação mais complexas. Convém no entanto perceber que uma das grandes vantagens do sistema está na simplicidade das operações, o que permite detectar potenciais situações de optimização ou de geração de código.

Exemplo 2.1

No exemplo que se apresenta abaixo, onde se faz a tradução de um bloco de código em linguagem C para RTL, pode-se confirmar o detalhe de uma descrição feita com este tipo de representação.

Código C	**RTL**		
		r[1] = &b	//Carrega o endereço de *b*
if(b>max)		r[2] = *r[1]	//Carrega o valor de *b*
max = b;		r[3] = &max	//Carrega o endereço de *max*
else		r[4] = *r[3]	//Carrega o valor de *max*
max = a;		r[2] ≤ r[4] ↑ L1	//Se *b* ≤ *max* salta *L1*
		r[5] = &max	//Carrega o endereço de *max*
		r[6] = &b	//Carrega o endereço de *b*
		r[7] = *r[6]	//Carrega o valor de *b*
		*r[5] = r[7]	//Carrega *max* com o valor de *b*
		↑ L2	//Salto incondicional para *L2*
	L1:	r[8] = &max	//Carrega o endereço de *max*
		r[9] = &a	//Carrega o endereço de *a*
		r[10] = *r[9]	//Carrega o valor de *a*
		*r[8] = r[10]	//Carrega *max* com o valor de *a*
	L2:		//Fim da função

♦

Já o sistema RTLS vai muito para além da formalização da representação intermédia, apresentando-se como uma solução concreta para o suporte da própria representação e tratamento de código, fornecendo, entre outras, rotinas de optimização, alocação e geração de código.

O RTLS foi implementado através de uma linguagem orientada ao objecto (*SmallTalk*), pelo que as suas componentes encontram-se sobre a forma de classes. A representação intermédia é construída através da instanciação dessas mesmas classes.

No essencial existem quatro classes: o *RTLProgram*, o *RTLFlowNode*, o *RTLTransfer*, e o *RTLExpression*. As três últimas são classes abstractas de dados, e apenas a primeira é uma classe de dados concreta.

O *RTLProgram* é a classe mais geral, cujas subclasses servem para reter todo o tipo de informação global que diga respeito ao código fonte, à descrição da máquina e de certa forma às instâncias das outras classes.

O *RTLFlowNode* contém as classes que permitem construir o grafo de fluxo de controlo que descreve a estrutura do programa. Os métodos desta classe incluem, rotinas de optimização de fluxo de controlo, tais como a eliminação de saltos ou a optimização de estruturas condicionais.

A principal função da classe *RTLTransfer* é caracterizar as operações de transferência de dados. Assim a cada instância está associada a informação que permite identificar o conjunto das instâncias que produzem os seus operandos e o conjunto das instâncias que utilizam o seu resultado, o que é essencial para grande maioria das rotinas de optimização.

O *RTLExpression* serve para representar todo o tipo de expressões, desde a representação de registos, memória, operações de acesso à memória, operações binárias, unárias e constantes, etc.

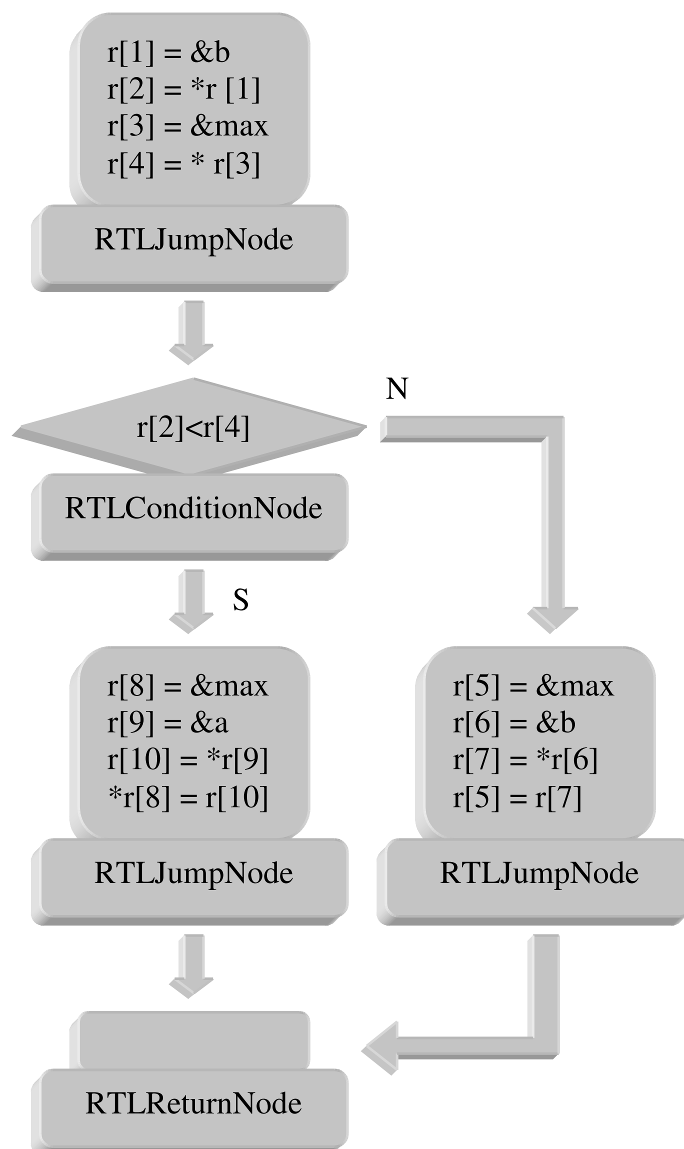


Fig. 2.7 – Diagrama de fluxo de dados do Exemplo 2.1.

Os métodos das várias classes permitem manter a consistência da representação e as estruturas de dados necessárias às rotinas de optimização, de geração de código, etc.

Após a construção da representação e da execução de todas as optimizações, determinam-se as instruções a utilizar para cada expressão e procede-se à alocação dos registos, fazendo o mapeamento entre os registos virtuais da RTL (pseudo-registos) e os registos físicos. Estas operações são normalmente implementadas como métodos da

classe *RTLProgram* com recurso à classe *MachineDescription*, que contém a informação sobre as características da máquina.

Este sistema (RTLS) possui diversas vantagens, de entre as quais se destaca a capacidade de desenvolver novas classes, possibilitando a representação de outros tipos de transferências e expressões, permitindo assim expandir a representação intermédia.

Outras vantagens advêm do facto desta representação estar integrada num sistema, com uma vasta diversidade de rotinas desenvolvidas e possuir uma classe própria para representar as características da máquina, o que serve de suporte e facilita a adaptação da RTL a novas arquitecturas. Convém no entanto não esquecer que apesar de todas as vantagens, não se trata de um sistema de geração de geradores, ou de optimizadores, mas sim de um sistema bem suportado, já com muitas das rotinas implementadas, que com algumas alterações, ou mesmo com desenvolvimento de novas classes e métodos pode ser, com maior ou menor dificuldade adaptado a novas arquitecturas.

Este tipo de representação será estudado com maior detalhe em capítulos posteriores, uma vez que serviu de base ao desenvolvimento do trabalho prático de dissertação desta tese.

2.2.1 Geração do Código Intermédio

Nesta secção pretende-se descrever alguns aspectos mais proeminentes da geração do código intermédio (GCI), a partir da árvore decorada e da tabela de identificadores.

Apesar da análise semântica ter uma finalidade distinta da GCI, a realidade é que na maior parte dos casos, esta última é feita em simultâneo com a primeira, utilizando como tal os mesmos mecanismos, como é o caso das gramáticas de atributos.

Não se pretende ao longo desta secção, entrar em detalhes de como é que tal é feito, mas apenas descrever determinadas situações típicas da GCI. Para tal, parte-se do princípio que a cada símbolo do analisador sintáctico se associou um conjunto de atributos e a cada uma das produções, um conjunto de regras semânticas para determinar esses atributos, com base nos quais se faz a GCI.

Tratamento de Tipos

Na generalidade dos processadores as instruções realizam-se sobre o mesmo tipo de operandos, por exemplo, a adição é feita sobre valores inteiros ou reais, mas raramente sobre um inteiro e um real. É função do analisador semântico detectar se os operandos são compatíveis entre si, cabe no entanto ao GCI inserir na representação intermédia os mecanismos necessários à conversão dos operandos, bem como decidir o tipo específico de operador a utilizar.

Todos os símbolos terminais, que sejam identificadores, possuem um tipo estabelecido implicitamente através do formato com que surgem no texto fonte ou então explicitamente através das declarações.

Depois para cada operação, é necessário verificar se todos os operandos possuem o mesmo tipo ou se são compatíveis. Se tal não acontecer é fundamental proceder à conversão para um tipo comum, o qual também representará o tipo do resultado da operação.

O processo de converter os operandos para um mesmo tipo comum, implica normalmente a inserção de novos nodos na árvore de sintaxe, que representem as operações de conversão.

Exemplo 2.2

Este exemplo demonstra como se processa o tratamento de tipos, para a situação em que se pretende somar uma constante e uma variável, ambos do tipo inteiro, e depois atribuir o resultado a uma variável do tipo real.

Código em C

```
int a;
float b;

b= a + 10;
```

Eq. 2.6

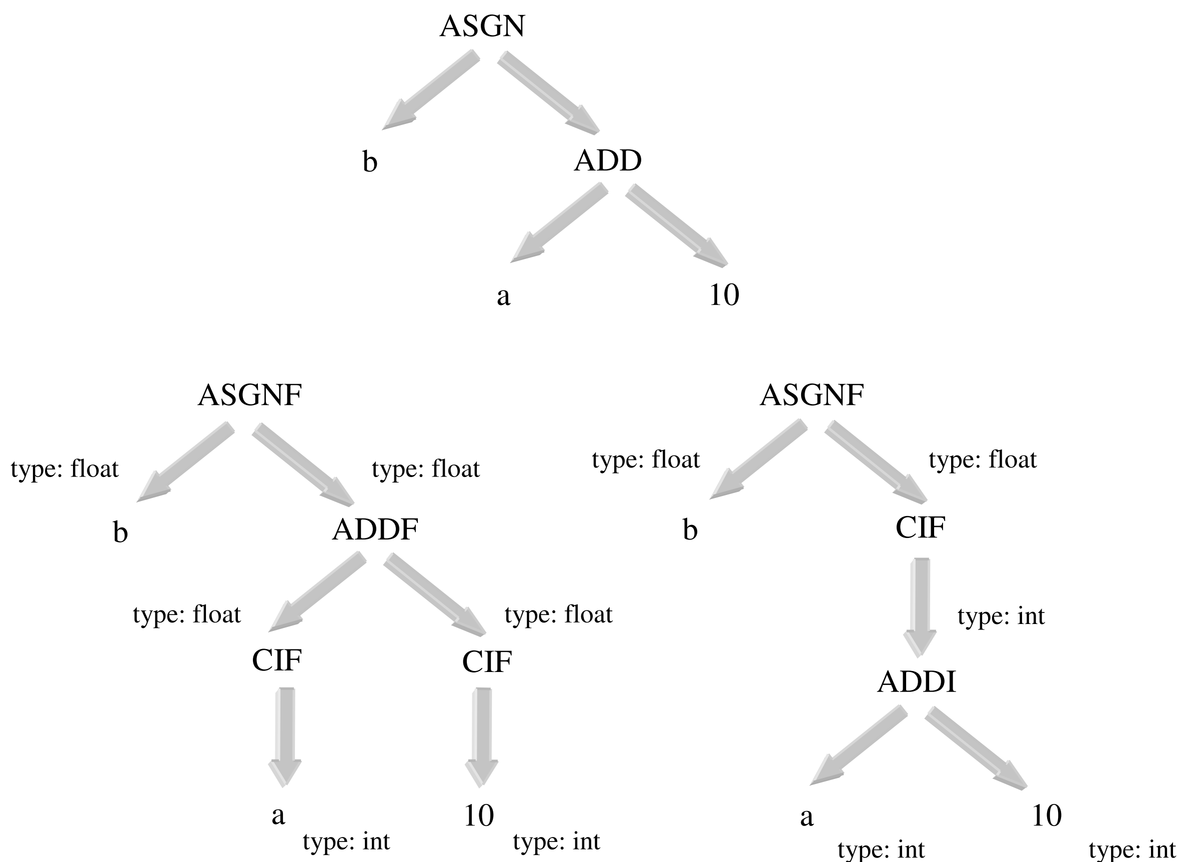


Fig. 2.8 – Representação da árvore de sintaxe e árvores intermédias da Eq. 2.6.

Para que seja possível realizar a operação da Eq. 2.6, é necessário converter a constante de valor 10 e a variável <u>*a*</u> para valores do tipo real (CIF) e aplicar o operador de adição para reais (ADDF), ou então, aplicar o operador de adição para inteiros (ADDI) sobre os dois operandos e converter o resultado para o tipo real. Em ambos os casos é ainda necessário realizar a atribuição da adição à variável <u>*b*</u> (ASGNF). A Fig. 2.8 representa a árvore de sintaxe e as árvores de representação intermédia para as duas soluções anteriores.

♦

Tratamento das Estruturas de Controlo

Dependendo da linguagem fonte, o GCI pode ter que tratar diversas estruturas de controlo, no entanto as mais vulgares são:

S -> **if** E **then** A | Eq. 2.7

if E **then** A_1 **else** A_2 | Eq. 2.8

while E **do begin** A **end** Eq. 2.9

Em que *E* representa uma expressão do tipo booleano e A_i uma expressão simples ou composta.

Para o GCI as estruturas de controlo só têm significado após se definir a representação numérica do resultado das expressões lógicas, isto porque noções como verdadeiro e falso são completamente abstractas para CGI. Para tal utiliza-se normalmente uma das duas seguintes convenções: a primeira, consiste em associar um valor numérico a cada um dos estados verdadeiro e falso, tal como definir zero para falso e um para verdadeiro, ou então zero para falso e qualquer outro valor para verdadeiro; a segunda abordagem representa o resultado através do próprio fluxo de controlo, ou seja, se o resultado da expressão é verdadeiro o programa continua a partir da posição *x*, caso contrário continua a partir da posição *y*. Esta segunda solução será abandonada por ser menos genérica.

Este tipo de preocupações não se coloca se a representação intermédia suportar o tipo de operadores utilizado, e de alguma forma cada um desses operadores possa ser relacionado com uma instrução máquina semelhante. Ficando neste caso a representação dos valores abstractos verdadeiro e falso, definida pela convenção utilizada pela própria máquina. No entanto, no caso em que não existe representação directa dos operadores há que contornar o processo de forma a se determinar o seu resultado numérico.

É comum utilizarem-se operadores do tipo booleanos na representação intermédia, tal como o **and**, **or**, e **not**, os quais possuem normalmente instruções equivalentes no código máquina. Mas os operadores do tipo relacional, tal como $>$, $<$, $\geq$, $\leq$, $\neq$ e $=$, não possuem normalmente representação directa. É respectivamente o caso da Eq. 2.10 e da Eq. 2.11.

a = b and c; Eq. 2.10

a = b < c; Eq. 2.11

Visto que as expressões que utilizam operadores relacionais, como as da Eq. 2.11, têm como resultado verdadeiro ou falso, que são conceitos abstractos para o GCI, é necessário que este resolva as expressões de forma a obter um resultado numérico possível de ser quantificado. Na prática, esta operação consiste em testar se a expressão $b < c$ é verdadeira, e nesse caso, atribuir à variável <u>*a*</u> o respectivo valor numérico convencionado para representar o valor lógico verdadeiro, procedendo-se de forma

semelhante para o caso da expressão ser falsa. Como exemplo, a Eq. 2.11 seria reescrita segundo a instrução condicional da Eq. 2.12.

if b < c **then** a=1 **else** a = 0; Eq. 2.12

Uma situação mais complexa ocorre quando a expressão condicional é composta por diversas sub-expressões, relacionadas por operadores do tipo **or** ou **and**. Nestas situações é necessário hierarquizar a sequência das operações e utilizar algumas optimizações que aproveitando as propriedades algébricas destes operadores, simplifiquem as expressões.

Exemplo 2.3

A Eq. 2.13 permite exemplificar o tratamento deste tipo de expressões.

a = (b < 20) or (b >40) Eq. 2.13

Aplicando a solução utilizada na Eq. 2.12, determina-se o valor da sub-expressão $b < 20$, da seguinte forma:

if b < 20 **then** a = 1 **else** E Eq. 2.14

De notar que no caso de b ser inferior a 20, já é condição suficiente para que a seja verdadeiro, caso contrário é necessário determinar o valor da sub-expressão $b > 40$.

if b < 20 **then** a = 1 **else**
    if b > 40 **then** a = 1 **else** a = 0

♦

Uma vez certificado que não ocorrem conflitos na representação dos resultados, pode então o GCI tratar de resolver as estruturas de controlo. Apenas como exemplo analise-se o comportamento do GCI para a Eq. 2.8.

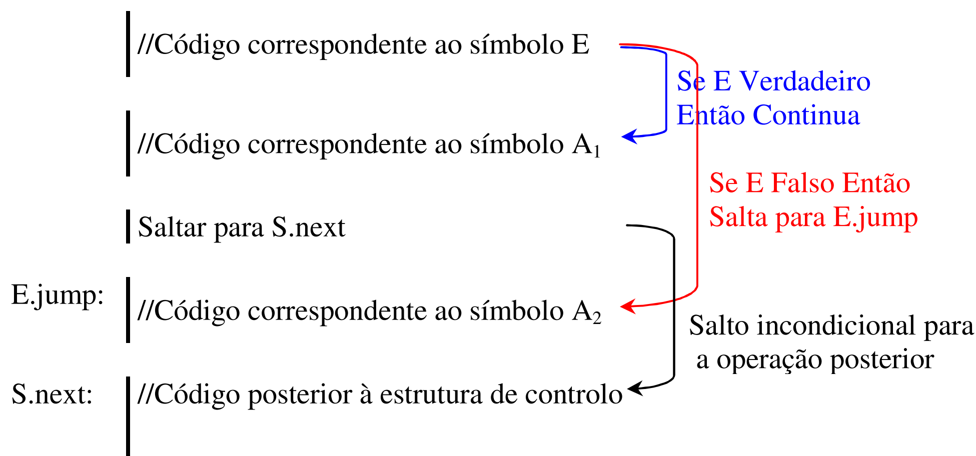


Fig. 2.9 – Exemplo da estruturação do código para expressões tipo a da Eq. 2.8.

Para este tipo de expressão pode-se convencionar que caso a condição resulte no valor lógico verdadeiro, então a execução continua através de A_1 e uma vez concluída a execução deste bloco, prossegue na instrução seguinte à expressão condicional. Caso a condição seja falsa, então a execução continua no princípio do bloco de código formado por A_2.

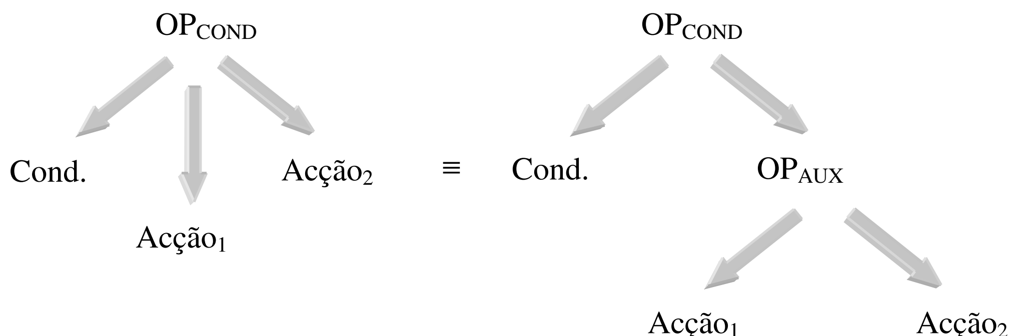


Fig. 2.10 – Conversão de uma operação de três operandos em duas de dois operandos.

Para tal, e uma vez que o GCI é implementado utilizando gramáticas de atributos, associa-se ao símbolo E (que representa a expressão condicional) um atributo (*jump*) no qual se armazena a posição onde o código continua a ser executado caso a expressão assuma determinado valor lógico (neste caso o valor falso). É ainda necessário associar ao símbolo S um segundo atributo (*next*) que identifica a instrução seguinte à expressão condicional, o qual permite conhecer a posição onde continua a execução do código após a conclusão de A_1. A Fig. 2.9, representa a organização da sequência do código que o GCI deve produzir para este tipo de expressão.

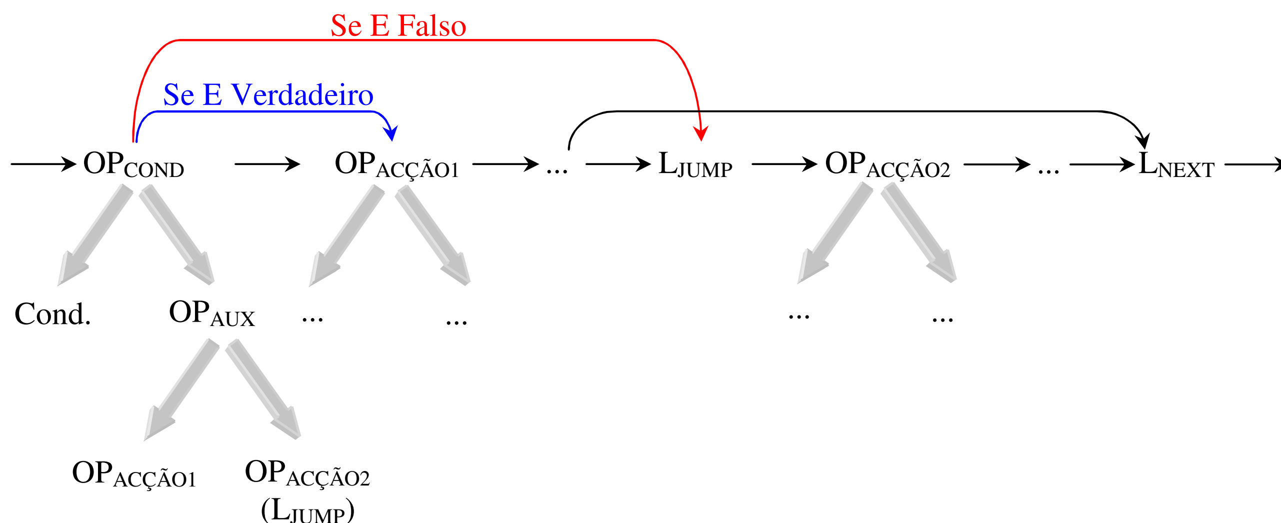


Fig. 2.11 – Decomposição da árvore binária numa floresta de árvores.

A utilização de árvores binárias na representação deste tipo de expressão é ligeiramente mais complicada, uma vez que a operação envolve três operandos, o da condição e as duas acções. Como se pode confirmar pela Fig. 2.10, a solução passa por

introduzir um operador fictício (OP_{AUX}), que permite decompor a operação condicional, que é do tipo ternário, num par de operadores binários.

Este tipo de representação pode tornar-se ineficaz no caso das acções serem formadas por várias expressões, uma vez que complica substancialmente a representação. Por essa razão é comum decompor as árvores numa lista de sub-árvores, designada por floresta, em que a árvore correspondente à expressão condicional apenas possui a condição e duas ligações (referências) para as primeiras árvores de cada uma das acções. A Fig. 2.11 exemplifica esta decomposição.

O GCI normalmente acrescenta uma árvore após as operações da acção A_1, cuja função é assinalar a presença da *label E.jump*. Tal é essencial, para o caso de se pretender gerar *assembly*, uma vez que as operações de *label* aparecem descriminadas entre o código.

Tratamento de funções

O tratamento de chamada e retorno de funções é uma das tarefas de maior complexidade com que o GCI se depara. Isto porque para invocar uma função, é necessário guardar o contexto de execução da função origem, passar os argumentos da função invocada, guardar o endereço de regresso e "saltar" para a posição inicial da função. No fim da função, é ainda necessário guardar o seu resultado, restaurar o contexto de execução da função origem e "saltar" para o endereço de regresso.

É função do GCI passar através da representação intermédia este conjunto de tarefas ao *back-end*, o que nem sempre é fácil de fazer devido às limitações da própria representação intermédia.

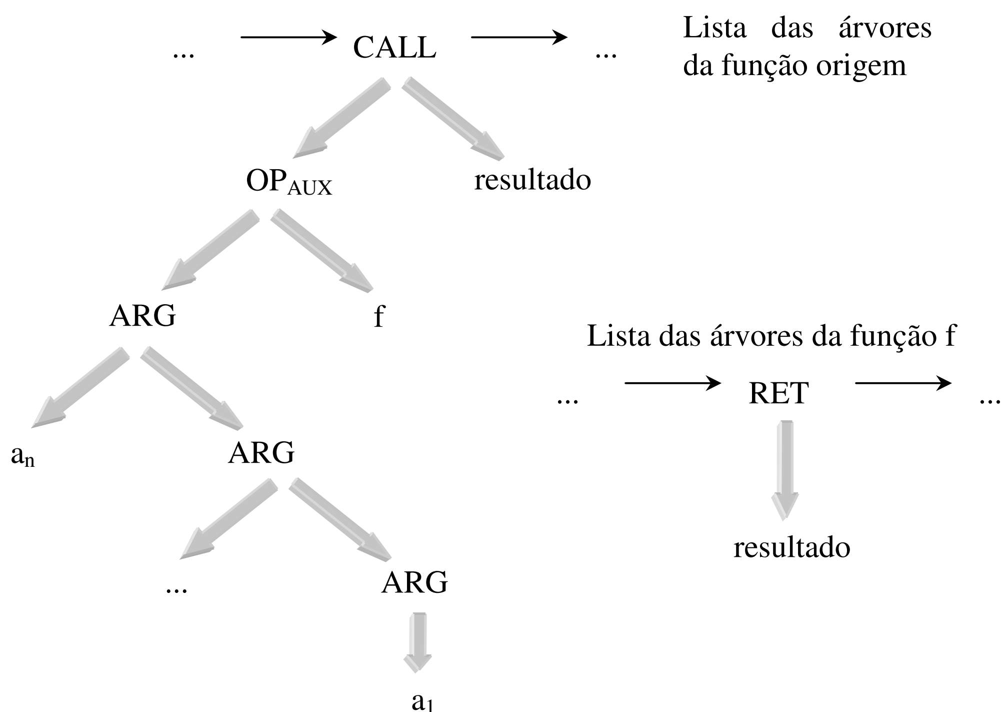


Fig. 2.12 – Estrutura das árvores de representação intermédia para uma função.

É imprescindível identificar duas fases bem distintas: uma primeira que corresponde à invocação da função, onde é necessário que a árvore represente o endereço e os

argumentos da função; e uma segunda, que corresponde ao retorno à função origem, onde a árvore deve representar o resultado a devolver e o endereço de retorno.

Na prática é vulgar converter o resultado da função num parâmetro de saída da função. Esta técnica permite que a função origem conheça a posição onde é devolvido o resultado antes mesmo de processar a função a invocar.

Na Fig. 2.12 encontra-se representada a lista de árvores da função origem (*fOrigem(...)*) e a lista de árvores da função invocada (*f(...)*). Na primeira lista, encontra-se a árvore responsável por invocar a função e na segunda lista, a árvore responsável por devolver a execução à função origem.

O operador que invoca a função é designado por CALL e possui três operandos, um para a lista de argumentos, outro para o endereço da função e ainda um terceiro com o endereço do valor devolvido pela função invocada. A lista de operandos é suportada recorrendo a um segundo operador auxiliar, designado por ARG.

```
... fOrigem(...)               int f ( a1, ..., an)
{                              int a1, ..., an;
  ...                          {
  y = f (a1, ..., an);           ...
  ...                            return x;
}                              }
```

A segunda lista (da função invocada) termina com a operação de retorno (RET), a qual possui pelo menos um operando, que é o valor a devolver. Pode no entanto, existir um segundo operando para representar o endereço (referência) da função origem.

2.3 *Back-End*

Nas secções anteriores descreveu-se resumidamente as funções de cada uma das fases do *front-end*, a interface entre este e o *back-end* e a geração do código intermédio. Pretende-se agora apresentar com algum detalhe as fases do *back-end*, ou seja, aquelas que dependem essencialmente das características da máquina, tal como do conjunto de instruções, dos modos de endereçamento, da quantidade de registos, etc. Normalmente estas fases pouco ou nada dependem da linguagem fonte.

As secções seguintes descrevem os processos para geração de código, alocação de registos e a optimização do código ao nível do *back-end*.

NOTA: Por simplicidade de representação, quando nesta secção se disser "geração de código", está-se a subentender "geração de código final".

2.3.1 Considerações sobre a Geração de Código

O funcionamento de um gerador de código, pode ser decomposto em pelo menos duas fases, a primeira, responsável pela selecção das instruções máquina a utilizar para cada expressão da representação intermédia; e a segunda, responsável por alocar os registos necessários para cada instrução.

Para além destas fases, cabe também ao *back-end* tratar de organizar o espaço de memória para a execução do programa ou então providenciar os mecanismos para tal.

É ainda habitual existir uma fase de optimização, normalmente designada por *peephole optimization*, que tanto se pode colocar antes ou depois da selecção das instruções. Esta tarefa é apresentada na secção 5.5.

Nesta secção e após uma breve descrição das diversas representações do código final, apresentam-se algumas das soluções utilizadas para a alocação do espaço de memória necessário à execução do programa, explicando o tratamento realizado pelo gerador de código final sobre a informação contida na tabela de símbolos. A secção continua com a descrição do tratamento das expressões da representação intermédia, com a alocação dos registos e termina com uma pequena abordagem à geração de código máquina.

Tipo de Código Final

O tipo de código final de um compilador, pode ser classificado como absoluto, relocatável ou código *assembly* [ASU86].

O código absoluto é uma representação em binário que utiliza endereços relativos, mas fixos, ou seja, as referências às posições de memória são feitas sempre através de um *offset* em relação à posição inicial da memória reservada para a execução do programa. A desvantagem principal encontra-se na utilização de referências fixas, que impede a compilação de um programa em módulos separados. Este pode, por outro lado, ser directamente carregado em memória e executado.

O código relocatável também é do tipo binário, no entanto, permite que os endereços funcionem como *offsets* relativos a uma referência externa, podendo ser posteriormente recalculados de forma a determinar a sua posição correcta. É este o tipo de código gerado por linguagens que permitam desenvolver aplicações em módulos separados, onde no fim é imprescindível realizar o processo de *linkagem*, de forma reunir todos os módulos num único executável, procedendo-se nesta última fase ao ajuste dos endereços.

Este tipo de código facilita o desenvolvimento de aplicações, uma vez que permite a reutilização de módulos pré-compilados, como é o caso das bibliotecas. No entanto, a geração de código relocatável, obriga o compilador a gerar informação extra, essencial para que o *linker* seja capaz de recalcular os endereços. Para além disso, acresce a fase de *linkagem* ao processo de geração de um executável.

A geração de *assembly* não produz directamente código binário, pelo que é necessário correr um assemblador para obter o código final, o que por si só aumenta o tempo de geração do executável. No entanto, esta representação é mais fácil de gerar, uma vez que não se depara com problemas de endereçamento, como por exemplo, determinar o endereço das variáveis ou a posição das *labels*.

Organização do Espaço de Memória

Uma das tarefas do gerador de código é estruturar o espaço de execução do programa compilado, garantindo mecanismos para guardar as variáveis do programa e toda a informação indispensável à sua execução.

O programa fonte é analisado, não como uma única estrutura, mas como um conjunto de funções, procedimentos, ou mesmo de blocos de código simples, onde seja possível, por exemplo, definir variáveis locais ao próprio bloco.

Para cada um destes blocos de código, é necessário reservar um espaço de memória, designado por registo de activação (*activation record* ou *frame*), onde se armazenam as variáveis locais ao próprio bloco de código e as variáveis temporárias que o compilador

determine serem úteis à sua execução. No caso das sub-rotinas, serve ainda para armazenar os parâmetros de entrada, os resultados a devolver pelas funções, a informação de *status* da máquina (tal como o valor dos diversos registos e o endereço de retorno) e, eventualmente, alguns apontadores para outros registos de activação necessários à execução da sub-rotina (por exemplo o registo de activação da função origem). A Fig. 2.13 apresenta um exemplo da estrutura de um registo de activação.

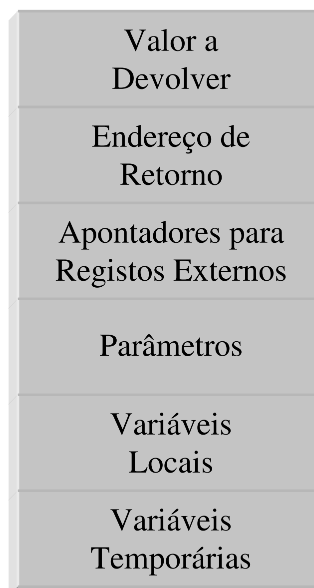


Fig. 2.13 – Estrutura do registo de activação.

Estratégias de Alocação

São duas as principais estratégias de alocação do espaço para os registos de activação, uma designada por alocação estática e outra por alocação de *stack*.

A alocação estática reserva o espaço em memória para os registos de activação durante a compilação, mais concretamente durante a geração de código. Tal solução, só é possível se as variáveis utilizadas são do tipo estático, condição essencial para se determinar o seu tamanho no momento da geração do código, de forma a que o compilador conheça o montante de memória a reservar para cada um dos registos de activação.

Desta forma, a estrutura exigida à organização dum programa em execução, consiste num bloco de código e num bloco para as variáveis estáticas, tais como os próprios registos de activação. Esta estrutura encontra-se representada na Fig. 2.14 a).

A alocação por *stack*, reserva o espaço necessário aos registos de activação em *run time*, ou seja durante a execução do programa e não durante a compilação. Para tal, utiliza uma *stack*, onde armazena os registos de activação conforme as respectivas rotinas (funções, procedimentos ou blocos de código) são invocadas. No fim da execução de cada uma, o respectivo registo é removido da *stack*.

Neste tipo de alocação a memória organiza-se em três blocos, um para o código das instruções, um para as variáveis estáticas, como por exemplo variáveis globais e um terceiro bloco onde se aloca dinamicamente o espaço para os registos de activação da *stack*. A estrutura encontra-se representada na Fig. 2.14 b).

Na alocação estática, como os registos de activação ficam embutidos no próprio código gerado, não é necessário criar mecanismos para realizar alocação em *run time*. Para além disso, facilita a referência de variáveis, acelera a execução do programa, e garante implicitamente que as variáveis retêm o seu valor mesmo após se sair da rotina da qual fazem parte, uma vez que os registos de activação ocupam uma posição fixa, utilizada apenas pela respectiva rotina.

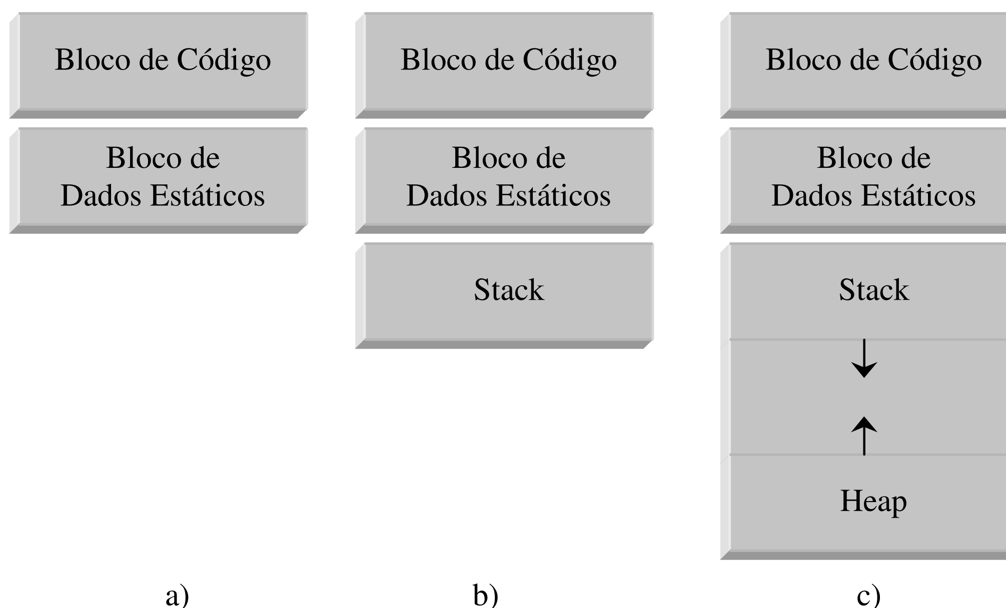


Fig. 2.14 – Organização da memória de execução de um programa, segundo as diferentes estratégias de alocação.

É exactamente nesta última vantagem, que reside a sua principal desvantagem. Como existe apenas um único registo para cada rotina, torna impraticável a utilização de recursividade. A tentativa de invocar uma rotina que já se encontre em execução, provoca a alteração dos campos do registo de activação, como é o caso do endereço de retorno ou o conteúdo dos registos, danificando assim a informação referente à primeira instância da rotina, causando danos irreparáveis na execução do próprio programa.

No caso da alocação por *stack* este problema já não se verifica, pois sempre que se invoca uma rotina, cria-se um novo registo de activação para armazenar a informação da nova instância. Desta forma existem tantos registos quantas as funções em execução.

A Fig. 2.15, apresenta um extracto de código em linguagem C, organizado segundo uma estrutura em árvore, em que a *root* representa o corpo principal do programa, (ex: *main( )* em C), e onde cada nodo descendente representa, da esquerda para a direita, a sequência das funções invocadas a partir do corpo principal.

Apesar da alocação por *stack* permitir trabalhar com variáveis do tipo dinâmico, é fundamental para isso, ter em conta determinados aspectos. Como a *stack*, é o único meio para passar e armazenar a informação necessária à execução do programa, também é necessário que lá fiquem armazenadas as variáveis dinâmicas. No entanto, estas não podem simplesmente ser guardadas nos registos de activação, pois nesse caso não seria possível definir uma estrutura de tamanho fixo, o que por si só, complicaria todo o sistema de referências. A solução encontrada consiste em guardar nos registos de activação apenas os endereços, ficando as variáveis propriamente ditas, colocadas na

stack mas fora dos registos de activação. Os endereços utilizados são relativos à posição que o registo de activação ocupa dentro da *stack*.

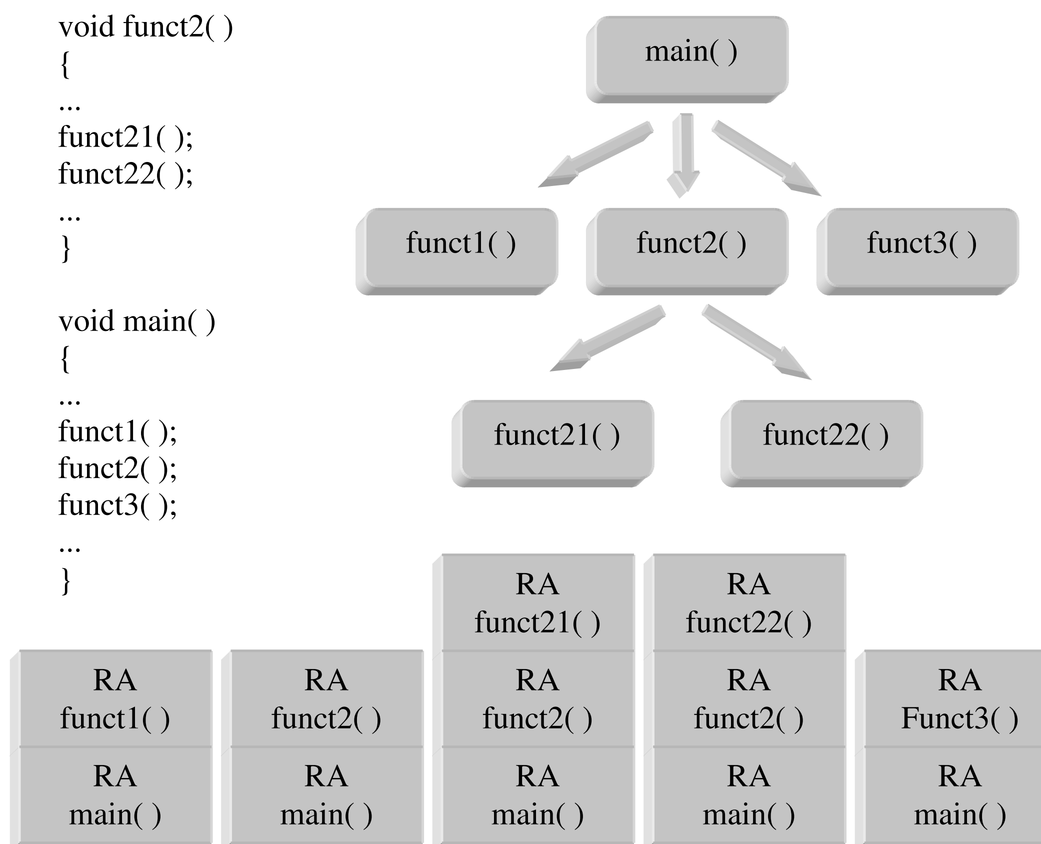


Fig. 2.15 – Exemplo da organização dos registos de activação na *stack*.

A utilização da *stack* nem sempre é a melhor solução para guardar as variáveis do tipo dinâmico. Por exemplo, no caso das funções que devolvem apontadores, torna-se indispensável manter o conteúdo das variáveis dinâmicas, mesmo após se sair da função.

Os compiladores que permitem este tipo de operações, utilizam um espaço de memória designado por *heap*, em que a política de alocação consiste apenas em atribuir o primeiro espaço vazio com tamanho suficiente para suportar a variável a alocar.

Normalmente a *heap* e a *stack*, utilizam o mesmo bloco de memória. Mas para que ambas as áreas não se sobreponham, convenciona-se que cada uma deve começar a alocar o espaço de que necessita, a partir dos extremos do bloco de memória, mas em sentidos opostos, como se representa na Fig. 2.14 c).

Antes de se explicar como toda esta organização se traduz na geração do código final, é útil descrever alguns aspectos sobre as características das instruções *assembly* e máquina.

Estrutura das Instruções e Modos de Endereçamento

O formato normal da estrutura de uma instrução consiste num operador, zero ou mais operandos e eventualmente um campo para o resultado. Em algumas máquinas os

operandos podem também servir para colocar o resultado. No entanto, nem todas as instruções obedecem a esta estrutura. Apresentam-se de seguida alguns exemplos ilustrativos das diversas instruções:

Operador	**Descrição**
NOP	Operação Nula

Operador	**Arg1**	**Arg2**	**Result.**	**Descrição**
ADD	A	B		Soma A com B e coloca o resultado em A.
ADD	A	B	C	Soma A com B e coloca o resultado em C.

Operador	**Arg1**	**Result.**	**Descrição**
CPL	A		Complementa A e coloca o resultado em A.
CPL	A	B	Complementa A e coloca o resultado em B.

Operador	**Arg1**	**Arg2**	**Arg3**	**Descrição**
JNE	A	B	C	Compara A com B, e caso sejam diferentes, salta para C.

Para se poder representar todos os exemplos atrás apresentados, recorre-se a um quadruplo, com a seguinte forma:

op arg1, arg2, result Eq. 2.15

Em que *op* identifica a operação, *arg1* e *arg2* são operandos e *result* pode ser um terceiro operando ou resultado.

Para os exemplos que se seguem, convencionou-se que cada um dos campos (op, arg1, arg2 e result) é representado por 1 byte, pelo que cada instrução tem como comprimento máximo 4 bytes. Sendo, no entanto, o operador (op) o único campo obrigatório.

A Unidade Central de Processamento (CPU), através do código *op*, não só determina a operação a realizar, como o número de argumentos e o tipo de endereçamento utilizado por cada um.

É possível que, cada um dos operandos utilize mais do que um modo de endereçamento [Gilm95]. Os mais vulgares são:

Endereçamento Herdado
Endereçamento Imediato
Endereçamento Directo
Endereçamento Registo Directo e Indirecto
Endereçamento Indexado
Endereçamento Relativo

No caso do *Endereçamento Herdado*, o operando é representado implicitamente no código do operador. É vulgarmente utilizado para representar registos específicos do processador. Por exemplo, a operação de multiplicar o conteúdo do acumulador A pelo acumulador B, (*MUL A,B*), pode ser representada apenas pelo código atribuído ao operador. Ou seja, basta o CPU identificar o operador, para que de imediato determine que operandos utilizar.

No *Endereçamento Imediato*, o operando representa o próprio valor a utilizar na operação (ex: #20). Neste caso, a instrução será composta por pelo menos dois bytes, um para o operador e outro para o operando que utiliza o endereçamento imediato.

No *Endereçamento Directo*, o operando representa o endereço onde se encontra o valor a utilizar pela operação (ex: 20). À semelhança do anterior, também este requer pelo menos dois bytes.

O *Endereçamento por Registo Directo ou Indirecto*, como o próprio nome indica, recorre à utilização de registos. No caso do endereçamento por registo directo, o valor a utilizar na operação encontra-se no próprio registo (ex: r[0] ou R0). No caso de endereçamento por registo indirecto o valor encontra-se no endereço apontado pelo registo (ex: *r[0] ou @R0).

É normal que a indicação do registo a utilizar, faça parte do próprio código do operador, pelo que não é obrigatório existir um byte reservado à identificação do registo. Por exemplo, no caso de um processador com 32 registos, em que são necessários 5 bits para assinalar o registo a utilizar, uma operação do tipo *ADD Ri, arg2*, consiste no seguinte conjunto de bits:

Byte do código op	**Byte de arg2**
y y y x x x x x	z z z z z z z z

Em que os *y*'s servem para identificar a operação *ADD* e os *x*'s o registo a utilizar. Os bits *z*'s representam o operando arg2.

O *Endereçamento Indexado* é representado na forma de ***base + offset*** (ex: 4(R0) ou @R0 + 4), em que normalmente a base consiste num registo e o *offset* num valor imediato. A base contém um endereço absoluto e o *offset* indica uma posição relativa à base. Este tipo de endereçamento necessita de mais um byte para o *offset*.

O *Endereçamento Relativo* é um caso específico do endereçamento indexado, em que o registo utilizado para a base é o *program counter*.

Nem todas as arquitecturas utilizam todos estes modos de endereçamento, no entanto as que o fazem normalmente estão incluídas na família de processadores CISC (*Complex Instruction Set Computer*), os quais caracterizam-se não só por utilizar diversas formas de endereçamento, como também instruções com tamanho variável e recorrerem a registos específicos para implementar determinado tipo de operações.

Já os processadores do tipo RISC (*Reduced Instruction Set Computer*), utilizam essencialmente instruções com endereçamento por registo ou endereçamento imediato, com tamanho fixo e em que todos os registos são de utilização geral. Mas como é inevitável ter que se carregar e despejar o conteúdo destes, de e para memória, existe um número muito limitado de instruções que podem referenciar posições de memória e que são normalmente designadas por instruções de LOAD e STORE.

Não se pretende aqui, discutir as vantagens e desvantagens de cada uma destas filosofias de concepção. Parte-se do princípio que ambas existem e que ambas necessitam de compiladores. Interessa apenas apurar, quais os aspectos a ter em consideração no desenvolvimento destes para cada uma das arquitecturas. Convém no entanto perceber que apesar de ambas as arquitecturas, necessitarem de seleccionar as instruções e alocar os registos, no caso dos CISC's, as tarefas são bastante mais complicadas, não só porque existem diversas opções para uma mesma instrução, como determinadas instruções utilizam registos específicos. No caso dos RISC, as tarefas são muito mais simples, porque normalmente só existe uma representação de cada instrução

e como os registos são todos de uso geral, o número de restrições que se coloca na alocação é bastante inferior.

Em contrapartida, a maior parte das instruções de um CISC, podem utilizar operandos directamente da memória, enquanto nos RISCs é necessário carregar previamente os valores nos registos, através de uma operação de LOAD.

Outro aspecto importante na escolha das instruções, é o seu tempo de execução, o qual depende em grande parte do tipo de endereçamento utilizado pelos operandos. Assim as instruções mais rápidas são as que utilizam operandos com endereçamento herdado ou endereçamento por registo, uma vez que o CPU com um único byte (*op*) identifica pelo menos um dos operandos e a instrução a executar. Para além disso, estes dois modos de endereçamento utilizam registos, os quais, como fazem parte da estrutura interna do CPU, possuem tempos de acesso inferiores aos necessários para aceder a qualquer valor da memória de dados (endereçamento directo) ou mesmo da memória de código (endereçamento imediato).

Geração de Código para os Registos de Activação

Pode-se agora regressar novamente para o problema da geração de código e mais concretamente à criação dos registos de activação para o caso da alocação estática.

Após a análise de cada rotina do código fonte e de se determinar a estrutura dos respectivos registos de activação, estes são colocados no bloco de dados estáticos, do qual o gerador de código guarda a posição inicial (*addr_inicial*), bem como a posição relativa (*offset*) de cada um dos registos de activação em relação à posição inicial. Para além disso, guarda também informação sobre a composição e respectiva organização de cada registo de activação.

Quando o gerador de código encontra uma chamada de uma função, calcula o endereço inicial do registo de activação desta, somando à posição inicial do bloco de dados estáticos, o respectivo *offset*.

O passo seguinte é preencher os campos do registo de activação, mas para tal há que os determinar, o que nem sempre é fácil. Por exemplo, o endereço de retorno, que obrigatoriamente deve ser preenchido pela função origem, pois apenas esta pode determinar onde continua a execução do programa após se sair da função a chamar, é calculado, somando à posição actual (posição da instrução que salvaguarda o endereço de retorno), o espaço em bytes ocupado pelas instruções que se encontram entre esta e a instrução a executar após se sair da função.

Exemplo 2.4

Este exemplo demonstra como se processa toda esta situação, apresentando o código gerado para a função origem.

Endereços	**Instruções**	**Comentários**
	MOV R0, #Addr_inicial	// Determina o endereço inicial do
	ADD R0, #offset	// registo de activação.
	MOV @R0+0, ...	// Salvaguarda no registo de activação
	MOV @R0+1, ...	// a informação necessária.
	...	
Addr_Actual:	MOV @R0+n, #Addr_Retorno	// Tal como o endereço de retorno.
	JUMP função	// Passa a execução à função a chamar.
Addr_Retorno:		

Segundo a convenção previamente definida, de que cada operador e operando ocupa um byte, determina-se o valor da posição de retorno, somando ao endereço *Addr_Actual* o tamanho da instrução *MOV @R0+n, #Addr_Retorno*, a qual ocupa 3 bytes (um para o operador *MOV*, uma para o *offset n*, e um terceiro para o *#Addr_Retorno*), com o tamanho da instrução *JUMP funcão*, que ocupa 2 bytes. Desta forma o valor de *Addr_Retorno*, é igual a *Addr_Actual* + 5 bytes, mas como Addr_Actual, corresponde neste caso ao valor do *program counter*, a instrução é reescrita da seguinte forma:

```
MOV @R0+n, @PC+5
```

Quando a execução passa para a função invocada, é salvaguardado o valor dos registos do processador. Esta operação pode também ser realizada pela função origem, no entanto, se tal for da responsabilidade da função invocada, permite ao gerador de código dispensar a operação de salvaguardar os registos, caso estes não sejam utilizados durante a execução da função.

No fim, a função deve devolver a execução do programa ao endereço de retorno. Para tal e pressupondo que o endereço do registo de activação se encontra em R0, é necessário realizar as seguintes instruções:

Instruções	**Comentários**
MOV R1, @R0+n JUMP R1	Devolve a execução à função origem.

♦

Em relação à alocação dos registos de activação na *stack*, o gerador de código necessita de conhecer o tamanho de cada um e a sua composição, e implementar os mecanismos necessários para lidar com a *stack*. De resto, é em tudo bastante semelhante à alocação estática. O Exemplo 2.5 ilustra a situação para a função origem e função a invocar.

Exemplo 2.5

No caso da função origem, é gerado o seguinte código, onde a constante *TRegAct* representa o tamanho do registo de activação:

Endereços	**Instruções**	**Comentários**
	MOV R0, SP	// Aloca e determina o endereço inicial
	ADD R0; #TregAct	// do registo de activação.
	MOV @R0-0, ...	// Salvaguarda a informação necessária,
	MOV @R0-1, ...	
	...	
Addr_Actual:	MOV @R0-n, @PC + 5	// tal como o endereço de retorno.
	JUMP função	// Passa a execução à função a chamar.
Addr_Retorno:	SUBB R0, #TregAct	// Repõe o valor do *stack pointer*.
	MOV SP, R0	// Liberta o espaço ocupado pelo
		// registo de activação.

Para o caso da função a invocar, o código é o seguinte:

Instruções	**Comentários**
MOV R1, @R0-n	// Devolve a execução à função origem.
JUMP R1	

Falta agora explicar como se processa a geração de código para as operações da representação intermédia.

2.3.2 Tratamento das Expressões da R.I.

Uma implementação eficiente e que produza código de qualidade considerável não é fácil de realizar. A possibilidade de utilizar mais do que uma instrução para substituir uma expressão da representação intermédia, associado à capacidade destas suportarem diversos modos de endereçamento e várias formas de comportamento em relação ao processo de armazenar o resultado, faz com que determinar uma solução que produza um código final de elevado desempenho, seja uma tarefa extremamente difícil. Mais ainda, não basta escolher as melhores instruções há também que ter em conta as limitações da máquina, como por exemplo, o número de registos disponíveis, ou as restrições colocadas na utilização destes.

O problema pode tornar-se mais complexo se o gerador de código também tentar reordenar a sequência das instruções, de forma a optimizar os recursos e diminuir o tempo de processamento. Nestas situações, há ainda que ter em conta a dependência entre dados, para garantir uma correcta execução do programa.

A solução mais simples consiste em utilizar uma e uma só instrução, para cada expressão ou conjunto de expressões da representação intermédia. No entanto, a qualidade do código obtido é muito pobre, não existindo qualquer garantia de se estar a utilizar as instruções mais correctas, pelo menos não para todos os casos, e obrigando mesmo a introduzir novas instruções de forma a manter a continuidade do fluxo de dados.

As poucas excepções ocorrem para as arquitecturas RISC, que seguem à risca a directiva de utilizar para cada operação da representação intermédia uma única instrução. Mesmo nestas condições, esta solução pode-se mostrar pouco eficiente, uma vez que maior parte das vezes é conveniente tratar as operações em conjuntos e não individualmente.

Soluções específicas

Os primeiros modelos de compiladores consideravam apenas quatro fases, a léxica, a sintáctica, a semântica e a geração de código. Era da responsabilidade desta última, a gestão dos registos, optimização do código e a selecção de instruções, tudo tratado como se fosse um único processo. Com este tipo de organização, não é de admirar que se considerasse esta, uma das fases mais complicadas de um compilador.

É dentro deste contexto que surge uma das soluções que está na base de alguns dos melhores compiladores actuais. Apareceu pela primeira vez no YC (Y compiler) [DF84a, DF84b], que teve por base o PO [DF80, GF88]. Trata-se como tal, de um

compilador com uma fase de optimização extremamente desenvolvida, a qual tem um papel fundamental em todo o processo, que é minimizar o número de decisões posteriores a tomar pelas fases de selecção e alocação de registos. É como tal pertinente que as optimizações sejam implementadas em conformidade com as características da própria máquina.

A estrutura deste compilador é composta por três fases (como se encontra representado na Fig. 2.16): a primeira é o *Expander*, responsável pela expansão de código; a segunda é o *Optimizer*, responsável pelas optimizações; e por fim, a fase de alocação de registos, o *Assigner*.

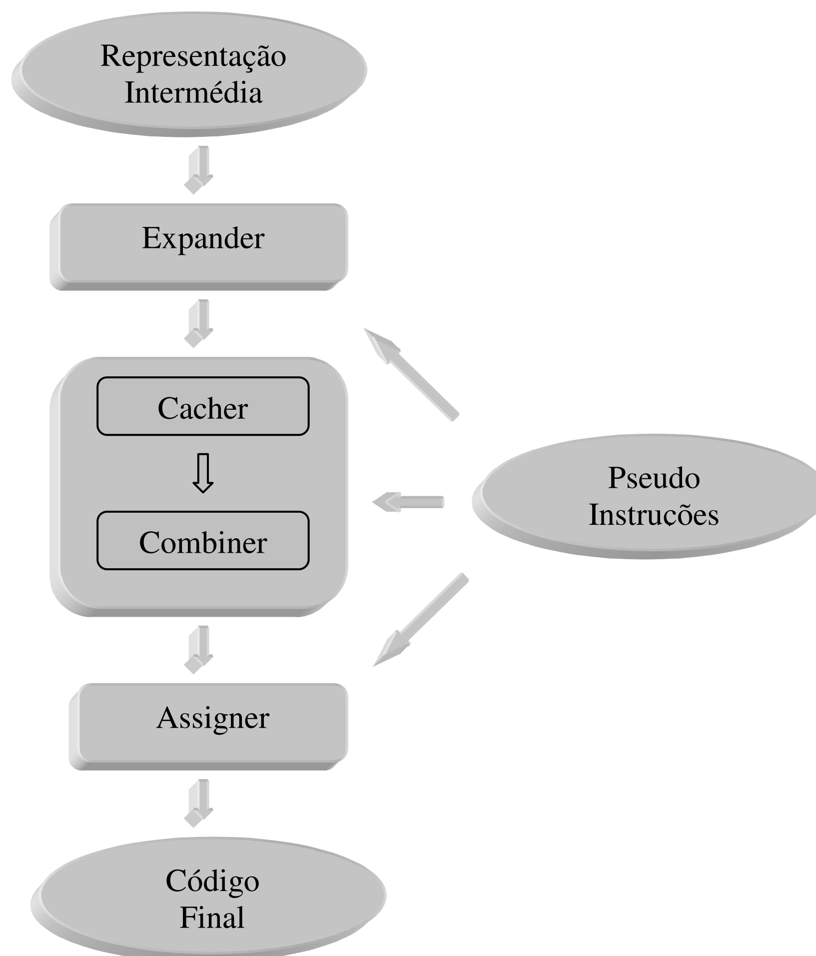


Fig. 2.16 – Representação da estrutura do compilador YC.

O *Expander* corresponde à geração do código intermédio do modelo que se encontra representado na Fig. 2.3. É da sua responsabilidade transformar a representação intermédia numa representação abstracta de muito baixo nível, com primitivas muito simples e de utilização geral, onde os operandos são representados por um conjunto infinito de pseudo-registos, muito semelhante à representação RTL.

O *Optimizer* é o grande responsável pela eficiência desta solução e é composto por duas sub-fases: o *Cacher*, responsável pela eliminação de sub-expressões e por assinalar as dependências entre registos; e o *Combiner*, responsável por combinar as operações da

representação intermédia e substitui-las pelas instruções correspondentes do processador.

O *Assigner* destina-se essencialmente a substituir os pseudo-registos por registos reais, uma vez que a selecção das instruções, fica praticamente reduzida a um processo de substituição de um para um, entre os operandos da representação abstracta e as instruções finais. A estrutura que representa a dependência de dados obtida pelo *Cacher*, serve para o *Assigner* gerir a alocação dos registos.

Mesmo com um número ilimitado de registos na representação abstracta, é necessário definir alguns critérios para a selecção das instruções. Critérios esses, que podem ter em consideração os seguintes aspectos:

1. Os operandos já se encontram em registos;

2. Os operandos podem ser necessários após a instrução actual;

3. O resultado contido num pseudo-registo é posteriormente alterado;

4. Pode ser necessário salvaguardar o conteúdo de um pseudo-registo, caso instruções posteriores sobreponham o seu conteúdo.

O objectivo dos pontos um e dois é maximizar a utilização dos registos em detrimento do endereçamento directo. O ponto três visa determinar se o resultado de uma operação é recalculado numa instrução posterior, o que permite dispensar a actualização do valor em memória. O ponto quatro serve para controlar um efeito indesejável do ponto anterior, que advém do facto de se manter o valor actual de uma variável em registo sem actualizar o valor em memória, uma vez que em determinadas instruções o resultado é guardado num dos operandos, pelo que se o valor deste for posteriormente necessário, há então que o salvaguardar em memória.

Caso a instrução permita referenciar separadamente a posição dos operandos e do resultado (três parâmetros), cabe ao *Assigner* verificar se pode ou não reutilizar os registos entretanto atribuídos para guardar o resultado.

Dos quatro critérios, apenas o primeiro serve para determinar o tipo de endereçamento a utilizar, os restantes assinalam a necessidade de salvaguardar o conteúdo dos registos.

Exemplo 2.6

Considere que determinada máquina suporta as operações de Subtracção (SUBB) e Atribuição/Load/Store (MOV), através das seguintes instruções (os respectivos custos encontram-se à direita):

SUBB	SUBB Reg, Addr	Reg = Reg - *Addr	2
	SUBB Reg_1, Reg_2	$Reg_1 = Reg_1 - Reg_2$	1
MOV	MOV Reg, Addr	Reg = Addr	2
	MOV Addr, Reg	&Addr = Reg	2
	MOV Addr, Addr	$\&Addr_1 = *Addr_2$	3

A seguir, apresentam-se duas expressões em linguagem intermédia e o tratamento destas, após o *Expander*, o *Cacher*, o *Combiner* e o *Assigner*:

Código Inicial	Expander	Cacher	Combiner	Assigner	Custo
	r[1] = &a	r[1] = &a			
	r[2] = * r[1]	r[2] = * r[1]	MOV r[2], &a	MOV R0, a	2
	r[3] = &b	r[3] = &b			
y = a - b	r[4] = * r[3]	r[4] = * r[3]	MOV r[4], &b	MOV R1, b	2
	r[5] = r[2] – r[4]	r[5] = r[2] – r[4]	SUBB r[2], r[4]	SUBB R0,R1	1
	r[6] = &y	r[6] = &y			
	*r[6] = r[5]	*r[6] = r[5]	MOV y, r[2]	MOV y, R0	2
	r[7] = &b				
	r[8] = * r[7]				
a = b - y	r[9] = &y				
	r[10] = * r[9]				
	r[11] = r[8] – r[10]	r[11] = r[4] – r[5]	SUBB r[4], r[2]	SUBB R1, R0	1
	r[12] = &a				
	*r[12] = r[11]	*r[1] = r[11]	MOV a, r[4]	MOV a, R1	2
				TOTAL	10

Como se pode verificar pelo exemplo, o *Expander* reescreve as expressões com o maior detalhe possível, o *Cacher* trata de eliminar expressões comuns, o *Combiner* realiza a selecção tendo em conta as instruções suportadas pela máquina e respectivos custos e por sua vez o *Assigner* atribui os registos físicos.

Caso o *Combiner* utilize na selecção os critérios definidos anteriormente, então começa por verificar a situação dos operandos da primeira instrução, de onde conclui que, quer a, quer b, ainda se encontram em memória. Mas enquanto a é necessário na primeira instrução, b à semelhança de y, apenas o será posteriormente.

O primeiro dilema é determinar qual a instrução da operação SUBB é que se deve utilizar, uma vez que são duas as opções possíveis:

	Opção 1	Custo	Opção 2	Custo
	MOV r[2], &a	2	MOV r[2], &a	2
	MOV r[4], &b	2	SUBB r[2], &b	2
	SUBB r[2], r[4]	1	MOV y, r[2]	2
	MOV y, r[2]	2		
Custo Total:		7		6
	y -> r[2]		y -> r[2]	
	b -> r[4]			

A primeira opção ao optar por carregar previamente a e b para registos, têm um custo de 7, enquanto que a segunda opção como só carrega um dos operandos, têm apenas um custo de 6. Mas como no primeiro caso b já fica em registo, possibilita obter alguns benefícios posteriores caso seja utilizado como operando. Nestas circunstâncias evita-se a realização de uma operação de *load* (MOV), o que permite poupar duas unidades, ou no caso de se optar por utilizar uma instrução que referencie o valor directamente de memória, permite poupar uma unidade. É como tal, correcto considerar que a opção 1 tem um custo potencial no pior dos casos de 6 unidades.

Há no entanto que ter em atenção, que antes de b ser novamente utilizado como operando, pode acontecer que faltem registos, podendo nestas circunstâncias a solução passar por libertar um dos que se encontram em utilização. Caso a escolha recaia sobre o registo utilizado por b, então perdem-se todos os benefícios potenciais.

Apesar dos custos serem iguais, seria legítimo pensar que o *Combiner* optasse pela opção 2, uma vez que o custo da primeira opção é apenas potencial.

No entanto, o *Combiner* pode também tirar proveito do facto de saber que a instrução SUBB, obriga a que o operando esquerdo se encontre sempre em registo. Pelo que caso não se realize a operação de *load* da variável b durante a primeira expressão, será forçosamente necessário realizá-la na segunda, o que permite garantir um custo potencial para a primeira opção de apenas 5 unidades

É possível confirmar pelo código que a seguir se apresenta, que o custo final da opção 1 é de 10 unidades, em oposição às 11 unidades da opção 2.

	Opção 1	**Custo**	**Opção 2**	**Custo**
	SUBB r[4], r[2]	1	MOV r[8], &b	2
	MOV a, r[4]	2	SUBB r[8], r[2]	1
			MOV a, r[8]	2
Custo Total:		7+3=10		6+5=11

Este exemplo serviu para demonstrar a complexidade do processo de selecção, nomeadamente quando em interligação com as restantes fases, mas também, para provar que o selector de instruções, neste caso representado pelo *Combiner*, necessita de contemplar um vasto conjunto de situações, que é tanto maior e complicado, quanto maior é o número de instruções e modos de endereçamento disponíveis pelo processador.

Não é difícil perceber que este modelo de compilador é muito pouco flexível no que diz respeito a portabilidade, uma vez que tal, implica reescrever na totalidade o *Combiner* e em grande parte as restantes fases da geração do código final.

Geração utilizando Programação Dinâmica

Uma das primeiras soluções desenvolvidas para geração de código independente das características da máquina, nomeadamente do formato das instruções, foi o PCC2 – Portable C Compiler [AJ76], que recorre aos princípios da programação dinâmica para implementar um algoritmo capaz de gerar código óptimo, para uma vasta gama de maquinas.

Como um dos objectivos definidos para o PCC2 foi construir um compilador portável, tentou-se criar um mecanismo de geração que fosse independente das características do processador, mas que permitisse gerar o conjunto de instruções óptimo. À semelhança da solução anterior, também aqui, a geração consiste em seleccionar as instruções e alocar os registos.

O algoritmo proposto, que funciona sobre árvores binárias, consiste em decompor o processo de geração, de forma a que este se realize ao nível de cada nodo. A solução óptima é então obtida através de uma combinação entre as soluções óptimas dos seus descendentes e as instruções possíveis de aplicar ao próprio nodo. A solução final é obtida após se processar todos os nodos da árvore segundo uma travessia *bottom-up*.

As decisões são feitas em *run time*, em que para tal se associa a cada um dos nodos, um vector com *k+1* elementos, em que *k* representa o número de registos disponíveis ([0, k]). Na i-ésima posição do vector, encontra-se o custo necessário para se obter uma solução (para o nodo em causa), utilizando apenas *i* registos, e na posição zero, o custo para realizar a operação em memória (sem registos).

Desta forma, o algoritmo começa por determinar os vectores dos nodos terminais, ou seja, o custo de utilizar directamente o respectivo identificador ou constante, com um ou mais registos. Depois prossegue através dos nodos intermédios, construindo os respectivos vectores, com base nas combinações de menor custo, entre as instruções possíveis de utilizar e os vectores das árvores descendentes. O Exemplo 2.7 ilustra o funcionamento deste algoritmo.

<u>Exemplo</u> <u>2.7</u>

Considere a árvore da Eq. 2.16 e um conjunto de instruções com custos e formatos semelhantes às utilizadas no exemplo da secção anterior (Exemplo 2.6).

a * b + c / (d - e) Eq. 2.16

A árvore e os respectivos vectores de custo encontram-se representados na Fig. 2.17 e é determinada considerando que existem apenas dois registos. De notar que no caso de um nodo necessitar de mais registos do que os disponíveis, o custo atribuído é infinito ou impossível.

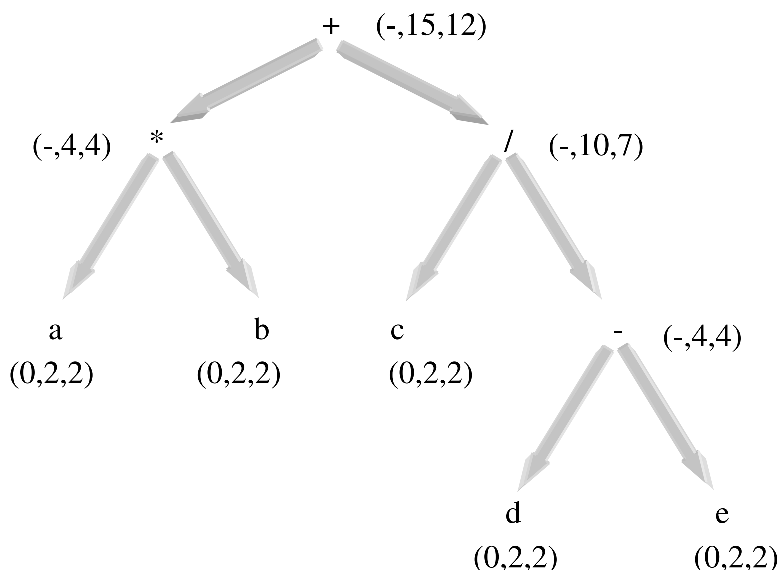


Fig. 2.17 – Representação da árvore da Eq. 2.10, com os respectivos vectores de custo.

Seja V, o vector custo, então o cálculo de V[0], para um qualquer nodo terminal, consiste no custo de utilizar a respectiva variável directamente de memória. Mas como por defeito as variáveis já se encontram em memória, então o custo atribuído a V[0] é de zero. V[1] e V[2] possuem os custos de colocar a variável num registo, quando existem respectivamente um e dois registos disponíveis.

Para os nodos onde ocorrem as operações de subtracção e de multiplicação, com um único registo obtém-se um custo de 4 unidades, duas provenientes da sub-árvore

esquerda (para colocar a variável em registo), mais duas da própria instrução e zero da sub-árvore direita. Com dois registos, o custo passa para 5 unidades, pelo que se opta por utilizar apenas um único registo.

Para os nodos onde ocorrem as operações de adição e divisão, em que ambos operandos são registos, o custo de V[1] tem que considerar a operação de converter o operando direito, de registo para endereço (2 unidades), mais o custo da instrução, num total de 4 unidades. Caso se encontrem dois registos disponíveis o custo é de apenas uma unidade.

O resultado final da selecção é então o seguinte:

```
MOV r[0], d          MOV R0, d
SUBB r[0], e         SUBB R0, e
MOV r[1], c          MOV R1, c
DIV r[1], r[0]       DIV R1, R0
MOV r[2], a          MOV R0, a
MUL r[2], b          MUL R0, b
ADD r[2], r[1]       ADD R0, R1
```

Ao contrário da solução anterior, esta já permite alguma independência entre o mecanismo de geração e as características das instruções. E tal deve-se essencialmente ao facto da escolha ser feita dentro de um contexto mínimo que se restringe a cada um dos nodos.

É lógico que faz falta conhecer a relação entre cada operador da representação intermédia e as instruções que o podem representar e identificar para cada uma destas, o tipo de endereçamento utilizado por cada um dos seus operandos e o respectivo custo.

Table-Driven Selector

Uma das primeiras tentativas de desenvolver um selector de instruções completamente independente das características do processador, foi apresentada por S. Graham [Graham80], que adapta as soluções utilizadas na análise sintáctica, sobre as quais já existe perfeito domínio, ao processo de selecção de instruções.

NOTA: No resto desta secção, parte-se do princípio que a representação intermédia se encontra representada sob a forma de árvores.

Os geradores de analisadores sintácticos constroem a partir da gramática de uma linguagem, um conjunto de tabelas com as quais o mecanismo de análise sintáctica reconhece as expressões. De forma similar, pretende-se obter a partir da descrição das instruções do processador, todo um conjunto de informação, representando de forma homogénea (tabelas), que possa ser disponibilizado a um selector que seja independente das características do processador, para que este realize a selecção das instruções.

Uma abordagem consiste em considerar que os não-terminais podem representar o tipo de endereçamento que resulta das instruções seleccionadas para cada nodo. Só que enquanto na análise sintáctica, o processo de reconhecimento das produções de uma gramática numa árvore, se processa naturalmente através de uma travessia tipo *top-down*, em que dado um não-terminal que representa todo o programa fonte se tenta identificar as produções que nele resultam e depois as produções que resultam nos não-terminais que se encontram RHS da primeira produção e por aí adiante. Na selecção das

instruções o processo funciona exactamente ao contrário, ou seja, a abordagem é feita segundo uma estratégia *bottom-up*, em que antes de se processar qualquer nodo é necessário processar previamente todos os seus descendentes de forma a determinar o tipo de endereçamento a utilizar para cada um dos seus operandos.

O que S. Graham propôs foi utilizar uma abordagem do tipo *top-down*, em que para cada nodo e após se determinar a respectiva instrução, se propaga pelos descendentes o tipo de endereçamento que deve resultar das instruções a estes atribuídas. Ou seja, mediante a instrução seleccionada, obtém-se um conjunto de não-terminais, um por cada operando, em que cada um passa a ser o não-terminal objectivo a atingir por cada uma das sub-árvores.

Este modelo apresenta diversas vantagens, mas uma das mais importantes, está na separação nítida entre o mecanismo de selecção, da estrutura de informação com a descrição do processador.

Outra vantagem igualmente importante, é que à semelhança das árvores de sintaxe, também aqui é possível acrescentar atributos e condições de contexto, aumentando assim a capacidade do selector, permitindo por exemplo: seleccionar as instruções mediante a compostura em que as operações surgem, adicionar funções de custo, ou até, realizar o próprio processo de alocação de registos. Esta abordagem foi posteriormente apresentada por Ganapathi e Fischer [GF82, GF84, GF85].

Infelizmente, a geração de código com base em parsers também possui algumas desvantagens. A principal advém do facto da selecção ser feita *on-the-fly*, ou seja, conforme se processam os nodos. É que mesmo que se utilizem funções de custo para determinar a melhor instrução a utilizar, tal escolha, é feita na melhor das hipóteses com o conhecimento que advém dos nodos superiores, mas sem qualquer conhecimento do que se passa nas sub-árvores. O que é certamente grave, uma vez que a selecção das instruções de um nodo encontra-se condicionada pelas escolhas anteriores. Pelo que uma escolha que aparentemente possa parecer acertada, pode-se traduzir numa catástrofe na solução final.

Outro aspecto importante, advém do facto, de que mesmo após se aplicar as condições de contexto, podem existir várias soluções possíveis. Nesta situação, a escolha recai sobre a primeira opção que ocorre, independente desta ser ou não a melhor. Há como tal que ter em atenção a ordem pela qual se descrevem as produções e mesmo assim nada garante a escolha acertada.

Comprova-se ainda, que na prática, uma gramática para descrever uma arquitectura do tipo CISC necessita de várias centenas de produções e resulta em tabelas demasiado grandes.

Como se poderá ver no capítulo 7, existem soluções que ultrapassam estas desvantagens, oferecendo praticamente os mesmos benefícios, mas normalmente a custo de degradarem o tempo de execução, por requererem duas travessias da árvore.

2.3.3 Alocação de Registos

A alocação dos registos é uma das fases de maior importância para a qualidade do código final, uma vez que um mau processo de alocação pode pôr em causa todo o esforço realizado pelas outras fases do processo de compilação. Pelo que, mesmo com os diversos estudos e experiências realizadas nesta área, continua ainda a ser uma das fases que maiores dificuldades coloca quanto à compreensão e implementação.

A função do sistema de alocação é determinar onde é que as variáveis e os valores manipulados por um programa devem residir ao longo da sua execução. Em que algumas das soluções possíveis são: memória, registos, *stacks*, etc.

Existem várias formas de implementar um sistema de alocação, no entanto, praticamente todas partilham o mesmo modelo de funcionamento, que é composto por duas fases: o *Allocator* responsável por determinar onde reside a variável ou valor; e o *Assigner* cuja função é realizar a gestão dos recursos, o que normalmente se resume à gestão dos registos. O modelo completo deste tipo de sistema encontra-se representado na Fig. 2.18.

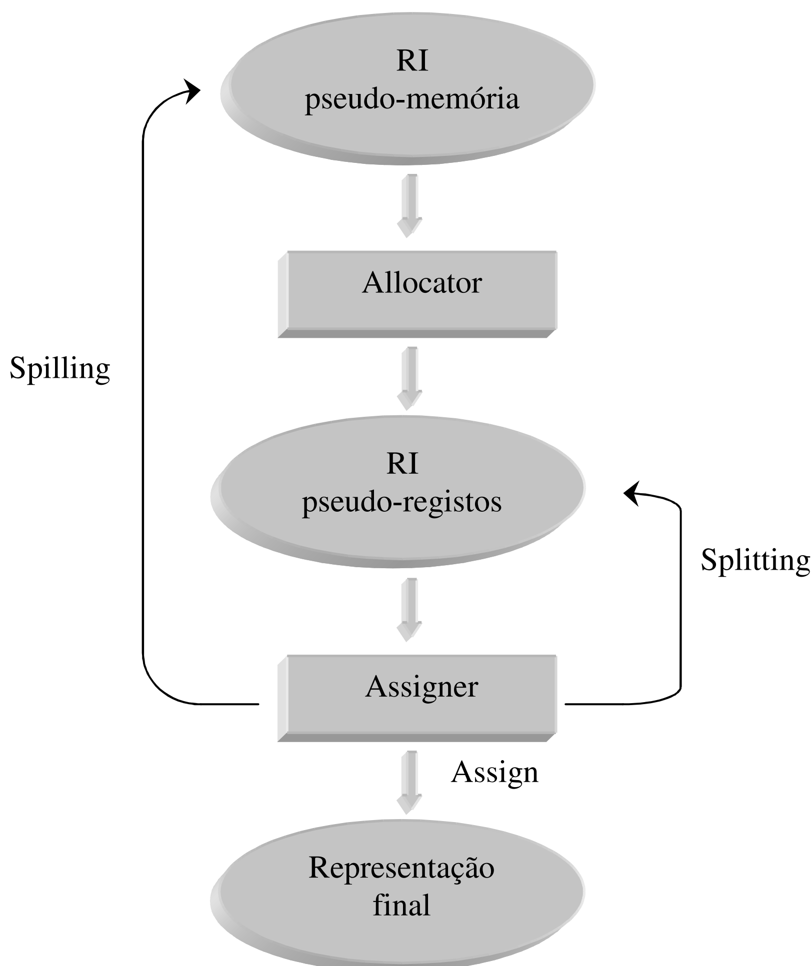


Fig. 2.18 – Estrutura de um sistema de alocação de registos.

Para uma apresentação mais detalhada deste modelo, há antes que definir a noção de pseudo-memória e pseudo-registo. A pseudo-memória é a representação atribuída por defeito às variáveis, através da qual se pretende representar uma qualquer posição que a variável possa ocupar, quer esta seja em memória, nos registos, ou noutro qualquer sítio. Os pseudo-registos surgem numa fase posterior, representando as variáveis candidatas aos registos físicos. Ambas as formas de representação são independentes de qualquer restrição física do processador ou sistema operativo, e utilizam-se ao nível do *front-end* e da representação intermédia.

É função do *Allocator* pegar nas variáveis, ainda na representação de pseudo-memórias e seleccionar quais são as que devem ser consideradas como pseudo-registos, indicando ainda o conjunto dos registos (físicos) que cada um destes pode utilizar. De notar que num processador os registos podem estar organizados em subconjuntos, segundo o fim a que se destinam, pelo que é imprescindível distinguir quais os que se devem utilizar em cada pseudo-registo.

Para conseguir realizar estas operações, o *Allocator*, necessita de possuir alguma informação sobre o processador, como por exemplo, o padrão de cada uma das instruções, os tipos de registos e respectivas quantidades, etc.

O *Assigner* pega na representação obtida após execução do *Allocator* e tenta atribuir a cada pseudo-registo um dos registos físicos. Caso existam registos disponíveis e em número suficiente, basta escolher os necessários e proceder à atribuição. Convém no entanto salientar que, mesmo com um número suficiente de registos disponíveis, o processo de selecção de quais utilizar pode ser extremamente importante para a eficiência final do processo de alocação (partilha de registos genéricos para funções específicas). Este e outros aspectos são discutidos na capítulo 6.

Mas o problema principal surge, caso não existam registos disponíveis em número suficientes para se proceder à atribuição. Nesta situação, é necessário optar por uma das três seguintes soluções:

1. Dar o problema da alocação como insolúvel;

2. Realizar o *spilling* de um registo, o que consiste em tentar substituir a sua utilização por uma posição de memória (endereçamento directo). O que obriga a reescrever as expressões da representação intermédia modificando todas as referências ao registo em causa, para a nova posição de memória. É o tipo de solução utilizada para as arquitecturas do tipo CISC, uma vez que estas normalmente comportam instruções com endereçamento directo. Um bom algoritmo de alocação deve ser capaz de quantificar o custo desta opção de forma a determinar a degradação do código final.

3. Realizar o *splitting* de um registo, o que consiste em libertar o registo, armazenando temporariamente o seu valor em memória, o qual deve ser reposto antes da próxima utilização. Esta solução obriga a inserir instruções para guardar e ler os valores dos registos de e para memória. Pode, em casos bem ponderados, ser uma solução melhor do que a anterior.

Nestas circunstâncias, e partindo do princípio que a solução só pode passar pelo ponto dois (*spilling*) ou pelo ponto três (*splitting*), é necessário que o *Assigner* escolha um, de entre os registos que se encontram ocupados (do mesmo tipo), para este seja "libertado" através das operações atrás referidas.

As principais diferenças entre as várias estratégias de alocação devem-se aos métodos utilizados na selecção dos registos a libertar, no operação a realizar (*spilling* ou *splitting*) e no tipo de abordagem utilizada pelo algoritmo de alocação (local ou global).

Não será demais lembrar que, de uma forma genérica, qualquer operação que se realize sobre registos, não só é mais rápida a executar, como possui um tamanho inferior. É como tal, conveniente maximizar a sua utilização em detrimento de outros modos de endereçamento, como por exemplo o endereçamento directo.

Após esta breve explicação da constituição de um sistema de alocação de registos, verifica-se que, as funções do *Allocator* se encontram intrinsecamente ligadas às fases anteriores de selecção e optimização, o que se comprova nos capítulos 5, 6 e 7. Por agora, pretende-se apenas focar o funcionamento do *Assigner*.

NOTA: De agora em diante, designar-se-á por alocação de registos as operações realizadas pelo *Assigner*.

O processo de alocação, torna-se bastante diferente conforme a abordagem utilizada, seja local ou global. As estratégias locais realizam a alocação, considerando apenas o bloco de código ou a expressão onde está inserida a instrução para a qual é necessário alocar o registo. Este tipo de alocação é utilizado em sistemas menos pretensiosos, principalmente em relação à qualidade final do código, mas que em contrapartida, permitem implementar compiladores simples e versáteis. As desvantagens, nomeadamente em relação à qualidade do código final, são em parte compensadas através da utilização de um bom sistema de selecção de instruções, ou através de uma vasta aplicação de optimizações. Esta última solução pode no entanto, comprometer o tipo de vantagens pretendidas para esta forma de alocação.

Estes algoritmos são simples, porque realizam a alocação localmente, o que dispensa a necessidade de conhecer o que se passa no resto do programa, basta uma breve análise local, para se determinar tudo o que é necessário ao processo de alocação. Justifica-se como tal, que em caso de falta de registos (disponíveis), se opte por soluções do tipo *spilling*, uma vez que, é apenas necessário substituir a instrução localmente, para que se consiga concluir a alocação com êxito. Mas se tal não for possível, recorre-se então a soluções de *splitting*, tendo sempre presente que este tipo de alocação tem como vantagem a simplicidade, dispensando como tal grandes processos de análise, tais como, os que são por vezes necessários para se determinar onde inserir as instruções de *splitting* e de *load*.

Uma das estratégias locais mais simples de implementar, tem por regra utilizar o registo cuja a próxima utilização se encontra a maior distância. Esta estratégia produz a solução local óptima, necessita no entanto de conhecer à priori quando é que determinado valor contido num registo será novamente utilizado.

Estratégias semelhantes, mas que necessitam de apenas uma passagem, sem que, no entanto, garantam a solução óptima (local), têm por base a escolha do registo cuja última utilização foi realizada há mais tempo, ou então, cuja primeira utilização se efectuou há mais tempo.

Uma solução alternativa para a alocação local, consiste em contar o número de vezes que um pseudo-registo é referenciado, o que pode ser feito aquando a execução do *Allocator*. Com o valor da contagem, é possível determinar o período de vida de uma variável num registo, pois sempre que este é referenciado durante o processo de alocação, a contagem sofre um decréscimo; dessa forma, quando atingir o valor zero é garantido que o registo já não é mais necessário, pelo que, se pode salvaguardar o seu conteúdo e libertá-lo. No caso de ser necessário realizar alguma operação de *spilling* ou de *splitting*, a contagem serve de indicador para determinar qual o registo a libertar, escolhendo-se o de menor valor na esperança de que, quanto menor a contagem, menor o número de referências posteriores a esse valor. Este critério é puramente heurístico, não garantindo qualquer tipo de solução óptima.

As estratégias de alocação global distinguem-se das anteriores, por realizarem a alocação, dentro de um contexto mais vasto, como por exemplo, sobre um programa ou

uma função. O que permite, uma distribuição mais eficiente dos registos, minimizando o número de operações de *spilling*, de *splitting* e de trocas entre registos. As contra partidas, traduzem-se na complexidade dos algoritmos, uma vez que, um grande número destes, recorre a elaboradas técnicas de análise, o que se traduz no aumento do tempo de execução, do tamanho do compilador, etc.

A alocação global é quase sempre um problema NP-Completo, pelo que as soluções têm sempre por base heurísticas que sem garantirem qualquer tipo de solução óptima, permitem simplificar substancialmente as soluções.

O *packing* é uma das estratégias de alocação global onde cada registo físico representa um *pack*, cuja dimensão é o período de execução do programa. A ideia base consiste em preencher cada *pack* (registo ao longo do tempo), mas de forma a que não ocorra sobreposição entre o período de vida útil de cada um dos pseudo-registo. Para tal é necessário utilizar técnicas de análise do fluxo de dados, de forma a permitir determinar o período de vida de cada variável.

Trata-se de uma solução genérica de alocação de recursos para a qual existem diversas implementações. As mais simples limitam-se a atribuir os pseudo-registos aos *packs* conforme estes estão disponíveis, adiando a execução das instruções quando não existem os recursos necessários. O que por motivos relacionados com as dependências entre dados, pode fazer com que o sistema de alocação caía em ciclo infinito (dead lock).

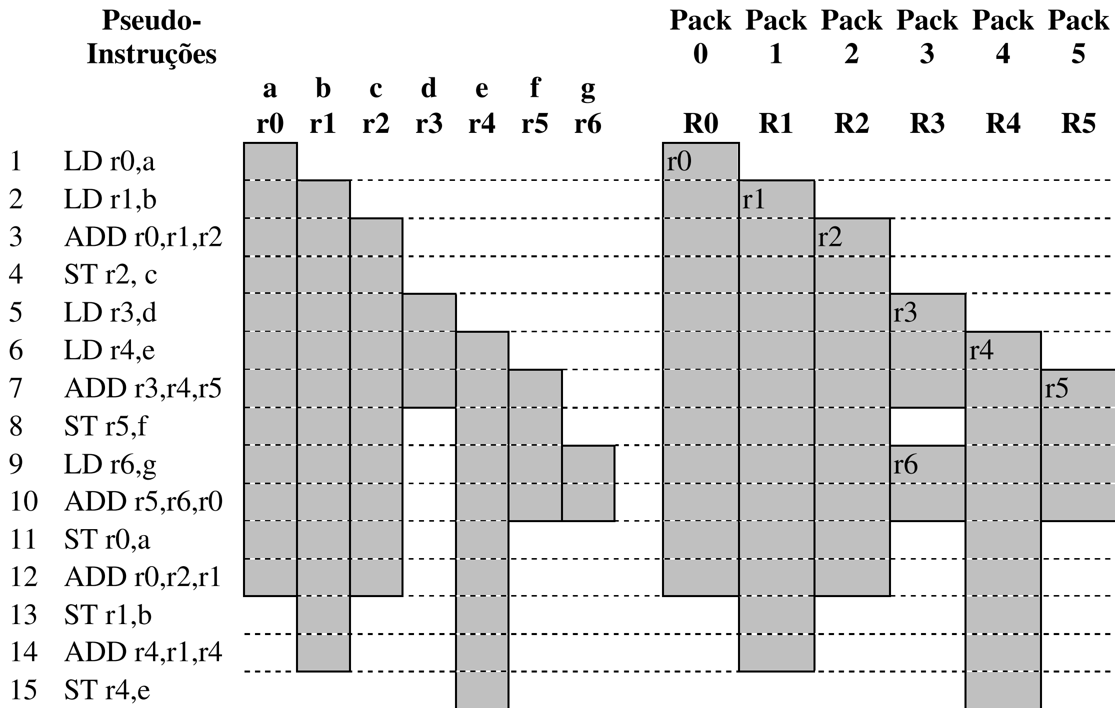
Existem implementações mais complexas, que tentam gerir melhor atribuição dos pseudo-registos aos *packs*, utilizando para além dos mecanismos de análise do fluxo de dados, os mecanismos de análise de dependências entre dados, maximizando assim a ocupação dos *pack* ao longo do tempo e detectando potenciais situações de ciclo infinito. Há, no entanto, que ter em atenção a carga computacional deste tipo de soluções, onde por vezes a utilização de heurísticas, permite simplificar consideravelmente os algoritmos, sem que para isso degrade significativamente o sistema de alocação.

Exemplo 2.8

O presente exemplo permite mostrar o funcionamento de um sistema de alocação global de *packing*, para a seguinte sequência de instruções:

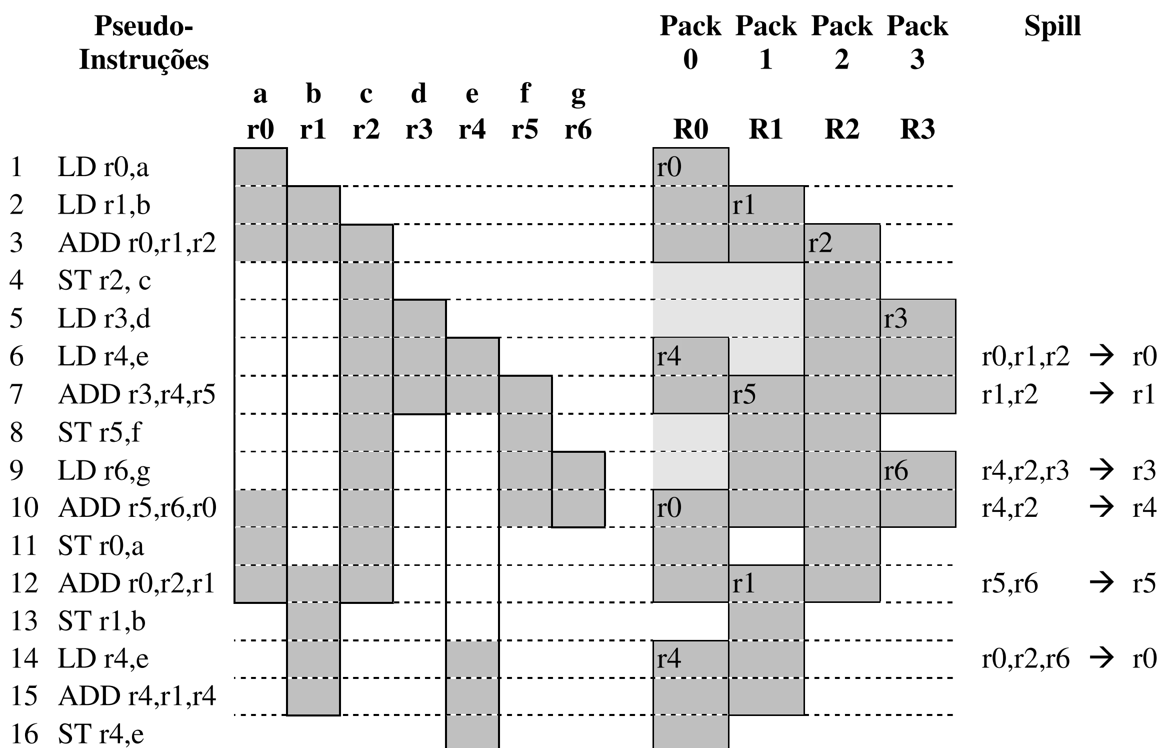
c = a + b
f = d + e
a = f + g
b = a + c
e = e + b

O seguinte esquema representa as respectivas pseudo-instruções, o período de vida de cada variável (pseudo-registo) e a distribuição destas pelos registos físicos, segundo o sistema de alocação por *packing*, onde cada registo é alocado a uma variável no início do seu período de vida e só no fim deste é que é libertado. Os registos são atribuídos conforme estão ou não disponíveis e segundo a ordem pela qual estão representados (de R0 para R5).



Como resultado da alocação, comprova-se que são necessários seis registos físicos para se concluir a geração do código final com sucesso. No entanto nem sempre é possível utilizar tantos registos como os idealmente necessários, pois por vezes só está disponível um número restrito de registos.

O seguinte esquema representa o processo de alocação por *packing*, utilizando apenas quatro registos, o que implica recorrer a operações de *splitting*, para concluir a alocação com êxito.



Devido ao número restrito de registos, torna-se necessário introduzir diversas operações de *splitting*, as quais são realizadas de forma implícita sempre que se salvaguarda o resultado das expressões em memória (operações de *store*). A única instrução necessária de introduzir foi a 14, com o objectivo de repor o valor da variável *e* em registo.

Do lado direito do esquema encontram-se representados os pseudo-registos possíveis de utilizar nas operações de *splitting* para cada instante. Estes são escolhidos, em primeiro lugar, de entre os pseudo-registos que já terminaram o seu período de vida (pelo que já não necessitam dos registos que ocupam), ou então, e só caso não exista nenhum pseudo-registo na condição anterior, de entre aqueles que não estão directamente envolvidos na operação que se está a realizar, mas de preferência, cuja próxima utilização se encontre a maior distância.

É possível verificar pelo exemplo anterior, que os resultados obtidos dependem da implementação e que uma boa implementação pode aumentar em muito a complexidade do sistema de alocação.

Ambas as estratégias de alocação, global e local, serão novamente abordadas no capítulo 6, onde ainda se descrevem outras implementações relacionadas com o trabalho desenvolvido ao longo desta tese.

2.3.4 Geração do Código Máquina

Para concluir esta já longa descrição do processo de compilação, falta apenas apresentar a geração do código máquina (GCM), que pode ser feita a partir da própria representação intermédia ou então a partir da respectiva linguagem *assembly*. Em ambos os casos, acrescenta ao processo de compilação toda uma nova fase responsável pela gestão dos blocos de código e de dados, e pela conversão das variáveis e das *labels* para endereços físicos.

Esta fase é assim responsável por alocar os blocos de memória, ou ficheiros, onde colocar as instruções a gerar, o que normalmente é feito para cada função ou rotina do programa. Em seguida, deve relacionar cada variável da tabela de identificadores com um endereço físico situado num dos blocos que foram previamente alocados para os dados. É vulgar determinar o endereço em relação à posição inicial do bloco e de tal forma que o espaço ocupado pela respectiva variável não sobreponha o espaço de outras variáveis. Numa etapa posterior e conforme se geram as instruções em código binário, substitui-se as variáveis e outros identificadores pelo respectivo valor do endereço.

Outro dos problemas a resolver pela fase de geração do código binário, é a gestão das *labels*, ou mais genericamente, das referências às posições do próprio código. De notar que a este nível de representação as estruturas de controlo são normalmente bastante simples e restringem-se quase sempre a saltos incondicionais ou condicionais. Desta forma os problemas a resolver encaixam-se normalmente numa das duas seguintes situações: saltos para posições anteriores ou saltos para posições posteriores à actual.

A primeira situação pode ser facilmente resolvida se a representação intermédia, de alguma forma identificar as posições onde devem existir *labels*, permitindo assim que o gerador de código relacione cada uma com a posição da instrução a ela associada. De tal

forma que qualquer referência posterior a uma *label* seja substituída pelo endereço da instrução.

O segundo caso é ligeiramente mais complexo, uma vez que as instruções referem *labels* que ainda não foram identificadas, pelo que não podem ser imediatamente e totalmente resolvidas. A solução passa então por gerar o código da instrução, designadamente o espaço a ocupar pelo endereço da *label*, sem que no entanto este contenha qualquer tipo de informação útil. Depois, é necessário manter uma listas com este tipo de instruções (inacabadas) para que logo que o endereço da respectiva *label* seja conhecido possam ser concluídas.

É ainda vulgar existir uma fase, após todo o processo de geração, responsável por optimizar o código final, tirando partido de situações que só a este nível é que são detectadas.

Por falta de tempo não foi possível incluir a geração do código máquina, no trabalho desenvolvido ao longo desta tese de mestrado. Apresenta-se no entanto no capítulo 9 mais alguns detalhes relacionados com esta fase e uma ferramenta desenvolvida com objectivo de auxiliar a sua construção.

3 BEDS – Sistema de Apoio ao Desenvolvimento de *Back-Ends*

Uma vez apresentadas as fases mais importantes de um compilador e alguns dos detalhes da sua implementação, é altura de entrar no tema de dissertação desta tese.

O objectivo foi desde o início estudar, e se possível desenvolver, um sistema de geração de *back-ends*, ou seja, pretendia-se estudar as soluções existentes, ver como funcionam e a partir disso tentar apresentar uma solução capaz de auxiliar o processo de desenvolvimento desta parte do compilador.

Com o estudo que se realizou, concluiu-se que a maior parte das ferramentas que, de alguma forma permitem gerar soluções portáveis, resolvem apenas parcialmente o problema da geração dos *back-ends*, ou seja, apenas produzem código para resolver determinadas fases. Com a desvantagem de que cada ferramenta possui as suas próprias restrições.

Mais ainda, todas as ferramentas partiam do princípio de que se utilizava uma determinada forma de representação intermédia, situação que se mostrou incontornável. E a grande maioria das ferramentas, ou só tratava da selecção de instruções, sem a parte da alocação, ou o contrário.

Existem no entanto algumas soluções que são excepção, pelo menos em relação a alguns dos pontos atrás assinalados, é o caso do GCC - *GNU C Compiler*, do *RTL System* e do BEG – *Back-End Generator*.

O primeiro é uma referência a todos os níveis de um compilador portável, quer ao nível do *back-end*, quer ao nível do *front-end*. Trabalha com base na representação intermédia referida no capítulo anterior, o *Register Transfer Language*, sendo como tal um dos descendentes do programa *PO* desenvolvido por Jack Davidson e Christopher Fraser [DF80].

Uma das suas componentes, o *Machine Description*, permite adaptar o compilador a novas arquitecturas, especificando praticamente tudo o que é necessário à obtenção de um *back-end*, desde os modos de endereçamento, à relação entre as expressões da RTL e as instruções máquina, ou até descrever optimizações, que tenham por base a substituição de expressões por outras.

Talvez por se tratar de um sistema construído de forma a permitir uma fácil modularização, quer do ponto de vista da linguagem fonte, quer do ponto de vista do código final, e por suportar praticamente todo o tipo de optimizações, o que levou certamente a muitas soluções de compromisso, faz com que seja um tanto ou quanto "pesado". No entanto e comprovando as potencialidades do modelo de compilador utilizado pela GNU, não só surgiram adaptações deste a outras linguagens fonte, como a um vasto conjunto de processadores.

O segundo projecto, conforme o descrito no capítulo anterior, não se trata de um compilador propriamente dito, mas sim um sistema que tenta fornecer todo o suporte necessário ao processo de compilação a partir da representação intermédia. Possui como tal um sistema de geração de código, o qual é muito semelhante ao utilizado pelo GCC, divergindo apenas na forma com que se descreve a arquitectura do processador, a qual é feita com a própria linguagem com a qual se desenvolveu o *RTL System*, o *SmallTalk*.

Como o nome indica e à semelhança do GCC, este sistema também utiliza a RTL como forma de representação intermédia, sendo como tal mais um descendente do trabalho desenvolvido por Jack Davidson e Christopher Fraser.

Convém ainda acrescentar que este segundo projecto é um exemplo do que se pretende obter como resultado do trabalho desta tese, uma vez que permite realizar praticamente tudo o que é pedido e necessário ao desenvolvimento de um *back-end*.

No terceiro projecto, a solução desenvolvida é bastante diferente das anteriores e tal deve-se ao facto do BEG não ser um compilador, mas um sistema gerador de algumas das fases de compilação (selecção de instruções e alocação de registos).

Julgo que qualquer comparação entre estes três sistemas será de alguma forma inadequada, uma vez todos eles foram desenvolvidos com objectivos diferentes, o que no caso do GCC, terá sido certamente garantir a qualidade do compilador na tarefa de geração de código, sacrificando sempre que tal fosse necessário a portabilidade. No caso do *RTL System*, julgo que o objectivo foi de certa forma levar mais adiante o trabalho já desenvolvido no PO mantendo os objectivos deste último, ou seja, desenvolver um optimizador portável. Já o BEG surge num contexto bastante diferente dos dois anteriores, onde o objectivo foi construir a partir de uma forma de reconhecimento específica, uma solução que permitisse desenvolver apenas as partes de um compilador que são intrinsecamente dependentes da arquitectura do processador.

É no entanto relevante para o tema desta tese destacar o tipo de solução adoptada por cada um dos sistemas para a selecção das instruções. É que nos dois primeiros casos o processo de selecção é do tipo *top-down*, quer sobre a forma de algoritmos de reconhecimento de *strings* ou do tipo *table driven*, que como já se referiu no capítulo 2 e como se poderá ver no capítulo 7, estão longe de ser soluções eficientes, pois nem sempre garantem a escolha da melhor instrução. No entanto, este tipo de soluções funciona bastante bem, graças aos processos de optimização que abundam no GCC e no *RTL System*, os quais permitem de certa forma um pré-processamento das expressões de forma a garantir a eficiência da fase de selecção.

Já o último sistema, para realizar a selecção de instruções recorre a um processo de *parsing*, do tipo *bottom-up*, que em conjunto com algumas técnicas de programação dinâmica, permite para uma dada expressão encontrar o conjunto de instruções que perfazem a solução óptima. Outras soluções há, em que nem sequer é necessário utilizar programação dinâmica para obter a solução óptima, como é o caso do *Bottom-Up Rewrite System* – BURS.

Pelo que uma diferença fundamental entre estes sistemas, de um lado o GCC e o RTLS, e do outro, o BEG e o BURS, está em delegar mais ou menos responsabilidades

à fase de selecção. Assim, os primeiros, com base em desenvolvidas técnicas de optimização garantem um processo de selecção eficiente; enquanto os segundos, partem do princípio que a escolha certa das instruções a utilizar é pura e simplesmente da responsabilidade do selector, pelo que em parte dispensam as optimizações.

Mas a simples questão de conterem ou não optimizações, levanta problemas relacionadas com a forma de representação, por exemplo, quer o BEG, quer o BURS, contentam-se com simples árvores de expressões (binárias), mas já o RTLS ou o GCC, necessitam de formas bem mais elaboradas.

De notar no entanto, que qualquer uma das soluções emprega técnicas de *parsing* para fazer a selecção, o que força à utilização de árvores de expressões como representação intermédia.

O RTLS ultrapassa este tipo de problema mantendo em simultâneo a representação em forma de árvores, com uma forma de representação sequencial, mas tudo isto graças a ter sido desenvolvido sobre uma linguagem orientada ao objecto e mesmo assim, com algumas dificuldades em relacionar as *RTLExpressions* (árvores) e os *RegisterTransfers* (representação sequencial).

Mas outros casos há, em que não é fácil manter uma representação só com base em árvores ou então, converter a representação utilizada para árvores, de forma a conseguir utilizar técnicas de *parsing* na selecção. Infelizmente não foi possível apurar qual a situação concreta do GCC, mas casos há em que contornam o problema fazendo a selecção com algoritmos de reconhecimento em *strings*, bastante menos eficientes.

Começa-se assim a notar que um dos aspectos fundamentais na concepção de um compilador, está na forma da representação intermédia escolhida, a qual vai certamente condicionar as restantes opções. Assim, de um lado temos compiladores com potentes mecanismos de análise e de optimização, e implicitamente, elaboradas formas de representação intermédia, que reduzem ao mínimo a intervenção de fases como a selecção ou mesmo a alocação (*assigner*); e do outro, os compiladores que ao dispensarem praticamente todas as formas de optimização, permitem trabalhar com representações mais simples, necessitando no entanto de melhores mecanismos de selecção e de alocação.

Como já se referiu, um dos objectivos do trabalho desenvolvido no contexto desta tese, é apresentar soluções que facilitem o desenvolvimento de compiladores, para tal e considerando os exemplos aqui apresentados, coloca-se a primeira grande questão: que tipo de compilador implementar e, implicitamente, que representação intermédia utilizar?

Se apenas se pretende um compilador pequeno, facilmente adaptável a novas arquitecturas e simples, então a solução será certamente implementar apenas as fases indispensáveis do compilador (análise léxica, sintáctica, semântica, selecção de instruções e alocação). Para este tipo de compilador o processo de desenvolvimento fornecido por sistemas como o BEG, é plenamente satisfatório.

Se no entanto, se pretende desenvolver um "grande compilador", capaz de realizar a mais eficiente das compilações, então não chega implementar as fases essenciais, há que acrescentar muitas mais e como tal trabalhar com modelos mais elaborados, como é o caso do GCC e do *RTL System*.

Após o estudo de ambas as alternativas e de analisar os prós e contras, o melhor que se pode pedir é uma solução que reúna o melhor de ambas. É exactamente com esse intuito que se desenvolveu um modelo que pretende integrar os dois tipos de soluções, compensando as falhas de uma com as capacidades da outra.

3.1 *O Back-End Development System*

A solução a apresentar, deve ser facilmente adaptável a novos processadores, assim como garantir as rotinas e respectivas infra-estruturas para os processos de optimização e alocação global. Mais ainda, pretende-se que estas rotinas sejam de alguma forma independentes, quer da linguagem fonte, quer do processador para o qual se pretende gerar o código, de forma a maximizar a sua reutilização no desenvolvimento de novos compiladores.

É como tal essencial garantir uma fase no processo de compilação totalmente independente das características do *front* e *back-end* do compilador e que suporte o acréscimo de infra-estruturas de apoio aos processos de optimização, ou seja, tem de permitir conter de forma acessível a informação sobre a estrutura do programa, da relação entre as variáveis, das operações, dos modos de endereçamento, etc, sem no entanto comprometer o processo de selecção do conjunto de óptimo de instruções, como o utilizado no BEG ou BURS.

De notar que, com estes objectivos, o compilador deixa de ser composto por apenas duas partes, o *front* e o *back-end*, e passa a possuir uma terceira, intermédia às duas anteriores, a qual é independente, quer das características da linguagem fonte, quer das características do processador para o qual se pretende gerar o código. Esta fase será a partir daqui designada por *middle-end*.

Para se obter um modelo com estas características, começou-se por estudar detalhadamente a implementação dos sistemas que tinham por base a RTL. Após esta fase analisou-se as necessidades dos sistemas de selecção com base em travessias *bottom-up* e por fim tentou-se integrar esta forma de selecção na RTL.

Desta forma desenvolveu-se toda uma infra-estrutura, cujas bases partiram por um lado pelo trabalho já desenvolvido pelos adeptos da RTL, nomeadamente por C. McConnell, J. D. Roberts e C. Schoening [MRS90, MRS??A] e o seu *RTL System*, e por outro lado pelos adeptos do BURS, tal como E. Pelegri-Llopart, C. Fraser e T. Proebsting. Destes últimos, destaca-se o trabalho desenvolvido por C. Fraser, no Light C Compiler (LCC) e as implementações da teoria do BURS feitas por T. Proebsting, o BURG e o IBURG. É esta última implementação que serve como ponto de partida para o sistema de geração de selectores de instruções no modelo a propor.

Infelizmente não foi possível estudar, nem o BEG, nem o GCC, pois a informação sobre a forma de implementação de ambos os sistemas é escassa e no caso do BEG nem sequer é disponibilizado os ficheiros fonte do sistema.

O esquema com todas as fases do modelo a propor para um sistema de desenvolvimento de *back-ends*, encontra-se representado na Fig. 3.1, o qual é composto por quatro tipos distintos de entidades, os módulos, as rotinas, as especificações e a representação. Os módulos representam as componentes a utilizar pelos *back-ends*, as rotinas são partes específicas dos módulos com funções muito próprias, as especificações são descrições a partir das quais o BEDS gera alguns dos módulos ou rotinas, e a representação como o próprio nome indica serve para representar as diversas fases do código a compilar.

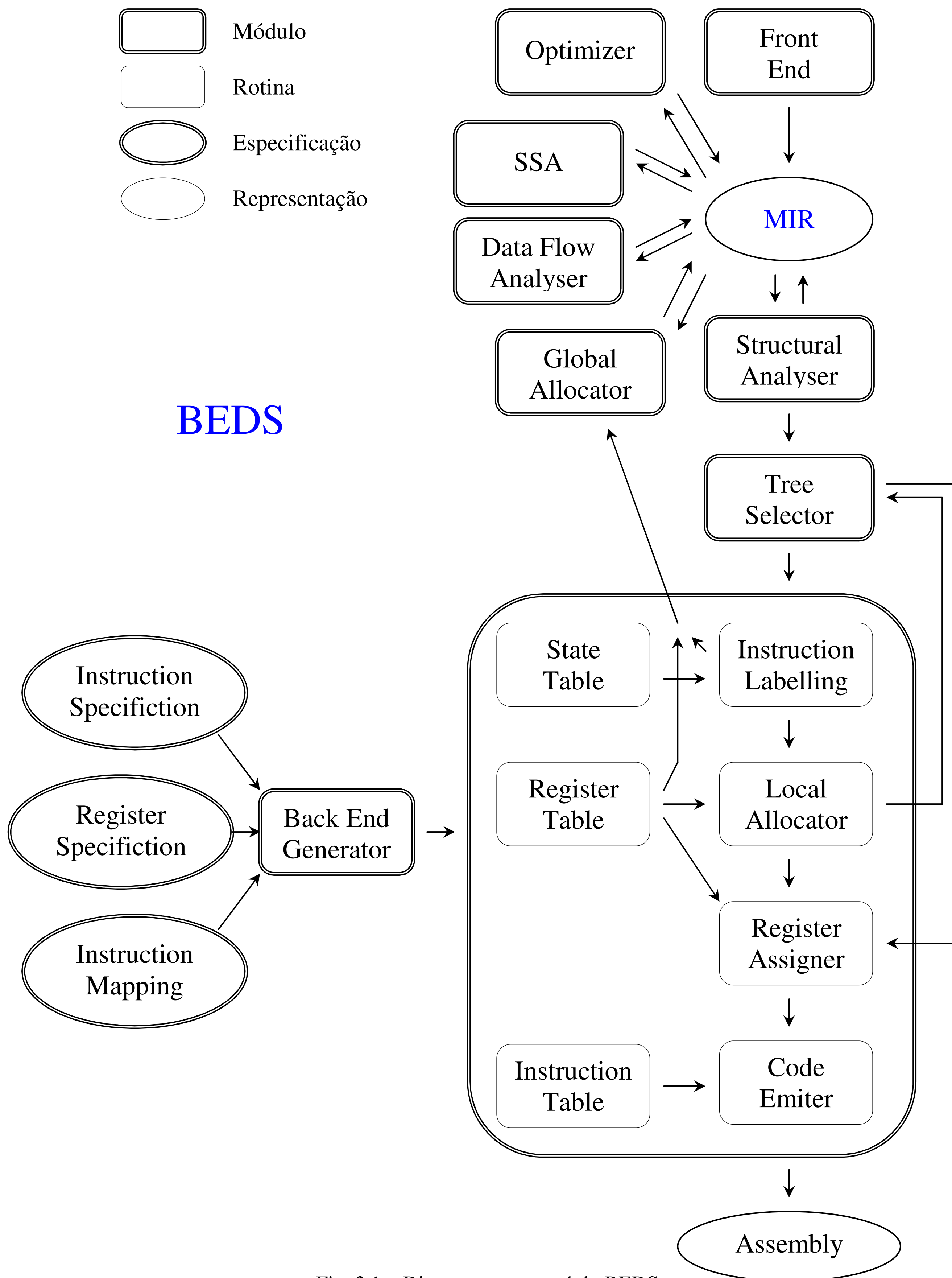


Fig. 3.1 – Diagrama estrutural do BEDS

De forma resumida o modelo divide-se em duas grandes partes, uma que funciona com base numa biblioteca, que fornece uma série de rotinas fortemente relacionadas com a representação intermédia, das quais fazem parte as rotinas para análise de fluxo de controlo e dados, optimização e alocação global, que serão explicadas nos capítulos 5 e 6; e uma por segunda parte, composta por um sistema de geração e modelação, que a partir de uma especificação com a informação referente ao processador e da relação desta informação com as expressões da representação intermédia, permite modelar determinadas rotinas e gerar alguns dos mecanismos necessários às tarefas de um *back-end*, como é o caso do selector de instruções, o sistema de atribuição de registos, ou o gerador de código final, que são abordados nos capítulos 7 e 8.

A implementação do modelo designou-se por BEDS – *Back-End Development System* e à representação intermédia utilizada por este deu-se o nome de MIR – *My Intermediate Representation*, a qual segue uma filosofia muito semelhante à da RTL, mas devidamente adaptada às necessidades das várias fases deste modelo.

A implementação de todo o BEDS incluindo a MIR, foi feita em C++, obedecendo ao paradigma da programação por objectos. Pelo que a biblioteca disponibilizada por este não é mais do que um conjunto de classes, sendo a representação intermédia feita por instanciação dos objectos dessas classes.

Desta forma o modelo obriga que o *front-end* transforme o código fonte numa representação MIR. Depois, recorrendo às potencialidades fornecidas pela biblioteca e pelo sistema gerador que compõem o BEDS, deverá ser possível desenvolver todas as fases do *back-end* de um compilador.

Apesar dos processos de optimização não pertencerem às funções de um *back-end*, o BEDS toma a responsabilidade de garantir as rotinas de optimização, ou pelo menos os recursos necessários à sua implementação, pois como já se viu é bastante vantajoso que estas rotinas sejam desenvolvidas sobre a representação intermédia, visto esta ser independente da linguagem fonte e das características do processador, permitindo assim a sua reutilização.

Para além disso, como de certa forma a representação intermédia é imposta, é certamente agradável para quem pretende utilizar este tipo de sistemas, que encontre implementado o maior número possível de mecanismos auxiliares; além de que, grande parte desses mecanismos são necessários às fases obrigatórias do *back-end*, como é o caso da alocação global.

Sobre a MIR trabalham essencialmente três tipos de rotinas, as relacionadas com os processos de optimização, as relacionadas com a alocação global e as relacionadas com a geração do código. Qualquer destas três rotinas serão analisadas com o devido detalhe respectivamente nos capítulos 5, 6 e 7. Fica no entanto feita uma pequena descrição de cada um dos módulos e de cada uma das rotinas.

Optimizer

O *Optimizer* representa uma classe com todas as funções de optimização. É um dos módulos do BEDS ao qual é sempre possível acrescentar mais alguma coisa, pelo que apenas se encontra implementado a título de exemplo.

As rotinas de optimização trabalham directamente sobre a MIR, necessitando por vezes das estruturas de informação fornecidas pelo SSA, *Data Flow Analyser*, *Control Flow Analyser (Structural Analyser)*, *Data Dependency Analyser* e o *Alias Analyser*. Estes dois últimos módulos ainda não se encontram implementados, pelo que todas as rotinas de optimização que deles necessitam estão por desenvolver.

Single Static Assignment

Este é um dos módulos inerentes à própria representação intermédia, pois como adiante se verá, a MIR não só tem por base a RTL como o *Single Static Assignment* (SSA), que é uma forma de realçar as dependências entre dados.

Destina-se a auxiliar o *front-end* a construir a própria representação MIR, convertendo a representação para SSA. Para além disso, disponibiliza rotinas para outros módulos, como é o caso da análise do fluxo de dados.

Data Flow Analyser

Este é um dos módulos de suporte a muitas das optimizações, mas também ao processo de alocação global. A sua função é criar estruturas de informação que permitam analisar o fluxo de dados, como por exemplo informar as posições onde uma dada variável se encontra "viva".

Global Allocator

Este módulo implementa a alocação global, com base no algoritmo de coloração de grafos proposto por Chaitin [CACCHM81, Chait82]. Trata-se de uma fase do processo de compilação que pode ter grande impacto na qualidade do código final, mas há que ponderar a sua utilização, uma vez que requer muito processamento. Recorre de forma directa ao *Data Flow Analyser*, ao SSA, ao *Instruction Labelling* e à *Register Table*. Mas de forma indirecta envolve praticamente todas as restantes fases.

Structural Analyser

A MIR é composta essencialmente por um grafo com muita informação associada, o qual pode assumir alguma complexidade, tornando a sua interpretação directa algo inacessível. Há como tal que, criar mecanismos que permitam uma análise detalhada da forma como o grafo está estruturado. O que normalmente se consegue através da identificação de padrões típicos, que formam partes do grafo, e da interligação entre eles. Este tipo de informação é essencial para determinados processos, como optimizações do fluxo de controlo ou para a "linearização" do grafo.

Esta análise pode ser obtida de diversas formas, mas no caso do BEDS utilizou-se uma abordagem designada por *Structural Analyse*, que tem por base os mecanismos de parsing dos analisadores sintácticos, adaptados à utilização sobre grafos.

Poder-se-á questionar a razão de ser deste módulo, uma vez que a informação por ele obtida, já é determinada através da análise sintáctica. No entanto é preciso ter presente que é necessário construir a representação intermédia de forma a poder utilizar os recursos fornecidos pelo BEDS, os quais por sua vez podem alterar a própria representação (mantendo o valor semântico). Pelo que as estruturas que originalmente existem no código fonte podem não existir na representação intermédia.

Tree Selector

Este módulo permite pegar na árvore de controlo do módulo anterior e obter uma lista de árvores de expressões, as quais serão posteriormente requisitadas, uma a uma, pelo *Instruction Labelling*.

Back-End **Generator**

O *Back-End Generator* é o sistema gerador que, a partir da especificação dos registos, das instruções do processador e da relação destas com a representação intermédia, permite gerar o selector de instruções e respectiva tabela de estados, a tabela dos registos, um sistema de alocação local, o *assigner* e as rotinas para emissão do código final.

Esta parte do BEDS pode funcionar independentemente do resto, aliás a ligação entre este módulo e as componentes que ela gera, com os restantes obtém-se como opção do sistema de geração e não por defeito.

Instruction Labelling

Esta é uma das rotinas do processo de compilação, gerada pelo módulo do *Back-End Generator*, a qual é parte essencial dos sistemas de selecção com base no BURS. A sua função é determinar o conjunto de padrões (instruções) que permitem cobrir cada uma das árvores ao menor custo, tendo por base o *State Table*. Na prática é nesta rotina que se faz a selecção das instruções, estando no entanto, o resultado obtido sujeito às restrições do processo de alocação.

Local Allocator

Trata-se de uma rotina também gerada pelo módulo do *Back-End Generator*, que é opcional e que permite alocar registos localmente, ou seja, ao nível de cada árvore de expressões.

Register Assigner

As rotinas de alocação, quer sejam locais ou globais, são responsáveis por determinar que operandos é que são candidatos a registos e que registos é que podem ser utilizados por cada operando. Cabe ao *Register Assigner* verificar de entre esses registos, quais os que se encontram livres, ou eventualmente ocupados, mas cujo conteúdo não seja mais necessário, e depois, destes, seleccionar e atribuir os necessários a cada um dos operandos.

Pode ainda acontecer, que do conjunto dos registos que um operando pode utilizar, nenhum se encontre disponível, nesta situação o *Register Assigner* deve avisar o sistema de alocação do sucedido, para que este opte por outro tipo de soluções, ou então ele próprio deve decidir o que fazer. Em ambas as situações, a solução poderá passar por realizar o *splitting* ou *spilling* de um dos registos, de forma a concluir a geração e a alocação com êxito.

Code Emiter

Após a selecção definitiva das instruções e do processo de alocação, cabe ao *Code Emiter* gerar o código final, que na actual situação do BEDS, resulta em código *assembly*. Pretende-se posteriormente criar a possibilidade de gerar o próprio gerador de código binário, com base na descrição das instruções e dos campos que as compõem.

State Table, Register Table e Instruction Table

Estas três rotinas não são mais do que estruturas de dados com a informação dos estados possíveis de se encontrar durante a selecção, dos registos do processador agrupados por conjuntos e do conjunto de instruções do processador.

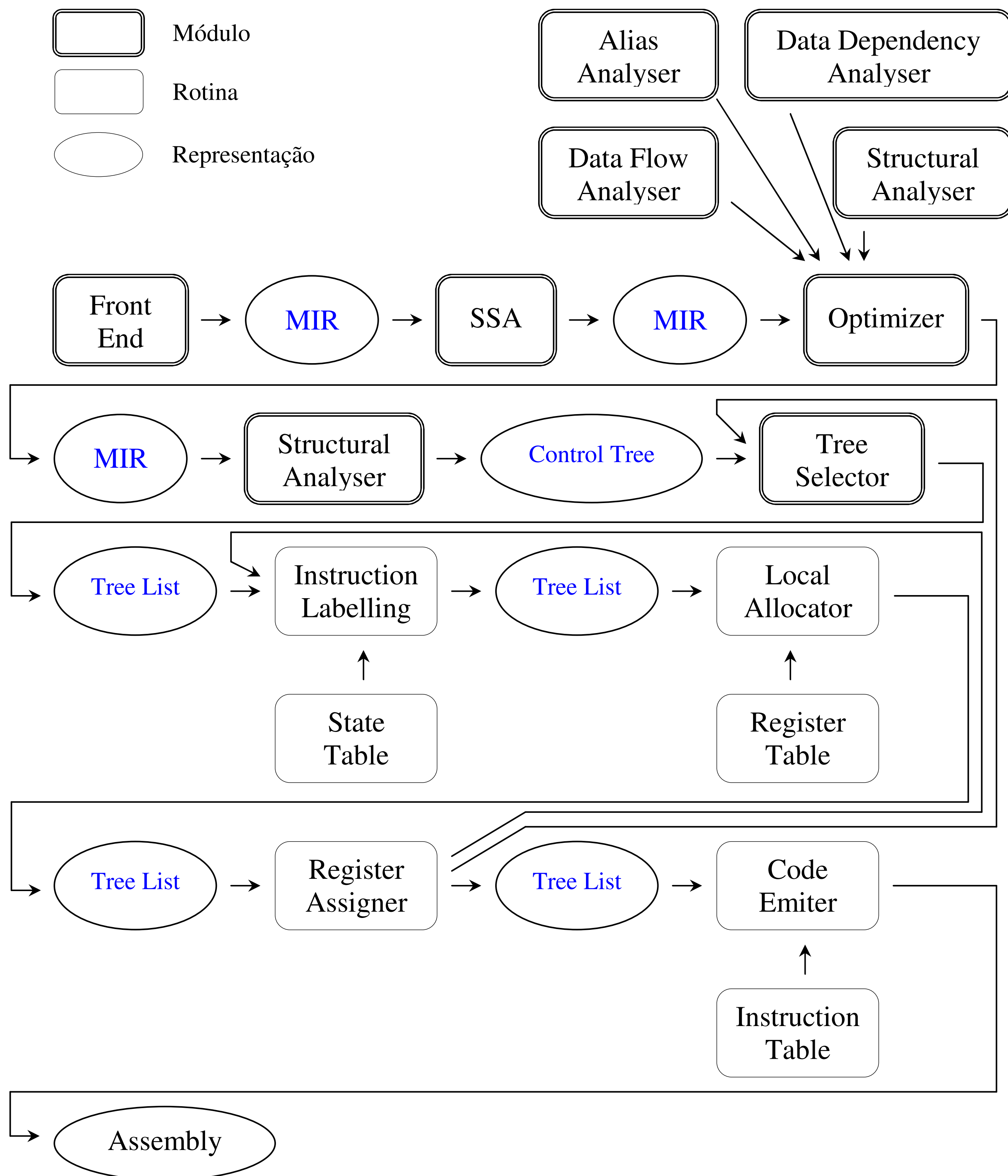


Fig. 3.2 – Modelo de compilador com alocação local.

No caso concreto do *State Table*, é possível substituir as estruturas estáticas por programação dinâmica, situação que foi explorada até ao limite no desenvolvimento do BEDS.

Nas duas próximas secções explica-se sucintamente como é que através do BEDS se pode conceber compiladores com sistemas de alocação local e global.

3.2 *Compilador com alocação local*

Para cada tipo de alocação impõe-se um modelo próprio de *back-end*, uma vez que o tipo de interligação entre as diversas fases é distinto. Para o caso da alocação local o modelo encontra-se representado na Fig. 3.2.

Neste modelo o BEDS gera um sistema de alocação do tipo *On-The-Fly Optimizado*, ou seja, a alocação é feita localmente para cada árvore de expressões, sendo os registos atribuídos conforme vão sendo necessários. Trata-se de um modelo optimizado, uma vez que, entre outras coisas, permite a reutilização de registos numa mesma árvore. Este processo de alocação será apresentado posteriormente com maior detalhe.

Uma vez gerado o código segundo a representação MIR, com auxílio do SSA, processam-se as optimizações ao nível da representação intermédia. Estas podem recorrer à análise do fluxo de controlo e de dados, à análise de dependência entre dados ou à análise de *aliases*. A única limitação imposta aos processos de optimização é de que estes não podem utilizar abordagens destrutivas sobre a MIR, ou seja, as alterações feitas devem continuar a garantir uma representação MIR.

Após esta fase, procede-se à análise estrutural para se construir a árvore de controlo, que cria uma estrutura de informação suplementar de forma a não alterar a MIR, o que é irrelevante para o presente caso, mas fundamental para o caso da alocação global.

A árvore de controlo é percorrida na fase do *Tree Selector*, de baixo para cima e da esquerda para a direita, processando cada folha de forma a criar uma lista de árvores de expressões.

O *Instruction Labelling* vai requisitando árvore a árvore ao *Tree Selector*, realizando para cada uma o *labelling*, de forma a determinar o conjunto de instruções que permitem criar o respectivo código ao menor custo possível. Após isso a árvore é transmitida ao *Local Allocator*, o qual é responsável por implementar uma política de atribuição para os registos. Cabe ao *Register Assigner* realizar a atribuição propriamente dita.

O *Local Allocator* liberta os registos que não se encontrem a ser utilizados. Este processo faz-se ao nível da própria árvore, ou seja, por defeito são libertados todos os registos ocupados pelos descendentes de um nodo, excepto o que é utilizado para manter o resultado do próprio nodo e os registos que guardam os resultados de sub-árvores de utilização comuns (estes só são libertados após a última utilização).

É com o trabalho conjunto destas três fases, o *Instruction Labelling*, o *Local Allocator* e o *Register Assigner*, que é possível garantir a plena geração de código. Tal só não acontece, se não for possível realizar o *labelling* (independentemente do custo), situação que só deverá ocorrer por falha da especificação.

Mas para que a geração de código final seja bem sucedida, é necessária uma forte integração entre as componentes envolvidas. Cabe no entanto ao *Register Assignment* determinar se é ou não possível concluir o processo de geração com sucesso, caso tal não o seja, deverá optar pelas seguintes alternativas: em primeiro lugar, tentar realizar o *splitting* da variável associada ao nodo da árvore de expressões para o qual não foi possível realizar a alocação, aumentando o custo da solução até aí proposta, na esperança que o *Instruction Labelling*, numa nova travessia, encontre uma outra solução

(que certamente terá um custo igual ou superior à solução anterior). Se mesmo assim não for possível gerar o código final, quer por não se conseguir realizar o *labelling* da árvore, quer por ainda assim, não ser possível realizar a alocação, então opta-se por uma solução de *spilling*. Neste caso, decompõem-se a árvore em duas (ou mais), de forma a que o resultado da sub-árvore para a qual não foi possível realizar a alocação, fique guardado em memória. A árvore original é assim reescrita de forma a utilizar o endereço dessa posição de memória invés da sub-árvore do qual este resulta.

Infelizmente a implementação do processo de *spilling* não é tão simples como se descreve, pelo menos no caso de se pretender obter uma solução que não comprometa a portabilidade do sistema. Além de mais nem sempre é fácil determinar o ponto pelo qual se deve dividir a árvore original. Os detalhes da implementação serão discutidos posteriormente no capítulo 6.

Para concluir falta o *Code Emiter*, que pega nos padrões das instruções e substitui os operandos pelos respectivos valores, quer estes sejam registos, endereços ou constantes e imprime para um ficheiro, gerando assim o código em *assembly*.

3.3 *Compilador com alocação global*

O BEDS permite também desenvolver compiladores com alocação global, estruturados segundo o modelo da Fig. 3.3.

Para tal e como já foi referido, recorre ao algoritmo de coloração de grafos proposto por Chaitin, o qual é descrito detalhadamente no capítulo 6. Nesta secção pretende-se apenas descrever o modelo do compilador e o funcionamento do *back-end*, para este tipo de alocação.

À semelhança da alocação local, também aqui a MIR é obtida com auxílio do SSA, onde em seguida se executam as optimizações, constrói-se a árvore de controlo, obtém-se a lista de expressões e faz-se o *labelling* das árvores. É a partir deste ponto que o funcionamento deste modelo e o modelo com alocação local difere.

A operação de *labelling* neste modelo não serve apenas para seleccionar as instruções, mas também para determinar os operandos candidatos à utilização de registos. É com essa informação que o sistema de alocação distribui os registos, tentando minimizar as transferências de dados, tal como as operações de *load* e *store* e as transferências entre registos.

Para além disso detecta se existem registos suficientes para realizar a operação de alocação. Caso tal não aconteça é função do sistema de alocação acrescentar as operações de *spilling* necessárias a uma correcta geração do código.

Repare-se que a obtenção da árvore de controlo e da lista de árvores para a realização das operações de *labelling*, não destrói a representação MIR, permitindo que esta seja reutilizada na alocação e implicitamente na análise de fluxo de dados.

Após esta fase é necessário obter uma nova lista com as árvores de expressões de forma a voltar a realizar-se o *labelling*. Não é no entanto necessário voltar a construir a árvore de controlo, uma vez que o processo de alocação, de princípio, não deverá alterar a estrutura do programa a compilar.

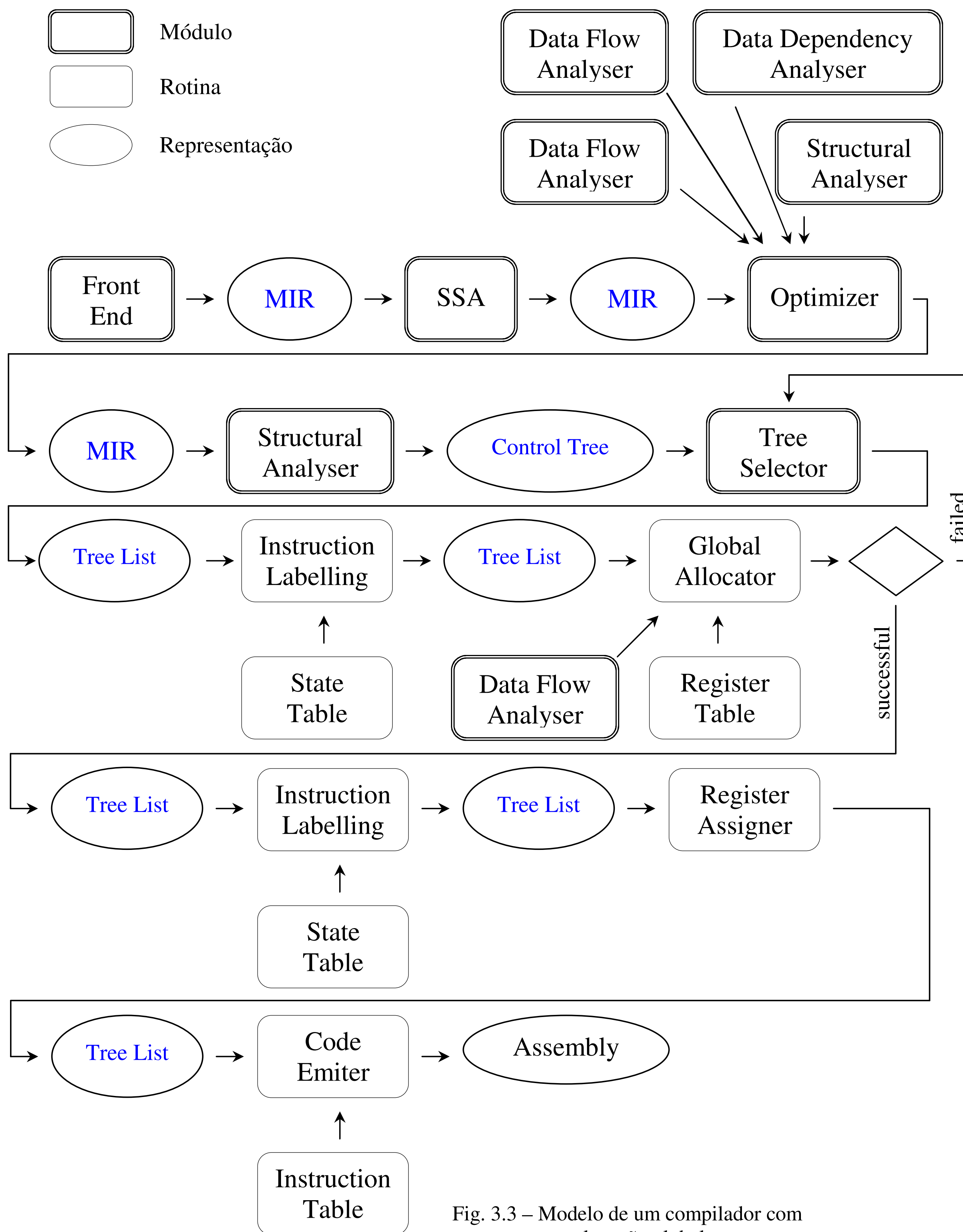


Fig. 3.3 – Modelo de um compilador com alocação global.

A nova operação de *labelling*, não deverá trazer grandes alterações, mas apenas acrescentar e actualizar as situações directamente envolvidas com as operações de *spilling*. Se tal não acontecer, existem duas alternativas: tentar atribuir os registos na esperança de se obter o código final, sem que para tal exista qualquer vantagem resultante do processo de alocação e correndo o risco de não ser possível realizar correctamente o *assigner*; ou então voltar a realizar o *labelling*, tantas vezes quantas as necessárias, e até se obter uma solução estável entre a fase da alocação e a fase de *labelling*. Só após se obter tal solução é que se deve atribuir os registos, concluindo depois a compilação com a passagem das árvores para o *Code Emiter*.

Com esta descrição conclui-se a apresentação da estrutura de um compilador, com alocação local e global, seguindo o modelo de concepção do BEDS.

4 My Intermediate Representation

Neste capítulo, pretende-se apresentar a MIR – My Intermediate Representation, explicando na medida do possível a razão de ser das suas diversas componentes e da relação destas com o *front-end* e com as restantes fases do BEDS. Em conjunto apresenta-se a noção de *Single Static Assignment* uma vez que também faz parte da estrutura base da MIR. Tentar-se-á ainda demonstrar as vantagens da sua utilização como forma de representação.

4.1 *Estruturação da MIR*

A especificação da representação intermédia, aqui designada por MIR, teve por base a especificação do *RTL System*. No entanto, conforme se foi desenvolvendo a implementação do BEDS, foi necessário modificar a especificação original. Tal não se deve a nenhuma incorrecção por parte do *RTL System*, mas sim ao facto da MIR integrar outros mecanismos diferentes dos utilizados neste sistema. Para além disso e apesar de ambas as representações partilharem muitas das características, a MIR sofreu alterações implícitas à própria estrutura do BEDS. Conceptualmente ambas representações partilham os conceitos base da RTL e do SSA.

Muitas das características da MIR estão associadas, não à representação intermédia, mas sim a imposições colocadas pelos vários módulos que compõem o BEDS. Há como tal que distinguir o que é específico da MIR, do que não é, como por exemplo, as estruturas associadas à MIR que permitem a análise dos dados. Dentro do possível apenas interessa focar neste capítulo as partes específicas da MIR, uma vez que as restantes serão apresentadas nos respectivos capítulos.

Assim sendo pode-se classificar as estruturas fornecidas pela MIR em quatro classes, a classe *Genérica*, com as infra-estruturas de apoio à representação dos programas, a classe *Expressions* que permite descrever as expressões da representação intermédia, a classe *DataTransfer* que permite descrever todo o tipo de transferências entre dados e a classe *FlowNode*, com os objectos que descrevem a organização do programa ao nível das estruturas de controlo.

Antes de prosseguir com a descrição de cada uma destas classes, e como se disse acima, vai-se previamente apresentar a noção de SSA.

4.2 *Single Static Assignment*

O *Single Static Assignment* é uma das formas de representação utilizados na concepção da MIR. A sua influência é tão forte que modifica por completo o processo de representação, pelo que antes de se descrever as classes da MIR, há que fazer a apresentação formal do SSA.

Desde há muito que se conhece as vantagens e desvantagens da utilização do SSA. Uma das desvantagens residia no próprio processo de determinar esta forma de representação. Tal facto restringiu a sua utilização até 1991, ano em que Ron Cytron em conjunto com outros investigadores, propõe uma forma eficiente de o conseguir. Desde então foram publicados diversos artigos demonstrando as vantagens da sua utilização nos processos de optimização. Tal advém das características implícitas que esta forma de representação possui, que permitem realçar as dependências entre dados.

Para tal há que garantir que cada variável possui uma e uma só atribuição, ou seja, as expressões devem ser redefinidas de forma a que cada atribuição deixe de ser feita a uma variável concreta e passe a ser feita a uma instância dessa variável. Instância essa que para todos os efeitos passa a ser considerada como uma variável independente.

O seguinte exemplo ilustra este tipo de representação, apresentando um bloco de código na sua representação normal e na respectiva representação em SSA.

dt_1	→	a = 0	$a_1 = 0$
dt_2	→	b = a + 1	$b_1 = a_1 + 1$
dt_3	→	a = 2	$a_2 = 2$
dt_4	→	c = a + b	$c_1 = a_2 + b_1$

Como só é permitido uma única atribuição por variável, torna-se muito fácil determinar quem é que utiliza o resultado proveniente de uma dada expressão ou quem por sua vez é responsável pelos resultados por esta utilizados. Esta informação é essencial para maioria dos processos de optimização.

Para além disso, o SSA permite implicitamente remover dependências do tipo *anti-dependence* e *output-dependence*. Por exemplo, na representação normal do exemplo anterior, existe uma anti-dependência entre dt_3 e dt_2, uma vez que dt_3 redefine o valor de uma variável utilizada por dt_2. Tal já não acontece na forma SSA uma vez que a_1 e a_2 são para todos os efeitos variáveis distintas.

O mesmo acontece entre dt_1 e dt_3, onde na representação normal existe uma *output-dependence* entre estas duas expressões, o que já não ocorre para a representação SSA.

A representação SSA é obtida, decompondo cada variável em tantas quantas as suas definições (atribuições), de tal forma que cada uma represente um instante da vida da variável original. Mas para que tal seja sempre verdade há que contornar determinadas situações, das quais fazem parte os dois seguintes casos:

```
i ← 0                    i ← 0
while ....               if ... then
        ...                      ...
```

```
                i ← i + 1                   i ← 10
                ...                         ...
fimwhile                            else
                                            ...
                                            i ← 20
                                            ...
                                    fimse
                                    ...
                                    j ← i
```

No primeiro exemplo, a variável *i* possui uma atribuição que é realizada dentro de um ciclo. O problema que se levanta é como fazer com que cada atribuição, que ocorre dentro do ciclo, corresponda a uma nova instância da variável original.

No segundo exemplo, a segunda e terceira atribuição da variável *i*, após a transformação SSA, passam a ser feitas a variáveis distintas (ex: i_2 e i_3). O que levanta a seguinte questão: como determinar qual a variável a utilizar na atribuição feita a *j* ?

Ambas as questões se resolvem através da introdução das funções Φ(...), da seguinte forma:

```
i1 ← 0                              i1 ← 0
while ....                          if ... then
                ...                         ...
                i2 ← Φ(i1, i2)              i2 ← 10
                ...                         ...
fimwhile                            else
                                            ...
                                            i3 ← 20
                                            ...
                                    fimse
                                    ...
                                    i4 ← Φ(i2, i3)
                                    j ← i4
```

A função Φ(...) é utilizada quando num dado ponto do programa existem duas ou mais instancias duma mesma variável, transformando-as todas numa única e nova instancia. Os parâmetros da função são como tal as instâncias que alcançam esse ponto do programa, onde o resultado é a instância proveniente do ramo do fluxo por onde se alcançou a actual posição do programa.

Se no entanto esta abordagem resolve o problema das atribuições, não deixa por si só de levantar outros problemas: o primeiro é como saber onde se devem inserir as funções Φ(...); e o segundo é como as representar no código final, de forma a que estas funcionem como o previsto?

As respostas a estas perguntas são dadas no capítulo sobre optimizações (capítulo 5), onde se descreve o processo de implementação das estruturas de apoio ao SSA.

4.3 *Classes da MIR*

Julgo que as explicações fornecidas sobre o SSA, serão suficientes para se poder iniciar a descrição das classes da MIR.

4.3.1 Classe Genérica

Esta classe é na realidade composta por três classes: *Program*, *Function*, e *IdentifierTable*.

A primeira serve para associar todo o tipo de informação referente ao programa que se pretende compilar. A segunda, permite manipular a informação referente aos blocos de código onde de alguma forma existam variáveis locais; é o caso das funções, procedimentos, etc. E por fim, a terceira serve para a implementação da tabela de identificadores.

Um aspecto crítico, a considerar na implementação destas classes, é a versatilidade que devem possuir para se adaptarem às necessidades das diversas linguagens fonte. É como tal natural, que possuam um conjunto de características que vão muito além do que em princípio seria necessário para um caso concreto.

Classe Program

Após se analisar, para várias linguagens, a organizado e a estrutura de um programa, concluiu-se que seria necessário criar um suporte para representar as variáveis, o código e as funções que funcionam a nível global.

A título de exemplo, para um compilador de C, a este nível encontravam-se declaradas todas as funções, através de objectos do tipo *Function* (uma vez que estas são globais), todas as variáveis tipo externo ou global e ainda o código implementado fora de qualquer função.

Desta forma a classe *Program* é composta pelos seguintes atributos:

mainTable	- Um objecto da classe *IdentifierTable* para os identificadores globais.
listOfFunctions	- O conjunto dos objectos tipo *Function*, que compõem o programa.
main	- O elemento do conjunto anterior que assinala a função principal.
flowNodes	- O conjunto de *FlowNodes* que descreve o código externo a qualquer função.
rootNode	- O nodo inicial do conjunto anterior.
blockMemory	- Simula o espaço de endereçamento da memória de dados.

Este último atributo serve para realizar a gestão da atribuição de endereços às variáveis do programa, o que é útil para a geração do código máquina.

Classe *Function*

A classe *Function* possui a informação referente a uma função, procedimento ou bloco de código com variáveis locais.

Há semelhança da classe anterior, também esta classe possui: uma tabela para os identificadores locais, um conjunto de funções locais, um conjunto de *FlowNodes* com o código da função, um apontador para o nodo inicial do conjunto anterior e um sistema próprio para a gestão do espaço de endereçamento da memória de dados. Para além disso, possui ainda os seguintes atributos:

name - Nome associado à *Function*.

level - Nível de encadeamento da *Function*.

type - Tipo de *Function*, que pode ser função, procedimento ou expressão composta.

parent - *Function* da qual a actual é descendente.

Os parâmetros de entrada e o valor de retorno das funções descritas pelos objectos desta classe, são assinalados através da tabela de identificadores.

Classe *IdentifierTable* x *CellIDTable*

Esta classe tenta implementar uma tabela de identificadores, composta por pares do tipo (chave, conj. de informação). A chave é do tipo *string* e o conjunto de informação é definido por defeito pela classe *CellIDTable*. É na realidade esta última classe que contém os atributos que permitem descrever um identificador, dos quais constam os seguintes:

IDscope - Assinala se o identificador é do tipo *label*, constante, global, local, parâmetro ou temporário.

IDclass - Assinala a classe do identificador, *auto*, *register*, *static* ou *extern*.

IDtype - Indica o tipo de identificador, *char*, *short*, *unsigned*, *float*, *double*, etc.

IDcrmfl - Indica que tipo de *Expressions* está associado ao identificador, estas são classificadas nas seguintes expressões:

- *IDconst* - Expressão do tipo *Constant*.
- *IDreg* - O conjunto de todos os *DataTransfers* onde se realizam atribuições para os *Registers* associados ao identificador.
- *IDmem* - Expressão do tipo *Memory*.
- *IDfunction* - Objecto da classe *Function*.
- *IDlabel* - Expressão do tipo *Label*.

Os identificadores das variáveis encontram-se sempre associados a uma expressão do tipo *Memory* e a uma ou mais expressões do tipo *Register*. Como mais adiante se pode confirmar, tal deve-se à forma da representação SSA. Para cada *Register* guarda-se a informação do *DataTransfer* ao qual está associado (definido).

4.3.2 Classe *Expressions*

Os objectos desta classe identificam todos os operadores que podem compor as expressões da representação intermédia.

A composição das expressões é feita recorrendo a árvores (de expressões), onde os nodos representam sempre operadores e onde os respectivos resultados funcionam como operandos do nodo imediatamente acima.

Na prática às árvores podem partilhar sub-árvores entre si, pelo que na realidade não são árvores mas sim grafos acíclicos directos (DAG), com um máximo de duas arestas por nodo.

A classe *Expressions* possui todos os atributos necessários a um qualquer operador, dos quais fazem parte os seguintes:

left - Representa a sub-árvore esquerda.
right - Representa a sub-árvore direita.
dTransfer - O *DataTransfer* ao qual a expressão está associada.
type - O tipo do resultado da expressão, *char*, *short*, *float*, etc.
operation - O nome do operador.

As subclasses de *Expressions* estão organizadas da seguinte forma:

Expressions
- *BinaryExpressions*
  - *AsgnExpression*
  - *AddExpression*
  - ...
- *UnaryExpressions*
  - *NegExpression*
  - ...
- *Storage*
  - *Memory*
  - *Register*
  - *Constant*
  - *BlockMemory*
  - *Label*
  - ...
- *JumpExpressions*
  - *Jump*
  - *CondJump*
  - *Call*
  - *Return*
  - ...

Como se pode confirmar existem quatro grandes grupos de operadores, os binários, unários, os que representam operandos e os operadores de salto. As classes *BinaryExpressions*, *UnaryExpressions* e *JumpExpression* não acrescentam novos atributos, servem apenas para classificar os operadores e inicializar alguns atributos dos

declarados para *Expressions*. Já o *Storage* possui um atributo, o *value*, que é uma união que permite armazenar qualquer valor (variável, constante, etc) de um qualquer tipo primitivo de dados da linguagem C (*char*, *short*, *float*, etc).

Não se pretende aqui, descrever extensamente todas as subclasses de *Expressions*, é no entanto necessário apresentar algumas dessas classes para que seja possível compreender a representação intermédia.

Classe BlockMemory

Esta é uma classe que não serve para descrever expressões, mas sim auxiliar o processo de atribuição de endereços às variáveis. É iniciada com um determinado tamanho que representa o espaço de memória em bytes, sem que no entanto esse espaço exista. Depois conforme se constrói a representação intermédia, é possível requisitar aos objectos desta classe, endereços para as variáveis, necessitando apenas que se indique o tamanho, em bytes, do espaço de memória a este associado. A própria classe garante a gestão do espaço de endereçamento.

Na prática não é mais que um array de bits, onde cada um representa um byte de memória e onde a gestão consiste em alocar e a desalocar o espaço conforme é necessário.

Classe Memory

Os objectos desta classe dão corpo às variáveis, estando normalmente associados a um *BlockMemory* e a um endereço por este atribuído. Para além disso encontram-se ligados à célula da tabela de identificadores cuja variável representam.

Classe Register

Os objectos desta classe são os recipientes, por omissão, do resultado das expressões. O seu objectivo é de alguma forma representar o resultado intermédios de cada uma das operações.

De notar que, apesar do nome desta classe ser *Register*, não significa que no código final os objectos dela instanciados sejam substituídos directamente por registos físicos.

Há semelhança da classe anterior, também estes objectos podem estar relacionados com uma célula da Tabela de Identificadores, aliás, para o pleno funcionamento de todas as estruturas do BEDS, deverá sempre existir esta relação. Este aspecto será explorado na descrição da classe *DataTransfer* e no capítulo sobre a alocação global.

Na representação intermédia estes objectos não chegam a fazer parte das árvores de expressões, encontrando-se apenas associados a estas, através dos *DataTransfer*. Aliás, são estes últimos os responsáveis pela instanciação dos objectos desta classe, pelo que existe sempre um *DataTransfer* para cada *Register* (o inverso já não é verdade).

Mais explicações sobre as subclasses de *Expressions*, serão fornecidas posteriormente conforme forem necessárias.

4.3.3 Classe DataTransfer

Como já foi dito, as árvores de expressões apesar de serem a forma de representação ideal para determinados processos, como é o caso da selecção de instruções, não são no entanto a melhor forma de representação para o caso das optimizações e mais

concretamente para os sistemas de análise, que servem de suporte à optimização ou alocação global.

Esta questão é tanto mais relevante quanto mais se trabalha na resolução deste tipo de problemas e nem sempre é fácil explicar. Mas muito genericamente pode-se dizer que as árvores não são suficientemente lineares para determinadas operações, por exemplo a ordem de execução das operações, é muito mais perceptível se estiver organizada segundo um array composto por triplos ou quádruplos, uma vez que dá a conhecer directamente a sequência das operações. A utilização das árvores é ainda mais penalizadora, quando se pretende relacionar nodos de árvores distintas.

Para além destes aspectos, as árvores não permitem o mesmo detalhe descritivo que uma representação com base em tuplos, uma vez que camufla determinadas trocas de dados.

Na concepção da grande maioria dos compiladores modernos, que possuem complexos métodos de optimização, não terão sido indiferentes estes aspectos, quando optaram por outras representações que não as árvores.

O *RTL System* possui uma classe de objectos, os *RegisterTransfer*, cuja finalidade é assinalar as transferências entre registos, classificando o tipo de troca de informação. Este tipo de objectos poderiam à primeira vista ser dispensados de uma representação intermédia, mas após um estudo mais detalhado, detecta-se a falta de uma entidade à qual se associe toda a informação sobre a transferência entre dados, que é de extrema importância para os processos de análise de fluxo de dados e análise de dependência entre dados. Para além disso a utilização dos *RegisterTransfer* permite em parte tornar a representação das árvores mais linear e representar as trocas entre dados que antes se encontravam camufladas nas árvores.

Surge assim, uma classe designada por *DataTransfer*, cuja a finalidade é, à semelhança do *RegisterTransfer*, controlar as trocas de dados e acrescentar a informação que faltava às árvores, em comparação com os tuplos. A alteração do nome deve-se a que os objectos tipo *DataTransfer*, não controlam apenas as transferências de, para e entre *Registers*, como se poderá ver mais adiante.

O que à primeira vista pode ser uma simples alteração conceptual, na prática trás algumas complicações, o problema é que não é fácil relacionar estes dois tipos de objectos. A solução mais viável é fazer com que cada operação da árvore de expressões corresponda a um *DataTransfer*, mas nem todos os objectos do tipo *DataTransfer*, estão associados a um nodo de uma árvore. Alguns porque descrevem as tais trocas que não são possíveis de representar numa árvore e outros porque descrevem alguns aspectos conceptuais, que não se encontram directamente relacionados com representação do código. No entanto desde que a representação das árvores de expressões se mantenha íntegra, os *DataTransfers* excedentes em nada afectam os processos que funcionam exclusivamente sobre árvores, uma vez que é função do *TreeSelector* determinar quais os *DataTransfers* é que são relevantes para a construção das árvores de expressões e desprezar os restantes. Assim sendo é possível existir *DataTransfers* com informação relevante apenas para os processos que funcionam sobre a MIR.

A estrutura hierárquica da classe *DataTransfer* encontra-se organizada da seguinte forma:

DataTransfer
- *Assignment*
- *AttribAssignment*
- *LabelAssignment*

JumpAssignment
CondJumpAssignment
ReturnAssignment
PhyAssignment

Antes de se descrever cada uma das classes descendentes de *DataTransfer*, há que apresentar os seus atributos:

flowNode	- *FlowNode* ao qual o *DataTransfer* pertence.
target	- *Expression* que representa o destino onde colocar o resultado do *DataTransfer.*
source	- *Expression* que representa o resultado a colocar em *target.*
flowDependents	- Conjunto dos *DataTransfers* que dependem do resultado do actual *DataTransfer*.
flowSupporters	- Conjunto dos *DataTransfers* do qual o actual *DataTransfer* depende.

É de realçar que nem todos os descendentes de *DataTransfer* utilizam o *target* e o *source*.

As duas últimas variáveis fazem parte do sistema de suporte à análise de dependência e de fluxo de dados, sendo a sua gestão feita durante o próprio processo de geração da representação intermédia.

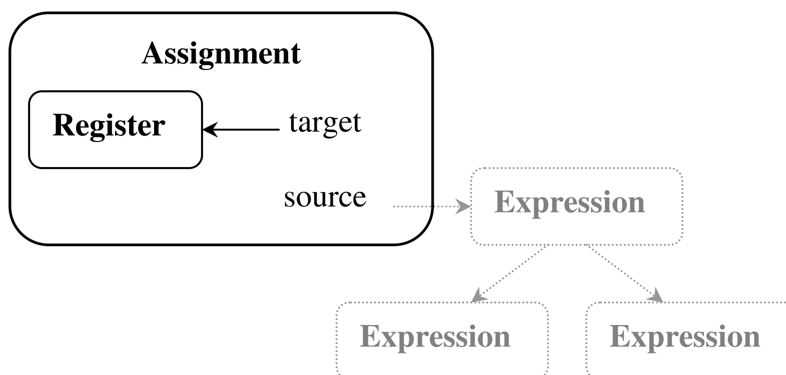


Fig. 4.1 – Classe *Assignment*.

Classe Assignment

A classe *Assignment* é o tipo de *DataTransfer* de utilização genérica, a única restrição é que o LHS tem que ser um *Register*, permitindo assim identificar trocas de dados entre uma qualquer expressão que se encontre do RHS para um pseudo-registo. Esta classe encontra-se representada na Fig. 4.1, onde se pode identificar o *target* do tipo *Register* e o *source* que representa uma qualquer *Expression*.

A atribuição de uma *Expression* a *source* actualiza as estruturas de suporte à análise do fluxo de dados. Para tal, o método responsável pela atribuição desta variável

determina quais os *DataTransfers* responsáveis pelos resultados dos descendentes de *source*. Uma vez conhecidos, actualiza os respectivos *flowDependents* e o *flowSupporter* do actual *DataTransfer*.

Classe AttribAssignment

Este tipo de *DataTransfer* permite identificar transferências de dados entre uma qualquer expressão e uma variável em memória. A sua função principal é indicar quando é que se deve actualizar o valor em memória de uma variável. Como tal, o *target* é sempre do tipo *Memory* e o *source* do tipo *AsgnExpression*, este último é uma classe derivada de *Expression*, do tipo *BinaryExpression*, cuja função é identificar directamente operações de atribuição.

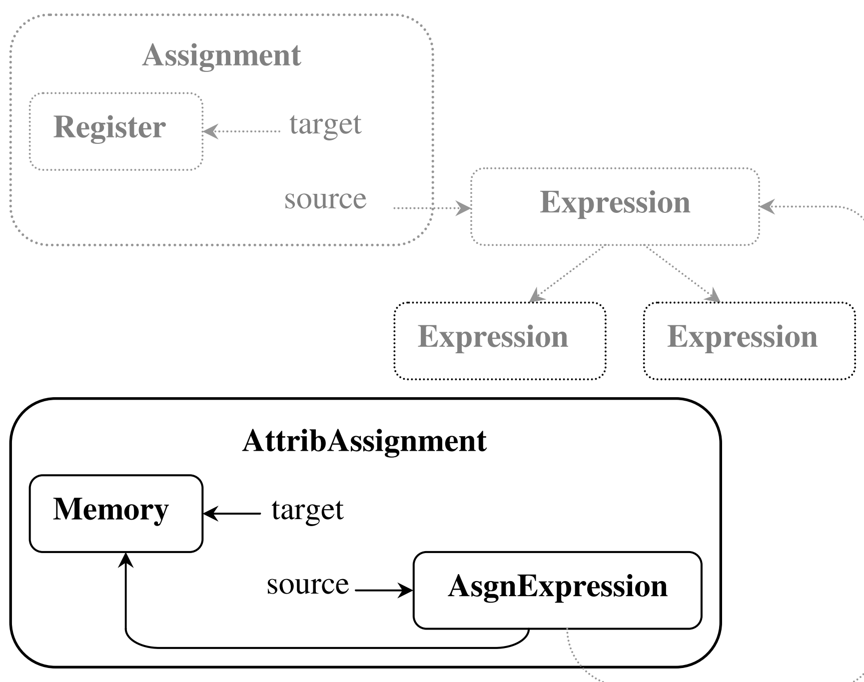


Fig. 4.2 – Classes *AttribAssignment* e *AsgnExpression*.

Por defeito o *Memory* fica associado ao descendente esquerdo do *AsgnExpression*, enquanto que no descendente direito fica a expressão que deriva no resultado a atribuir à variável, a qual deve possuir um *Assignment* próprio conforme se encontra ilustrado na Fig. 4.2.

Esta classe possui uma variável, a *nextAtrib*, que guarda a indicação de quais os *DataTransfers* responsáveis pela próxima atribuição a esta variável. A sua gestão pode ser feita aquando da geração da representação intermédia ou após esta estar feita. A primeira opção necessita da colaboração da Tabela de Identificadores, mais concretamente da variável *IDreg* e de alguma colaboração do gerador da representação intermédia. Já a segunda solução utiliza um algoritmo próprio.

Classe LabelAssignment

Esta classe destina-se a identificar as posições onde é necessário inserir *labels*. Encontra-se normalmente associada aos objectos da classe *FlowNode*, como se poderá

ver mais adiante. O único atributo que contém é o *label*, que identifica uma *Expression* do tipo *Label*. Esta última classe apenas possui um identificador, que é único e é atribuído aquando da instanciação deste tipo de objectos.

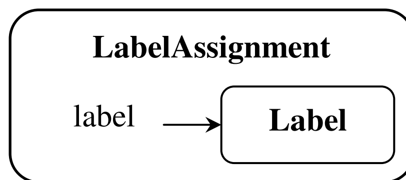


Fig. 4.3 – Classes *LabelAssignment* e *Label*.

Classe JumpAssignment

Este tipo de *DataTransfers* serve para identificar operações de salto incondicional. A sua utilização está sempre associada a um determinado tipo de *FlowNode*, o *JumpNode*.

Esta classe possui apenas uma única variável, a *jpexp*, que é do tipo *Jump*. Esta última permite que lhe seja associado um objecto da classe *Label*, e cuja função é identificar a posição para onde se deve efectuar o salto. A ligação entre estas classes encontra-se representada na Fig. 4.4.

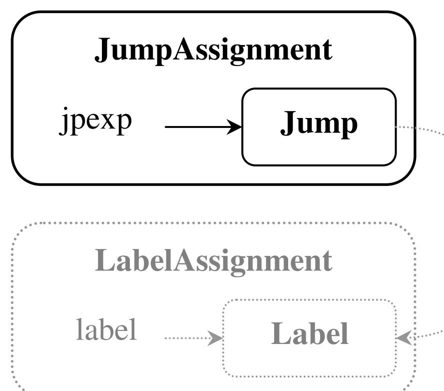


Fig. 4.4 – Classes *JumpAssignment* e *Jump*.

Classe CondJumpAssignment

Este é o tipo de *DataTransfer* que identifica operações de salto condicional. À semelhança da classe anterior, a sua utilização está directamente associada a um determinado tipo de *FlowNode*, o *CondJumpNode*.

Esta classe possui apenas uma variável, a *cjpexp*, da classe *CondJump*. Esta última descreve toda a operação necessária à execução de um salto condicional. Como tal, está associada a uma condição de teste e a dois objectos do tipo *Label*, para os quais se realizam os saltos, mediante o valor da condição.

Como esta operação necessita de três argumentos, é necessário recorrer a alguns mecanismos auxiliares de forma a manter a coerência com o sistema de expressões binárias utilizadas pelo BEDS.

Para tal o *CondJump* possui um objecto da classe *ArgExpression*, que é um descendente de *Expression* e cuja finalidade é unicamente decompor expressões com mais do que dois argumentos em expressões binárias. Desta forma, à variável *left* de *CondJump* está associada a condição e à variável *right* a *ArgExpression*, cujos descendentes são as duas *Labels*. A representação esquemática encontra-se na Fig. 4.5.

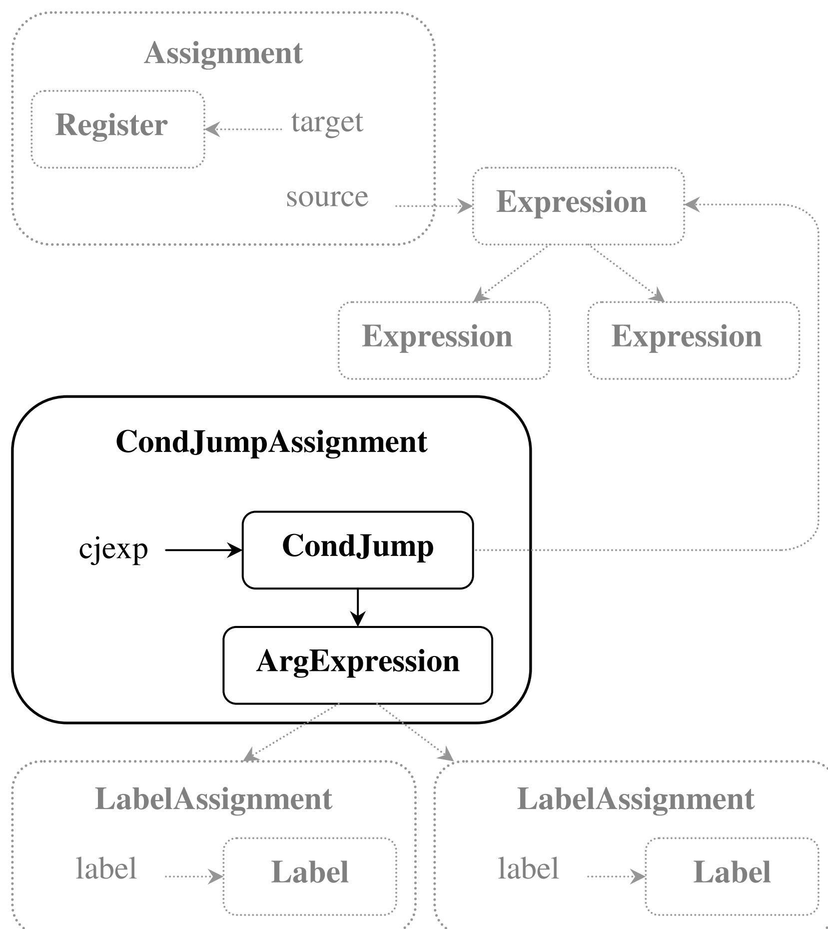


Fig. 4.5 – Classes *CondJumpAssignment*, *CondJump* e *ArgExpression*.

Classe CallAssignment

Os objectos desta classe pretendem descrever as chamadas de funções ou procedimentos. O *target* está ligado a um *Register* e o *source* a uma *Expression* do tipo *Call*. Este, por sua vez, possui como descendente esquerdo um objecto do tipo *Function* que assinala a função/procedimento a invocar e do lado direito a lista de argumentos, ou seja, a lista das expressões das quais resultam os argumentos. Quando existe mais do que um argumento é necessário utilizar o *ArgExpression*. A Fig. 4.6 ilustra a utilização desta classe.

Classe ReturnAssignment

O *ReturnAssignment* serve para identificar as operações de retorno de funções, as quais estão sempre associadas aos *FlowNodes* do tipo *ReturnNode*.

Esta classe possui uma variável da classe *Return* que é descendente de *Expression*, a qual está associada, caso exista, à expressão que dá origem ao valor a retornar. A relação entre estas classes encontra-se representada na Fig. 4.7.

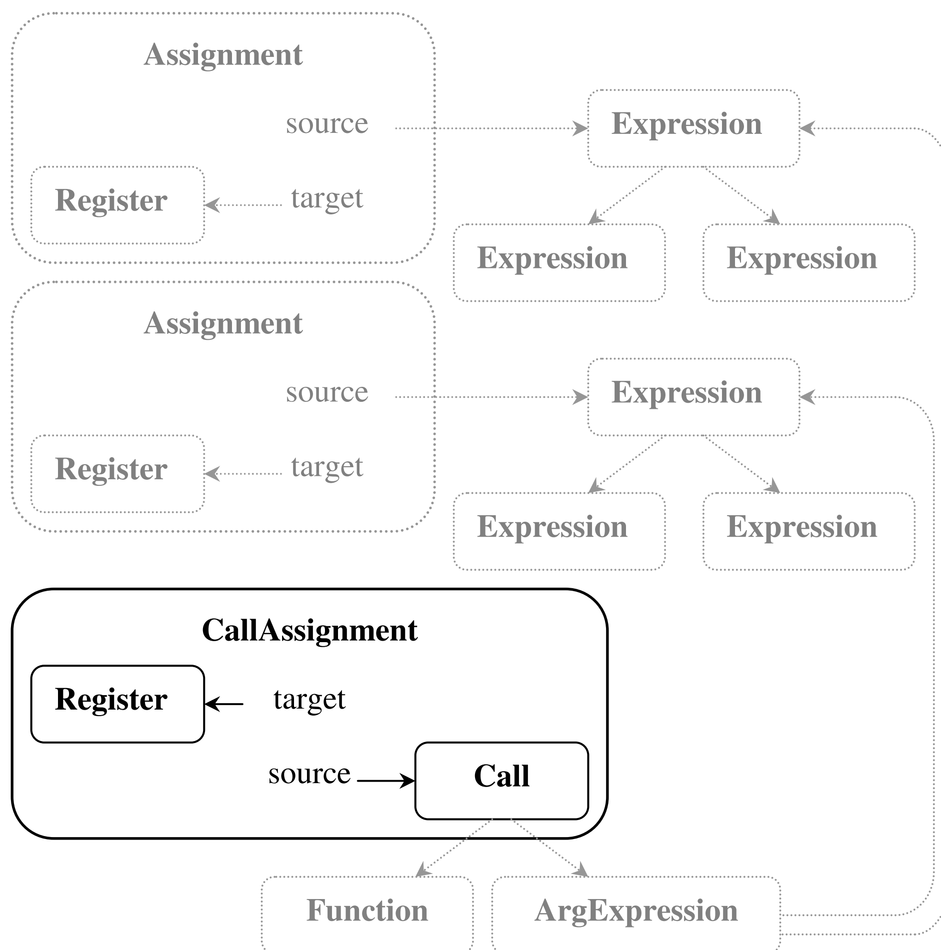


Fig. 4.6 – Classes *CallAssignment* e *Call*.

Classe PhyAssignment

Esta classe é exclusiva dos mecanismos necessários ao SSA, a sua função é assinalar a presença de funções Φ(...). Não possui qualquer variável específica uma vez que a informação de que necessita é determinada através dos mecanismos que implementam a análise do fluxo de dados.

Como se poderá ver no capítulo 5.3, estes *DataTransfer* são inseridos pelo processo que converte a representação não-SSA em SSA. É ainda possível inserir estes *DataTransfer* aquando da geração da representação intermédia, mas infelizmente tal só é possível para linguagens bem "comportadas", ou seja, linguagens sem operações do tipo *goto* ou com vários *returns*.

Quanto à classe *DataTransfer*, como forma de representação, está quase tudo dito. O que falta referir está relacionado com as estruturas de dados que foram necessárias introduzir para o funcionamento dos restantes módulos do BEDS, mas tal será feito nos

respectivos capítulos. Convém apenas sublinhar que, apesar dessas estruturas estarem incluídas noutros projectos, são elas que dão razão de ser aos *DataTransfer*.

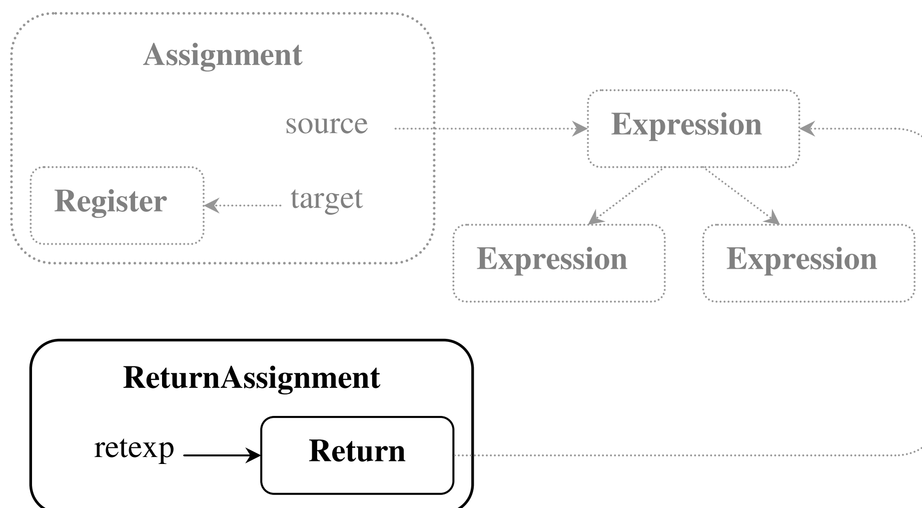


Fig. 4.7 – Classes *ReturnAssignment* e *Return*.

4.3.4 Classe FlowNode

Apesar dos objectos da classe *Expression* e da classe *DataTransfer*, por si só já formarem uma excelente forma de representação, é por vezes necessário ter uma perspectiva mais global de como está estruturado o programa a representar.

Para tal existe a classe *FlowNode* cujos objectos servem para agrupar os *DataTransfers* em conjuntos, os quais quando interligados entre si permitem representar as estruturas de controlo de um programa, sendo esta a sua principal função. Mas, a utilização dos *DataTransfer* como um conjunto, permite ainda decompor a representação possibilitando um tratamento localizado, o que facilita muitas das tarefas do *back-end*. Para além disso permite uma melhor gestão da informação, uma vez que esta é feita por conjuntos e não por indivíduos. É evidente que a maior parte dessa informação não está relacionada com a caracterização dos *DataTransfers* mas sim com a estruturação do programa.

As variáveis mais importantes desta classe, são:

funct	- *Function* ao qual o *FlowNode* pertence.
startNode	- Indica se o actual *FlowNode* é o nodo inicial do programa.
endNode	- Indica se o actual *FlowNode* é o nodo final do programa.
inEdges	- Sequência com os *FlowNodes* que estão ligados ao actual *FlowNode*.
outEdges	- Sequência com os *FlowNodes* ao qual o actual *FlowNode* se liga.

As duas últimas variáveis fazem parte do suporte à análise de fluxo de controlo e representam respectivamente o conjunto dos nodos antecessores e o conjunto dos nodos sucessores. A ordem, segundo a qual os nodos estão guardados em *inEdges* e em *outEdges* é extremamente importante. No primeiro caso, serve para relacionar os *FlowNodes* com os argumentos das funções Φ(...). Assim a posição ocupada pelo argumento na função Φ(...), é a mesma que ocupa o *FlowNode*, responsável por determinar o valor desse argumento, em *inEdges*. Na realidade não é o *FlowNode* quem determina o valor do argumento, mas sim um dos *DataTransfers* contidos nesse *FlowNode*. No caso do *outEdges* existe uma relação entre a posição que o *FlowNode* ocupa e a semântica a ele associado. Esta situação é explicada mais adiante.

A classe *FlowNode* encontra-se estruturada da seguinte forma:

- *FlowNode*
  - *GenericNode*
    - *JumpNode*
    - *CondJumpNode*
    - *ReturnNode*
  - *IntervalNode*

Apenas os objectos da classe *GenericNode* e respectivos descendentes servem para a descrição da representação intermédia. O *IntervalNode* funciona como estrutura de apoio e é utilizado essencialmente na análise estrutural para a construção da árvore de controlo.

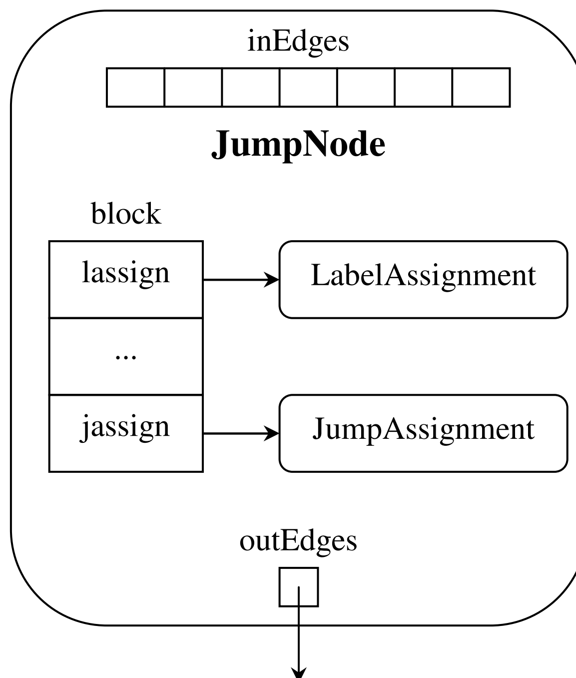


Fig. 4.8 – Classe *JumpNode*.

Classe GenericNode

Esta classe não é directamente utilizada na representação intermédia, pois destina-se apenas a conter as variáveis que são comuns às classes que dela descendem.

As variáveis que a compõem são: o *block*, que é uma sequência com todos os *DataTransfers* contidos em *GenericNode*; e a *lassign*, do tipo *LabelAssignment* que permite associar uma *Label* ao início do *GenericNode*. O *lassign* é sempre o primeiro *DataTransfer* de *block*.

Classe JumpNode

Esta classe serve para representar qualquer bloco de instruções (*DataTransfers*), que termine com uma operação de salto incondicional, como tal possui uma variável, a *jassign*, do tipo *JumpAssignment* com toda a informação associado à operação de salto. O *jassign* ocupa sempre a última posição de *block*. A Fig. 4.8 ilustra a composição desta classe.

Classe CondJumpNode

Os blocos de instruções que terminem com uma operação do tipo salto condicional são representados por objectos da classe *CondJumpNode*. As principais diferenças em comparação com a classe anterior, é que esta possui no lugar do *JumpAssignment*, um *CondJumpAssignment* e o *outEdges* está ligado a dois *GenericNode*, um por onde continua a execução do programa caso a condição seja verdadeira e outro para o caso desta ser falsa.

Classe ReturnNode

Esta classe serve para representar os conjuntos de instruções que terminam com uma operação do tipo retorno. Ao contrário das duas classes anteriores não possui elementos em *outEdges* e o último *DataTransfer* de *block* é um *ReturnAssignment*.

Classe IntervalNode

Este é um tipo de *FlowNode* cuja principal característica é permitir conter outros *FlowNodes*. Destina-se a ser utilizado fundamentalmente pela análise estrutural. A qual, como já atrás se disse, tem por objectivo construir a árvore de fluxo de controlo a partir do respectivo grafo, em que para tal é necessário reconhecer determinados padrões pré-definidos, formados por vários nodos e que representam estruturas condicionais e cíclicas conhecidas, os quais são posteriormente substituídos por um único objecto desta classe. O objectivo final é reduzir o grafo de fluxo de controlo a um único nodo deste tipo. As variáveis desta classe são:

flowNodes - O conjunto de todos os *FlowNodes* que compõem o *IntervalNode*.

region - Identifica o tipo de estrutura de controlo cujos *FlowNodes* contidos em *IntervalNode* representam.

rootNode - Identifica o *FlowNode* inicial do conjunto contido em *IntervalNode*.

O *IntervalNode* pode possuir diversos *FlowNodes* em *inEdges* e em *outEdges*, a sua ordenação vai depender das convenções estabelecidas para cada tipo de *region*, encontrando-se já definidas as seguintes:

Null	- Valor por defeito atribuído a *region*.
Block	- Representa um bloco simples de código.
IfThen	- Representa uma estrutura do tipo *If* ... *Then* ... .
IfThenElse	- Representa uma estrutura do tipo *If* ... *Then* ... *Else*... .
Case	- Representa uma estrutura do tipo *Switch* ... *Case* ... .
Improper	- Representa estruturas não cíclicas, normalmente contendo *goto*'s.
SelfLoop	- Representa uma estrutura do tipo *While* ..., sem corpo (ciclo de espera).
WhileLoop	- Representa uma estrutura do tipo *While* ... *do* ... .
Terminal	- Representa um único *GenericNode*.
Proper	- Serve para representar estruturas não identificáveis com nenhuma das anteriores.

Após esta descrição da classe *FlowNode*, pode-se dizer que o grafo de controlo de fluxo (CFG) de um programa, é, na representação MIR, formado por um conjunto de *GenericNodes*, o qual contém mais dois elementos do que o número de vértices do CFG original, que são o *startNode* e *endNode*.

Conclui-se assim a apresentação da MIR, de onde se espera que as explicações fornecidas tenham sido suficientes para compreender esta forma de representação e permitam perceber os restantes capítulos desta tese.

5 Optimização

A fase de optimização, como já foi dito, é uma das fases mais importantes de um compilador moderno e tende a ganhar cada vez mais relevo. Tal facto deve-se a que as exigências actuais não se padecem com a construção de simples tradutores de código fonte em código final, com base em técnicas *ad doc*. É exigido muito mais do que isto, pretende-se compiladores que retirem o máximo proveito das potencialidades do processador para o qual são construídos.

De notar que a velocidade de execução de um programa não depende só da velocidade do processador, mas também da eficiência do código objecto; além de que a quantidade de recursos necessárias à execução desses mesmos programas é em grande parte determinada pelo processo de compilação.

Não é por uma qualquer razão que esta área é uma das mais activas de entre as muitas que compõem o universo das áreas associadas ao desenvolvimento de geradores de código. Não pelos processos de optimização em si, mas sim pelas diferentes formas como estes são implementados, as quais têm evoluído a par dos modelos de representação e das infra-estruturas possíveis de associar a estes modelos. É como tal praticamente impossível falar de optimizações, sem falar das várias formas de análise fundamentais à sua implementação. Pretende-se aqui (neste capítulo) focar aquelas que são relevantes aos vários projectos que compõem o BEDS, das quais fazem parte a análise do fluxo de controlo, a análise do fluxo de dados e alguns conceitos sobre a análise de dependência entre dados.

5.1 *Análise do fluxo de controlo*

A análise do fluxo de controlo permite manter o conhecimento sobre a estrutura de um programa, fornecendo informação de como estão organizados os diversos blocos de código e da relação entre estes. Para tal, recorre-se normalmente a estruturas em forma de grafos orientados e consequentemente aos algoritmos por estes utilizados. Dentro deste contexto estes grafos serão designados por *grafos de fluxo de controlo (CFG).*

Como se viu no capítulo anterior, as infra-estruturas de suporte a este tipo de análise encontram-se implementadas através das variáveis da classe *FlowNode*, denominadas *inEdges* e o *outEdges*. Estas não são mais do que duas listas de adjacências, onde o *outEdges* guarda os *FlowNodes* adjacentes segundo o fluxo de controlo e o *inEdges* os *FlowNodes* adjacentes segundo o sentido contrário ao fluxo de controlo. Estas duas estruturas em conjunto com *flowNodes* da classe *Function*, que armazena todos os nodos de uma função/procedimento, mantêm toda a informação necessária sobre a organização de um programa (função). De notar que *inEdges* e *outEdges* não são mais do que os conjuntos dos antecessores e dos sucessores de um vértice. A Fig. 5.1 mostra um exemplo de um *CFG* e as respectivas listas de adjacências.

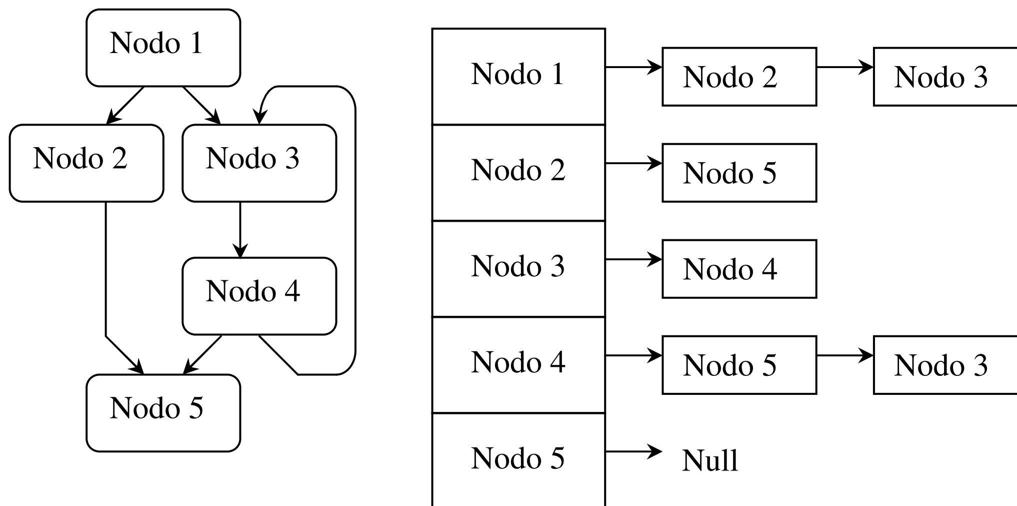


Fig. 5.1 – Exemplo de um CFG e da respectiva lista de adjacências.

A Fig. 5.2 ilustra a representação MIR das listas de adjacências dos três primeiros nodos do CFG da Fig. 5.1.

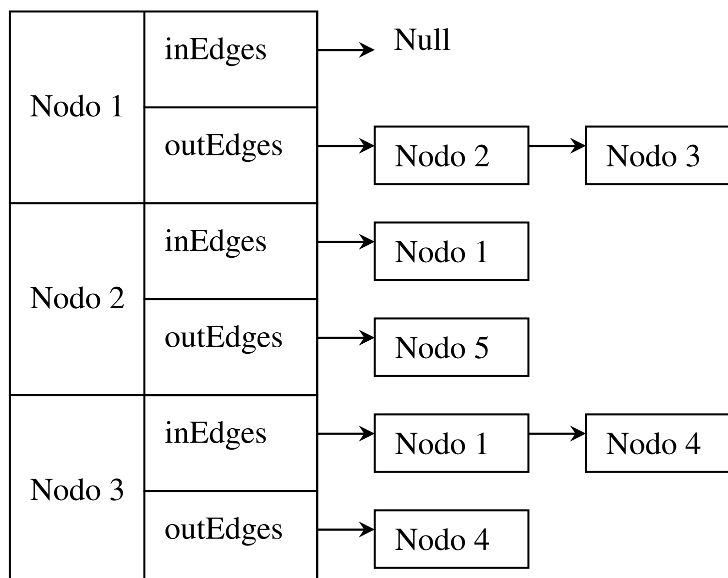


Fig. 5.2 – Representação da lista de adjacências dos 3 primeiros nodos do grafo da Fig. 5.1.

5.1.1 Árvore do fluxo de controlo

Apesar da informação associada ao *inEdges* e ao *outEdges* permitir identificar todos os caminhos possíveis de um programa, falta-lhe alguma semântica, como por exemplo, perceber que estruturas de controlo se encontram representadas no grafo, ou quais as componentes dessas estruturas. Este tipo de informação pode por vezes, ser facilmente perceptível, mas nem sempre tal acontece. Por exemplo, no grafo da Fig. 5.1 facilmente se percebe que o *Nodo 1* representa parte de uma estrutura condicional do tipo *If ... then ... else ...* . Convencionando que o descendente esquerdo representa o percurso a executar caso a condição seja verdadeira, então também é fácil perceber que o *Nodo 2* corresponde ao conjunto de instruções a executar nessa situação, mas não é tão fácil perceber qual ou quais os nodos a executar caso a condição seja falsa.

Para tal faz falta construir o que se designa por árvore de controlo. Como o nome indica, trata-se de uma árvore em que cada nodo pode possuir vários descendentes e onde apenas as folhas representam nodos simples; os restantes nodos da árvore servem apenas para representar composições de nodos do grafo.

Como se poderá ver em secções posteriores, a árvore de controlo é fundamental para os processos que necessitam de "linearizar" o grafo de fluxo como, por exemplo, para a selecção de instruções.

Existem, pelo menos, duas técnicas possíveis de utilizar na construção das árvores de controlo a partir do respectivo grafo de fluxo: uma utiliza a análise de intervalos e outra a análise estrutural. Esta última foi a técnica utilizada no BEDS que não é mais que uma forma alternativa de realizar a análise de intervalos.

Antes de se apresentar estas técnicas, é conveniente definir uma das propriedades dos grafos, a redutibilidade.

Se for possível decompor um grafo em vários sub-grafos, cada um formado por um subconjunto dos nodos do grafo original e por todas as transições do mesmo, que interligam os nodos do sub-grafo entre si, de tal forma que, este possa ser substituído por um nodo do tipo *IntervalNode* e se for possível aplicar este processo de decomposição/substituição até se obter um único nodo do tipo *IntervalNode*, que represente todo o grafo original, diz-se então que este é redutível.

A redução obedece sempre a um conjunto de transformações com as quais deverá ser possível reduzir o grafo a um único nodo. No caso de grafos não redutíveis é por vezes possível isolar a parte que não o é e reduzi-la a um único nodo, o qual é classificado como região *Imprópria* e que funciona como uma espécie de região desconhecida, ou seja, que não obedece a nenhuma das transformações descritas. Outra hipótese consiste em decompor a parte não redutível através de uma técnica designado por *node splitting*, de forma a tornar o grafo redutível.

As duas técnicas que se apresentam em seguida permitem realizar o processo de redução do grafo de fluxo.

5.1.2 Análise de intervalos

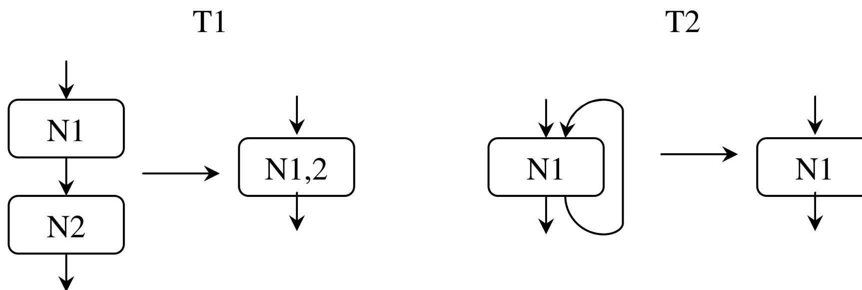
A utilização da análise de intervalos como forma de implementar a análise do fluxo de controlo passa, em primeiro lugar, por dividir o grafo em regiões que obedeçam a padrões pré-definidos. Estas regiões compostas por dois ou mais nodos são posteriormente substituídas por um único nodo abstracto (*IntervalNode*). O qual, para

todos os efeitos representa o conjunto de nodos que compõem a região, designadamente nas ligações com os nodos exteriores.

É através de sucessivas identificações das regiões e respectivas substituições, que é possível reduzir o grafo a um único nodo do tipo abstracto, o qual representa a raiz da árvore de controlo e onde os nodos que o compõem são os seus descendentes directos na árvore. As folhas desta são sempre os nodos do grafo de fluxo.

Exemplo 5.1

Pretende-se mostrar a aplicação de uma forma de análise de intervalo, conhecida por *T1-T2 interval analysis*, na redução do grafo da Fig. 5.1. Este tipo de análise utiliza apenas as duas seguintes transformações para reduzir o grafo:



O processo de redução encontra-se representado na Fig. 5.3, onde numa primeira fase são detectados dois blocos, um do tipo *Imprópria* e uma sequência simples (T1), ambos são substituídos. Numa segunda fase detecta-se um *SelfLoop* e numa terceira fase restam dois nodos que formam uma região *Imprópria*. Esses dois nodos da região *Imprópria*, não são mais reduzidos, mas são aglutinados dando origem ao nodo único que representa todo o grafo. A respectiva árvore de controlo encontra-se representada na Fig. 5.4.

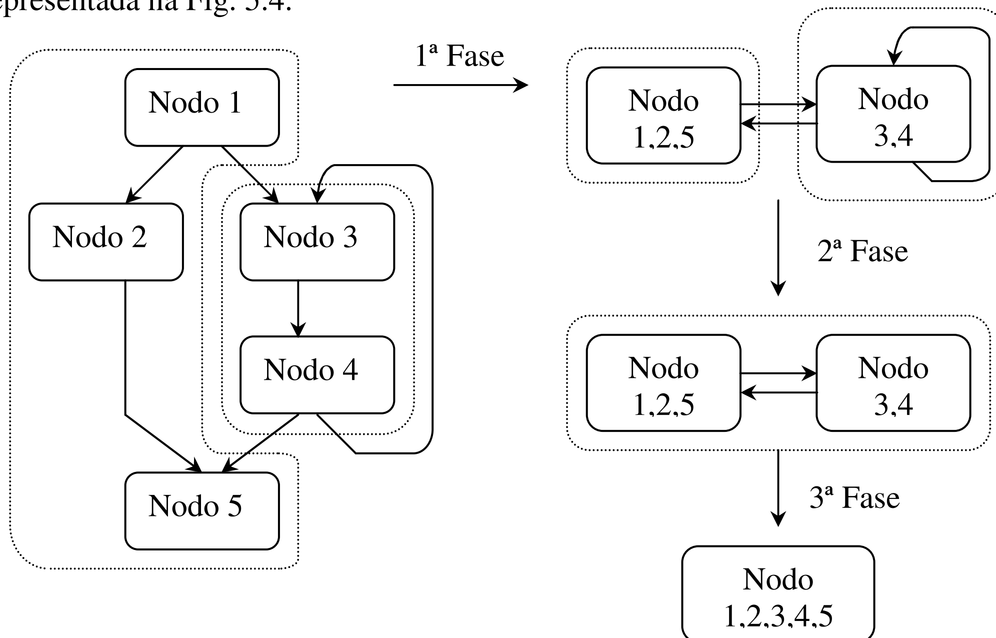


Fig. 5.3 – Redução do grafo da Fig. 5.1, segundo a análise T1-T2.

♦

Devido ao número muito restrito de transformações e da forma como estas são aplicadas, este tipo de análise não permite obter grandes resultados. Torna-se ainda menos atractiva quando utilizada para reconhecer padrões que se encontram em situações muito enredadas. Em contrapartida é simples de perceber e implementar, e se considerar-mos que foi uma das primeiras soluções propostas para este tipo de análise, até que é aceitável.

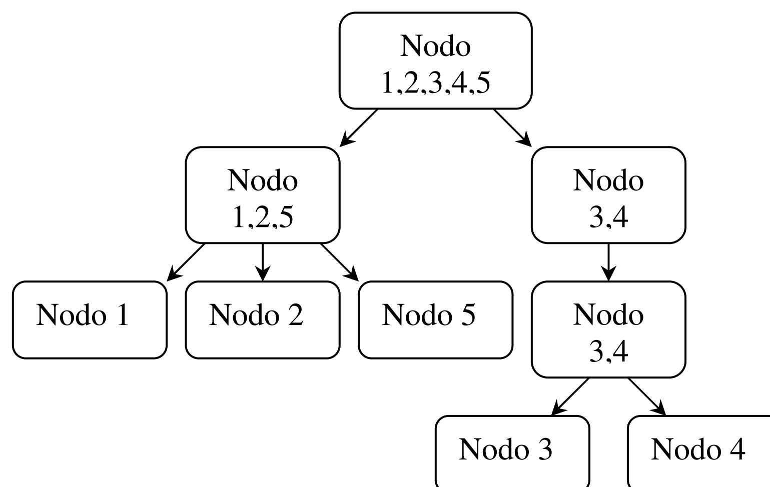


Fig. 5.4 – Árvore de controlo resultante da análise T1-T2 para o grafo da Fig. 5.1.

5.1.3 Análise Estrutural

Com o objectivo de melhorar a forma de reconhecimento, permitindo distinguir qualquer tipo de região que se pretenda definir, independentemente do seu enredo, surgiu uma solução com base nos algoritmos da análise sintáctica para reconhecimento de expressões, os quais foram sujeitos a diversas alterações de forma a viabilizar a sua utilização sobre grafos, uma vez que por defeito são aplicados a estruturas em árvore (de sintaxe).

Com este tipo de abordagem é possível definir praticamente qualquer tipo de região, nomeadamente as regiões que identificam as estruturas de controlo da própria linguagem fonte. Mas mais do que isso, permite detectar regiões que apesar de não se encontrarem descritas directamente pela linguagem fonte, existem pela própria forma como se encontra organizado o programa. Estas situações são vulgares para as linguagens que permitem a utilização conjunta de *if*'s e *goto*'s, através dos quais é possível implementar estruturas de controlo mais elaboradas, como *while*, *repeat until*, etc, sem que tal seja detectado no reconhecimento da linguagem fonte.

O algoritmo de reconhecimento foi retirado do *Advanced Compiler Design and Implementation*, de Steven Muchnick [Much97].

Descrição do Algoritmo de Análise Estrutural

O algoritmo utilizado começa por realizar uma travessia do tipo pré-order sobre o grafo (à semelhança das árvores), marcando todos os nodos com um número de ordem

que indica a "camada" a que este pertence em relação a todo o grafo. Para tal foi necessário acrescentar à classe *FlowNode* as seguintes variáveis:

hasBeenVisited - Indica se o nodo já foi visitado no processo de travessia do grafo.

topNumber - Indica o número de ordem do nodo atribuído aquando da travessia.

interval - Identifica o *IntervalNode* do qual o nodo faz parte.

Depois pega no nodo de menor ordem e a partir dele tenta identificar algum tipo de região. É importante dizer, que uma das poucas restrições desta análise, é que as regiões só podem ter um único ponto de entrada, pelo que o nodo seleccionado acaba por ser o único que possui antecessores exteriores à região a identificar.

A identificação das regiões, no algoritmo utilizado, faz-se em duas fases: numa primeira tenta-se construir a partir do nodo inicial uma região do tipo acíclico; caso não seja possível, passa-se a uma segunda fase, a qual testa se o nodo inicial faz parte de uma região do tipo *Impróprio* ou do tipo cíclico.

Após se ter determinado o tipo de região, esta é reduzida a um nodo abstracto, obrigando a redireccionar todas as transições que se fazem entre os nodos que pertencem à região e os nodos exteriores a esta e actualizando a variável *interval* de cada nodo.

Sempre que ocorre uma redução, inicializa-se o processo, o qual se repete até que não seja possível executar mais nenhuma redução, ou seja, até que se obtenha um único nodo.

Tipo de regiões

Como já se referiu, as regiões podem ser de três grandes classes: acíclica, cíclica ou impróprias. As primeiras e como o próprio nome indica, caracterizam-se por não conterem caminhos cíclicos dentro da região que descrevem. A solução implementada permite detectar os seguintes tipos: *Block*, *IfThen* e *IfThenElse*, que se encontram representados na Fig. 5.5.

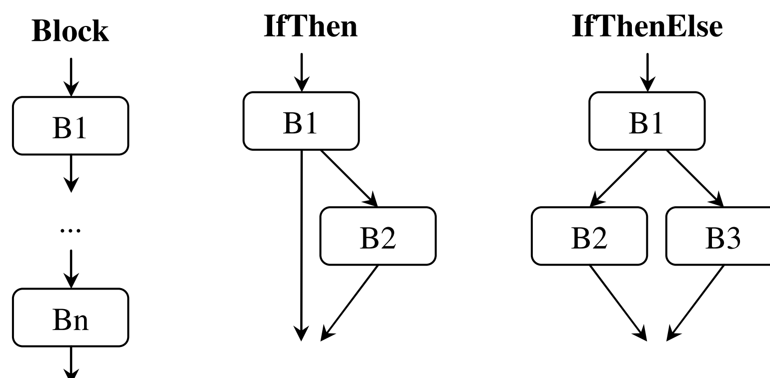


Fig. 5.5 – Exemplos de regiões acíclicas.

Nas regiões cíclicas existe sempre caminho entre dois quaisquer nodos. A solução implementada permite identificar as seguintes regiões: *SelfLoop*, *WhileLoop* e o *NaturalLoop*, que estão representados na Fig. 5.6.

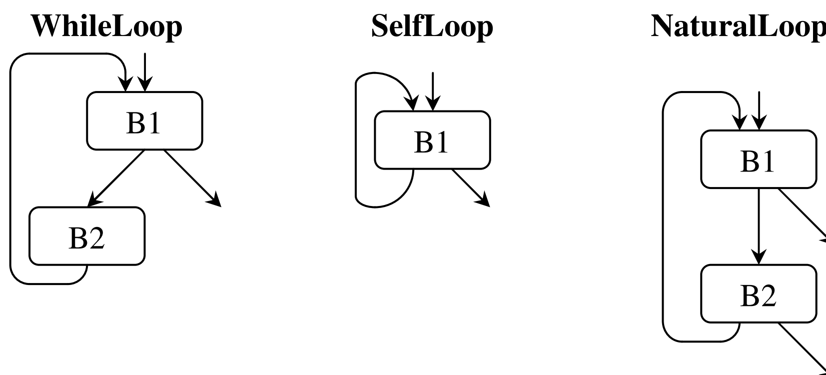


Fig. 5.6 – Exemplos de regiões cíclicas.

As regiões *Impróprias* envolvem normalmente sub-regiões cíclicas, mas que não se encaixam em nenhuma das anteriores. Na Fig. 5.7 encontram-se alguns exemplos deste tipo de regiões.

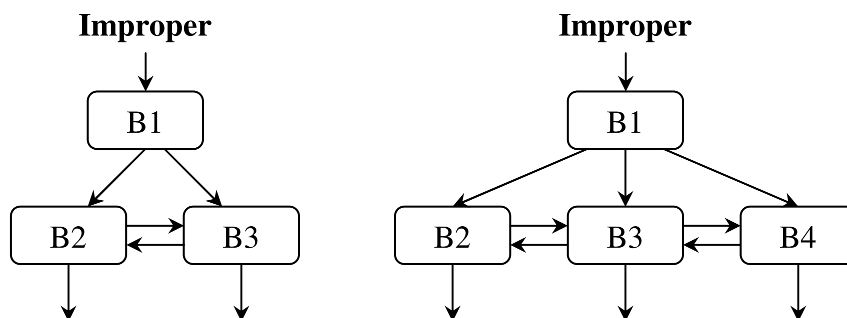


Fig. 5.7 – Exemplos de possíveis regiões impróprias.

Exemplo 5.2

Pretende-se ao longo deste exemplo mostrar como todo o processo da análise estrutural se desenrola para o grafo da Fig. 5.1.

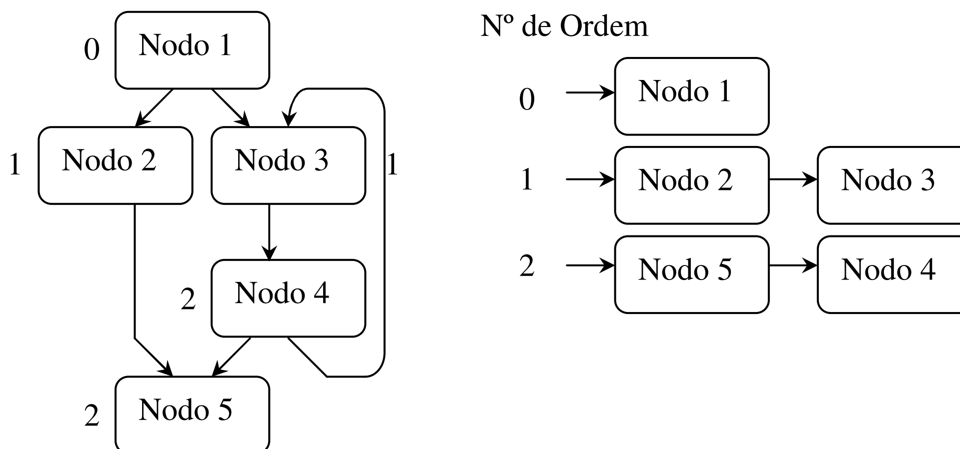


Fig. 5.8 – Ordenação *preorder* do grafo.

A análise começa por realizar uma travessia *preorder* do grafo, numerando os nodos pela ordem segunda a qual devem ser processados. O que no presente caso resulta na ordenação apresentada na Fig. 5.8. De seguida escolhe-se o nodo de menor ordem, que neste caso é o *Nodo 1* e tenta-se reconhecer alguma das regiões atrás definidas. Como tal não é possível, então escolhe-se o seguinte nodo de menor ordem, que pode ser o *Nodo 2* ou o *Nodo 3*. O *Nodo 2* também não permite identificar nenhuma região, mas já o *Nodo 3* permite identificar uma região do tipo *Block*, composta por ele próprio e pelo *Nodo 4*. Procede-se então à redução obtendo-se o grafo representado na Fig. 5.9.

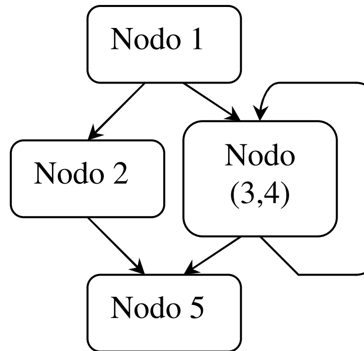


Fig. 5.9 – Resultado da primeira iteração da análise estrutural.

Volta-se a realizar a travessia *preorder*, ordenando novamente os nodos, onde a única alteração a registar, deve-se aos nodos 3 e 4, que passam ambos a ser de ordem 2. De seguida selecciona-se o nodo de menor ordem, ou seja o *Nodo 1*, mas como este não permite identificar nenhuma região, torna-se necessário escolher o próximo nodo de menor ordem, que tanto pode ser o *Nodo 2* ou o nodo abstracto (*Nodo 3,4*). Este último permite identificar uma região do tipo *SelfLoop*. Reduz-se novamente o grafo e volta-se a realizar a travessia *preorder*. Escolhe-se novamente o nodo de menor ordem, o qual continua a ser o *Nodo 1*, mas que agora já permite identificar uma região do tipo *IfThenElse*. Mais uma vez reduz-se o grafo e executa-se todo o procedimento, detectando-se agora uma região do tipo *Block*; após a sua redução, é concluído o processo, por se ter já atingido um só nodo. As várias fases encontram-se representadas na Fig. 5.10.

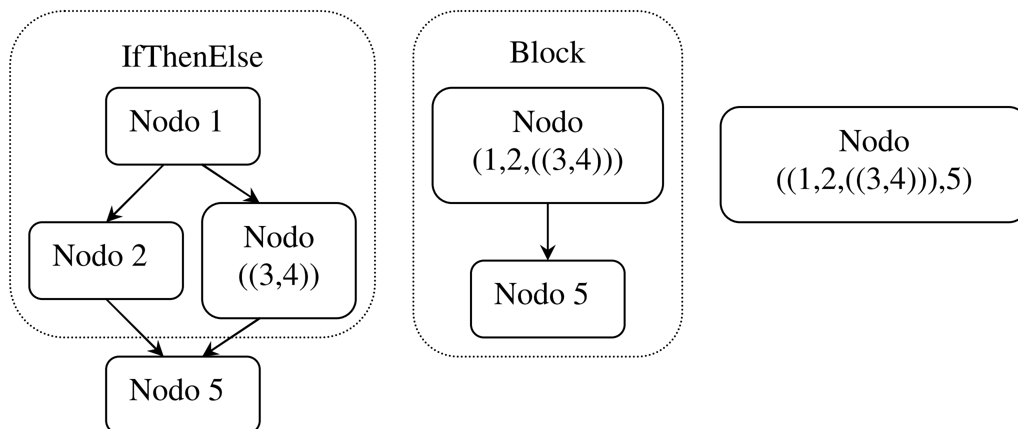


Fig. 5.10 – Restantes fases do processo de redução do grafo.

O nodo final como já se disse representa a raiz da árvore de controlo de fluxo, que para este exemplo se encontra representada na Fig. 5.11.

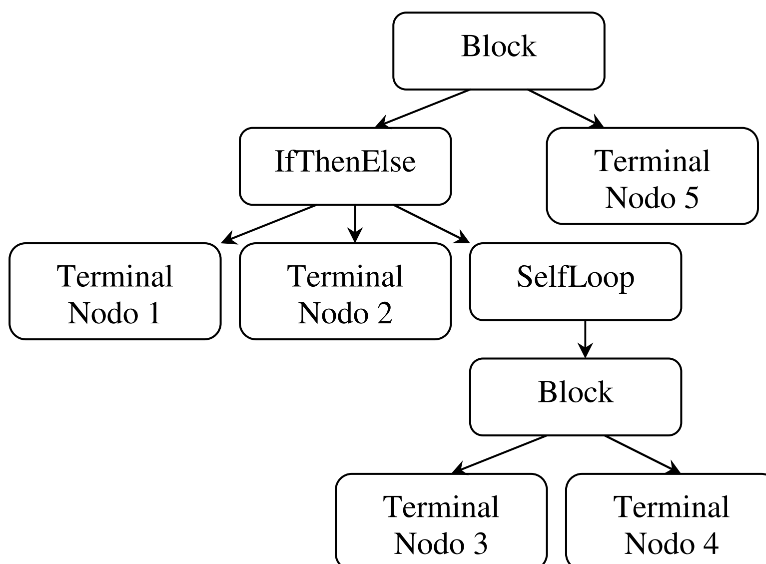


Fig. 5.11 – Árvore de controlo do grafo da Fig. 5.1.

5.2 *Análise do fluxo de dados*

A análise de fluxo de dados, pelo tipo de informação que permite obter, é uma das principais formas de análise, funcionando de suporte a um grande número de rotinas do processo de compilação. A sua função é determinar a forma como os dados (variáveis) são manipulados no decorrer de um programa, função ou bloco de código. São várias as informações que este tipo de análise e respectivas estruturas de dados permitem obter, tais como: as posições onde ocorrem as definições de uma variável; as posições onde se utiliza determinada definição de uma variável; as posições das definições que alcançam determinada utilização; e que variáveis alcançam determinada posição do programa.

Nas secções que se seguem apresenta-se a forma como o BEDS se propõe implementar os mecanismos necessários ao processo de análise do fluxo de dados, na tentativa de obter, entre outras, as informações atrás referidas.

5.2.1 Listas de definições

As listas de definições guardam para cada variável o conjunto das posições do programa, função ou bloco de código, onde se (re)definem as respectivas variáveis, isto é, onde lhes é (re)atribuído um valor. Para tal e como já se viu, na classe *CellIDTable* definiu-se uma variável designada por *IDreg*, que armazena todos os *DataTransfers* cujas expressões dos LHS são *Registers* que representam a variável associada ao *CellIDTable*. Desta forma é possível conhecer todos as posições do programa onde uma dada variável é (re)definida.

Por forma a simplificar outras fases da análise do fluxo de dados e aumentar a eficiência das pesquisas, optou-se por utilizar não uma lista de *DataTransfers*, mas sim uma lista de (*FlowNode*, *DataTransfer*). Desta forma, dado um *IDreg* de uma variável e um *FlowNode* é possível determinar todos os *DataTransfers* que a definem nesse nodo. De notar que não é obrigatório que assim seja, visto ser possível determinar a partir de um qualquer *DataTransfer* o *FlowNode* ao qual pertence.

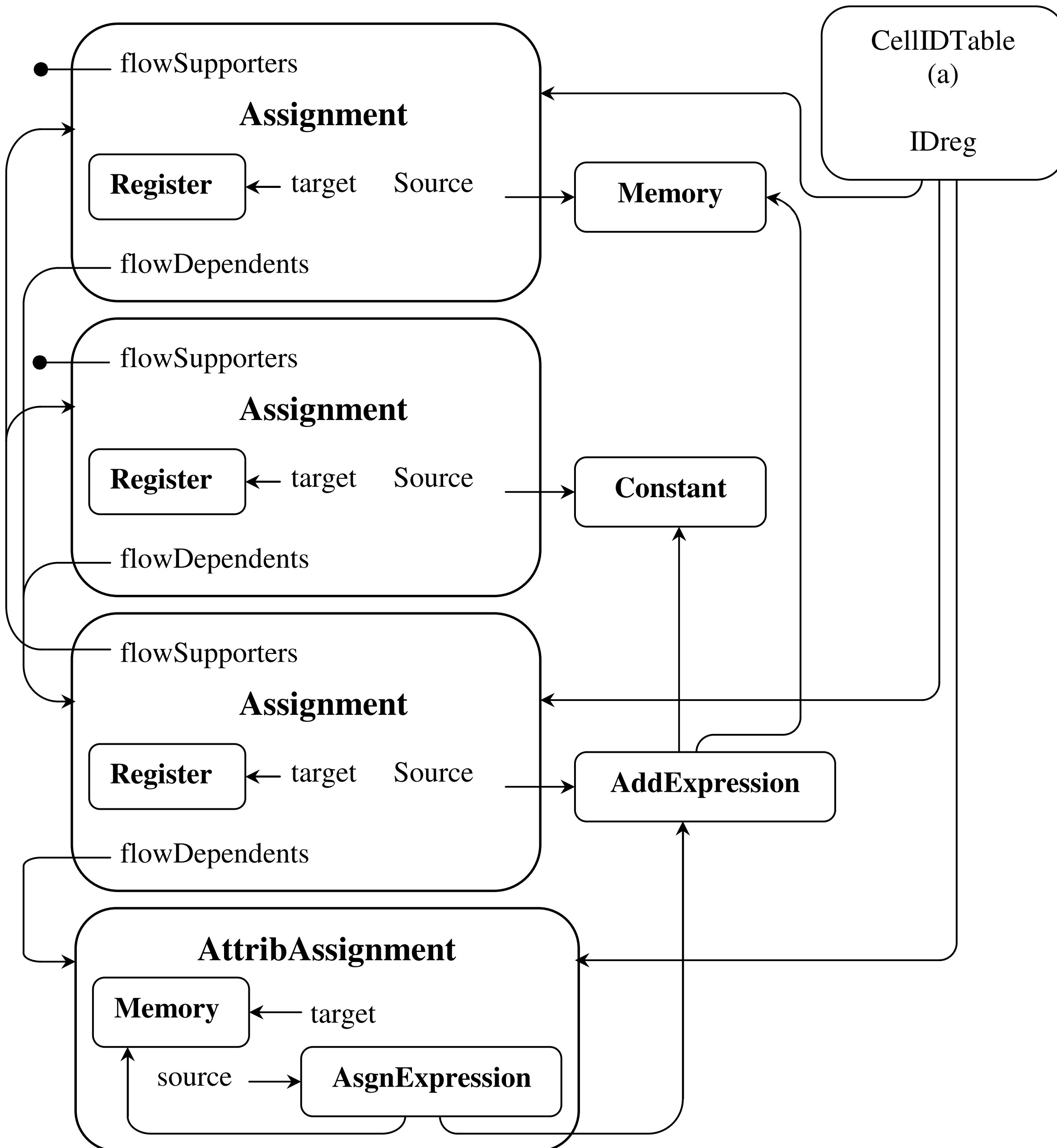


Fig. 5.12 – Exemplo da representação MIR da expressão a = a+1.

5.2.2 DU-Chain e UD-Chain

Outro tipo de informação, que a análise do fluxo de dados e respectivas estruturas deve fornecer, é, para cada definição de uma variável, a lista das posições onde esta é utilizada, a qual se designa por *Definition Uses - Chain*, ou simplesmente por *DU-Chain*.

O BEDS não implementa nenhum mecanismo dinâmico para determinar este tipo de informação, opta antes por um sistema estático. Para tal, existe uma estrutura de dados associada a cada *DataTransfer*, designada por *flowDependents*, a qual foi apresentada em capítulos anteriores, onde se descreveu como sendo a lista dos *DataTransfers* onde é utilizado o resultado do *DataTransfer* ao qual *flowDependents* está associada.

A sua gestão é relativamente simples, uma vez que é automaticamente preenchida quando o resultado do respectivo *DataTransfer* é utilizado por um outro *DataTransfer*. Para tal, há que lembrar que cada *DataTransfer* possui uma variável, designada por *source* a qual representa uma *Expression*. Esta por sua vez descreve uma árvore de expressões que pode possuir um ou dois descendentes, os quais também são *Expressions* e também estão associados a *DataTransfers*. Desta forma, quando se faz a atribuição de *source*, determinam-se os seus descendentes e os respectivos *DataTransfers*, aos quais se actualizam os *flowDependents*.

Consegue-se assim, manter esta estrutura sempre actualizada, desde que ao destruir-se um *DataTransfer* este seja removido de todos os *flowDependents*. Para tal, é necessário que cada *DataTransfer* conheça quais os *DataTransfers* responsáveis pelos resultados por ele utilizado, o que se consegue associando a este uma segunda lista, designada por *flowSupporters*, que é gerida em simultâneo com o *flowDependents*. A Fig. 5.12 exemplifica, para a expressão: a = a +1, a relação entre estas estruturas.

Existe ainda outro tipo de informação que a análise do fluxo de dados deve fornecer, que é a lista das definições que alcançam determinada utilização de uma variável, o que é vulgarmente designado por *User Definitions – Chain*, ou *UD-Chain*.

No entanto, como a utilização do SSA garante que em qualquer ponto de um programa só se pode encontrar presente uma definição por variável, significa que as *UD-Chains* nestas circunstâncias são compostas apenas por um ou nenhum elemento. Pelo que se optou por não determinar directamente esta informação, uma vez que dado um *DataTransfer* é sempre possível determinar a definição de uma variável por ele utilizada, através do *flowSupporter*. No entanto é de acentuar que este último não é uma *UD-Chain*.

5.2.3 Iterative Data Flow Analysis

Apesar dos mecanismos até aqui descritos já permitirem, por si só, fornecer bastante informação, não se pode ainda falar de um processo de análise. Para além de mais, espera-se que a análise do fluxo de dados permita obter mais informação do que aquela que os mecanismos até aqui apresentados fornecem, como por exemplo, determinar o alcance das variáveis num programa.

De forma genérica, a análise do fluxo de dados deve permitir controlar o processo evolutivo de determinadas características associadas aos dados, ao longo da execução de um programa. Uma das quais é a própria presença (alcance) da variável, ou seja, determinar quando é que esta está presente ou deixa de estar ao longo da execução do

programa. No entanto, não se deve restringir a análise do fluxo de dados a este exemplo, pois existem outras características inerentes aos dados cuja evolução pode ser traçada através deste tipo de análise, como por exemplo, a evolução do valor das variáveis.

Existem diversas formas de implementar a análise do fluxo de dados, recorrendo inclusive à análise estrutural e à análise de intervalos, apresentadas em secções anteriores. O método utilizado pelo BEDS é um dos mais simples de perceber e de implementar; designa-se por *Análise Iterativa* e, como se pretende, trabalha sobre grafos orientados. Estes no entanto devem sempre possuir um nodo inicial e um nodo final, os quais serão posteriormente reconhecidos respectivamente por *startNode* e *endNode* (variáveis de *FlowNode*). Caso o grafo contenha mais do que um nodo de saída, é necessário reestruturá-lo de forma a que todos esses nodos passem apontar para um único nodo (de saída). A descrição que se faz desta análise e das respectivas definições foi retirada do capítulo 8 do livro *Advanced Compiler Design and Implementation* [Much97], mas o algoritmo final, utilizado na análise do alcance das variáveis é da autoria de Gary Kildall [Kild73].

Conceito de reticulado e funções de fluxo

A análise do fluxo de dados processa-se operando sobre elementos que representam características das variáveis ou expressões. Os quais, devem ser independentes dos valores de entrada e do fluxo de execução de um programa (em *run time*). À estrutura algébrica composta por esses elementos, designa-se por reticulado.

Este assume formas distintas, mediante o tipo da análise a implementar, pelo que, deve ser visto como um conceito abstracto, pelo menos, enquanto não se especificar concretamente o tipo de características a controlar com a sua utilização.

Assim sendo, define-se abstractamente, um reticulado L como sendo um tipo de entidade, que comporta um conjunto de elementos, que permite caracterizar o estado da(s) característica(s) a controlar em determinada posição do grafo.

Os operadores dos reticulados, são definidos conforme o tipo dos elementos que o compõem. Mas, praticamente todos os reticulados suportam os dois seguintes operadores abstractos: o *meet* e o *join*, representados respectivamente por $\sqcap$ e por $\sqcup$.

No caso da análise do alcance (presença) das variáveis, os reticulados são implementados com base em vectores de bits, onde o conjunto de todos os reticulados é representado por BV^n, em que n indica a dimensão dos vectores. Cada bit representa uma definição de uma variável, o mesmo é dizer que, o valor de n depende do número de definições que ocorrem no programa. Cada vector está normalmente associado a um nodo do programa.

As operações *meet* e *join* são neste caso, implementadas através dos operadores lógicos de conjunção e disjunção. No entanto, como os reticulados são sequências de bits, não nos interessa realizar estas operações sobre os reticulados em si, mas sim sobre os elementos que as compõem, o que é equivalente a realizar para cada um dos operadores (conjunção e disjunção) o produto vectorial entre reticulados,.

Define-se então, a conjunção entre dois reticulados L_1 e L_2, que se representa por $L_1 \wedge_L L_2$, como:

$$<x_1,..., x_n> \wedge_L <y_1,..., y_n> = < x_1 \wedge y_1, ..., x_n \wedge y_n> \qquad \text{Eq. 5.1}$$

onde $\wedge$ é a operação de conjunção entre valores binários.

A disjunção entre reticulados, representada por $L_1 \vee_L L_2$, define-se de forma semelhante, ou seja:

$$<x_1,..., x_n> \vee_L <y_1,..., y_n> = < x_1 \vee y_1, ..., x_n \vee y_n> \quad \text{Eq. 5.2}$$

onde $\vee$ é a operação de disjunção entre valores binários.

Genericamente, diz-se que, para um qualquer reticulado *L*, existem dois elementos, únicos, designados por *top* e *bottom* e representados respectivamente por $\top$ e por $\perp$, tal que:

$$\forall\ x \in L,\ x \sqcup \top = \top \text{ e } x \sqcap \perp = \perp \quad \text{Eq. 5.3}$$

No caso da análise do alcance das variáveis, o mesmo será dizer que:

$$\forall\ x \in L,\ x \vee \top = \top \text{ e } x \wedge \perp = \perp \quad \text{Eq. 5.4}$$

onde os valores de *top* e *bottom* que respeitam a equação anterior, são respectivamente 1 e 0.

O *top* simboliza o valor dos elementos do reticulado no início do primeiro nodo do programa (*startNode*) e o *bottom* o valor dos elementos do reticulado no fim do último nodo do programa (*endNode*). No presente caso, pode-se dizer que se um dado elemento (bit) de um reticulado está a zero, então a definição associada a esse elemento do reticulado, alcança a posição (nodo) do programa onde se encontra localizado o reticulado, caso contrário o bit está a um.

Como anteriormente se referiu, os reticulados existem como forma de representar a evolução das componentes de um programa, quer estas sejam variáveis, expressões, ou qualquer outra forma de representação da informação. A sua distribuição ao longo de um programa varia consoante o tipo da componente a controlar, podendo estar posicionadas, por exemplo, no início, ou no fim de cada nodo, em cada expressão, etc. Mas traduzindo sempre a evolução das componentes a analisar, através das diferenças que existem de reticulado para reticulado e que resultam das transformações impostas pelo programa nas componentes a analisar. As funções que descrevem as diferenças de reticulado para reticulado, são designadas por *funções de fluxo* e formalmente definidas da seguinte forma:

$$F_B: L \rightarrow L \quad \text{Eq. 5.5}$$

Em que *L* representa o conjunto dos reticulados e *B* representa o conjunto de instruções do programa que é responsável pela transformação F_B. Ou seja, para cada conjunto de instruções entre dois quaisquer reticulados *A* e *B*, tal que *A* precede *B* (entre reticulados e segundo o fluxo do programa), existe sempre uma *função de fluxo* que traduz o efeito destas instruções sobre um qualquer valor do reticulado *A*, de forma a obter o valor do reticulado *B*.

O objectivo da análise do fluxo de dados consiste então, em determinar o valor final de todos os reticulados do programa, o que normalmente se designa por solução MOP – *meet over-all paths*. Para tal, começa-se por escolher um possível valor a atribuir ao reticulado de entrada (do programa), a partir do qual se determinam os restantes reticulados, utilizando as *funções de fluxo* que ocorram ao longo de todos os caminhos possíveis do grafo, desde o *startNode* e de nodo em nodo até chegar ao *endNode*. E sempre que, para um dado nodo onde exista pelo menos um reticulado, convirjam dois ou mais caminhos, realiza-se o *meet* entre os reticulados precedentes que alcançam esse nodo.

Formalmente define-se MOP da seguinte forma: dado um grafo *G*, formado por um conjunto de *FlowNodes*; um conjunto P_B com todos os caminhos possíveis que vão desde o *startNode* até à posição *B*; uma função de fluxo $F_B(...)$ correspondente ao bloco de instruções associado a *B*; uma função $F_P(...)$, que represente a composição das funções de fluxo que ocorrem ao longo de um dos caminhos entre o *startNode* e a posição *B*, tal que:

$$F_P = F_{Bn} \circ ... \circ F_{B1} \qquad \text{Eq. 5.6}$$

e o valor do reticulado inicial do programa, representado por $V_{inicial}$, define-se então a MOP da seguinte forma:

$$MOP(B) = \bigcap_{p \in P_{B_i}} F_P(V_{inicial}), B_i = startNode, ..., B_n \qquad \text{Eq. 5.7}$$

O algoritmo da Análise Iterativa

Após a apresentação de alguns conceitos fundamentais aos processos de análise do fluxo de dados e mais concretamente da análise do alcance das variáveis, é chegada a altura de ver como se procedeu à sua implementação. Mas antes, faz falta definir a noção de alcance. Diz-se então que, uma variável alcança determinado ponto do programa, função ou bloco de código, se e só se, existir pelo menos um caminho entre uma qualquer definição dessa variável e esse ponto.

Praticamente todas as formas de análise, dentro das quais se inclui a *Análise Iterativa*, que pretendam obter este tipo de informação, recorrem à utilização de reticulados compostos por sequências de bits, posicionados à entrada ou à saída de cada bloco simples de código. Cada bit da sequência representa a definição de uma variável e mediante o seu valor, assinala se esta alcança ou não a posição onde se encontra posicionada a sequência.

A implementação desta forma de reticulados sobre a estrutura da MIR, é feita associando a cada *FlowNode* um vector de bits, designado por *in*, que indica quais são as variáveis que alcançam o início de cada *FlowNode*. Para esse vector define-se a seguinte função:

$$AliveIn: FlowNode \rightarrow BitVector \qquad \text{Eq. 5.8}$$

que recebe como parâmetro um *FlowNode* e devolve o vector de bits *in*.

É de notar que os vectores de bits estão associados aos *FlowNodes* e não aos *DataTransfers* como seria o ideal. É que se assim não fosse, seria necessário uma quantidade substancial de memória, sem que tal acarretasse um grande acréscimo na informação fornecida. No entanto, os mecanismos fornecidos pelo BEDS permitem que se determine este tipo de informação para cada *DataTransfer*, recorrendo a uma solução mista, que consiste em determinar que variáveis é que alcançam cada nodo e depois, localmente e de forma dinâmica, determinar quais as que alcançam determinado *DataTransfer*.

Parte-se do princípio que inicialmente as variáveis não alcançam qualquer ponto do programa, o mesmo será dizer que os vectores de bits estão todos preenchidos com uns, de tal forma que:

$$\text{AliveIn}(i) = \mathbf{1} = <1...1> , \forall i \in \{\textit{FlowNode}\} \qquad \text{Eq. 5.9}$$

e que é sempre verdade que:

$$\text{AliveIn}(\text{startNode}) = \mathbf{1} = <1...1> \qquad \text{Eq. 5.10}$$

Define-se ainda, uma função que dado um nodo devolve um vector de bits assinalando que variáveis são (re)definidas dentro desse nodo (0- (re)definida, 1- inalterada), da seguinte forma:

$$\text{Def: FlowNode} \rightarrow \text{BitVector} \qquad \text{Eq. 5.11}$$

O que não é mais do que a função de fluxo do respectivo *FlowNode*. De notar que *Def(...)* não assinala mais do que o início de cada definição.
O vector de bits que se obtém à saída do *FlowNode* resulta da operação de *meet* entre o resultado de *Def(...)* e o resultado de *AliveIn(...)*, de tal forma que:

$$\text{AliveOut: FlowNode} \rightarrow \text{BitVector} \qquad \text{Eq. 5.12}$$

$$\text{AliveOut}(B) = \text{AliveIn}(B) \sqcap \text{Def}(B) \qquad \text{Eq. 5.13}$$

ou seja:

$$\text{AliveOut}(B) = \text{AliveIn}(B) \wedge_L \text{Def}(B) \qquad \text{Eq. 5.14}$$

O algoritmo é obtido a partir da iteração do seguinte conjunto de equações, para todos os nodos do grafo e até que se obtenha uma solução estável para todos os reticulados:

$$\text{AliveIn}(i) = \begin{cases} V_{inicial}, & i = \textit{startNode} \\ \bigcap_{p \in inEdges(i)} \text{AliveOut}(p), & i \neq \textit{startNode} \end{cases} \qquad \text{Eq. 5.15}$$

O que resulta no seguinte algoritmo:

Proc DF_IterateAnalyse (inout: N, in: startNode, in: $V_{inicial}$)
N :Set(FlowNode)
startNode :FlowNode
$V_{inicial}$:Reticulado
Início
    b, p : FlowNode
    WorkList :Queue(FlowNode)
    effect, totaleffect : Reticulado
    Para ∀ p ∈ N \ {startNode} Fazer

```
        Enqueue( WorkLst, p)
        AliveIn(b) ← T
    FimPara
    AliveIn(startNode) ← V_inicial
    Repetir
        b ← Dequeue( WorkList,)
        totaleffect ← T
        Para ∀ p ∈ Antecessores(b) Fazer
            effect ← AliveOut(p)
            totaleffect ← totaleffect ⊓ effect
            Se AliveIn(b) ≠ totaleffect Então
                AliveIn(b) ← totaleffect
                Enqueue( WorkList, b)
            FimSe
        FimPara
    Até Que WorkList = ∅
Fim
```

Fig. 5.13 – Algoritmo para determinar o alcance das variáveis.

O algoritmo começa por criar um *queue* auxiliar, a *WorkList*, com todos os nodos do grafo excepto o *startNode*, depois inicializa os reticulados, com $V_{inicial}$ para o *startNode* e com *top* para os restantes. À partida estes dois valores (*top* e $V_{inicial}$) são iguais, mas pode acontecer que se pretenda controlar a evolução das variáveis que são definidas como parâmetros ou externamente ao grafo, pelo que $V_{inicial}$ deve traduzir o estado dessas variáveis.

De seguida o algoritmo entra na fase iterativa, onde começa por remover um elemento de *WorkList* e determinar através dos nodos precedentes, que variáveis alcançam o nodo em causa. Sempre que ocorrem alterações no respectivo reticulado, o nodo é reposto na *WorkList* para voltar posteriormente a ser processado. O algoritmo termina quando os valores dos reticulados estabilizarem.

O BEDS não se fica por esta forma de análise; é que, após determinar o alcance das variáveis, o BEDS permite determinar o período de vida útil das mesmas. De reparar que este último tipo de análise tem objectivos diferentes da anterior, dado que o período de vida útil de uma variável termina após a sua última utilização. Pelo que se considera que uma variável está "viva" num determinado ponto *B* do grafo, se e só se existir uma definição dessa variável que alcance esse ponto e se existir pelo menos uma utilização a partir desse ponto (inclusive).

Para tal foi necessário definir mais uma função, designada por *Use(...)*, a qual permite determinar para um dado nodo quais as variáveis (definições) nele utilizadas (0- utilizada, 1- não utilizada), da seguinte forma:

$$\text{Use: FlowNode} \rightarrow \text{BitVector} \qquad \text{Eq. 5.16}$$

Uma vez que todas as variáveis alcançam o *endNode*, então o resultado da análise do alcance resulta num reticulado com todos os valores a $\perp$. É este o ponto de partida para se realizar a análise do período de vida útil.

A ideia base consiste em percorrer o grafo, desde o *endNode* até ao *startNode*, testando que variáveis (definições) estão a ser utilizadas, mantendo apenas essas como

activas, as restantes deixam de o estar até que se encontre uma utilização ou respectiva definição.

Para esta forma de análise, continua-se a utilizar o mesmo tipo de reticulados da análise anterior, ou seja vectores de bits, também eles implementados com base nas estruturas e funções da análise anterior, nomeadamente com base no vector *in* e nas funções *Def(...)* e *AliveIn(...)*.

A MOP é obtida aplicando iterativamente o seguinte conjunto de instruções a todos os nodos do grafo (excepto o *startNode*) até se obter uma solução estável:

$$\text{AliveIn(i)} = \begin{cases} ((\bigcap_{s \in \text{outEdges}(i)} \text{AliveIn(s)}) \vee \text{AliveIn(i)}) \wedge \text{Use(i)}, & i \neq endNode \\ \\ \text{AliveIn(i)} \wedge \text{Use(i)}, & i = endNode \end{cases} \quad \text{Eq. 5.17}$$

O que resulta no seguinte algoritmo:

```
Proc DF_AliveAnalyse ( inout: N, in: startNode, in: V_inicial)
N :Set(FlowNode)
startNode :FlowNode
V_inicial :Reticulado
Início
    b, p : FlowNode
    WorkList :Queue(FlowNode)
    effect, totaleffect : Reticulado
    Para ∀ p ∈ N \{startNode} Fazer
        Enqueue(WorkList , p)
    FimPara
    Repetir
        b ← Dequeue( WorkList,)
        totaleffect ← T
        Para ∀ p ∈ Sucessores(b) Fazer
            Se p = endNode Então
                effect ← T
            Senão
                effect ← AliveOut(p)
            FimSe
            totaleffect ← totaleffect ⊓ effect
        FimPara
        totaleffect ← (totaleffect ∪ AliveIn(b)) ⊓ Use(b)
        Se AliveIn(b) ≠ totaleffect Então
            AliveIn(b) ← totaleffect
            Enqueue( WorkList, b)
        FimSe
    Até Que WorkList = ∅
Fim
```

Fig. 5.14 – Algoritmo para determinar o período de vida das variáveis.

No caso do BEDS, a implementação de ambas as formas de análise, necessita de algo mais que permita relacionar as variáveis da tabela de identificadores e respectivas definições com as posições no vector de bits. Tal é feito de forma dinâmica e deve ser realizado antes de se proceder a qualquer um dos processos de análise do fluxo de dados. O processo em si é muito simples, consiste em percorrer o *IDreg* de cada um dos identificadores, atribuindo um número de ordem a cada uma das definições. Como tal e de forma a obter-se a máxima eficiência nos processos de análise, é conveniente que todas as variáveis se encontrem declaradas na tabela de identificadores, quer estas sejam variáveis explícitas, variáveis artificiais, resultados de sub-expressões ou constantes cujo valor não possa ser utilizado directamente.

Muito mais há para dizer sobre a análise do fluxo de dados, podendo-se mesmo afirmar que o aqui foi exposto é apenas o início. No entanto e com o objectivo de não tornar mais extensa esta descrição, decidiu-se apenas apresentar o que já foi implementado no BEDS. Mais alguns detalhes serão descritos em secções posteriores, nomeadamente na apresentação das optimizações e da alocação global.

5.3 *Single Static Assignment* – Implementação

Após se ter apresentado a SSA e respectivas vantagens e desvantagens no capítulo 4.2, e uma vez que, já se descreveu grande parte da MIR, é então chegado o momento de apresentar os mecanismos e respectivas implementações, fornecidos pelo BEDS, para se obter este tipo de representação.

Como já foi dito, parte-se do princípio que a representação SSA é determinada a partir da representação normal, onde o resultado de cada expressão é atribuído a uma variável ou utilizado para determinar uma condição de teste.

O processo normal de conversão desenrola-se em três fases: na primeira fase inserem-se as funções Φ(...); na segunda fase, cada uma das variáveis é “instanciada”, tantas quantas as vezes as suas definições (atribuições), tendo já em conta as funções Φ(...), uma vez que estas também são definições; e na terceira fase actualizam-se as referências em relação às instâncias de cada variável.

No BEDS o processo encontra-se simplificado, uma vez que, ao se descrever um programa em MIR, de imediato se garante que cada variável possui uma e uma só definição. Para tal, há que relembrar que, as definições (atribuições às variáveis) são descritas através da classe *AttribAssignment*, onde o valor atribuído é sempre determinado previamente através de um *Assignment*, como se encontra representado na Fig. 4.2. Como cada *Assignment* possui do LHS o seu próprio *Register*, o qual, ao representar o resultado da variável e ao ser único, permite representar uma “instância” singular da variável por cada definição.

O processo de conversão para uma representação SSA fica assim, reduzido à inserção das funções Φ(...) e ao reescrever das expressões em relação a estas.

5.3.1 Dominance Frontiers

Na inserção das funções Φ(...), o primeiro problema é determinar onde é que estas devem ser inseridas. Uma solução simples, consiste em colocar por cada variável e em

todos os nodos para onde convirjam dois ou mais caminhos uma função Φ(...). Mas facilmente se percebe, que se trata de uma solução extrema, uma vez que, nem em todos os nodos necessitam de uma função Φ(...) por cada variável.

A solução mais eficiente consiste em colocá-las, em posições do grafo, designadas por *Dominance Frontiers*, que não são mais do que nodos do grafo alcançados por duas ou mais definições de uma mesma variável. Esta solução designa-se por *Minimal SSA Form*.

Para uma explicação mais formal do conceito de *Dominance Frontiers*, faz falta definir o significado de nodo dominador (*Dominator*). Trata-se de um conceito que se aplica aos grafos e que é definido da seguinte forma: um nodo *a* domina um nodo *b*, que se representa por *a dom b*, se e só se, todos os caminhos possíveis que vão desde o nodo inicial do grafo (*startNode*) até o nodo *b*, incluírem o nodo *a*. Diz-se então, que *a* pertence ao conjunto dos nodos dominadores de *b*, o qual se representa por *dom(b)*.

Define-se ainda que *a* é o dominador imediato de *b*, com $a \neq b$, e representa-se por *a idom b*, se e só se, *a dom b* e não existe nenhum nodo *c*, tal que $c \neq a$ e $c \neq b$, para o qual *a dom c* e *c dom b*. Nestas circunstâncias, *a* pode também ser designado por *idom(b)* e é sempre único.

Define-se ainda que *a domina estritamente (strictly dominates) b* e representa-se por *a sdom b*, se e só se, *a dom b* e $a \neq b$.

Sejam então, *A* e *B* nodos de um grafo, tal que em *A* ocorre uma definição da variável *V*, representada por V_1 e *A sdom B*, então qualquer expressão de *B* vê a variável *V* como sendo V_1. Supondo agora um terceiro nodo *C*, tal que $C \in Sucessores(B)$, mas *B não sdom C*, o que implica, que *C* possui vários nodos precedentes, pelo que, pode ocorrer que *C* veja para além de V_1 outras definições de *V*. Diz-se então que *C* é uma *dominance frontier* de *A*, pelo que necessita obrigatoriamente de uma função Φ(...) para a variável *V*.

Por questões de eficiência computacional, não se pretende determinar as *dominance frontiers* em relação a cada uma das variáveis, mas sim em relação a cada nodo do grafo, o que permite uma solução independente das definições. Caso contrário, por cada nova função Φ(...) inserida e uma vez que estas são definições, é necessário recalcular novamente as *dominance frontiers*.

Diz-se então que, a *dominance frontier* do nodo *X*, a qual se representa por *DF(X)*, é o conjunto de todos os nodos *Y*'s, tais que, *X* é dominador de um dos antecessores de *Y*, mas *X* não domina estritamente *Y*.

$$DF(X) = \{Y : \exists\, P \in \text{Antecessores}(Y), (X \text{ sdom } P) \wedge \neg(X \text{ sdom } Y)\} \qquad \text{Eq. 5.18}$$

O método aqui utilizado para determinar as *dominance frontiers* foi apresentado por Chaitin *et al.* [CFRWZ91] e tem por base a árvore de dominadores (*dominator tree)*, que tal como o nome indica é uma estrutura em forma de árvore que contém os mesmos nodos do grafo, mas onde cada um possui como ascendente o nodo que imediatamente o domina, ou seja, o ascendente do nodo *X* na árvore é sempre o *idom(X)*.

Apesar de ser possível determinar *DF(X)* a partir da definição anterior e da árvore de dominadores, ficou provado que a ordem desse algoritmo é quadrática. Chaitin, propôs então, uma solução de ordem linear, provando que, para *DF(X)* contribuem dois tipos de nodos, uns que serão designados por $DF_{local}(X)$ e outros por $DF_{up}(X,Z)$, de tal forma que:

$$DF(X) = DF_{local}(X) \cup \bigcup_{Z \in Children(X)} DF_{up}(X,Z) \qquad \text{Eq. 5.19}$$

Os $DF_{local}(X)$ são os sucessores de X, que não são estritamente dominados por X, tal que:

$$DF_{local}(X) = \{ Y \in Sucessores(X): \neg(X \text{ sdom } Y)\} \qquad \text{Eq. 5.20}$$

O que pode ser reescrito da seguinte forma:

$$DF_{local}(X) = \{ Y \in Sucessores(X): idom(Y) \neq X \} \qquad \text{Eq. 5.21}$$

Os $DF_{up}(X,Z)$ são os nodos, que não sendo estritamente dominados por X, pertencem aos DF dos nodos estritamente dominados por X (que são os nodos descendentes de X na árvore de dominadores), tal que:

$$DF_{up}(X, Z) = \{Y \in DF(Z): \neg (idom(Z) \text{ sdom } Y)\} \qquad \text{Eq. 5.22}$$

o que, também pode ser reescrito da seguinte forma:

$$DF_{up}(X, Z) = \{Y \in DF(Z): idom(Y) \neq X)\} \qquad \text{Eq. 5.23}$$

para todo e qualquer Z cujo dominador imediato seja X, de tal forma que:

$$\bigcup_{Z \in N: idom(Z)=X} DF_{up}(X,Z) \equiv \bigcup_{Z \in Children(X)} DF_{up}(X,Z) \qquad \text{Eq. 5.24}$$

onde os *Children(X)* são os descendentes de X na árvore de dominadores, ou seja, os nodos para os quais X é o dominador imediato.

O algoritmo para determinar as DF de cada nodo obtém-se da Eq. 5.19 e da árvore de dominadores, o que resulta no seguinte:

```
Para cada X ∈ Sequência(travessia bottom-up da árvore de dominadores), fazer:
    DF(X) ← ∅
    Para ∀ Y ∈ Sucessores(X) Fazer
        Se idom(Y) ≠ X Então
            DF(X) ← DF(X) ∪ {Y}
        FimSe
    FimPara
    Para ∀ Z ∈ Children(X) Fazer
        Para ∀ Y ∈ DF (Z) Fazer
            Se idom(Y) ≠ X Então
                DF(X) ← DF(X) ∪ {Y}
            FimSe
        FimPara
    FimPara
```

Fig. 5.15 – Algoritmo para determinar as DF através da árvore de dominadores.

De notar que, através da equivalência da Eq. 5.24 é possível determinar as *DF*, sem que, para tal seja necessário construir a árvore de dominadores, da seguinte forma:

```
Proc DF_AliveAnalyse( in: N, in: endNode) :Dictionary( FlowNode, Set( FlowNode))
N : Set(FlowNode)
endNode : FlowNode
Início
    DF : Dictionary( FlowNode, Set(FlowNode))
    x, y : FlowNode
    W, SW :Set(FlowNode)
    more :Booleano
    W ← {endNode}
    SW ← ∅
    Para ∀ x ∈ N Fazer
        DF(x) ← ∅
    FimPara
    Repetir
        more ← False
        Enquanto W ≠ ∅ Fazer
            x ← W.get( )
            W ← W – {x}
            SW = SW ∪ {x}
            DF(x) ← Sucessores(x)
            Para ∀ y ∈ IDom(x) Fazer
                DF(x) ← DF(x) ∪ DF(y)
            FimPara
            DF(x) ← DF(x) – IDom(x)
        FimEnq
        Enquanto SW ≠ ∅ Fazer
            x ← SW.get( )
            SW ← SW – {x}
            Para ∀ y ∈ Antecessores(x) Fazer
                Se DF(y) = ∅ Então
                    W ← W ∪ { y }
                    more ← True
                FimSe
            FimPara
        FimEnq
    Até Que more = False
Fim
```

Fig. 5.16 – Algoritmo para determinar as DF sem a árvore de dominadores.

Foi esta última, a solução utilizada no BEDS que se encontra implementada na classe *DFSet* do módulo SSA, a qual por sua vez, utiliza as classes *IDom* e *IDomSet*, para determinar para cada nodo, o conjunto de nodos dos quais é dominador imediato.

5.3.2 Inserção das Funções Φ(...)

O processo de inserção das funções Φ(...) é realizado para cada uma das variáveis de forma independente, determinando-se, para cada uma, o conjunto de nodos do grafo onde ocorrem as respectivas definições. As funções Φ(...) são então inseridas nas *dominance frontiers* de cada um desses nodos. Convém no entanto, relembrar que as próprias funções Φ(...) contribuem para o número de definições de cada uma das variáveis.

A inserção das funções Φ(...) passa por pegar em cada definição de uma variável e testar se já existe uma função Φ(...) para cada um dos nodos pertencentes à DF do nodo onde ocorre a definição. Caso exista, basta então acrescentar a instância da variável proveniente da actual definição aos parâmetros da função Φ(...). Caso contrário, é necessário que antes se insira a função Φ(...). O algoritmo final para colocar as funções Φ(...) e respectivos parâmetros, é o seguinte:

```
Proc PlacePhyFunctions ( inout: N, in: DF)
N :Set(FlowNode)
DF : Dictionary( FlowNode, Set(FlowNode))
Início
    IterCount ← 0
    Para ∀ x ∈ N Fazer
        HasAlready(x) ← Work(x)← 0
    FimPara
    W ← ∅
    Para ∀ v ∈ Tabela de Identificadores Fazer
        IterCount ← IterCount +1
        Para ∀ x ∈ Definitions(v) Fazer
            Work(x) ← IterCount
            W ← W ∪ { x }
        FimPara
        Enquanto W ≠ ∅ Fazer
            x ← W.get( )
            Para ∀ y ∈ DF(x) Fazer
                Se HasAlready(x) < IterCount Então
                    InserirFunção( x, v, y)
                    HasAlready(y) ← IterCount
                    Se Work(y) < IterCount Então
                        Work(y) ← IterCount
                        W ← W ∪ { y }
                    FimSe
                Senão
                    InserirArgumento( x, v, y)
                FimSe
            FimPara
        FimEnq
    FimPara
Fim
```

Fig. 5.17 – Algoritmo para inserir as funções Φ(...).

Esta função encontra-se implementada através da classe *PlacePhyFunctions* do módulo SSA. A função *InserirFunção(x,v,y)* representa o procedimento necessário para inserir uma função Φ(...) para a variável *v*, no nodo *y*, com um argumento proveniente da definição de *v* no nodo *x*. A função *InserirArgumento(x,v,y)* acrescenta um argumento à função Φ(...) da variável *v*, proveniente da definição desta no nodo *x*.

5.3.3 Actualização das referências

O passo seguinte, consiste em actualizar as referências feitas às definições e que entretanto foram substituídas (parcialmente) pelas funções Φ(...), bem como actualizar os respectivos *flowSupporters* e *flowDependents*. Qualquer um destes procedimentos poderia ter sido feito aquando da inserção das funções Φ(...), no entanto, tal solução mostrou-se extremamente complexa e de difícil implementação, pelo que se optou apenas por actualizar as referências após a inserção das funções.

O processo em si consiste em percorrer o grafo segundo o fluxo de controlo a partir de cada um dos *PhyAssignments* inseridos (*DataTransfer* das funções Φ(...)), até que se encontre uma outra definição da variável associada à função Φ(...) ou se alcance o *endNode*. Actualizando entretanto, as expressões e os respectivos *flowDependents* e *flowSupporters*. O algoritmo encontra-se implementado através da função *ActualizeExpressions* do módulo SSA.

5.3.4 De SSA para a Forma Normal

Apesar da representação SSA permitir utilizar técnicas de análise muito eficientes, quer a nível das optimizações ou da alocação, torna-se no entanto inevitável que, a partir de determinada altura do processo de compilação se reverta a representação para a forma normal, ou pelo menos se "elimine" as funções Φ(...). É que estas, apesar de serem perfeitamente possíveis numa representação intermédia, não têm no entanto, qualquer significado ao nível do código máquina ou mesmo em *assembly*.

O problema pode ser ilustrado através da Fig. 5.18, onde se encontra representada uma função Φ(...), com três parâmetros, em que cada um, e após o processo de alocação, se encontra fisicamente colocado em posições diferentes. Assim, v_1 está no registo r_x, v_2 num endereço e v_3 no registo r_y. O problema que se coloca é, como se pode fisicamente representar v_4, uma vez que este pode resultar de um qualquer dos três elementos r_x, endereço e r_y.

A solução pode ser extravagante, mas na realidade é a única forma relativamente simples de ultrapassar este problema. Consiste em alocar um registo ou endereço para v_4 e inserir instruções em cada um dos nodos *N1*, *N2* e *N3*, de tal forma que os resultados das respectivas instâncias de *v* sejam colocados no espaço alocado para v_4. A solução para o exemplo da Fig. 5.18 encontra-se representada na Fig. 5.19.

Há assim um acréscimo de instruções que podiam ser dispensadas caso não se utilizasse a representação SSA. Em compensação, é possível minimizar o seu número através de um processo de optimização designado por eliminação de código morto, que permite, nesta circunstâncias, eliminar as funções Φ(...) cujos resultados não são posteriormente utilizados.

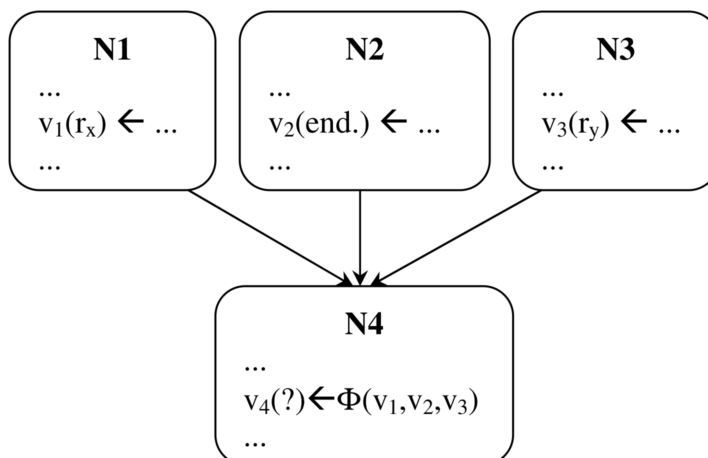


Fig. 5.18 – Exemplo de uma função Φ(...) após o processo de alocação.

É ainda possível minimizar o número de instruções através do próprio processo de alocação global, principalmente se este tiver por base algoritmos por coloração. É que estes algoritmos utilizam grafos de interferências para realizar a alocação, através dos quais é possível fazer com que as várias instâncias de uma variável sejam alocadas a uma mesma posição física, eliminando assim a necessidade de realizar qualquer procedimento para as funções Φ(...). Mais alguns detalhes sobre a possibilidade de utilizar o processo de alocação para minimizar o número de instruções a inserir, serão fornecidos no capítulo 6.

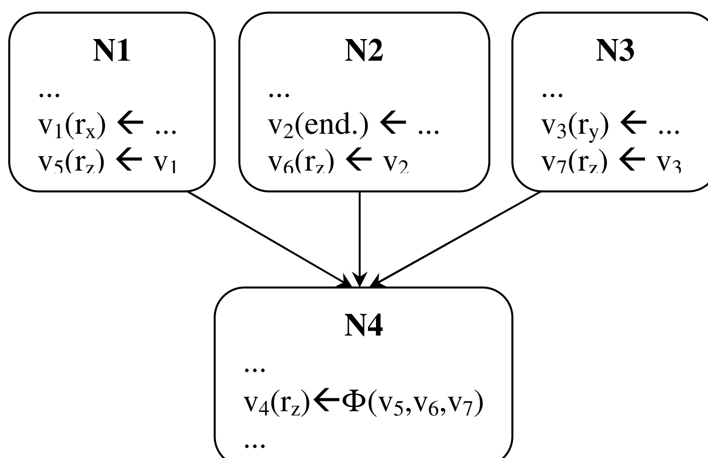


Fig. 5.19 – Exemplo de como se pode resolver a alocação para as funções Φ(...).

5.4 *Optimizações Intermédias*

Até aqui muito se tem falado sobre optimizações, mas na realidade pouco ou nada foi dito sobre estas. No entanto, apresentaram-se praticamente todos os mecanismos necessárias à sua implementação.

De notar que praticamente nenhuma optimização, por si só, permite obter grandes ganhos, quer ao nível da eficiência, quer ao nível do tamanho do código, e que a aplicação de uma optimização pode melhorar um destes factores à custa da degradação de outros. Mais, algumas optimizações devem ser aplicadas por diversas vezes, visto que os efeitos colaterais de outras optimizações podem provocar novas oportunidades de melhoramento. Além disso, a ordem pela qual são aplicadas é também um factor importante para o resultado final.

É importante também perceber que uma optimização só deve ser aplicada se for segura, ou seja, se for garantido que em toda e qualquer situação, as alterações por esta realizadas não modificam o comportamento do programa original. Há ainda que ponderar a sua utilização, pois os benefícios esperados podem não ocorrer para todas as situações possíveis, podendo mesmo em determinadas circunstâncias degradar a qualidade do código final.

Esta secção pretende descrever algumas formas de optimização, tentando dar especial relevo àquelas que já são possíveis de implementar no BEDS e às que facilmente se implementam através da representação SSA.

Convém antes realçar que no BEDS, grande parte do trabalho dentro desta área está por realizar e que as poucas excepções se encontram praticamente por testar (só foram implementadas com o objectivo de verificar o funcionamento das infra-estruturas descritas nas secções anteriores).

Constant Propagation e Constant Folding

O *Constant Propagation* é uma forma de optimização, cujo objectivo é substituir as variáveis pelo seu valor, quando se confirma que este é constante. O *Constant Folding* ou *Constant Expression*, é uma generalização da primeira, em que se substitui o resultado de uma expressão pelo seu valor, quando se prova que este é constante.

Ambas as formas são facilmente implementáveis, principalmente quando se utiliza a representação SSA. No primeiro caso, basta detectar se uma determinada instância de uma variável é constante, tal que:

$$v_x \leftarrow Constante$$

e depois através do *flowDependents*, modificar todos os *DataTransfer* em que v_x é utilizado. No segundo caso, a única coisa que altera é o processo para determinar se v_x é constante, uma vez que este pode resultar de uma qualquer expressão. Para tal, testa-se recursivamente, se os seus descendentes são constantes, se tal acontecer então faz-se a substituição.

Algebraic Simplifications

As propriedades algébricas podem ser utilizadas para simplificar determinadas expressões. Algumas das propriedades mais utilizadas, são:

$i + 0 \equiv 0 + i \equiv i - 0 \equiv i$ $\qquad$ $0 - i = -i$

$i * 1 \equiv i + 1 \equiv i / 1 \equiv i$ $\quad$ $i * 0 = 0 * i = 0$

$-(-i) \equiv i$ $\quad$ $i + (-j) = i - j$

$b \wedge T \equiv T \wedge b \equiv b$ $\quad$ $b \vee F = F \vee b = b$

$i\hat{}2 \equiv i * i$ $\quad$ $2 * i = i + i$

$i * 5 \equiv$ $t_1 = i << 2$, $t_2 = t_1 + i$ $\quad$ $i * 7 \equiv$ $t_1 = i << 3$, $t_2 = t_1 - i$

A implementação deste tipo de optimizações é bastante semelhante à das duas optimizações anteriores, mas aqui, cada caso deve ser tratado individualmente, ou então agrupados segundo determinadas características.

Algebraic Reassociation

Trata-se de um tipo de optimização com base nas propriedades algébricas, mais concretamente na propriedade associativa, comutativa e distributiva. Os exemplos típicos são:

$a * b + a * c \equiv a * (b + c)$

$(i - j) + (i - j) + (i - j) \equiv (4 * i - 4 * j) \equiv 4 * (i - j)$

Convém no entanto salientar que, este tipo de transformações pode criar potenciais situações de *overflow*.

Copy Propagation

Consiste em eliminar expressões do tipo $v_y \leftarrow v_x$, substituindo nas expressões posteriores as utilizações de v_y por v_x.

Também esta é uma das optimizações facilmente implementáveis em SSA, uma vez que basta pegar nos *flowDependents* do *DataTransfer*, correspondente à expressão $v_y \leftarrow v_x$ e substituir a utilização de v_y por v_x e actualizar os respectivos *flowDependents* e *flowSupporters*.

Unreachable Code Elimination

Consiste em eliminar blocos de código (*FlowNodes*), para os quais se prove que em toda e qualquer circunstância estes nunca serão executados, como por exemplo:

```
vi ← 0
...     /*Valor de vi mantém-se constante */
Enquanto vi Fazer
        ...
FimEnq
```

onde facilmente se percebe que a estrutura cíclica nunca será executada, uma vez que a expressão de teste resulta sempre em falso.

O processo pela qual esta optimização avalia se o código é ou não alcançável, pode, na sua forma mais simples, assemelhar-se ao *Constant Folding*, mesmo ao nível da implementação. É no entanto possível utilizar abordagens mais agressivas, que normalmente recorrem a processos de optimização do tipo *Induction Variable*, que é apresentado mais adiante e cuja implementação é bastante mais elaborada.

Uma variante desta forma de optimização é o *If Simplification*, que se especializa, como o nome indica, em estruturas do tipo *If...Then...Else...* .

Common Sub Expression Elimination

Trata-se de uma das mais eficientes formas de optimização, pelo menos para os processadores mais convencionais. Consiste em eliminar sub-expressões iguais, mantendo apenas a primeira ocorrência e modificando as referências posteriores das restantes sub-expressões (entretanto eliminadas). O seguinte exemplo demonstra a aplicação deste tipo de optimização:

$a \leftarrow 4 * i / b$		$t_1 \leftarrow 4 * i$
$c \leftarrow d + 4 * i$	$\equiv$	$a \leftarrow t_1 / b$
		$b \leftarrow d + t_1$

A sua implementação é algo mais complicada que as anteriores, pois necessita dos mecanismos da análise de controlo do fluxo de dados. Onde os reticulados representam o conjunto das expressões que se encontram disponíveis em determinada posição do grafo. Na prática, estes são implementados através de um conjunto de (apontadores para as) expressões.

De notar que uma expressão só está disponível em determinada posição do grafo, se desde a posição onde esta é definida até ao ponto onde está disponível, não ocorre nenhuma definição da variável que representa o seu resultado, nem de nenhum dos seus operandos.

A substituição das sub-expressões passa por verificar se num determinado nodo, existem sub-expressões que já se encontrem representadas no reticulado associado a esse nodo. Se tal ocorrer, então é porque essa sub-expressão já foi previamente calculada e não foi entretanto modificada, o que permite proceder à remoção da sub-expressão, desde que se actualize todas as suas referências posteriores.

Code Hoisting

Trata-se de uma optimização que partilha os mecanismos de análise da optimização anterior, e cujo objectivo, é determinar se existem expressões comuns aos vários caminhos do grafo, que possam ser colocadas em evidência. O exemplo seguinte tenta ilustrar esta situação:

Se a > 0 Então		Se a > 0 Então
a ← a – 1		a ← a - 1
b ← a – c	$\equiv$	Senão
Senão		q ← a + 1
a ← a + 1		FimSe
b ← a – c		b ← a - c
FimSe		

Loop Invariant Code Motion

Consiste em detectar dentro das estruturas cíclicas, expressões que independentemente das iterações do ciclo, produzam sempre o mesmo resultado. Estas, uma vez que se mantêm inalteradas ao longo da execução da estrutura cíclica, podem ser deslocadas para fora desta.

Normalmente este tipo de expressões surgem, não por erro de quem programa, mas como efeito indesejável de outras optimizações e do próprio processo de geração de código. O seguinte exemplo ilustra esta situação:

Código Fonte		**RI s/optimização**		**RI c/optimização**
$i \leftarrow 1$		$i \leftarrow 1$		$i \leftarrow 1$
Enquanto i <100 Fazer	L_1:	$t_1 \leftarrow i > 99$		$t_2 \leftarrow$ addr a
a[i] $\leftarrow i + 1$		Se t_1 goto L_2		$t_3 \leftarrow t_2 - 4$
FimEnq		$t_2 \leftarrow$ addr a	L_1:	$t_1 \leftarrow i > 99$
		$t_3 \leftarrow t_2 - 4$		Se t_1 goto L_2
		$t_4 \leftarrow 4 * i$		$t_4 \leftarrow 4 * i$
		$t_5 \leftarrow t_3 + t_4$		$t_5 \leftarrow t_3 + t_4$
		$i \leftarrow i + 1$		$i \leftarrow i + 1$
		$*t_5 \leftarrow i$		$*t_5 \leftarrow i$
		goto L_1		goto L_1
	L_2:		L_2:	

Esta optimização é relativamente fácil de implementar, desde que se considere que uma expressão só é constante dentro de um ciclo, caso:

- Se trate de uma expressão constante (mesmo fora do ciclo);

- Ou se, as definições de todos os seus operandos não pertencem ao ciclo;

- Ou ainda se, as definições pertencem ao ciclo, mas são elas próprias constantes.

Induction Variable

A indução entre variáveis pode ser definida da seguinte forma: sendo *x* e *y* duas variáveis, diz-se que *x induz y* se, qualquer variação no valor de *x* afecta *y* numa quantidade proporcional.

Facilmente se percebe que através desta propriedade matemática, é possível realizar uma série de simplificações sobre as expressões. Permitindo mesmo, provar que, em determinadas circunstâncias, uma dada condição resulta sempre em verdadeiro ou em falso, sem que para tal esta seja constante. O seguinte exemplo tenta ilustrar esta forma de optimização:

Para i ← 1 Até 10 Fazer		t1 ← 200
a[i] ←200 – 4 * i		Para i ← 1 Até 10 Fazer
FimPara	≡	t1 ← t1 – 4
		a[i] ← t1
		FimPara

De notar que, sempre que *i* aumenta em uma unidade, *a[i]* reduz-se em quatro unidades, pelo que *i induz a[i]*. Aproveitando tal conhecimento, é possível remover para fora do ciclo a parte que não depende de *i*.

Esta é também uma das optimizações que utiliza mecanismos da análise do fluxo de dados.

Dead Code Elimination

Esta optimização já foi referida em secções anteriores, e como se viu, tem por objectivo eliminar código (expressões), cujo resultado não é posteriormente utilizado.

Trata-se de uma das optimizações, que mais facilmente se implementa no BEDS, basta para tal, verificar se o conjunto dos *flowDependents* de cada *DataTransfer*, está vazio, se tal ocorre, significa que não existe nenhuma expressão posterior que utilize este resultado.

Value Numbering

Uma optimização mais elaborada do que a eliminação de sub-expressões comuns, é *Value Numbering*, que tem por objectivo, eliminar expressões equivalentes, mas não necessariamente iguais. O seguinte exemplo ilustra uma situação onde é possível aplicar esta forma de optimização:

```
i ← ...                      i ← ...
j ← i + 1          ≡         j ← i + 1
k ← i                        k ← i
p ← k + 1                    p ← j
```

5.5 *Optimizações do Código Final*

As optimizações cujo efeito é local e se processam sobre um número restrito de instruções, designam-se por *Peephole Optimizations*. Normalmente dependem das características das instruções do processador, pelo que se utilizam normalmente sobre os níveis de representação mais próximos da linguagem final (*assembly* ou código máquina).

Pretende-se nesta secção apenas apresentar alguns casos ilustrativos, possíveis de serem implementados no próprio processo de selecção de instruções e mesmo ao nível da MIR, e que podem acarretar bastantes benefícios para o código final.

Loop Inversion

Esta optimização consiste em transformar estruturas cíclicas do tipo *while...* ou *for...*, em estruturas tipo *repeat...until...* ou *repeat...while...* . Trata-se como tal, de uma optimização que funciona essencialmente sobre o grafo do fluxo de controlo e eventualmente sobre as expressões de teste. No entanto, o seu efeito só pode ser medido a nível da representação final, como ilustra o seguinte exemplo:

Código fonte

```
i ← 1                              i ← 1
a ← 1                              a ← 1
Enquanto i < 100 Fazer     ≡       Repetir
        a ← a * i                          a ← a * i
        i ← i + 1                          i ← i + 1
FimEnq                             Enquanto i < 100
```

RI s/ optimização

	i ← 1
	a ← 1
L_1:	t_1 ← i > 99
	Se t_1 goto L_2
	a ← a * i
	i ← i + 1
	goto L_1
L_2:	

RI c/optimização

	i ← 1
	a ← 1
L_1:	a ← a * i
	i ← i + 1
	t_1 ← i < 100
	Se t_1 goto L_1
L_2:	

Straightening e Branch Optimization

Estas optimizações têm por objectivo eliminar saltos consecutivos, agrupando os blocos de código (nodos) correspondentes a cada salto.

É uma optimização que se implementa ao nível do grafo do fluxo de controlo e é bastante simples. Consiste em detectar no grafo nodos do tipo *JumpNode* consecutivos, em que cada um possua um único antecessor (ver Fig. 5.5) e substitui-los por um único nodo.

Esta optimização não só permite simplificar o grafo como reduzir o número de instruções do código final.

RI s/ optimização

L1:	...
	a ← b+ c
	goto L2
L6:	...
	goto L4
L2:	b ← c * 2
	a ← a + 1
	if c goto L3
L5:	

≡

RI c/optimização

L1:	...
	a ← b + c
	b ← c * 2
	a ← a + 1
	if c goto L3
L6:	...
	goto L4
L5:	...

Strength Reduction

São todas as optimizações que têm por objectivo substituir determinadas instruções por outras de menor custo, como tal, são dependentes das características de cada processador. O seguinte exemplo ilustra este tipo de optimização:

a ← b * 2	≡	a ← b << 2
a ← a + 1	≡	inc a

Existem muitas mais optimizações, mas por uma questão de espaço e de forma a não alongar mais este capítulo sobre optimização, tiveram que ficar de fora.

Espera-se no entanto, que as optimizações aqui apresentadas tenham sido suficientes para ilustrar o que é possível de realizar com estas, bem como as suas as vantagens e as potencialidades que o BEDS fornece para sua implementação.

6 Alocação de Registos

Após a introdução realizada no capítulo 2 sobre a alocação de registos, pretende-se agora apresentar as soluções utilizadas pelo BEDS, na alocação global e local. São ainda descritas todas as infra-estruturas de suporte a estes processos, bem como a interligação com os restantes módulos do BEDS.

6.1 *Alocação global*

Uma das melhores soluções encontradas para a alocação global utiliza um algoritmo de coloração de grafos, através do qual se procura determinar o número mínimo de cores necessárias para colorir todos os vértices do grafo, de forma a que não existam dois vértices adjacentes com a mesma cor. Esta técnica foi utilizada por Chaitin *et al.* [CACCHM81, Chait82] na concepção de um algoritmo para a alocação global de registos. Esta é uma das melhores soluções propostas para resolver a alocação devido à uniformidade com que se propõe tratar as diversas restrições inerentes a cada processador, designadamente em relação aos registos, e por propor uma solução sistemática na resolução da alocação.

Propõe-se ainda retirar o máximo proveito do processo de alocação, maximizando a utilização dos registos. Para tal, delega nas fases de optimização e selecção de instruções a responsabilidade de propor a utilização de registos para o maior número possível de operandos. Assume, ainda a responsabilidade de minimizar o número de instruções adicionais necessárias às operações de *splitting*, reduzindo como tal o número de vezes em que é necessário salvaguardar o conteúdo dos registos para memória de forma a disponibilizar os registos.

NOTA: A partir deste ponto e até ao final do capítulo, as definições de uma variável, as variáveis temporárias e os pseudo-registos, passam a ser designados, só e simplesmente por variáveis. E registo físico apenas por registo.

O algoritmo proposto por Chaitin começa por construir, para cada procedimento ou função, o respectivo grafo de interferências. Onde cada vértice representa uma variável,

ou um registo e os ramos assinalam as interferências entre as várias variáveis e entre estas e os registos. De tal forma que, se numa dada posição do procedimento/função estão duas variáveis "vivas"; ou se uma variável não pode utilizar um determinado registo, então existe um ramo no grafo que liga os respectivos vértices. Esse ramo representa a impossibilidade, para o primeiro caso, de atribuir um mesmo registo físico a duas variáveis que se encontrem simultaneamente "vivas" e, para o segundo caso, de a variável ocupar o registo em causa.

Após a construção do grafo é necessário colori-lo, atribuindo uma cor a cada um dos vértices, mas de forma a que não existam vértices adjacentes com a mesma cor, o que posteriormente corresponderá a registos distintos.

Caso exista um ramo entre dois vértices, em que um represente um registo e o outro uma variável, significa então, que esta não pode assumir a mesma cor do registo, o que posteriormente se irá traduzir pela interdição da utilização do mesmo por parte da variável.

É graças a esta última característica, que este algoritmo deve grande parte do seu sucesso, pois através dela é possível representar as restrições que se colocam na utilização de cada um dos registos.

Ao longo do processo de coloração do grafo pode acontecer que não seja possível que todos os vértices adjacentes possuam cores distintas. Nessa situação é indispensável ter em consideração outros aspectos que permitam decompor o período de vida de uma ou mais variáveis de forma a reduzir o número de adjacentes que estas possuam, tornando assim possível colorir o grafo.

Ao nível da representação intermédia, a decomposição do período de vida de uma variável, corresponde a realizar sobre esta, uma ou mais, operações de *splitting* e respectivas operações de *load*.

Os detalhes sobre a selecção das variáveis candidatas a operações de *splitting*, bem como a descrição deste tipo de operação são apresentados nas secções seguintes.

Após a conclusão do processo de coloração é apenas necessário realizar a atribuição concreta dos registos, o que não é mais que fazer a correspondência entre cores e registos.

No entanto, antes de se construir o grafo de interferências, há que determinar de entre todos os pseudo-registos (variáveis) quais os reais candidatos aos registos físicos, uma vez que, e por questões relacionadas com o formato das instruções do processador, nem todas as variáveis podem permanecer em registos, mesmo que se encontrem alguns disponíveis. A forma de selecção utilizada pelo BEDS é descrita na secção seguinte.

6.1.1 Selecção dos candidatos

É fundamental não esquecer que na construção da MIR, as variáveis representadas por objectos do tipo *Memory*, antes de utilizadas, devem ser colocadas num *Register* através de um *DataTransfer*. É este *Register* que para todos os efeitos representa a variável, onde a função do *DataTransfer* é essencialmente assinalar a presença da operação de *load* do valor que se encontra em memória para o registo, à semelhança dos *AttribAssignments* cuja função é assinalar as operações de *store*. Pelo que em termos de optimizações e mesmo da alocação, se trabalha essencialmente sobre os *Register*, tentando mesmo minimizar a presença dos *DataTransfer* destinados a operações de *load* e dos *AttribAssignments*.

Desta forma, ao nível da MIR todos os operandos são por defeito candidatos aos registos. É no entanto inconsistente tentar realizar a alocação sem discriminar quais são realmente, os operandos que devem permanecer em registos, ou que pelo menos se tente que permaneçam. Isto porque, em primeiro lugar, é insustentável considerar que todos podem permanecer em registos ao longo de todo o programa, e em segundo lugar, porque em última instância decidir se um operando ocupa ou não um registo depende em muito das instruções do processador.

Desta forma, a nível da MIR e de todos os processos que se desenrolam exclusivamente sobre ela, o objectivo é tentar que os operandos sejam vistos como um tipo uniforme de dados, em que todos utilizam uma mesma forma de representação (os *Registers*). Mas quando se trata de gerar o código final, da alocação ou de outros processos directamente relacionados com as características do processador, é normalmente necessário, discriminar quais os operandos que realmente podem ou devem permanecer em registos, bem como determinar em que momentos é que tal deve ser feito.

Convém ainda salientar que apenas as árvores de expressões passam para a fase de selecção de instruções, das quais não constam os *Registers*, pelo que estes deixam de ter qualquer significado ao nível da fase de selecção.

O BEDS determina quais os operandos que devem ou não permanecer em registos, realizando uma pré-selecção das instruções, considerando para tal, que não existe qualquer restrição por parte da alocação. Desta forma obtém a sequência óptima de instruções para o correspondente código da representação intermédia, de onde é possível saber quais os operandos que devem permanecer em registos, bem como o tipo concreto de registo a utilizar, entre outras informações relevantes para a construção do grafo de interferências.

Surge assim, pela primeira vez, a necessidade de interligar a representação MIR com as fases posteriores do processo de compilação, nomeadamente com a selecção das instruções. Esta não é no entanto, a única interligação necessária para se proceder com a alocação, é também essencial consultar a *Register Table*, que é obtida pelo *Back-End Generator* a partir da *Register Specification* (ver Fig. 3.1), o qual é parte integrante da descrição do processador. Tudo com o objectivo de determinar o número de registos disponíveis, bem como as restrições a colocar na utilização de cada um.

De forma a se conseguir realizar a pré-selecção das instruções, torna-se pertinente "linearizar" a representação intermédia. Para tal, existe o *Tree Selector* cuja função é percorrer a árvore de controlo de fluxo, através de uma travessia *bottom-up* e da esquerda para a direita, processando cada um dos *IntervalNodes*, com o objectivo de construir uma lista de árvores de expressões, segundo a ordem de execução (ver capítulo 5.1).

Posteriormente, passa-se cada uma das árvores para o selector de instruções, o qual trata de determinar quais as instruções idealmente necessárias, para executar o código representado por estas.

Genericamente, as árvores são obtidas a partir dos *AttribAssignments* e de alguns outros *DataTransfers*, nomeadamente dos responsáveis pelas expressões de salto e de *labelling*. Por defeito, os *Assignments* não contribuem directamente para a obtenção das árvores, mas apenas para as sub-árvores que as compõem.

É de notar que uma mesma sub-expressão pode fazer parte de várias árvores, o que poderia levar a pensar que seria necessário controlar o processo de selecção destas sub-árvores, de forma a que cada uma seja processada uma única vez. No entanto, e como se poderá ver no capítulo 7, a solução óptima (conjunto de instruções) de uma árvore, é

determinada a partir das soluções óptimas de cada uma das suas sub-árvores. O que implica que uma sub-árvore possui sempre a mesma solução óptima, independentemente da restante árvore onde se insere. Como tal, não advém nenhum mal em se pré-seleccionar as instruções de uma sub-árvore por diversas vezes uma vez que o resultado é sempre o mesmo. O principal inconveniente é o tempo que se desperdiça.

No entanto, aquando da selecção final das instruções, tal já não pode acontecer, uma vez que isso implicava que se gerasse código por cada vez que a sub-árvore fosse processada, levando a que todo o esforço feito durante a representação intermédia para se detectar sub-expressões comuns fosse anulado. Como tal e de forma a homogeneizar o tratamento das árvores por parte do selector de instruções, implementou-se um mecanismo de controlo, que assinala se uma dada sub-árvore já foi ou não processada.

Convém ainda salientar, que nem a criação da árvore de controlo, nem a criação da lista de árvores destroem a representação intermédia, limitam-se apenas a decorar a representação com mais alguma informação e a gerarem algumas estruturas de informação temporárias. Pelo que o processo de alocação pode prosseguir sem qualquer impedimento.

Antes de se continuar com a descrição das restantes fases do processo de alocação, é conveniente perceber o que é que se ganhou com esta pré-selecção das instruções. Em primeiro lugar, funciona como um teste de viabilidade da geração de código, uma vez que se tal não for possível sem qualquer tipo de restrição em relação à fase de alocação, então é garantido que também não é possível considerando essas restrições.

De notar que só não é possível gerar código se não existir pelo menos uma instrução, ou conjunto de instruções, que permitam realizar cada uma das operações presentes na MIR.

Em segundo lugar, permite simplificar algumas expressões e simultaneamente eliminar algumas variáveis do tipo temporário. É que em determinados casos, algumas sub-árvores (de expressões) são representadas ao nível do código final por uma única instrução. Como tal, ao nível da MIR e uma vez que nesta fase do processo de compilação já se executaram praticamente todas as operações de optimização, exclusivas deste nível de representação, é possível reduzir cada uma dessas sub-árvores a uma única operação representada por um só nodo, eliminando assim uma série de *DataTransfers* e respectivas variáveis temporárias (*Registers*).

Em terceiro lugar e como já se viu, permite determinar quais os pseudo-registos que devem permanecer em registos físicos, bem como o tipo concreto de registo.

O *Tree Selector* para processar as estruturas de controlo segue o tratamento standard referido na secção 2.2.1. A selecção de instruções é apresentada no capítulo 7.

6.1.2 Construção do grafo de interferências

Independente da forma como se encontra representado o grafo de interferências, este possui um conjunto de vértices e um conjunto de ramos. Como já foi dito, fazem parte do conjunto dos vértices todos os registos e todos os pseudo-registos seleccionados como possíveis candidatos a um registo físico, onde os pseudo-registos representam variáveis explícitas do código fonte, variáveis temporárias ou até mesmo constantes. E os ramos assinalam a interferência entre variáveis ou entre variáveis e registos.

Antes de se apresentar como é que se constrói o grafo de interferências, é conveniente explicar que se um processador possui vários conjuntos de registos que não interfiram entre si ou seja, quando a utilização de um registo de um dos conjuntos em

nada afecta o outro conjunto, e caso as variáveis também não interfiram entre si, de tal forma que nenhuma possa ocupar um qualquer registo dos dois conjuntos, é então possível obter vários grafos de interferências em vez de um. Este facto permite simplificar a própria construção dos grafos e o próprio processo de alocação. Tal situação é vulgar, por exemplo, para os processadores que separam por completo as instruções/registos para inteiros e para vírgula flutuante.

Representação do grafo de interferências

Para construir o grafo de interferências há que possuir uma estrutura de dados que o represente. Como o processo de alocação, designadamente no que se refere ao tratamento do grafo de interferências, acarreta uma carga elevada de processamento, é fundamental que as estruturas de dados utilizadas sejam o mais eficientes possíveis, nomeadamente quando se pretende determinar se dois vértices são ou não adjacentes, ou determinar quantos e quais são os adjacentes de um dado vértice. De reparar que os processos de pesquisa necessários aos dois exemplos anteriores, diferem substancialmente, uma vez que no primeiro caso, é conveniente possuir uma estrutura que permita acesso aleatório, enquanto no segundo é necessário um acesso sequencial.

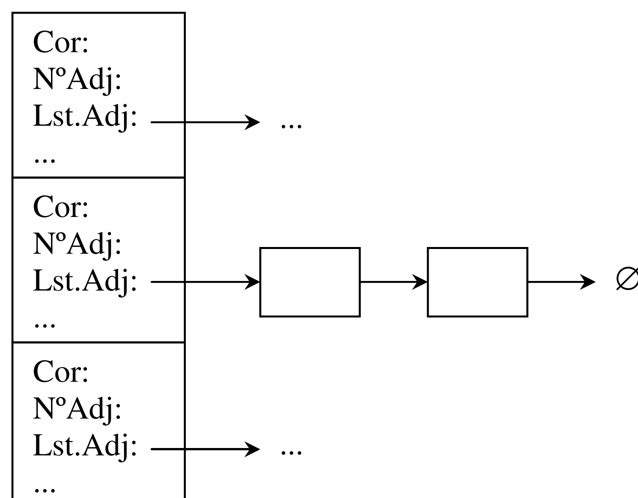


Fig. 6.1 – Representação esquemática da lista de adjacências.

Assim sendo e com base na própria representação proposta por Chaitin, achou-se por bem utilizar uma matriz de adjacências, cuja única finalidade é assinalar a presença de ramos entre vértices e que se adequa perfeitamente às pesquisas do tipo aleatório.

Como se trata de um grafo não orientado a matriz resultante é simétrica, pelo que na prática apenas se utiliza a semi-matriz inferior (ou então a semi-matriz superior), de tal forma que, dado dois índices i e j, correspondentes a dois vértices, então o valor na posição (i, j) = (j, i) = AdjMtx[max(i, j), min(i, j)].

No entanto, detecta-se que na maioria dos casos, o número de vértices é substancial e que a relação entre esse número e o número médio de adjacentes por vértice é muito elevada, fazendo com que a matriz de adjacências em determinadas circunstâncias seja pouco eficiente, como por exemplo, quando é necessário determinar os adjacentes de cada um dos vértices. Como tal, Chaitin propõe também e em simultâneo a utilização de uma lista de adjacências, formada por uma sequência de listas, em que cada elemento da sequência serve ainda para armazenar alguma informação referente a cada um dos

vértices, tal como o número de adjacentes ou a cor atribuída, como se encontra representado na Fig. 6.1.

Para se obter a representação completa do grafo, começa-se por construir a matriz de adjacências e depois a partir desta constrói-se a respectiva lista. Ambas estruturas serão modificadas mediante as alterações que ocorram no grafo. Faz, no entanto, falta que se conheça previamente o número de vértices, ou seja, o número de variáveis directamente relacionadas com o processo de alocação, o que é necessário para definir o tamanho das estruturas de dados.

Para o conjunto de ramos do grafo contribuem as interferências entre variáveis que se encontrem simultaneamente "vivas", as interferências entre variáveis e registos, e ainda a relação que existe entre registos. Desta forma, a matriz de adjacências é construída com auxílio de duas funções, *AreAlive(...)* e *Interfere(...)*, em que ambas devolvem valores do tipo booleano.

Na primeira, dadas duas variáveis indica se ambas se encontram simultaneamente "vivas". Na segunda, dada uma variável e um registo, indica se existe alguma interferência entre ambos. A Fig. 6.2 representa uma matriz de adjacências para 4 registos e 5 variáveis, mostrando as áreas de interferência de cada uma das funções.

R_0 R_1 R_2 R_3 V_1 V_2 V_3 V_4

R_1 R_2 R_3 V_1 V_2 V_3 V_4 V_5

True

Interfere(...)

AreAlive(...)

Fig. 6.2 – Estrutura da matriz de interferências.

Como o grafo de interferências é não-orientado, é possível representá-lo utilizando apenas a semi-matriz inferior (ou superior), o que quando conciliado com o facto do grafo ser acíclico, permite eliminar a primeira linha e a última coluna da matriz , para o caso de se estar a utilizar a semi-matriz inferior, ou a última linha e a primeira coluna, par o caso de se estar a utilizar a semi-matriz superior.

Interferências entre variáveis

A função *AreAlive(...)* é implementada com base na análise do período de vida útil das variáveis, a qual permite obter os vectores de bits que assinalam se uma variável está ou não "viva" em determinado ponto do programa. Através destes é relativamente simples determinar se duas variáveis estão simultaneamente "vivas". Para tal, basta pegar na definição da primeira variável e verificar se para todos os seus *flowDependents*, a segunda variável está ou não "viva". De reparar que este teste não garante que as duas variáveis não se encontrem simultaneamente "vivas", mas garante que não interferem entre si. Um exemplo desta situação encontra-se ilustrado na Fig. 6.3.

Interferências entre registos e variáveis

A função *Interfere(...)* permite colocar restrições à utilização de cada um dos registos físicos. Restrições essas que são determinadas conjugando a informação da pré-selecção das instruções com a da *Register Table*.

Convém nesta altura salientar que na especificação das características do processador, designadamente no *Register Specification*, é possível declarar mais do que um conjunto de registos, de tal forma que os elementos de cada um sejam utilizados apenas em determinadas circunstâncias. Estas são determinadas pela pré-selecção ou por convenções impostas para a própria utilização dos registos como por exemplo, para o caso da passagem de parâmetros. É ainda possível que um mesmo registo seja partilhado por diversos conjuntos e até declarar registos fictícios, compostos por registos reais ou por outros registos fictícios. Mais alguns detalhes relacionados com a declaração de registos são apresentados no capítulo 8.

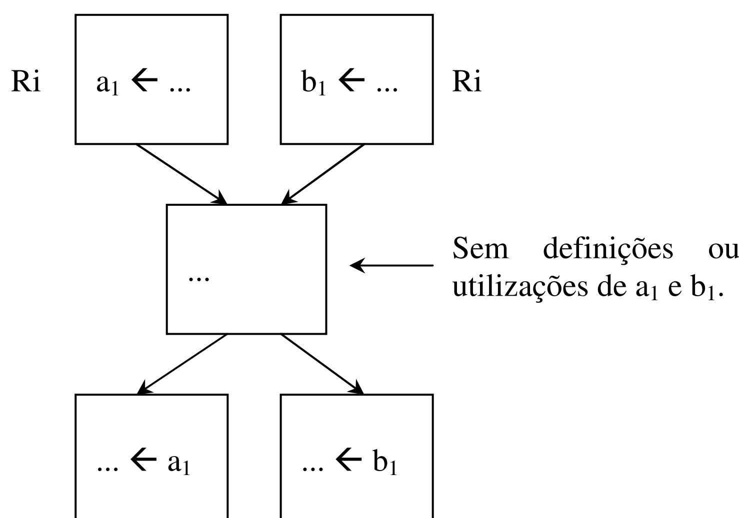


Fig. 6.3 – Exemplo com duas variáveis simultaneamente vivas, mas que aparentemente não interferem uma com a outra.

Desta forma para uma dada variável, a função *Interfere(...)* verifica qual o conjunto de registos cujos elementos podem ser utilizados para conter essa variável, o que é determinado pela pré-selecção. Depois consultando o *Register Table* verifica quais os registos que compõem esse conjunto e restringe a utilização de qualquer outro registo que não se encontre aí incluído, colocando para tal ramos entre o vértice correspondente à variável e os vértices dos registos que não pertençam ao conjunto.

Infelizmente para que este tipo de interferência funcione, é necessário que a variável ao longo da sua "vida" ocupe sempre o mesmo tipo de registo, o que nem sempre é verdade. Esta situação é bastante delicada, uma vez que as operações de troca entre registos ocorrem ao nível das instruções e não da representação intermédia. E mesmo que se tente introduzir um *Assignment* que represente essas trocas ao nível da MIR, o que ocorre é que estes tendem a desaparecer através dos processos de optimização (eliminação de cópias). Para além do mais, as trocas entre registos são operações que não têm representação nas árvores de expressões e ao nível do selector de instruções acabam por ficar camufladas (ver capítulo 7 - chain rules).

No actual estado do BEDS, este é um dos problemas que se encontra por resolver; a solução mais viável é considerar a troca entre registos como uma nova definição (*AttribAssignment*), que como tal e devido ao SSA, resultará numa variável independente, à qual se pode atribuir um novo registo. De notar no entanto, que este tipo de atribuição é feita a uma variável temporária, das que normalmente se encontram associadas aos *Assignments*, onde representam os resultados dos nodos intermédios das árvores de expressões. Como tal, esta nova forma de atribuição também deve permitir a sua utilização na construção de outras árvores, sem no entanto perder a capacidade de representar uma atribuição do tipo normalmente associado ao *AttribAssignment*.

Restrições entre registos

Outra forma de restrição advém dos próprios registos. Um dos exemplos ocorre quando o processador em causa possui registos compostos, ou seja, registos formados por outros registos, do tipo $Rc = (Rs_1 , Rs_2)$. Nestas situações a utilização de um dos componentes, por exemplo Rs_1, inviabiliza a utilização de Rc.

De notar que a nível do grafo de interferências, os registos têm que interferir sempre entre si, de forma a que após a coloração cada um possua a sua própria cor, permitindo assim relaciona-los com as (cores das) variáveis. Pelo que a solução que permita representar esta forma de restrição, não passa, como à primeira vista se poderia pensar, por manipular os ramos entre registos, daí ter que se desenvolver uma alternativa como a seguir se explica.

A utilização de registos compostos, pressupõe a existência de variáveis que os utilizem, ou dito de outra forma, que utilizem múltiplos registos simples, o que a nível do grafo de interferências poder-se-ia descrever como um vértice que possui mais do que uma cor.

É com base nesta ideia, que se tentou encontrar uma forma de representar este tipo de restrição no grafo de interferências. Para tal há que recordar, que cada vértice tem de ser colorido com uma cor diferente das dos seus adjacentes, o que significa que após a sua coloração reduziu em um, o número de cores que se encontram disponíveis para colorir cada um dos seus adjacentes. Se o vértice em causa utilizar um registo composto por n registos simples, então na realidade está a reduzir em n o número de cores disponíveis para todos os vértices a ele adjacentes.

Ao nível do grafo de interferências a utilização de registos compostos, afecta a forma de contabilizar o número de adjacentes, pelo que se um vértice representa uma variável que utiliza registos do tipo composto, formados por n registos simples, então a contribuição desse vértice em relação aos seus adjacentes, também é de n.

É ainda importante chamar atenção que nestas circunstâncias o processo de atribuição das cores complica-se substancialmente, uma vez que pode ocorrer que um vértice que necessite de um registo composto por n registos simples, no fim do processo de coloração realmente consiga ter disponíveis as n cores (e como tal registos), mas que mesmo assim não consiga formar com esses registos simples, o registo composto de que necessita. Isto porque na realidade o processo de coloração desconhece o que são registos compostos, sabe apenas que há determinadas variáveis que necessitam de mais do que um registo.

É como tal fundamental implementar um sistema de atribuição de cores, cujas regras de selecção não consistam apenas em utilizar a primeira cor possível de aplicar (diferente da cor de qualquer um dos vértices adjacentes).

A solução utilizada no BEDS funciona com base em duas heurísticas. A primeira aplica-se para os vértices que necessitam de mais do que um registo (simples), nestas

circunstâncias as cores são escolhidas de forma a constituírem um registo (composto) do tipo requerido para o vértice. A segunda heurística, aplica-se aos vértices que necessitam apenas de um registo, consiste em escolher uma cor de forma a não inutilizar possíveis registos compostos que se possam ainda formar com as cores que se encontram livres. Ambas as heurísticas necessitam de conhecer a composição dos registos e da relação destes com o tipo a atribuir a cada variável, estes aspectos serão discutidos no capítulo 8. Necessitam ainda de conhecer de antemão a cor respeitante a cada um dos registos simples, o que obriga a colorir logo de início os vértices a estes associados e só depois os restantes.

6.1.3 Coloração do grafo

Já se abordou a construção do grafo de interferências; é agora necessário perceber como se consegue colorir o grafo de forma a que não existam dois vértices adjacentes com a mesma cor.

Antes de se avançar com mais explicações, é conveniente definir o número cromático de um grafo *G*, como sendo o número mínimo de cores necessárias para o colorir. Para isso define-se **grau de um vértice**, que se representa por *k*, como sendo o número de vizinhos a que o vértice está ligado e, define-se ainda, **ordem de um grafo**, como sendo o maior grau de todos os vértices. O número máximo de cores disponíveis para colorir o grafo é representado por *n*.

Está demonstrado que provar que é possível colorir um grafo *G* com *n* cores, tal que *n* é igual ou superior a 2, é um problema NP-Completo. Pelo que, a solução proposta por Chaitin *et al.* associa aos algoritmos de coloração, algumas heurísticas que permitem acelerar o processo de coloração do grafo, tornando assim possível obter uma solução em tempo finito (aceitável). Posteriormente surgiram outras soluções [BCKT89, BCT92, CH90, CK91, GSO94, PF96] com base na proposta por Chaitin *et al.* que diferem essencialmente no tipo de heurística utilizada.

Em qualquer dos casos, o mecanismo utilizado é essencialmente o mesmo e consiste em remover os vértices do grafo, um a um e segundo uma ordem estabelecida pelas heurísticas.

Chaitin *et al.* propõem que se removam primeiro os vértices cujo grau é inferior ao número de cores (*n*) disponíveis para colorir o grafo. O que no presente caso e considerando a possibilidade de utilizar registos compostos, se traduz por remover qualquer vértice, cuja diferença entre o número de cores disponíveis e o grau do vértice, seja igual ou superior ao número de registos (simples) que este necessita. Com estas condições, é garantido que é possível colorir o vértice com uma cor diferente da de todos os seus adjacentes.

Praticamente todas as implementações deste algoritmo utilizam esta heurística. Tal consenso já não se verifica, para a situação em que não existem vértices com um grau inferior ao número de cores. É que nesta situação, a simples remoção do vértice do grafo, não garante a existência de cores suficientes para o colorir. Mas também não implica o contrário, é que podem existir vários vértices adjacentes com uma mesma cor (mas que não são adjacentes entre eles), podendo como tal existir cores disponíveis. Veja-se o exemplo da Fig. 6.4, em que todos os vértices possuem um grau igual ou superior a três, mas mesmo assim é perfeitamente possível colorir o grafo com apenas três cores, sem que fiquem dois vértices adjacentes com cores iguais.

É como tal imprescindível utilizar outros mecanismos para que se torne possível concluir a coloração do grafo. Um método pessimista consiste em considerar que não existe nenhuma combinação de cores que permita colorir o grafo. Pelo que a solução passa pela sua reestruturação, de forma a que se torne possível obter um resultado conclusivo. Uma das abordagens para a reestruturação, consiste em realizar o *splitting* de uma ou mais variáveis, das que ainda se encontram representadas no grafo. O que corresponde a decompor o período de "vida" útil de cada uma, em vários segmentos mutuamente exclusivos, como se se tratassem de diversas variáveis, à semelhança do que se faz para o SSA, mas agora aplicado de forma restrita.

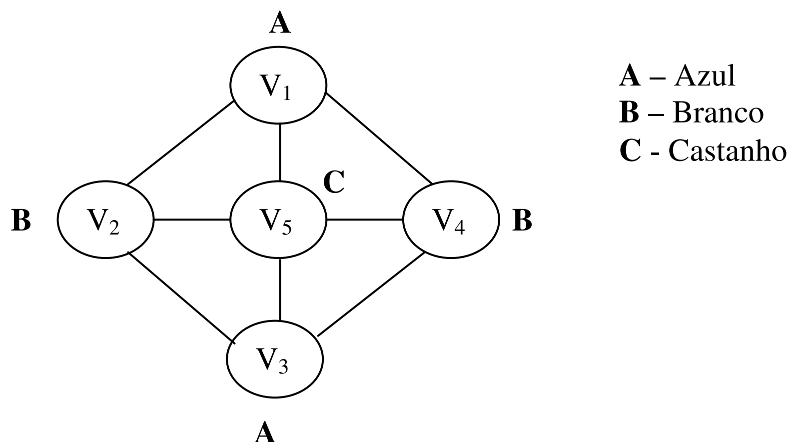


Fig. 6.4 – Exemplo de como é possível colorir um grafo com n=3 e todos os $K \geq 3$.

Desta forma reduz-se o número de interferências entre variáveis permitindo assim continuar aplicar a primeira heurística descrita, ou então, decompor ainda mais algumas variáveis.

Ao nível do grafo, tais modificações correspondem a decompor um ou mais vértices, em tantos quantos os segmentos anteriores. Mantendo em relação aos vértices originais, as restrições deste com os registos, mas reformulando as interferências em relação às outras variáveis.

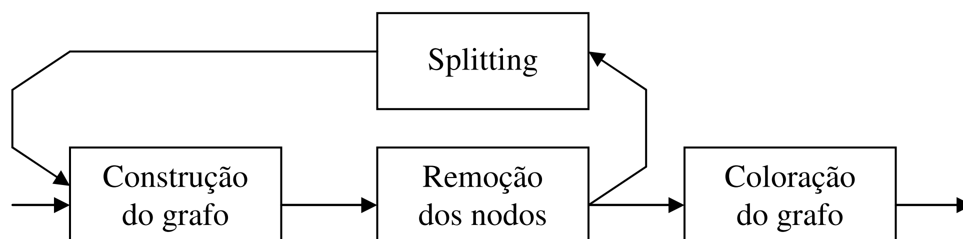


Fig. 6.5 – Fases do processo de coloração do grafo.

Existem outros métodos de remoção mais optimistas que, por exemplo, consideram que é sempre possível obter uma solução que permita colorir o grafo, limitando-se, como tal, a continuar a remover os vértices na esperança de que, aquando da atribuição das cores tal pressuposto se confirme, caso contrário apelam ao *splitting*.

A coloração propriamente dita, só começa quando todos os vértices tiverem sido removidos do grafo. Nesta fase pressupõem-se que estes foram armazenados

temporariamente numa estrutura, de preferência do tipo *stack*, de forma a garantir o acesso aos mesmos pela ordem inversa da qual foram removidos do grafo.

O processo de coloração consiste em pegar em cada um dos vértices que se encontram na *stack* e inseri-los novamente no grafo. Mas conforme tal acontece, é atribuído uma cor a cada um, a qual é escolhida tendo em consideração as cores entretanto atribuídas aos vértices adjacentes.

Caso existam registos compostos, é necessário controlar mais do que as cores dos vértices adjacentes; faz falta também evitar atribuir cores que inviabilizem a utilização de registos compostos. É como tal, essencial identificar o mais antecipadamente possível, as cores atribuídas a cada um dos registos físicos, por forma a controlar se determinada cor representa ou não, um registo que faça parte de uma composição. Em tal situação, deve-se evitar a utilização dessa cor, excepto se não existirem mais alternativas, ou então, se algum dos vértices adjacentes já possuir uma cor que pertença à composição em causa.

No fim, cada registo possui uma cor única entre os registos, que o identifica, pelo que a distribuição destes pelas variáveis consiste numa simples comparação de cores.

6.1.4 *Splitting* de variáveis

Um dos pontos críticos da alocação, está na selecção das variáveis a utilizar em caso de ser necessário introduzir operações de *splitting*. Uma má selecção pode levar a incluir uma série de instruções que em nada resolvam o processo de coloração do grafo, deteriorando assim a qualidade final do código.

O objectivo é seleccionar a variável que, segundo a implementação do processo de *splitting*, incorra na menor penalização possível. A solução ideal seria, como tal, determinar o custo real do *splitting* das várias variáveis, mas devido à carga computacional tal torna-se completamente inviável.

A forma mais simples para se determinar o custo de *splitting*, consiste em utilizar o número de definições e de utilizações de cada uma das variáveis, de tal forma que:

$$\text{Custo}_{\text{spill}}(v_i) = \sum_{\text{def} \in v_i} f_1(\text{def}) + \sum_{\text{use} \in v_i} f_2(\text{use}) \qquad \text{Eq. 6.1}$$

Devido à representação SSA, no BEDS cada variável possui uma só definição, pelo que o primeiro somatório resulta sempre no mesmo valor, podendo como tal ser removido.

É ainda necessário avaliar os potenciais benefício que podem resultar do *splitting* de cada uma das variáveis, uma vez que este tipo de operação permite normalmente diminuir o número de restrições que existem no grafo.

Os benefícios podem assim, ser medidos através do número de adjacentes que cada vértice possui. Convém no entanto reparar que não se trata de uma relação linear e que por vezes inflaciona de forma especulativa os benefícios decorrentes do *splitting*.

Assim, a variável seleccionada deve ser escolhida de entre as que compõem o grafo, de tal forma que maximize o valor da seguinte equação:

$$\text{Custo}_{\text{final}}(v_i) = \frac{\text{Grau}(v_i)}{\text{Custo}_{\text{spill}}(v_i)} \qquad \text{Eq. 6.2}$$

Existem diversas optimizações ou adaptações que permitem outro tipo de relações entre o grau de cada vértice e o custo de *splitting* da respectiva variável.

Uma vez escolhida a variável falta descrever como é que se implementa a operação de *splitting*, propriamente dita, o que normalmente é feito com base em heurísticas relativamente simples e numa abordagem muito optimista.

No caso concreto do BEDS, como só existe uma definição o que implica um único *store*, optou-se por inseri-lo logo que seja possível, isto é, testa-se se existe alguma utilização da variável nas imediações posteriores à definição, se tal ocorrer, o *store* é colocado após a utilização, caso contrário, é colocado imediatamente a seguir à definição. No caso das definições provenientes dos *PhyAssignments* não se chega sequer a inserir as operações de *store*.

O problema maior surge com o posicionamento dos *load*, primeiro porque podem existir vários; segundo, porque podem encontra-se ou não no mesmo caminho; terceiro, porque a possível posição de inserção pode ser ainda mais caótica do que própria situação que se pretende resolver; e quarto, porque pode afectar a própria representação SSA.

Percebe-se como tal, que a solução terá que ser sempre de compromisso e que nem sempre produz os resultados desejados.

Convém no entanto ter em conta as seguintes situações:

1. Evitar colocar operações de *load* e de *store* dentro de estruturas cíclicas, uma vez que a penalização é ainda maior.

2. Inserir as operações de *load* no mesmo nodo do grafo de fluxo de controlo, onde a variável é utilizada, ou então num dos seus nodos dominadores. Caso contrário poderá ainda ser necessário inserir alguns *PhyAssignments*.

3. Criar mecanismos para determinar a "pressão sobre os registos", quantificando assim o número de variáveis ou de cores "vivas" ao longo do programa, o que permite avaliar se determinada posição do programa se encontra ou não sobrecarregada, ao ponto de não se poder inserir operações de *splitting*. No BEDS facilmente se consegue determinar a pressão sobre os registos, contando o número de elementos do reticulado que assinalam a presença de variáveis "vivas" (ver capítulo 5.2 – Análise do período de vida).

4. Actualizar as estruturas da representação, designadamente os *flowDependents*, *flowSupporters*, o SSA, grafo de interferências, etc. Pelo que é conveniente manter as listas das *dominance frontiers* de cada um dos nodos.

5. E por último, não confiar que os quatro pontos anteriores sejam suficientes para garantir um bom processo de inserção de instruções de *splitting*.

Infelizmente, no BEDS, ainda não houve oportunidade de implementar um sistema de inserção de instruções de *splitting* que considere todos os aspectos atrás referidos e que seja eficiente. Na realidade, a solução implementada ainda é muito provisória, pelo que os *load*s são inseridos imediatamente antes da sua utilização.

No Apêndice E encontra-se um exemplo que ilustra como todo o processo se desenrola para esta forma de alocação.

6.2 *Alocação local*

Com já foi referido, o BEDS também fornece mecanismos para realizar a alocação ao nível local, mais propriamente dito ao nível de cada uma das árvores de expressões.

O processo inicia-se após se terem realizado todas as tarefas inerentes à representação intermédia e já numa fase em que se pretende gerar o código final. Para tal e à semelhança da alocação global, é necessário "linearizar" o grafo de controlo de fluxo e realizar a selecção das instruções. Só que agora, a alocação é feita ao mesmo tempo que a selecção, ou seja, para cada árvore de expressões determinam-se as instruções a utilizar, alocando em seguida os registos necessários. O processo prossegue com a próxima árvore.

Como já se viu, a selecção das instruções permite determinar o tipo de registos a utilizar para cada operando. Com base nessa informação e caso existam registos livres escolhe-se um e atribui-se, caso contrário, o algoritmo de alocação apela a uma série de estratégias simples, mas que normalmente surtem efeito, de forma a conseguir concluir o processo de alocação. Em casos extremos chega-se mesmo a recorrer a operações de *splitting*.

Por defeito, cada registo é apenas utilizado ao nível da própria instrução em que se insere, sendo libertado imediatamente após a sua conclusão. É como tal normal que estes sejam reutilizados por diversas vezes numa mesma expressão.

A única excepção à regra anterior surge para a situação em que a informação contida no registo deve transitar para a próxima instrução ou ser utilizada posteriormente numa outra expressão.

6.2.1 Gestão dos registos

Após a selecção das instruções invoca-se o alocador local, que percorre a árvore de expressões numa travessia do tipo *post-fix*.

Fazendo uma breve revisão sobre a relação entre uma árvore de expressões e as instruções, pode-se dizer que a cada nodo de uma árvore, na sua forma mais simples, se encontra associado uma instrução cujos operandos são determinados, sintetizando os resultados das suas sub-árvores e cujo resultado, dará por sua vez origem a um dos operandos do nodo antecessor, pelo que os valores sintetizados, que transitam de nodo para nodo, não são mais do que endereços de memória, constantes, registos, etc.

Os registos surgem assim através das transferências explícitas de valores de memória ou de constantes para registos, normalmente representadas por instruções de *load* ou de *move*, ou então como meio necessário para conter o resultado de uma instrução.

O processo de alocação, como já se disse, atravessa a árvore de expressões, realizando para cada nodo, as seguintes operações:

1- Faz a alocação da sub-árvore esquerda;
2- Faz a alocação da sub-árvore direita;
3- Testa a necessidade de alocar algum registo para a instrução associada ao nodo;
4- Caso tal aconteça, verifica se existem registos livres e se assim for procede à atribuição;
5- Caso contrário, apela às tácticas descritas na próxima secção;

6- Finaliza libertando todos os registos ocupados directa, ou indirectamente, pela instrução, excepto se:
   6.1- Algum for utilizado para manter o resultado,
      6.1.1- Então há que verificar se o mesmo é utilizado noutras árvores;
      6.1.2- Se isso acontecer é iniciado um contador com o número de utilizações posteriores menos uma.
   6.2- Existirem registos que se encontrem nas condições do ponto 6.1.2.;
      6.2.1- Nestas circunstâncias reduz-se o contador.

Para que tudo funcione correctamente, há que proceder à libertação de todos os registos, excepto dos que se encontram nas condições do ponto 6.1.2.

É de notar que cada nodo tem correspondência directa com um *DataTransfer*, através do qual é possível verificar se o respectivo resultado é utilizado por mais do que uma sub-árvore ou então se é utilizado mais do que uma vez numa mesma árvore, basta para tal contar o número de elementos de *flowDependents*. É com base nesse valor que se inicializa o contador a que se refere o ponto 6.1.2.

Convém ainda salientar que, após se realizar a alocação de uma das sub-árvores e se esta resultar num registo, então este não poderá ser utilizado pela outra sub-árvore.

O algoritmo de alocação tira proveito do facto de muitos processadores utilizarem diversas formas de endereçamento para um dos seus operandos, mas não para os restantes, processando em primeiro lugar estes últimos, aumentando assim a probabilidade de se concluir a alocação, uma vez que se encontram mais registos livres. Só por último é que se trata dos operandos com maior número de formas de endereçamento, pois em caso de não ser possível alocar um registo, é mais fácil conseguir obter uma solução com base numa operação de *spilling*.

É por esta razão que se optou por realizar uma travessia da esquerda para a direita, uma vez que maioria das arquitecturas CISC admite instruções em que o segundo operando permite mais do que uma forma de endereçamento.

6.2.2 *Spilling*, *splitting* e reconversão de registos.

Em caso de falta de registos, encontram-se implementadas três formas para se tentar concluir a alocação com êxito.

A primeira consiste em realizar o *spillling* do operando correspondente à sub-árvore para a qual não se conseguiu concluir a alocação. Há como tal, que reformular as instruções seleccionadas, de forma a que o operando que até aqui utilizava um registo, o deixe de fazer, optando por outra forma de endereçamento, resolvendo a questão utilizando instruções mais dispendiosas.

Para tal, o que acontece é que a alocação dá o trabalho por inacabado devolvendo ao processo de selecção, a indicação do nodo onde ocorreu a falta de registo. Este volta a seleccionar as instruções, mas agora colocando a infinito o custo associado à instrução que dava origem à utilização do registo. Desta forma, faz com que a selecção opte por outras instruções que podem eventualmente não utilizar registos, tornando assim possível concluir a alocação.

Uma variante desta solução consiste em manipular todos os custos de forma a garantir que as novas instruções não utilizam registos para o nodo em causa. No entanto tal solução aumenta em muito a probabilidade de não se conseguir concluir o próprio processo de selecção de instruções.

A segunda e terceira hipótese, que só são utilizadas no caso da primeira não resultar, aplicam-se de forma mutuamente exclusiva. A segunda consiste em verificar se existe algum registo que se encontre alocado mas que não pertença à árvore em causa. Nesse caso liberta-se esse registo indicando que a sub-árvore da qual resultou deixou de estar processada. A terceira solução consiste em realizar o *splitting* cortando a árvore em duas.

De notar que a segunda solução implica voltar a gerar código para a sub-árvore da qual derivou o registo que foi libertado, o que pode acarretar custos elevados. A terceira solução consiste em inserir um *store* do resultado da sub-árvore onde ocorreu a falha do registo e posteriormente um *load* a carregar esse mesmo resultado para o operando por onde se dividiu a árvore.

Os custos de ambas as soluções encontram-se determinados pelo processo de selecção das instruções, como se poderá ver no capítulo seguinte.

De princípio a terceira solução é mais económica, para além de que nem sempre é possível aplicar a segunda, pois pode acontecer que não existem registos nas condições exigidas. Só que por vezes não só existem, como apenas se limitam a realizar o *load* de um valor, pelo que o custo associado é bastante baixo, sendo como tal preferível.

Conclui-se assim a descrição dos processos de alocação implementados no BEDS. Alguns detalhes referentes à especificação dos registos, bem como da relação das rotinas de alocação com o processo de selecção, serão apresentados no capítulo 8.

7 Selectores de código e a sua geração

Na continuação do que se disse no capítulo 2 sobre a selecção de instruções, pretende-se agora aprofundar mais esta fase da compilação, apresentando outras soluções, bem como mecanismos de geração dos próprios Selectores, subindo assim um nível no processo de desenvolvimento dos compiladores, à semelhança do que se faz para as fases de análise léxica e sintáctica, através de ferramentas como o lex e o yacc. Mas, enquanto que estas são dependentes das características da linguagem fonte, a geração de um Sistema de Selecção de Instruções é dependente das características do processador.

Como se poderá verificar no resto deste capítulo, as soluções utilizadas recorrem essencialmente a técnicas genéricas de reconhecimento de padrões em estruturas em árvore, mas que aqui são empregues na construção de sistemas de selecção de instruções, por vezes referidos também como Sistemas de Geração de Código.

7.1 *Introdução aos geradores de selectores*

Antes de se avançar com a apresentação dos selectores de instruções e com a descrição do sistema desenvolvido para o BEDS, convém perceber alguns aspectos importantes, relacionados com esta etapa do processo de compilação.

Os sistemas aqui descritos são genericamente concebidos como autómatos, e como tal compostos por duas partes, uma formada por um mecanismo responsável por "navegar" na máquina de estados e outra, por um conjunto de estados e respectivas tabelas de transições que caracterizam um determinado autómato, de tal forma que, dado um operador, o seu estado actual e o(s) símbolo(s) entretanto reconhecido(s), o mecanismo passe ao próximo estado do sistema.

A cada uma das transições (ou a cada um dos estados) está associado um conjunto de acções a executar no caso deste ser alcançado, acções essas que podem consistir na geração das instruções máquina.

Ao contrário do mecanismo de "navegação", os estados e as tabelas de transições são dependentes dos padrões a reconhecer, dos respectivos custos (ou das funções de custo), das acções a executar, etc. No caso concreto dos Selectores de Instruções, os factores, que caracterizam os estados e tabelas, advêm da representação intermédia e do processador para o qual se pretende gerar as instruções. No entanto, a forma de representar a máquina de estados e a relação desta com o mecanismo de navegação do autómato é constante. Daí se colocar a possibilidade de subir um nível no desenvolvimento desta fase do processo de compilação, reformulando os objectivos, de forma a ser possível desenvolver um sistema que, a partir da descrição dos padrões, respectivos custos e instruções, permita gerar toda a informação necessária ao autómato (descrição dos estados, tabelas de transições, etc). Esta fase de concepção do Selector de Instruções engloba algum pré-processamento do processo de selecção, daí ser por vezes designada por fase de pré-processamento, em oposição à fase de reconhecimento das árvores de expressões (selecção de instruções), designada por *run-time*.

Outro aspecto importante na construção dos sistemas de geração de selectores, é a forma como estes implementam os mecanismos que permitem escolher qual ou quais os padrões a utilizar, em caso de existirem várias alternativas. Algumas soluções fazem a escolha, avaliando em *run-time* o contexto de execução. Pelo que a construção das tabelas de transições é em parte independente dos custos atribuídos aos padrões, o que por si só, permite separar o mecanismo de selecção, do mecanismo de avaliação, fazendo com que este último seja implementado conforme as exigências e características do sistema.

Outras soluções há, em que a avaliação dos padrões está intrinsecamente associada ao mecanismo de selecção, pelo que é fundamental o conhecimento prévio dos respectivos custos, de forma a se conseguir construir o sistema de selecção.

7.2 *Tree Pattern Matching*

A utilização das técnicas de reescrita na implementação de interpretadores, foi apresentada por Wasilen e por Weingart [Wasil72, Weing73]. Segundo estes autores, a técnica consiste em, dada uma expressão inicial, reconhecer as sub-expressões que a compõem e substitui-las por outras, segundo regras pré-definidas e até que não seja possível simplificar mais a expressão inicial.

Em 1982, Hoffmann e O'Donnell [HO82] apresentaram uma série de processos de reescrita para árvores e mostraram como estes podiam ser utilizados na geração automática de interpretadores de código, tais como selectores de instruções. As soluções propostas utilizam duas abordagens distintas, uma do tipo *top-down* e outra do tipo *bottom-up*, as quais estão na base de muitos dos actuais selectores de instruções. Ambas as soluções foram desenvolvidas a partir do algoritmo de reconhecimento de *strings* de Knuth-Morris-Pratt [KMP77].

A solução do tipo *top-down*, consiste em construir uma árvore, de tal forma que todo e qualquer caminho desde o nodo raiz até uma folha, represente uma das expressões a reconhecer (o equivalente às *strings* do algoritmo de Knuth-Morris-Pratt). Depois, utilizando o algoritmo de Aho e Corasick [AC75], constrói-se com base na árvore um autómato, em que os nodos correspondem a estados e os ramos que os interligam a transições da máquina de estados. O nodo raiz representa o estado inicial.

Hoffmann e O'Donnell utilizando este algoritmo, reduziram o reconhecimento de padrões em árvores a um problema de reconhecimento de *strings*, onde em vez de caracteres, se tinham os símbolos das expressões. No entanto e ao contrário das *strings*, as expressões obrigam a relacionar os símbolos reconhecidos com os respectivos descendentes da expressão, ou seja, a partir de um determinado estado é possível identificar vários símbolos válidos, mas cuja utilização se encontra condicionada ao descendente pelo qual prossegue o reconhecimento, pelo que é necessário relacionar o símbolo com o descendente para o qual é reconhecido. Para tal, inserem-se alguns estados intermédios, tantos quantos os descendentes de cada um dos nodos e reescrevem-se as próprias expressões de forma a considerar estes novos estados.

O reconhecimento dos padrões faz-se segundo uma travessia *preorder* da árvore de expressões e os mecanismos de avaliação de custos são normalmente implementados em *run-time*, através de programação dinâmica. Este é o modelo utilizado pelo Twig [Tjiang85], que consegue assim, gerar a cobertura óptima das árvores de expressões.

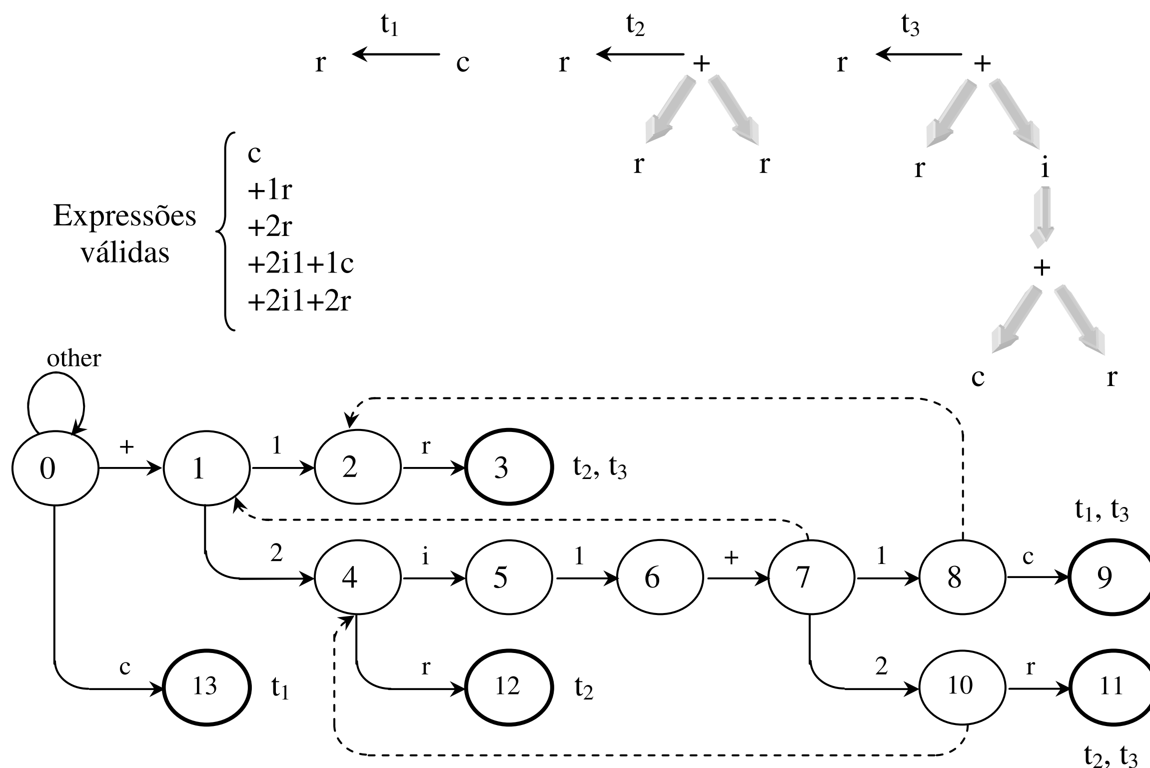


Fig. 7.1 – Padrões e máquina de estados do *top-down* pattern matching.

A Fig. 7.1 representa um exemplo para um sistema de reconhecimento tipo *top-down*, retirado de um artigo de Aho *et al.* [AGT89]. Na figura, para além dos padrões t_1, t_2 e t_3, estão representadas as expressões possíveis de construir com os mesmos, considerando o descendente pelo qual se está a reconhecer o próximo símbolo, e a respectiva máquina de estados.

Provou-se que o tempo necessário para construir a árvore e a máquina de estados é directamente proporcional à soma do comprimento das palavras chave. E que o tempo necessário ao reconhecimento de uma *string* é directamente proporcional ao seu tamanho. Trata-se como tal de uma solução extremamente rápida.

Hoffmann e O'Donnell foram também pioneiros nas estratégias do tipo *bottom-up*, apresentando várias soluções, em que de um modo geral, o objectivo é determinar para cada nodo da árvore de expressões, todos os padrões ou parte de padrões, que se ajustam a esse nodo. Seja então *n* um qualquer nodo da árvore, tal que *op* representa a respectiva operação e *k* o número de descendentes de *n*. Pretende-se então determinar o conjunto *M* de todos os padrões possíveis de utilizar em *n*. Para tal, parte-se do principio que o mesmo já foi feito para cada um dos nodos descendentes de *n*, onde os respectivos conjuntos são representados por $M_1,...,M_k$. Pode-se então dizer que *M* é formado por todos os padrões $op(t_1,...,t_k)$, tal que $t_i \in M_i$, para $1 \leq i \leq k$, obtidos através da combinação dos elementos de $M_1,...,M_k$, de tal forma que cada $op(t_1,...,t_k)$ seja um padrão ou parte de um padrão válido, segundo as regras pré-definidas. Só após se determinarem todos os *M*'s, é que é possível obter o conjunto dos padrões que descrevem a árvore de expressões.

Este tipo de solução, permite tirar proveito das gramáticas não divergentes, em que a quantidade total de todos os *M*'s é finita, proporcionando assim, a possibilidade de os determinar em pré-processamento.

A escolha dos padrões neste tipo de solução, ou é feita com base numa avaliação em *run-time* do contexto da operação, ou então construindo as tabelas de transições de forma a que estas traduzam sempre a melhor solução, ou pelo menos aquela que se espera ser a melhor. De notar, que como as tabelas de transições são construídas em pré-processamento, com base nos custos individuais dos padrões e sem qualquer tipo de avaliação do contexto de execução, pode ocorrer que em *run-time* a solução seleccionada não corresponda à melhor escolha.

É ainda possível evitar a construção das tabelas determinado em *run-time* os padrões utilizáveis para cada nodo. Esta solução acarreta um aumento considerável no tempo de execução, uma vez que o processo de reconhecimento engloba grande parte do processo de construção das tabelas. Por outro lado, reduz substancialmente o pré-processamento e elimina os recursos necessários à manutenção das tabelas.

Hoffmann e O'Donnell apresentaram uma comparação entre as diversas soluções, designadamente no tipo de abordagem utilizada, *top-down* ou *bottom-up*, e concluíram que esta última, apesar de obrigar a uma maior carga de pré-processamento, garante sem dúvida, um reconhecimento das expressões mais rápido e é mais versátil. Este último aspecto é extremamente importante para a implementação dos mecanismos de avaliação de custos e para o caso de se pretender dar outro tipo de utilização a esta forma de reconhecimento.

Não se pretende aqui apresentar as soluções propostas por Hoffmann e O'Donnell uma vez que neste capítulo são apresentados outros trabalhos, com base nas mesmas soluções, mas que são mais dedicados ao processo de selecção e apresentam características mais especificas.

7.3 *Bottom-Up Rewrite System*

Um outro investigador que trabalhou muito na área da geração de sistemas de reconhecimento de padrões em árvores, foi Pelegri-Llopart [PG88]. Do trabalho por ele desenvolvido surgiu o BURS – *Bottom-Up Rewrite System*, um sistema de reconhecimento de padrões que partilha muitas das características das soluções do tipo *bottom-up* apresentadas por Hoffmann e O'Donnell. Possui no entanto um conjunto de

propriedades muito próprias: primeiro, tenta maximizar a carga computacional da fase de pré-processamento de forma a reduzir o tempo de execução em *run-time*; segundo, não se trata apenas de um sistema de reconhecimento mas também de reescrita, ou seja, permite aquando do reconhecimento reescrever a própria árvore alargando assim o universo das aplicações; terceiro, a avaliação dos custos da utilização de cada um dos padrões é feita em pré-processamento, mas garante, com base nos custos pré-definidos, a solução final óptima.

O reconhecimento das expressões necessita de uma travessia *bottom-up* da árvore, na qual se identifica o estado de cada nodo da árvore de expressões. Este processo, designa-se por *labelling* e é determinado com base na tabela de transições do operador associado ao nodo e no estado dos seus descendentes. Depois através de uma segunda travessia tipo *top-down* e mediante o valor da *label* atribuída e do não-terminal objectivo a alcançar, identificam-se os padrões a utilizar e realizam-se as respectivas acções.

7.3.1 Exemplo do funcionamento do BURS

O presente exemplo foi retirado de um artigo de Proebsting [Proeb95] e serve para elucidar o funcionamento de um sistema de selecção com base na teoria do BURS.

A informação de entrada para um Sistema de Geração de Geradores de Código (Selectores de Instruções) do tipo BURS, consiste num conjunto de regras, que definem os padrões a reconhecer, o custo de cada uma, o símbolo não-terminal que substitui o padrão e a respectiva acção. O exemplo considera as seguintes regras:

Regra	LHS		RHS	Custo	Acção
1	goal	→	reg	(0)	“”
2	reg	→	Reg	(0)	“”
3	reg	→	Int	(1)	“mv $0, $1”
4	reg	→	Fetch(addr)	(2)	“ld $0, $1”
5	reg	→	Add(reg, reg)	(2)	“add $1, $2”
6	addr	→	reg	(0)	“st $0, $1”
7	addr	→	Int	(0)	“mv $0, $1”
8	addr	→	Add(reg, Int)	(0)	“add $1, $2”

De notar que o RHS, pode ser composto por operadores binários, unários e sem operandos (neste último caso temos os símbolos terminais *Reg* e *Int*). Pode ainda conter apenas um único símbolo não-terminal e neste caso diz-se tratar de regras em cadeia (*chain rules*).

Os símbolos do LHS são sempre do tipo não-terminal, onde *goal* representa o não-terminal objectivo a atingir para o nodo raiz da árvore.

A Fig. 7.2 representa a árvore de expressões para a qual se pretende determinar a cobertura e o conjunto de acções a realizar. Associado a cada nodo encontra-se o estado que o descreve, com tantas regras quantos os símbolos não-terminais da gramática.

Cada terminal (operador com zero operandos) possui um único estado, os restantes (operadores unários e binários) podem possuir vários estados, dependendo o seu número da combinação destes com os não-terminais de cada um dos operandos.

A título de exemplo, o estado do símbolo terminal *Int* é construído com base na regra 3, que deriva o não-terminal *reg* ao custo de 1 unidade e pela regra 7, que deriva o não-

terminal *addr* ao custo de 0 unidades. O símbolo goal é obtido através da regra 1 aplicada após a regra 3, pelo que o custo final é a contribuição dos custos das duas regras ou seja 1 (1+ 0).

Já no caso do operador *Add*, o não-terminal *reg* é obtido através da regra 5, a um custo de 2 unidades mais uma proveniente da regra 3 do descendente direito. O não-terminal *addr* obtém-se através da regra 8, a um custo de 0 unidades. O *goal* é obtido aplicando-se a regra 1 após a regra 5, o que totaliza um custo de 3 unidades.

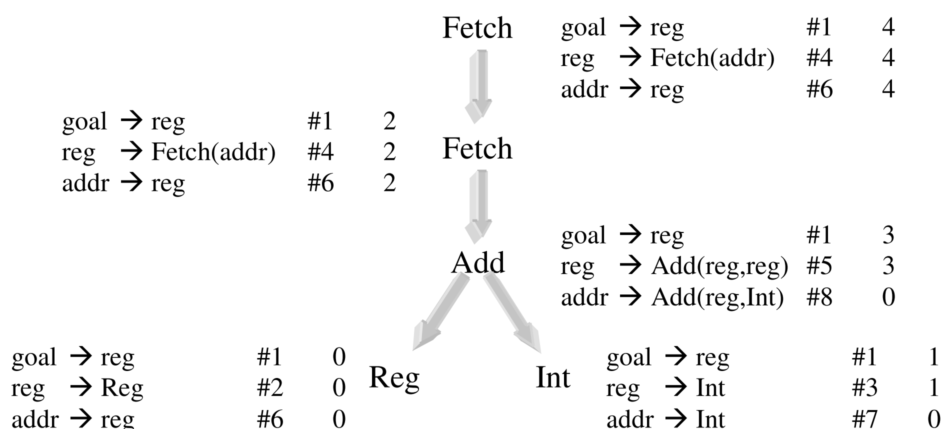


Fig. 7.2 – Exemplo de uma árvore de expressões após o *labelling*.

De notar que um estado, só pode ser composto por regras de um mesmo operador ou então por regras em cadeia. Estas últimas, como já se disse, são regras que se caracterizam por possuir no RHS apenas um não-terminal (ex: #1, #6).

Após o *labelling*, é necessário realizar uma segunda travessia no sentido *top-down*, para se seleccionar os padrões e se executar as respectivas acções. Neste caso e pressupondo *goal* como o não-terminal objectivo para o nodo raiz, obtém-se então a seguinte sequência de padrões e de acções:

Regra	**LHS**		**RHS**	**Custo**	**Acção**
1	goal	→	reg	0	
4	reg	→	Fetch(addr)	2	ld $0, $1
6	addr	→	reg	0	st $0, $1
4	reg	→	Fetch(addr)	2	ld $0, $1
8	addr	→	Add(reg, Int)	0	add $1, $2
2	reg	→	Reg	0	
			Total:	**4**	

No caso da selecção de instruções é conveniente que as acções sejam realizadas aquando do regresso da travessia *top-down*, isto é, numa travessia *post-fix*, de forma a que a ordem de execução seja a correcta.

A implementação de um sistema deste tipo acarreta diversas dificuldades, mas a principal está em determinar os estados e as tabelas de transições, uma vez que todo o sistema de avaliação de custos, que normalmente é implementado através de programação dinâmica e como tal em *run-time*, passa a ser feito em pré-processamento.

Outro aspecto que acarreta algum cuidado é a própria representação destas estruturas, que devem ser codificadas de forma a maximizar a velocidade de execução e minimizar espaço ocupado. Alias, este último é um dos aspectos que todos os sistema tipo *bottom-up* têm de considerar, uma vez que na maior parte dos casos, as tabelas de transição são extremamente grandes. É, assim, habitual que estes sistemas se façam acompanhar por mecanismos de compressão de tabelas e de minimização de estados [Chase87, DDH84].

Estudos mais detalhados de optimização do BURS, foram desenvolvidos por Henry [FHP91, GHS82, HD89, Henry84, Henry89, Henry91], tendo este conseguido obter técnicas, que apesar de bastante elaboradas, permitem reduzir o número de estados, o que por si só reduz o espaço e o tempo de construção das tabelas.

Ao longo da próxima secção, pretende-se mostrar como obter os estados e construir as tabelas de transições, bem como apresentar algumas das optimizações possíveis de utilizar para aumentar a eficiência deste sistema.

7.4 *Implementação do BURS*

A descrição aqui feita, do processo de construção dos estado e tabelas de transições para um sistema do tipo BURS, tem por base uma das mais fieis implementações desta teoria, o *Bottom-Up Rewrite Generator* – BURG. Este sistema foi desenvolvido por Proebsting [FHP91], um dos investigadores mais activos dentro desta área, tendo mesmo apresentado outras implementações de geradores de selectores de instruções, como é o caso do IBURG, que foi o modelo utilizado para desenvolver o sistema de selecção do BEDS, e o WBURG que tem a vantagem de apenas necessitar de uma única travessia (*bottom-up*) para realizar o *labelling* e a selecção dos padrões.

Proebsting também contribuiu com algumas optimizações para o modelo do BURS, como é o caso do *triangle trimming* e do *chain-rule trimming*. Ambas as técnicas permitem optimizar o tamanho das tabelas, através da eliminação dos estados que não contribuem para obter melhores custos, sem que para tal agravem a eficiência do gerador (selector de instruções).

Versões mais simplificadas da teoria do BURS, foram apresentadas por Balachandran, *et al.* [BDB90], que ao eliminarem a capacidade de reescrita e ao introduzirem algumas simplificações na aplicação dos mapas de indexação, permitem gerar sistemas mais simples e eficientes.

7.4.1 Construção dos mecanismos do BURS

Antes de se explicar como se constroem as tabelas de transições, convém perceber o que estas realmente são e como funcionam. Na Fig. 7.3 encontra-se representado uma hipotética tabela de transições para um hipotético operador *Add*.

Como se trata de uma tabela de 4x4, significa que o conjunto de regras deste sistema apenas dá origem a 4 estados distintos e que o operador *Add* possui dois operandos, identificados como *left* e *right*. Pode-se ainda verificar que o descendente direito (*right*), só pode advir dos estados 2 ou 3 e o descendente esquerdo (*left*) dos estados 1,2 ou 4. Qualquer nodo com este tipo de operador só se pode encontrar no estado 1 ou 4.

Como se verifica pela Fig. 7.3 é possível alcançar um mesmo estado a partir de situações distintas, como é o caso de *Add(#1,#3)* e *Add(#2,#3)*.

Add

left \ right	#1	#2	#3	#4
#1			#1	
#2			#1	
#3				
#4		#4		

#1=Add(#1,#3)
#1=Add(#2,#3)
#4=Add(#4,#2)

Fig. 7.3 – Representação da tabela de transições do operador Add.

Genericamente, a função de *labelling* é estruturada da seguinte forma:

```
int label(Nodo *p){
      ...
      switch(arity[p→op]){  // arity indica o número de operandos de p→op
      case 0: ...
              p->state = state(p→op, 0, 0);
              ...
              break;
      case 1: ...
              p->state = state(p→op, label(p→left),0);
              ...
              break;
      case 2: ...
              p→state = state(p→op, label(p→left), label(p→right));
              ...
      }
...
return (p->state);
}
```

Fig. 7.4 – Estrutura da função de *labelling*.

Onde *Nodo** representa um apontador para um nodo da árvore, composto pelo operador (*p→op*), pelo estado (*p→state*) e por dois descendentes (*p→left*, *p→right*). A função *state(...)* determina o estado do nodo com base nos estados dos seus descendentes.

O algoritmo utilizado na implementação do BURG, para determinar os estados e construir as tabelas de transições, considera dois conjuntos: o *State*, onde se colocam os estados conhecidos e o *WorkList*, onde se encontram os estados a processar. Ambos estão inicialmente vazios.

O primeiro passo, que é realizado através da função *ComputeLeafStates( )*, consiste em analisar todos os símbolos terminais, construindo os respectivos estados e adicionando-os ao *State* e ao *WorkList*.

Depois, retira-se um a um, os estados do *WorkList* e determina-se para cada um dos operadores, com pelo menos um operando, quais as transições nele induzidas por esse estado, quando combinado com os restantes estados já processados. Se o resultado for um estado distinto dos já determinados, então é adicionado aos conjuntos *State* e

WorkList, para que também ele possa posteriormente ser processado. O ciclo repete-se até que o conjunto *Worklist* se encontre vazio.

A função principal do algoritmo encontra-se representado na Fig. 7.5, através do qual se pode verificar, que *Worklist* é uma estrutura do tipo *stack* e *State*, um conjunto, em que alguma forma se salvaguarda a relação entre cada um dos seus elementos e os índices das tabelas de transições. *Operators* é o conjunto de todos os operadores com um ou mais operandos.

```
Proc Main( )
Var Global States : Set(Estados)
Var Global Worklist : Stack(Estados)
Var state : Estado
Var op : Operador
Início
    States = ∅
    Worklist = ∅
    ComputeLeafStates( )
    Enquanto Worklist ≠ ∅ Fazer
        state = Pop(Worklist)
        Para ∀ op ∈ Operators Fazer
            ComputeTransitions(op, state)
        FimPara
    FimEnq
Fim
```

Fig. 7.5 – Algoritmo da função principal, para geração do BURS.

Como já foi dito, um estado deve conter a informação sobre as regras que permitem sintetizar ao menor custo cada um dos símbolos não-terminais. Na prática, por cada símbolo não-terminal, o estado possui um item composto pelo próprio símbolo não-terminal, pela regra que o permite sintetizar e pelo respectivo custo. Caso este seja infinito, significa então que a partir desse estado não existe nenhuma regra que derive nesse *não-terminal*.

O custo de cada item de um estado necessita de ser normalizado, isto é, o seu valor só deve depender do próprio estado e não do contexto onde este ocorre. O contra exemplo desta situação, encontra-se representado na Fig. 7.2, onde o operador *Fetch* possui dois estados distintos devido ao contexto em que está inserido. No entanto, como ambos traduzem exactamente a mesma situação, devem então ser substituídos por um único estado, representado da seguinte forma:

LHS		**RHS**	**Regra**	**Custo**
goal	→	reg	#1	0
reg	→	Fetch(addr)	#4	0
addr	→	reg	#6	0

Consegue-se assim reduzir o número total de estados e como tal o tamanho das tabelas e o tempo de geração das mesmas. De notar que este processo é fundamental

para que o número de estados não seja potencialmente infinito, o que inviabilizaria a construção de todo o sistema.

Desta forma, o custo relativo ou custo delta é determinado subtraindo a cada não-terminal o valor do de menor custo. O algoritmo de normalização encontra-se representado na Fig. 7.6.

```
Proc NormalizeCosts(state : Estado)
Var delta : Inteiro
Var n : Não Terminal
Início
      delta = min∀i. {state [i].cost}
      Para ∀ n ∈ Nonterminals Fazer
               state [n].cost = state[n].cost - delta
      FimPara
Fim
```

Fig. 7.6 – Algoritmo de normalização de custos.

Os estados dos símbolos terminais determinam-se com base nas regras em que o símbolo surge isolado do RHS da produção, retendo-se apenas as que permitem sintetizar cada um dos não-terminais ao menor custo. Depois normalizam-se os custos e aplicam-se as regras em cadeia na tentativa de se conseguir sintetizar os não-terminais que ainda não foram alcançados através das primeiras regras, o que por si só, permite aumentar o espaço de soluções do selector de instruções e por vezes reduzir os custos. O processo termina após a inserção dos novos estados em *State* e em *Worklist*. O algoritmo encontra-se representado na Fig. 7.7.

```
Proc ComputeLeafStates( )
Var leaf :Terminal
Var r : Regra
Var state : Estado
Var n : NãoTerminal
Início
      Para ∀ leaf ∈ Leaves Fazer
               state = ∅
               Para ∀ r: n ← leaf Fazer
                        Se r.cost< state [n].cost Então
                                 state [n] = {r.cost,r}
                        FimSe
               FimPara
               NormalizeCosts(state)
               Closure(state)
               WorkList = Append(WorkList, state)
               States = States ∪ {state}
               leaf.state = state
      FimPara
Fim
```

Fig. 7.7 – Algoritmo para determinar os estados do símbolos terminais.

O algoritmo do *Closure(state)*, consiste em aplicar iterativamente as regras em cadeia ao estado, até que não se produzam mais alterações.

```
Proc Closure(state : Estado)
Var r : Regra
Var n, m : Não Terminal
Var cost : Inteiro
Início
    Repetir
        Para ∀ r: n -> m tal que m ∈ Nonterminals Fazer
            cost = r.cost + state [m].cost
            Se cost < state [n].cost Então
                state [n] = {cost,r}
            FimSe
        FimPara
    Até que não ocorram alterações no estado
Fim
```

Fig. 7.8 – Algoritmo para aplicar as regras em cadeia.

O passo seguinte consiste em remover um a um, os estados de *Worklist* e testá-los para todo e qualquer operador com mais do que um operando. Este procedimento é feito através da função *ComputeTransitions(op, state)*, que analisa o efeito de *state* para todas as dimensões de *op*. Para tal, determina um pseudo-estado designado por *pState* e que é formado apenas pelos itens de *state* que podem contribuir para a *i*-nésima dimensão de *op*. Na prática, como estes pseudo-estados só servem para determinar as transições, não necessitam de armazenar a informação referente às regras, mas apenas os símbolos não-terminais e o custo necessário para os sintetizar.

Para o correcto funcionamento do algoritmo, é necessário manter para cada uma das dimensões de cada um dos operadores, os respectivos *pStates* (*op.reps[i] = {pStates}*), bem como a relação entre estes e o estado de onde derivam (*op.map[i][state] = pState*).

O processo prossegue combinando o novo *pState* com todos os *pState* entretanto determinados para cada uma das restantes dimensões. Depois avalia-se o custo de cada uma destas combinações, quando aplicada às várias regras que utilizam *op*, formando um estado com aquelas que permitem obter ao menor custo cada um dos não-terminais. O qual passa a representar *op* aquando da ocorrência da respectiva combinação dos *pStates*. No caso deste ainda não ter sido previamente determinado, é então reunido ao *State* e adicionado ao *Worklist*. O algoritmo final é o seguinte:

```
Proc ComputeTransitions(op : Operador , state : Estado)
Var i, cost : Inteiro
Var pSate, result : Estado
Início
    Para i de 1 Até op.arity Fazer
        pState = Project(op, i, state)
        op.map[i][state] = pState
        Se pState ∉ op.reps[i] Então
```

```
                    op.reps[i] = op.reps[i] ∪ {pState}
                    Para ∀(s1,...,pState, sop.arity) tal que ∀j≠i, sj ∈ op.reps[j] Fazer
                        result = ∅
                        Para ∀ r:n->op(m1,...,mop.arity) Fazer
                            cost = r.cost + pState[mi].cost + Σj≠i sj[mj].cost
                            Se cost < result[n].cost Então
                                result[n] = {cost,r}
                            FimSe
                        FimPara
                        Trim(result)
                        NormalizeCosts(result)
                        Se result ∉ States Então
                            Closure(result)
                            WorkList = Append(WorkList, result)
                            States = States ∪ {result}
                        FimSe
                        op.transitions[s1,..,pState,..,sop.arity] = result
                    FimPara
                FimSe
            FimPara
Fim
Proc Project(op : Operador, i : Inteiro, state : Estado) : Estado
Var pState : Estado
Início
    pState = ∅
    Para ∀ n ∈ Nonterminals Fazer
        Se ∃ r: m-> op(n1,...,n,...nop.arity) Então
            pState[n].cost = state [n].cost
        FimSe
    FimPara
    NormalizeCosts(pState)
    Retornar pState
Fim
```

Fig. 7.9 – Algoritmo para determinar os estados dos não-terminais.

7.4.2 Optimizações de Proebsting

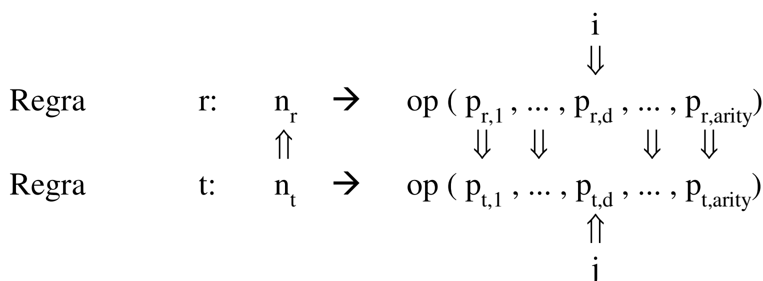
Proebsting desenvolveu algumas técnicas de optimização para o BURS [Proeb91, Proeb92a, Proeb92b, Proeb95a], que designou por *Triangle Trimming* e *Chain Rule Trimming*. Ambas foram aplicadas ao BURG através da função *Trim(state)* do algoritmo anterior. Estas optimizações tiveram por base outras duas, propostas por Henry e designadas por *sibling* e *demand trimming*, que partilham os mesmos princípios, divergindo apenas nos algoritmos utilizados.

A ideia base deste tipo de técnicas consiste em fazer com que dois ou mais estados não idênticos o passem a ser, de forma a se poder eliminar um deles, acelerando assim o processo de geração dos selectores, uma vez que se reduz o número de combinações possíveis de se obter para cada operador e implicitamente o tamanho das tabelas.

Os algoritmos propostos por Proebsting e por Henry, divergem na forma como se propõem alcançar estes objectivos. Mas ambos tentam eliminar todos os não-terminais, para os quais se prove, que não existe nenhuma solução feita com base neles, que não possa ser obtida através de outros não-terminais a um custo igual ou inferior. Com este processo consegue-se eliminar alguns dos itens que compõem os estados, reduzindo assim algumas das diferenças que possam existir. A remoção dos não-terminais não deve afectar a capacidade de determinar a cobertura óptima de uma qualquer árvore de expressões.

A técnica de *Triangle Trimming* verifica para todos os estados, se existe um par de símbolos não-terminais (*i*, *j*), tal que *j* possa substituir *i*, em qualquer dimensão de toda e qualquer regra onde *i* é utilizado, produzindo o mesmo resultado mas a um custo não superior. Nessa situação, *i* é removido do respectivo estado.

Diz-se então, que se para um dado operador *op*, existirem duas regras, *r* e *t*, onde seja possível utilizar na dimensão *d*, respectivamente os não-terminais *i* e *j* (quer directamente, quer por regras em cadeia), tal que:



É possível prova que *j* pode substituir *i*, se for possível derivar cada um dos $p_{t,x}$ a partir de $p_{r,x}$ ($p_{r,x} \Rightarrow p_{t,x}$), para x =1...arity $\wedge$ x≠d, e n_r a partir de n_t ($n_t \Rightarrow n_r$), de tal forma que:

$$\begin{aligned}&state[i].cost + r.cost + Cost(i \Rightarrow p_{r,d}) \geq \\ &state[j].cost + t.cost + Cost(j \Rightarrow p_{t,d}) + Cost(n_t \Rightarrow n_r) + \sum_{k \neq d} Cost(p_{r,k} \Rightarrow p_{t,k})\end{aligned}$$

NOTA: A operação $n_1 \Rightarrow n_2$ significa que é possível sintetizar o não-terminal n_2 a partir do não-terminal n_1, aplicando-se a n_1 uma ou mais regras em cadeia até se obter n_2. E $Cost(n_1 \Rightarrow n_2)$, representa o menor dos custos para sintetizar n_2 a partir de n_1.

```
Proc ChainRuleTrimming(state :Estado)
Var n, m : Não Terminal
Var c : Inteiro
Início
    Para ∀ n ∈ state Fazer
        Para ∀ m ∈ state Fazer
            c = Cost(m ⇒n)
            Se state[n].cost ≥ state[m].cost + c Então
                state[n] = {∝, ⊥}
            FimSe
        FimPara
    FimPara
Fim
```

Fig. 7.10 – Algoritmo de *Chain Rule Trimming*.

O *Chain Rule Trimming*, há semelhança da técnica anterior, tenta também reduzir o número de estados eliminando potenciais diferenças que possam existir entre estes. Só que agora a ideia consiste em tentar eliminar itens de um estado, que possam posteriormente ser sintetizados a partir de outros itens do mesmo estado, a um custo menor ou igual, utilizando regras em cadeia. Trata-se como tal de uma espécie de *anti-Closure(...)*, podendo levar a pensar que existe alguma contradição nesta forma de optimização. Mas convém reparar que a função *Trim(...)*, responsável por ambas as formas de optimização, é invocada antes da normalização dos custos e da condição que avalia se o estado até ai determinado pertence ou não ao conjunto *State*. Só no caso deste não pertencer, é que se executa a função *Closure(...)*. A qual garante que os não-terminais (associados aos itens) eliminados pelo *chain rule trimming*, voltam a ser alcançados, mas com a vantagem de entretanto se ter reduzido o número de estados. O procedimento encontra-se representado na Fig. 7.10.

7.4.3 Optimizações de Chase

Chase desenvolveu um método de compactação de tabelas [Chase87], utilizado por Pelegri na implementação do BURS, e que consiste em analisar se existem entradas (linhas ou colunas) que sejam iguais. Se tal acontecer, eliminam-se todas as ocorrências excepto uma e através da utilização de vectores de indexação reorienta-se as entradas da tabela, como se encontra representado na Fig. 7.11.

Comprova-se experimentalmente, que a maior parte das linhas e colunas das tabelas de transições, não contém informação útil e mesmo quando contêm, é na maior parte das vezes redundante. Condições que permitem reduzir substancialmente o tamanho das tabelas.

Pelegri, no sentido de conseguir uma maior compactação, faz a gestão dos vectores de indexação ao nível do bit, pelo que os elementos que compõem os vectores possuem apenas os bits necessários, arredondados à potência de dois mais próxima (2,4,8,...), para representar o número de estados da respectiva tabela. Este tipo de tratamento advém do facto de se ter verificado que a utilização de bytes ou múltiplos de bytes, implicava algum desperdício de espaço.

Exemplo 7.1

A Fig. 7.11 elucida o processo de compactação de tabelas utilizando vectores de indexação, em que numa primeira fase reduzem-se as linhas e só depois as colunas da tabela.

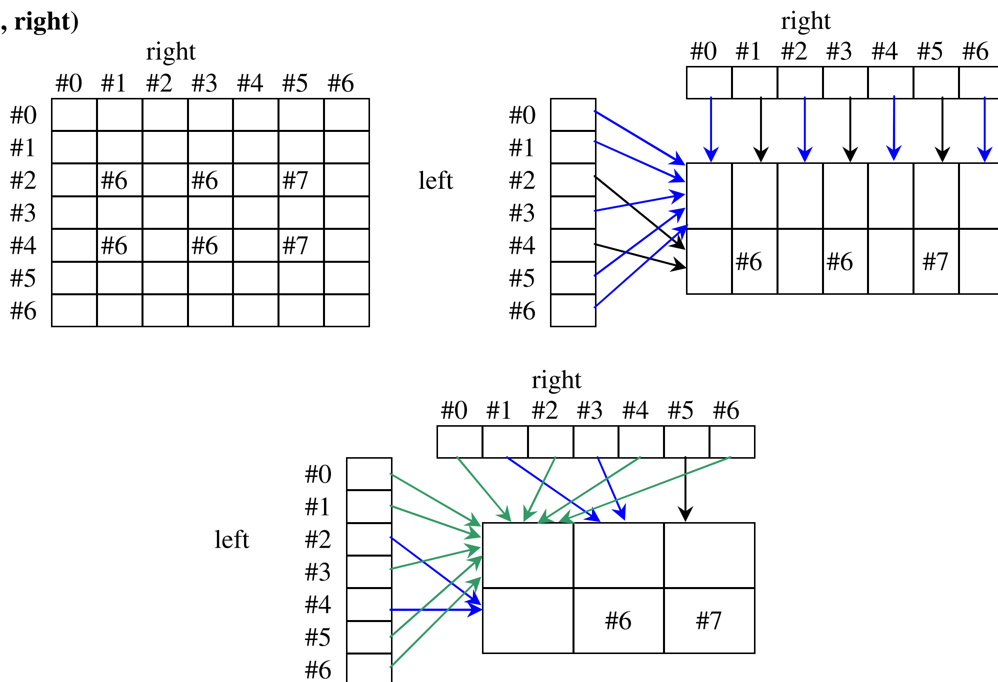


Fig. 7.11 – Compactação das tabelas, recorrendo a vectores de indexação.

7.4.4 Optimizações de Henry

O trabalho desenvolvido por Henry [GHS82, HD89, Henry84, Henry89, Henry91], consistiu em transformar a informação contida nas tabelas de transições, numa solução mista de código (*hard code*) e dados, com o objectivo de diminuir o tamanho e o tempo de acesso às tabelas.

Como se pode verificar pela função *ComputeTransitions(...)* da Fig. 7.9, os estados são determinados em função de um operador. É como tal natural, que os respectivos identificadores sejam valores praticamente sequenciais ou então muito próximos. Acontece no entanto, que se o número total de estados for superior a 256, passa a ser necessário utilizar dois bytes por cada um dos elementos das tabelas de transições, mesmo que estas apenas guardem meia dúzia de valores distintos.

Henry, contorna este problema aplicando diversas tácticas. Uma consiste em ajustar os valores dos identificadores, de tal forma que só seja necessário utilizar um byte por cada elemento da tabela. Seja então [a,b] o intervalo, no qual se encontram todos os identificadores contidos numa determinada tabela (excepto o nulo), tal que *a* e *b* representam respectivamente o estado com o menor e o maior identificador. É então possível reescrever o intervalo, da seguinte forma:

$$[\ 1, b\text{-}a\text{+}1] + a - 1 \qquad \text{Eq. 7.3}$$

O que a nível da tabela de transições, consiste em subtrair *a-1* a cada elemento cujo valor é diferente de nulo. Aquando da operação de *labelling*, ao valor obtido da tabela adiciona-se *a-1*. Esta reformulação apenas compensa se *b-a +1 < 256*, o que permite definir as tabelas com apenas um byte por elemento. Caso tal não aconteça, Henry, opta por decompor a tabela de forma a que cada componente não indexe mais do que 256 estados distintos, para tal, há que antes realizar algumas permutas entre colunas e entre linhas, de forma a minimizar os elementos em comum das várias sub-tabelas. A decomposição faz-se segundo o esquema da Fig. 7.12.

Henry também utiliza os mecanismos de compactação de tabelas propostos por Chase, mas ao contrário da implementação de Pelegri que trata as tabelas como entidades individuais, Henry opta por as tratar como uma entidade única e tira proveito disso, para compactar também os vectores de indexação. O Exemplo 7.2 ilustra como é que tal acontece.

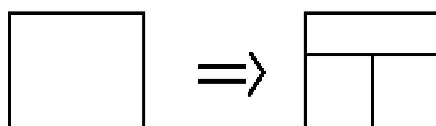
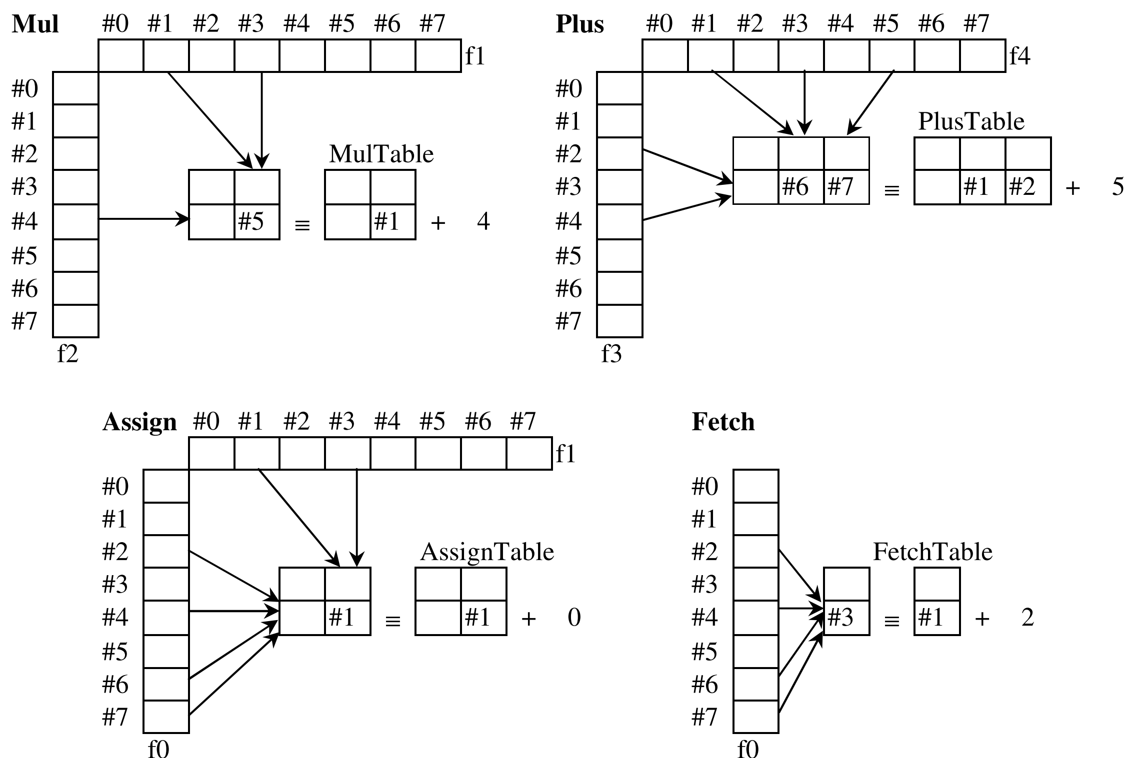


Fig. 7.12 – Esquema de decomposição de tabelas.

<u>Exemplo 7.2</u>

Pressupondo que de uma dada gramática e após a aplicação dos vectores de indexação, resultaram as seguintes tabelas de transições:



De forma a simplificar a representação, não se colocaram as ligações entre os vectores e os estados nulos. Ao lado de cada tabela “comprimida” encontra-se a tabela equivalente após se aplicar a reformulação da Eq. 7.3.

Mesmo para um problema pequeno como este, verifica-se que existem vectores de indexação que são comuns a mais do que um operador, como por exemplo, o vector do *Fetch* e o vector do operando esquerdo de *Assign* (f0). A prática demonstrou que este tipo de ocorrências é vulgar, pelo que Henry propõe a partilha dos vectores pelas tabelas de transições, passando estas a ser uma única entidade.

De forma a reduzir ainda mais o espaço, os elementos de cada vector são constituídos pelo número de bits correspondente à potência de 2 (2,4,8,...) mais próxima da quantidade a representar. Assim, por exemplo, f0 como só faz referência aos índices 0 e 1 das tabelas, apenas necessita de um bit por elemento. Na prática utilizam-se 2 bits, uma vez que facilita os mecanismos de compactação e manipulação binária. Os vectores são então representados da seguinte forma:

```
static struc {  unsigned int f0:2;
                unsigned int f1:2;
                unsigned int f2:2;
                unsigned int f3:2;
                unsigned int f4:2;
}mapv[ ] = {    { 0, 0, 0, 0, 0,},        //#0
                { 0, 1, 0, 0, 1,},        //#1
                { 1, 0, 0, 1, 0,},        //#2
                { 0, 1, 0, 0, 1,},        //#3
                { 1, 0, 1, 1, 0,},        //#4
                { 0, 0, 0, 0, 2,},        //#5
                { 1, 0, 0, 0, 0,},        //#6
                { 1, 0, 0, 0, 0,},        //#7
              // f0,f1,f2,f3,f4
};
```

Fig. 7.13 – Esquema da organização dos mapas de indexação.

Como já se explicou, a operação de *labelling* consiste em identificar o operador do nodo, por exemplo, através de uma estrutura tipo *switch...case...*, e depois consultando a respectiva tabela e com base nos estados dos nodos descendentes, determina-se o estado do nodo actual. Neste exemplo concreto, os procedimentos de consulta das tabelas realizam-se através das seguintes macros:

```
#define AssignState(l,r)     (temp = AssignTable[mapv[l].f0][mapv[r].f1]?temp+0:0)
#define FetchState(l)        (temp = FetchTable[mapv[l].f0]?temp+2:0)
#define MulState(l,r)        (temp = MulTable[mapv[l].f2][mapv[r].f1]?temp+4:0)
#define PlusState(l,r)       (temp = PlusTable[mapv[l].f3][mapv[r].f4]?temp+5:0)
```

Fig. 7.14 – Macros para determinar o *labelling*.

Henry, para além dos vectores de indexação das tabelas de transições, codifica também a informação sobre os estados, necessária para se determinarem as regras a aplicar. Pressupondo então, que os estados obtidos para a gramática deste exemplo, são os seguintes:

Estado	nt	Regra	Estado	nt	Regra	Estado	nt	Regra
	reg	7		reg	0		reg	6
	con	0		con	1		con	0
1	addr	0	2	addr	3	3	addr	0
	x	0		x	0		x	0
	y	0		y	0		y	0
	reg	0		reg	0		reg	0
	con	2		con	0		con	0
4	addr	3	5	addr	0	6	addr	4
	x	0		x	8		x	0
	y	9		y	0		y	0

Estado	nt	Regra
	reg	0
	con	0
7	addr	5
	x	0
	y	0

Em que *reg*, *con*, *addr*, *x* e *y*, são não-terminais da gramática. Os dois últimos não-terminais (*x* e *y*) surgem através do processo de normalização da gramática, pelo que não são incluidos na codificação.

De forma a juntar a informação sobre os estados a *mapv*, à que os decompor, agrupando a informação pelos vários não-terminais, de tal forma que:

```
     f5    f6    f7                   f5+2  f6+0  f7+5
     0     0     0     // #0          0     0     0
     0     0     7     // #1          0     0     2
     3     1     0     // #2          1     1     0
     0     0     6     // #3    ≡     0     0     1
     3     2     0     // #4          1     2     0
     0     0     0     // #5          0     0     0
     4     0     0     // #6          2     0     0
     5     0     0     // #7          3     0     0
//  addr   con   reg                  addr  con   reg
```

Como os identificadores das regras podem assumir valores muito elevados e uma vez que os valores por cada vector (f_i) são muito próximos, aplica-se a reformulação da Eq. 7.3.

Henry desenvolveu várias outras optimizações, algumas das quais para o seu próprio sistema, o CODEGEN. Julgo no entanto que as mais relevantes são as que aqui se apresentaram, tanto que foram integradas na maior parte dos sistemas com base no BURS.

7.5 *IBURG*

O BURS é sem dúvida um dos processos mais rápidos para realizar a selecção ou a reescrita de árvores. É exactamente nestes dois aspectos, que residem as suas principais vantagens. Infelizmente a estas contrapõem-se também algumas desvantagens, uma das quais, advém do próprio processo para determinar os estados e gerar as tabelas de transições, o qual é extremamente complexo, pelo menos quando se pretende obter sistemas optimizados.

Mas mais inconveniente, pode ser a impossibilidade de avaliar o contexto de execução em *run-time*, uma característica que é completamente antagónica à filosofia de concepção do BURS, mas que pode ser extremamente importante para a implementação de determinados sistemas, como é o caso da alocação de registos. Com a agravante, de que a maioria dos selectores de instruções permite realizar este tipo de operações.

Uma solução de compromisso, de nome IBURG, foi desenvolvida por Fraser *et al.* [FHP92] e partilha muitas das características do BURG, designadamente a gramática, o processo *bottom-up* de *labelling* e o *top-down* de selecção das regras, e todo um conjunto de pequenas operações. Aliás, quer Fraser, quer Proebsting, participaram ambos na concepção do BURG e do IBURG.

Mas as diferenças também são relevantes, é que o IBURG ao optar por uma solução mais *hard-code*, dispensa a construção das tabelas de transições e, em parte, dos próprios estados. Como tal, deixa de ser necessário conhecer previamente o custo de cada uma das regras da gramática.

Lógico que o sistema de reconhecimento se torna mais lento, mas com a vantagem de permitir utilizar programação dinâmica para avaliar os custos em *run-time*. Convém no entanto aqui esclarecer, que o IBURG só utiliza custos estáticos, mas que facilmente podem ser substituídos por funções de custo.

7.5.1 Operação de *labelling*

É na operação de *labelling*, ou mais concretamente no *state(...)*, em que o BURG e o IBURG mais se distinguem. É que no primeiro caso, como os custos são previamente conhecidos torna-se possível determinar os estados que caracterizam um qualquer nodo de uma expressão válida da gramática. Já no segundo caso, como os custos podem variar, faz com que o número de estado seja teoricamente infinito, o que inviabiliza qualquer tentativa de os quantificar ou de determinar. Por outro lado, como não se conhecem os estados, também não é possível relacioná-los com os nodos das árvores de expressões, o que leva a que cada nodo se caracterize por possuir um estado próprio que não partilha com mais nenhum.

A estrutura de um estado é assim composta pelos seguintes campos:

```
struct state {
    int op;
    struct state *left, *right;
    short cost[n_NT];
    short rule[n_NT];
};
```

Em que *n_NT*, representa o número de não-terminais da gramática.

A estrutura da operação de *labelling* em si, é em tudo semelhante à que se encontra representada na Fig. 7.4, mas a função de *state(...)*, apesar de utilizar os mesmos princípios, é bastante diferente. Esta começa por alocar dinamicamente um estado, de seguida inicializa os campos, colocando por exemplo, os custos a infinito. Depois com base no operador e nos estados dos nodos descendentes determina o estado do nodo actual.

Caso o operador seja do tipo terminal, o respectivo estado é estático, pelo que apenas é necessário preencher os campos com os custos e as regras que permitem sintetizar cada um dos não-terminais ao menor custo. Neste aspecto o IBURG é em tudo semelhante ao BURG, uma vez que as regras são seleccionadas em pré-processamento, utilizando mesmo um algoritmo semelhante ao da Fig. 7.7.

Para operadores com um ou mais operandos, é necessário determinar em run-time, o custo de utilização de cada regra. O que não é mais do que adicionar ao custo da própria regra, o custo de sintetizar cada um dos seus operandos a partir dos nodos descendentes. A regra passa a vigorar para esse estado, caso seja a que permite sintetizar o não-terminal que se encontra do seu LHS ao menor custo. O Exemplo 7.3 ilustra esta situação para duas regras distintas de um mesmo operador.

Exemplo 7.3

O seguinte trecho de código pertence à função *state(...)*,obtida de uma gramática que possui entre outros elementos, um operador (*Plus*), quatro regras, duas onde o operador é utilizado e mais duas regras em cadeia (*reg:addr*, *con:addr*), e ainda três símbolos não-terminais, o *reg* (*reg_NT*), o *addr* (*addr_NT*) e o *con* (*con_NT*). As variáveis *p*, *left* e *right*, representam respectivamente o estado do nodo actual, e os descendentes esquerdo e direito.

```
case PLUS:
    /* 10 reg: Plus( reg, addr)       2 */
    c = left→cost[reg_NT] + right→cost[add_NT] + 2
    if( c < p->cost[reg_NT]){
            p→cost[reg_NT] = c;
            p→rule[reg_NT] = 10;
    }
    /* 11 addr: Plus( con, reg)       1 */
    c = left→cost[con_NT] + right→cost[reg_NT] + 1
    if( c < p->cost[addr_NT]){
            p→cost[addr_NT] = c;
            p→rule[addr_NT] = 11;
            /* 20 reg: addr           2 */
            /* 21 con: addr           0 */
            closure_addr(p);
    }
```

♦

Se para um não-terminal nt_x, existir pelo menos uma regra do tipo $nt_y : nt_x$, então o IBURG gera uma função de nome $closure_nt_x(p)$, em que dado um estado p, avalia se $Custo(nt_y : nt_x) + p \rightarrow cost[nt_x] < p \rightarrow cost[nt_y]$ e se tal acontecer, substitui o custo de

p→cost[nty] por *Custo*($nt_y : nt_x$) + *p→cost*[nt_x], ou seja, aplica ao estado *p* todas as regras em cadeia em que nt_x surge do RHS.

Caso exista alguma regra do tipo $nt_z : nt_y$, então *closure_*nt_x*(p)* invoca *closure_*nt_y*(p)*, e assim por adiante. O Exemplo 7.4 ilustra uma destas situações.

Exemplo 7.4

Na continuação do exemplo anterior, temos a seguinte função de *closure*:

```
static void closure_addr(struct state *p) {
      /* 20 reg : addr           2 */
      if (p→cost[addr_NT] + 2 < p→cost[reg_NT]) {
              p→cost[reg_NT] = c + 2;
              p→rule[reg_NT] = 20;
      }
      /* 21 con : addr           0 */
      if (p→cost[addr_NT] + 0 < p→cost[con_NT]) {
              p→cost[con_NT] = c + 0;
              p→rule[con_NT] = 21;
      }
}
```

♦

Pelo que já foi dito, verifica-se que nada impede a substituição do custo de cada uma das regras por funções de custo, uma vez que não existe qualquer compromisso na estrutura do algoritmo com os custos atribuídos aquando a especificação das regras.

7.5.2 Selecção das regras

A selecção das regras no IBURG não podia ser mas simples, uma vez que toda a informação necessária encontra-se representada no próprio estado, pelo que basta conhecer qual o não-terminal objectivo e realizar uma travessia *top-down*, para se determinarem as regras a utilizar.

Apenas para compactar um pouco a forma de representar os estados, os autores do IBURG utilizam vectores de regras, um por cada um dos símbolos não-terminais, mas de tal forma, que cada vector contém apenas as regras que sintetizam determinado não-terminal. O Exemplo 7.5 elucida esta situação.

Exemplo 7.5

Ainda dentro do contexto dos exemplos anteriores, seria possível obter os seguintes vectores de regras:

```
static short decode_reg[ ]={0, 10, 20}
static short decode_addr[ ]={0, 11}
static short decode_con[ ]={0, 21}
```

Possibilitando a redefinição de *struct state* da seguinte forma:

```
struct state {
	...
	struct {
		unsigned reg_nt:2;
		unsigned addr_nt:2;
		unsigned con_nt:2;
	} rule;
};
```

Conseguindo assim reduzir o espaço ocupado por cada estado, o que pode no entanto não ser muito relevante, pelo menos quando se considera apenas uma árvore. Mas acontece por vezes ser necessário realizar o *labellling* a todas as árvores e só depois é que se determinam as regras, como é o caso do sistema de alocação global do BEDS, nestas circunstâncias e uma vez que estamos a falar de muitas árvore, este sistema de compactação pode-se mostrar muito vantajoso.

7.6 *O selector de instruções do BEDS*

O selector de instruções que acompanha o BEDS, ao qual passaremos a designar por BBEG - **B**EDS **Back-End G**enerator, foi construído com base no IBURG. O pouco que se alterou está essencialmente relacionado com a integração deste último no BEDS, pelo que as alterações se verificam mais ao nível do gerador do que ao nível do selector de instruções. Alias, a única alteração directamente relacionada com a selecção e que eventualmente é digna de nota, é a utilização de funções de custos em vez de custos constantes.

Convém aqui salientar que o BBEG não restringe a sua utilização ao contexto de desenvolvimento do BEDS, alias o seu funcionamento é por defeito independente deste último, conforme se vai esclarecer no próximo capítulo.

Um dos aspectos em que o BBEG é claramente diferente do IBURG, é na gramática utilizada, uma vez que esta, no BBEG, não só serve para descrever as instruções e a relação destas com as expressões da RI, mas também para especificar o *register set*, relacionar os identificadores da RI com os identificadores do gerador, activar os módulos de alocação, etc. Todos estes aspectos também influenciam o próprio gerador, uma vez que este para além das funções típicas do IBURG, passa a gerar a informação sobre os registos, as macros que interligam o BBEG com o BEDS, os algoritmos de alocação e desalocação, as funções para emitir as instruções, alguns mecanismos de coordenação entre o sistema de alocação e o de selecção, etc.

O BBEG permite ainda associar às regras, condições de contexto, procedimentos a realizar após a selecção, e relacionar os operandos e operadores das expressões com os parâmetros das instruções. Como se disse acima, muitos destes aspectos são abordados mais detalhadamente ao longo do próximo capítulo.

8 Utilização do Sistema BEDS

Este capítulo começa por apresentar os passos fundamentais no desenvolvimento de um compilador dentro da infra-estrutura do BEDS, depois descreve-se a gramática da linguagem utilizada pelo BBEG, o processo de geração da representação intermédia e a utilização dos diversos módulos que compõem o BEDS.

Antes de se avançar, convém lembrar que o desenvolvimento de compiladores está longe de ser uma tarefa que se possa considerar simples, mesmo quando se utilizam ferramentas próprias.

8.1 *Estrutura de um projecto*

O BEDS não é por si só um sistema completo, quer quando visto como uma ferramenta de apoio ao desenvolvimento de compiladores, ou como um conjunto de componentes que podem integrar o compilador. Permite no entanto auxiliar o desenvolvimento de algumas das suas componentes, quer gerando-as a partir de uma descrição ou fornecendo-as directamente através de um conjunto de classes, as quais devem ser parametrizadas à medida de cada caso.

Antes de se descreverem os passos essenciais à construção de um compilador, é de todo conveniente esclarecer as diferentes fases temporais que envolvem o BEDS e a sua aplicação ao desenvolvimento de compiladores. A fase primordial, que precede todas, é a própria construção dos sistemas de geração de geradores, ou seja a construção do BBEG e a implementação das classes que compõem o BEDS. Como não é directamente relevante para a construção de um compilador não é aqui abordada.

A fase seguinte, que será considerada como primeira fase de um projecto para a construção de um compilador, consiste em utilizar os mecanismos construídos na fase anterior, designadamente o BBEG, para que com base numa descrição (especificação) gere algumas das componentes que vão integrar o compilador. Componentes essas que, por serem de alguma forma dependentes de factores externos ao BEDS, encontram-se inacabadas, ou em *stand-by*, no processo de concepção e desenvolvimento.

Esta fase inclui: a especificação das características do processador e da relação que estas possam ter com a representação intermédia; a selecção do processo de alocação, se local ou global; o desenvolvimento das funções de custo, de avaliação das condições de contexto e eventualmente das funções para gerar código máquina; e o próprio processo de geração do selector de instruções e das rotinas de alocação (local).

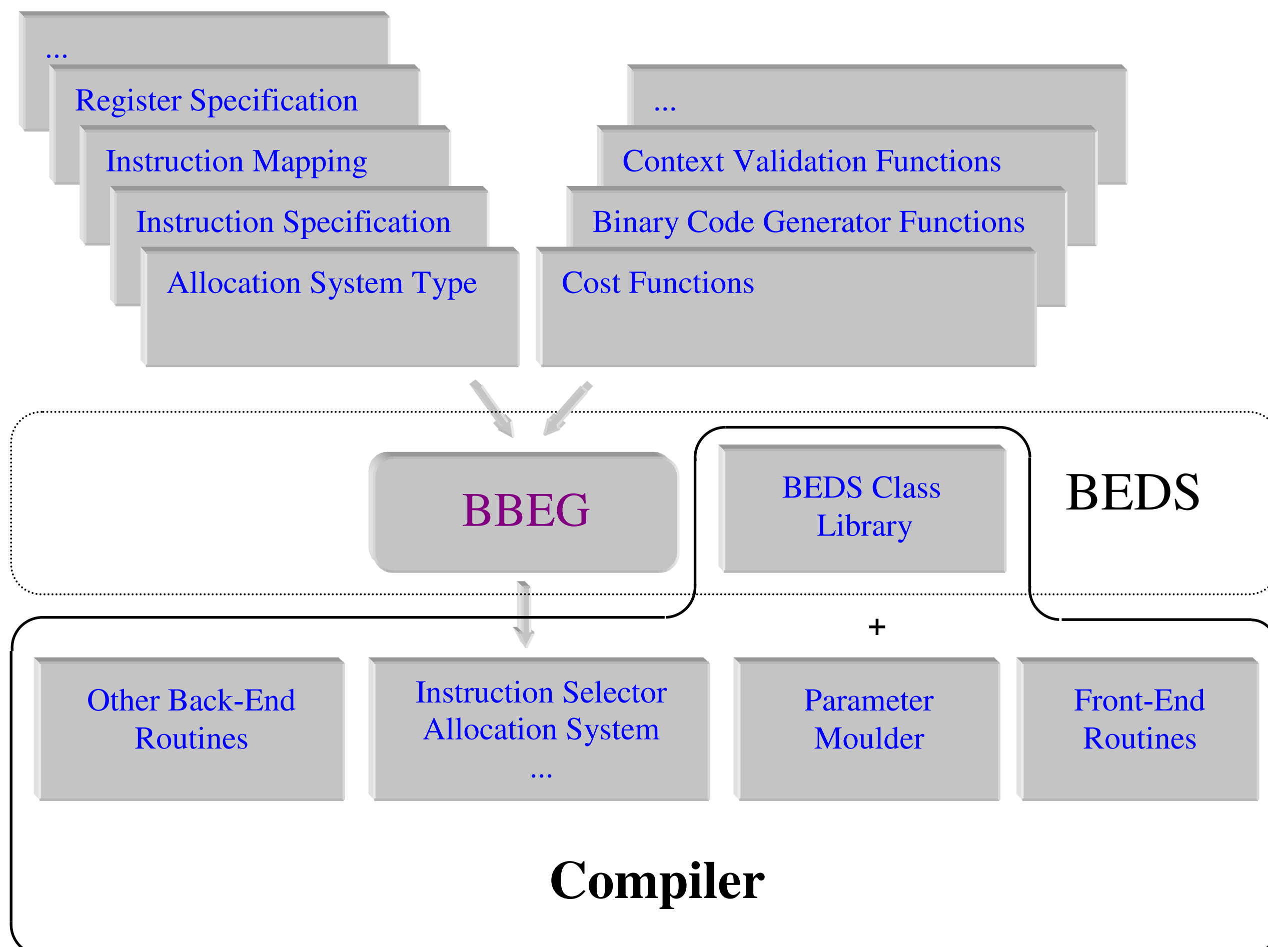


Fig. 8.1 – Modelo com os vários intervenientes no desenvolvimento de um compilador.

A construção do compilador, propriamente dita é feita na segunda fase e consiste em reunir as rotinas obtidas da fase anterior com as classes do BEDS e com as rotinas do *front-end*, de tal forma que, na terceira e última fase (a da compilação do programa fonte), seja possível construir a representação intermédia e aplicar todos os processos necessários para se obter o código final.

Assim, na segunda fase é necessário associar ao gerador da representação intermédia, as rotinas que permitem instanciar os objectos das classes do BEDS, "parametrizando" eventualmente alguns desses objectos de forma a moldá-los às necessidade concretas do compilador que se está a construir (*Parameter Moulder* da Fig. 8.1). A instanciação das classes aplica-se à construção da representação intermédia, mas também à invocação das rotinas de optimização e geração de código final.

Na realidade as funções da segunda fase acabam por ficar incluídas num nível semelhante ao da primeira fase, mas agora do *front-end*, isto porque, quer o analisador

semântico, quer o gerador de código intermédio, são normalmente construídos utilizando ferramentas de geração (lex, yacc, ox, etc).

A terceira fase é apenas simbólica uma vez que já nada tem haver com a construção do compilador, mas apenas com a sua execução, ou seja com o seu funcionamento aquando da compilação de um programa fonte. É no entanto importante distingui-la das restantes de forma a manter claro a função das fases anteriores.

A próxima secção apresenta a gramática da linguagem de especificação do BBEG, a BBEGL - *BEDS Back-End Generator Language*, e descreve alguns dos aspectos inerentes à primeira fase de um projecto para a construção de um compilador. As duas secções seguintes descrevem como utilizar os mecanismos fornecidos pelo BEDS para integrar o processo de compilação (segunda fase do projecto de construção de um compilador).

8.2 *BEDS Back-End Generator Language*

Como já se disse, o BBEG foi desenvolvido utilizando os mecanismos de geração do IBURG e como tal as componentes por este geradas, mas a gramática foi completamente reformulada. Assim, foi necessário desenvolver de raiz o reconhecedor desta nova linguagem de especificação, para tal, recorreu-se a duas outras ferramentas de concepção de processadores de linguagens: o Eli [GHKSW90] para construir o analisador léxico, sintáctico e semântico do BBEG, e o Lex para situações localizadas para as quais não se justificava mais do que uma simples análise léxica.

O Eli é uma ferramenta de concepção de processadores de linguagens, composta por diversos programas, cada um responsável por auxiliar a concepção de uma das fases do processador. É sem dúvida uma referência dentro deste tipo de ferramentas pelo que, à semelhança do Lex, dispensa qualquer tipo de apresentação mais detalhada.

A escolha destas ferramentas deve-se à maturidade de ambas, à documentação existente, conhecimento e apoio disponível, e essencialmente por utilizarem C/C++ como linguagens de suporte, o que é fundamental para a integração dos mecanismos gerados com os módulos do BEDS.

Após esta breve apresentação sobre as ferramentas utilizadas no desenvolvimento do BBEG e uma vez que não é sobre elas que se pretende aqui falar, pode-se então começar a descrever como se procede a relação entre os componentes gerados pelo BBEG e as restantes componentes do *back-end* e do *front-end*, bem como explicar quais os mecanismos necessários de garantir para que este relacionamento se proceda conforme o esperado. A secção conclui com a descrição da linguagem de especificação do BBEG.

8.2.1 Interface entre *front-end* e *back-end*

Como o BBEG foi concebido para ser completamente independente de qualquer uma das restantes componente do BEDS, resulta que o código por ele gerado também se mantém independente dessas rotinas, pelo que não é obrigado a partilhar com este as mesmas estruturas de informação, designadamente as utilizadas na representação das árvores de expressões, ou a forma como identifica os operadores, uma vez que por

defeito não existe qualquer relação entre os operadores das árvores que provêm do *front-end* e os operadores definidos na especificação a utilizar pelo BBEG.

Foi como tal necessário definir uma interface que disciplinasse o diálogo entre ambas as fases. Após várias experiências concluiu-se que esta deveria ser composta por duas partes distintas, uma independente da especificação e que fornecesse os mecanismos de acesso por defeito, e uma segunda parte, intrinsecamente dependente da especificação e que permitisse tratar os casos pontuais, como por exemplo, o acesso a determinados operadores/operandos, as funções de custo e de contexto, etc.

Para não comprometer as soluções utilizadas no *front-end* optou-se por implementar a interface com base em macros, as quais devem ser fornecidas pelo utilizador, ou então no caso deste pretender utilizar as rotinas geradas com os restantes módulos do BEDS, pode, sob pedido explícito (na especificação), serem geradas pelo próprio BBEG.

As macros encontram-se divididas em dois grupos, um com as macros que permitem o acesso aos campos das árvores de expressões e que são utilizadas por praticamente todos os processos do *back-end*, mas fundamentalmente pelas funções de *labelling*; e um segundo grupo, que comporta as macros utilizadas pelas rotinas de alocação e geração de código. Integrada no primeiro grupo está ainda uma enumeração que permite identificar os operadores das árvores de expressões.

As macros a utilizar por omissão nas operações de *labelling*, são:

STATE_TYPE	- Representa o tipo de estrutura de dados dos estados, que por defeito consiste em *struct state* *.
NODEPTR_TYPE	- Designa o tipo do nodo das árvores de expressões, que no caso do BEDS consiste *RTLExpression* *.
OP_LABEL(p)	- Permite aceder ao operador do nodo *p*, normalmente do tipo inteiro.
LEFT_CHILD(p)	- Permite aceder (ler) ao descendente esquerdo do nodo *p*.
RIGHT_CHILD(p)	- Permite aceder (ler) ao descendente direito do nodo *p*.
STATE_LABEL(p)	- Permite aceder (ler/escrever) ao estado do nodo *p*.
PANIC	- Função a utilizar em caso de erro.

E as macros a utilizar, por omissão no sistema de alocação local e na geração de código, são:

SETPHYVAL(p,r,g,t)	- Assinala que o nodo *p*, para a regra *r* utiliza o registo *g*, que é do tipo *t*. Em que:
g	- É um apontador do tipo *void* *, para uma estrutura com a informação sobre o registo utilizado no resultado da regra *r*.
t	- Indica o tipo de registo utilizado em *g*, se *SimpleRegister* ou *CompRegister*.

GETPHYVAL(p,r,n)	- Permite aceder ao registo atribuído ao nodo *p* em relação à regra *r*, de tal forma que se *n* é:
0	- Devolve o registo utilizado pela regra *r*.
1	- Devolve o registo utilizado pela regra anterior a *r*.
2	- Devolve o registo da última regra aplicada a *p*.
GETPHYTYPE(p,r,n)	- Permite aceder ao tipo de registo atribuído ao nodo *p* em relação à regra *r* e ao valor de *n*.
RESULTSTATE(p)	- Permite aceder ao estado do resultado do nodo *p*.

Para uma melhor compreensão das macros SETPHYVAL, GETPHYVAL e GETPHYTYPE, há que explicar que o processo de selecção pode atribuir a cada nodo mais do que uma regra, em que uma está associada à operação (do nodo) e as restantes são regras em cadeia. Como estas podem utilizar recursos distintos para manter os resultados, surge a necessidade de criar mecanismos de suporte e gestão das regras dentro dos nodos, como é o caso das macros atrás definidas. A sua utilização, como já se disse, destina-se às funções de alocação e desalocação local, e geração do código final (*assembly*).

As *Expressions* da MIR já comportam meios para gerir os recursos alocados a cada uma das regras de um nodo, recursos esses que podem ser de três tipos distintos: o *SimpleRegister*, o *CompRegister* e o *STermInf*. Este último destina-se a representar recursos que não sejam registos. Trata-se de uma estrutura composta por uma união, que permite representar um qualquer tipo de dado primitivo da linguagem C, e por uma *string* que descreve o formato segundo o qual deve o valor ser imprimido, o qual respeita as convenções de formatação de texto da instrução *printf(...)* que acompanha a linguagem C.

Outro aspecto importante que o utilizador deve ter em conta, é garantir que os componentes do *back-end* identificam correctamente os operadores das árvores de expressões. Para facilitar a interligação entre ambas as partes, o BBEG utiliza os lexemas que identificam os operadores na especificação das regras. Cabe ao utilizador definir o valor desse símbolos (lexemas), por exemplo, através de um conjunto de *#define*, ou através de uma enumeração, mas sempre como sendo valores do tipo inteiro.

Caso se pretenda utilizar os mecanismos gerados pelo BBEG com os restantes componentes do BEDS, é possível obter todas estas infra-estruturas da interface, bastando para tal que o utilizador dê indicações de tais pretensões, como se poderá ver mais adiante.

8.2.2 Estruturas para representação dos registos

Como já foi dito por inúmeras vezes, o BEDS integra a componente de alocação de registos, permitindo meios para os descrever, disponibilizando depois essa informação de forma a poder ser utilizada pelas rotinas de alocação, quer sejam locais ou globais.

Como o conjunto de registos varia de processador para processador e por vezes de forma bastante distinta, torna-se difícil criar uma forma única e simples de os descrever.

As diferenças mais importantes relacionam-se com o número de registos, a separação em subconjuntos (mediante a utilização a que se destinam), a utilização de um registo por vários subconjuntos e a existência de registos compostos por mais do que um registo simples.

O BBEG soluciona todos estes casos com base em duas componentes distintas. Uma em que fornece as estruturas com a informação individual de cada um dos registos, quer estes sejam simples ou compostos, e outra que associa a cada símbolo não-terminal o conjunto de registos que este pode utilizar.

A primeira componente propõe-se a representar todos os registos simples e compostos, sem no entanto traduzir qualquer informação respeitante à sua utilização. Recorre para tal às seguintes estruturas:

```
/*  Registos Simples  */
typedef struct simplereg {
      char *reg;
      NODEPTR_TYPE p;
} SimpleRegister;

/*  Registos Compostos  */
typedef struct compreg {
      char *reg;
      int *comp;
      NODEPTR_TYPE p;
} CompRegister;
```

Em que *reg* é a *string* que representa o registo, *p* é apenas utilizado aquando da alocação, assinalando o nodo da árvore de expressões (mais recente) ao qual o registo está alocado, e *comp* (só para o CompRegister) é um array de inteiros que identifica os registos simples que compõem o registo. O Exemplo 8.1 ilustra a utilização destas estruturas.

Exemplo 8.1

Para um processador com quatro registos simples, R_1, R_2, R_3 e R_4, e dois registos compostos D_1 e D_2, formados respectivamente por (R_1,R_2) e (R_3,R_4), obtêm-se as seguintes estruturas de dados:

```
SimpleRegister bbeg_TERM[ ] = {
            /*  1  */         "R1", NULL,
            /*  2  */         "R2", NULL,
            /*  3  */         "R3", NULL,
            /*  4  */         "R4", NULL,
            /*     */         NULL, NULL,
};

int compreg1[ ] = { 2, 3, -1};
int compreg2[ ] = { 0, 1, -1};
```

```
CompRegister bbeg_COMPREG[ ] = {
        /* 1 */        "D1", compreg1, NULL,
        /* 2 */        "D2", compreg2, NULL,
        /*   */        NULL, NULL, NULL,
};
```

Para limitar o tamanho dos *arrays*, acrescentou-se um último elemento, normalmente do tipo nulo ou inteiro com valor –1.

Para relacionar os registos com símbolos não-terminais utiliza-se a seguinte estrutura:

```
typedef struct regset {
    int *reg, *creg;
} RegSet;
```

onde *reg* e *creg* apontam respectivamente para os vectores com os registos simples e compostos que o não-terminal pode utilizar. O Exemplo 8.2 demonstra como empregar esta estrutura.

<u>Exemplo</u> <u>8.2</u>

Na continuação do exemplo anterior e pressupondo que existem três não-terminais (nt_1, nt_2 e nt_3), em que o primeiro tanto pode utilizar R_3 como R_4, o segundo não utiliza qualquer registo e o terceiro pode utilizar apenas os registos compostos. Obtêm-se então as seguintes estruturas de dados:

```
int nt1_reg[ ] = { 2, 3, -1};
int nt1_creg[ ] = { -1};
RegSet bbeg_nt1 = {nt1_reg, nt1_creg};

int nt2_reg[ ] = { -1};
int nt2_creg[ ] = { -1};
RegSet bbeg_nt2 = {nt2_reg, nt2_creg};

int nt3_reg[ ] = { -1};
int nt3_creg[ ] = { 0, 1, -1};
RegSet bbeg_nt3 = {nt3_reg, nt3_creg};

RegSet *bbeg_RegisterNTSet[ ] = {
        /* 0 */        NULL;
        /* nt1 */      &bbeg_nt1,
        /* nt2 */      &bbeg_nt2,
        /* nt3 */      &bbeg_nt3,
};
```

Os elementos *nt?_reg* e *nt?_creg* identificam as posições que os registos ocupam em *SimpleRegister* e em *CompRegister*, terminando ambos em –1. Os não-terminais

são identificados em *bbeg_RegisterNTSet* implicitamente através da posição que ocupam.

Há apenas a acrescentar que o utilizador pode requisitar ao BBEG para que este gere todas estas estruturas de informação e respectivas definições com base nos dados da especificação.

8.2.3 Gramática do BBEGL

Uma vez apresentados os aspectos relacionados com a interface e com a representação dos registos, é chegada a altura de descrever a BBEGL. Esta é formada por sete partes distintas: a primeira é apenas formal e identifica o nome da descrição (*descr_name*); a segunda, é onde se activam as *flags* que assinalam quais as componentes a gerar (*descr_flags*); a terceira e sétima, permitem introduzir blocos de código em C que são colocados, respectivamente, antes e após as rotinas a gerar pelo BBEG (*start_code*, *end_code*); a quarta, quinta e sexta, formam o corpo principal da descrição (*body*), mas encontram-se, no entanto divididas para permitirem especificar, com mais clareza, respectivamente, os operadores, os símbolos não-terminais e as regras.

As produções aqui utilizadas para descrever a gramática do BBEGL, obedecem à seguinte estrutura:

axioma ':' lista de símbolos terminais e não-terminais '.'

Corpo da especificação, cabeçalho e *flags*

Assim, qualquer especificação deve estar em conformidade com a seguinte produção:

Description : descr_name descr_flags start_code body end_code.

tal que:

descr_name : 'MACHINE' 'DESCRIPTION' term ';' .

em que *term* representa uma *string* válida segundo as convenções utilizadas para representar os identificadores na linguagem C. As *flags* são descritas com base nas seguintes produções:

descr_flags : 'FLAGS' ':' lst_of_flags .

lst_of_flags : lst_of_flags flag / flag .

flag : 'SET' IDENT ';' .

O BBEG por enquanto apenas admite quatro *flags*, que são:

RTLSINTERFACE - Força a incluir as bibliotecas e as macros necessárias para aceder às componentes da MIR, designadamente às expressões.

RTLSGENERATOR - Gera os mecanismos necessários para emitir código (*assembly*), ou seja, os padrões das instruções, a formatação segundo o qual cada operando deve ser imprimido, eventuais funções de conversão e a função que, dado o nodo actual, a regra a aplicar e os nodos descendentes, gera a respectiva instrução.

RTLSALLOCATOR - Indica ao BBEG que deve gerar as funções para realizar alocação/desalocação local.

REGTABLES - Gera as estruturas de suporte para a descrição e gestão dos registos, e a respectiva informação.

O corpo principal da descrição, contém a declaração dos operadores (*dcl_op*), dos operandos (*descr_opernd*) da representação intermédia e a descrição das regras (*block*), de tal forma que:

```
body        :  declaration rules_block .

declaration :  dcl_op descr_opernd / descr_opernd dcl_op / descr_opernd .
```

Declaração dos operadores

Os operadores são definidos segundo as seguintes produções:

```
dcl_op      :  'OPERATORS' ':' lst_of_ops  .

lst_of_ops  :  lst_of_ops  op  / op .

op          :  term ['=' function] ';' .

function    :  term '(' ')' [ ',' '"' PTYPE '"'] .
```

Como se pode verificar, é possível associar a cada operador uma macro (*function*), que permite aceder ao valor correspondente, ou no qual se traduz o operador. Estas macros devem no entanto devolver um valor do tipo *UValue*, que não é mais que uma união que comporta qualquer um dos tipos primitivos de dados da linguagem C. Pode-se ainda associar a cada macro, o formato segundo o qual deve o valor ser imprimido e que permite também indicar qual o campo da união a utilizar. No caso do formato não ser fornecido, o valor é tratado como sendo do tipo decimal.

O utilizador pode optar por não indicar qualquer tipo de macro, nesta situação, quer o valor a visualizar, quer o formato segundo o qual deve ser imprimido, são determinados por síntese e segundo a regra que se aplica, utilizando as macros descritas na secção 8.2.1. É no entanto conveniente que se defina o formato dos operadores terminais, ou seja, daqueles que não possuam operandos, uma vez que os restantes são determinados por síntese com base nestes.

Declaração dos não-terminais

Os não-terminais são símbolos que surgem na descrição das regras e que servem para representar os valores intermédios das operações, que podem ocorrer sob a forma de registos, endereços, constantes, etc.

A primeira parte desta componente da especificação é a definição do não-terminal *TERM*, que não é mais do que o conjunto de todos os registos físicos simples do processador. Faz-se aqui realçar que neste não-terminal não devem ser representados registos compostos.

De seguida devem-se declarar todos os restantes não-terminais que ocorrem na descrição das regras, que na forma mais simples são só e simplesmente declarados, sendo o seu valor determinado através das macros apresentadas na secção 8.2.1. Estas na realidade não fazem mais do que sintetizar o resultado do operador do qual resultou o não-terminal.

É no entanto possível associar a cada não-terminal um conjunto de registos ou uma macro. A primeira situação permite declarar os registos compostos e viabiliza a construção de subconjuntos de *TERM*, o que é extremamente útil para agrupar os registos segundo o tipo de aplicação a que se destinam. A segunda situação possibilita a substituição das macros utilizadas por defeito e definidas na secção 8.2.1, permitindo assim um tratamento personalizado do respectivo não-terminal.

Está actualmente em desenvolvimento uma solução que permite definir os não-terminais que representam registos através de outros não-terminais. Mas por falta de disponibilidade e uma vez que esta solução implica algumas modificações nos processos de alocação, ainda não foi possível a sua implementação.

Esta componente da especificação termina opcionalmente com a declaração do axioma, ou seja, do não-terminal no qual todas as árvores de expressões devem derivar.

As produções que descrevem esta parte da gramática são:

```
descr_opernd : 'OPERANDS' ':' term_decl [lst_of_sets axiom_decl] .

term_decl    : 'TERM' '='  lst_of_reg ';' .

lst_of_reg   : lst_of_reg ',' register / register .

register     : term .

term         : IDENT .

axiom_decl   : 'AXIOM' '=' term ';' .

lst_of_sets  : lst_of_sets def_set / def_set .

def_set      : term ';' / term '=' lst_of_elem ';' / term '=' function ';' .

lst_of_elem  : lst_of_elem ',' element / element .

element      : term / term '<' lst_of_elem '>' .
```

Exemplo 8.3

Na continuação dos dois exemplos anteriores e pressupondo que existem apenas três não-terminais que utilizam registos, um (*reg*) que de forma genérica usa qualquer registo, outro (*ireg*) que apenas utiliza R1 e um terceiro (*dreg*) onde se utilizam os registos compostos D1 e D2, obtém-se então a seguinte especificação:

OPERANDS:

```
TERM = R1, R2, R3, R4
reg = R1, R2, R3, R4
ireg = R1
dreg = D1<R1,R2>, D2<R3,R4>
...
```

Especificação das regras

É chegada finalmente a altura de apresentar a parte da gramática que define a linguagem de especificação das regras, de qual (especificação) se vai derivar o selector de instruções. A declaração de cada uma é composta por cinco campos distintos: *RULE*, *CONDITIONS*, *COMPUTE*, *COST* e *EMIT*. O primeiro permite descrever o padrão a reconhecer nas árvores de expressões e também o fluxo dos dados; o *CONDITIONS* permite inserir as condições de contexto, em C, que viabilizam, ou não, a utilização da respectiva regra; o *COMPUTE* permite processar algum código em C após a geração do código final; o *COST* atribui um custo à regra, quer através de uma constante ou de uma função que devolva um valor inteiro; e o *EMIT*, não é mais do que a *string* com o código final a emitir. Esta *string* pode conter campos que façam referência aos operadores ou operandos de *RULE*.

As produções desta componente da especificação, são:

```
rules_block     :   'RULES' 'PATTERN' ':' lst_of_rules .

lst_of_rules    :   lst_of_rules rule_pat / rule_pat .

rule_pat        :   rule computations .

        rule            :   'RULE' ':' rule_result '::=' rule_pattern ';' .

        rule_pattern    :   term '(' rule_pattern ',' rule_pattern ')' /
                            term '(' rule_pattern ')' /
                            term .

        rule_result     :   term '<' regid '>' .

        regid           :   '$'CONST .

computations    :   pre_cond pos_pat cost_pat emit .
```

```
pre_cond    :    'CONDITIONS' ':' COD_C  ';' .

pos_pat     :    'COMPUTE' ':' COD_C ';'.

cost_pat    :    'COST' ':' custo ';' .

custo       :    CONST / IDENT '(' ')' .

emit        :    'EMIT' ':'  COD_EMIT ';' .
```

Como se pode verificar pelas produções, os padrões das regras apenas devem ser descritos com base em operadores binários, unários ou sem operandos, e que é necessário associar ao não-terminal que deriva da regra o campo de onde advém o seu resultado (*regid*). Caso este seja *$0* significa então que o resultado não advém de nenhum dos elementos do RHS do padrão, devendo de princípio ser alocado ou obtido através de uma macro. No caso de se pretender referir a algum dos elementos do RHS (*$n*, com n>0), a contagem deve ser feita começando em 1 e contando todos os elementos, excepto parêntesis, quer estes sejam operadores ou operandos. O Exemplo 8.4 ilustra alguns casos desta situação.

Exemplo 8.4

```
RULES PATTERN:

    RULE:   reg < $0 > ::= con                //Aloca ou calcula o valor de reg
    ...
    EMIT:   %{MOV $0, #$1\n%}

    RULE:   lab < $1 > ::= LABEL              //Utiliza o valor do operando LABEL
    ...
    EMIT:   %{ LABEL $1\n%}

    RULE:   ireg < $2 > ::= ADDP(ireg,con)  //Utiliza o valor do ireg do RHS
    ...
    EMIT:   %{ADDP $2, #$3\n%}
```

O Exemplo 8.4 permite também verificar a utilização do *EMIT* que, como já se disse, comporta uma *string* que admite campos sob a forma de *$n*, em que *n* refere um dos elementos de *RULE*. Esta *string* aceita também os caracteres especiais normalmente utilizados em funções como o *printf(...)*. O Apêndice A ilustra alguns casos da especificação de um hipotético processador.

Geração das rotinas

Uma vez concluída a escrita da especificação e admitindo que fique gravada no ficheiro *file.spec*, esta é então submetida ao BBEG através do seguinte comando:

```
>bbeg file.spec
```

O que dá origem a dois ficheiros, o *backend.c* e o *backend.h*, que contêm todas as rotinas, protótipos e definições necessárias. Estes ficheiros devem depois ser compilados com as rotinas do *front-end* e com as bibliotecas do BEDS.

8.3 *Mecanismos para a geração da RI*

Uma vez obtida a interface entre *front-end* e o *back-end*, as rotinas de alocação, selecção e geração de código, quer através do BBEG, ou implementando manualmente, completam-se as componentes que faltavam às bibliotecas do BEDS para se poder construir completamente o *back-end*. Pode-se agora passar à segunda fase da construção de um compilador.

De notar que o papel das rotinas do *back-end* é passivo, ou seja, não cabe a estas inicializar a construção da RI ou o processo de geração de código, tal é função do *front-end*. Assim, esta segunda fase passa essencialmente por dotar o *front-end* dos mecanismos necessários para conseguir, aquando a compilação (terceira fase), construir a RI e processar as optimizações e a geração de código.

Esta secção propõe-se a demonstrar o que é necessário fazer para criar os módulos que venham a gerar a RI e a próxima secção descreve como despoletar os processos de análise, optimização e geração de código.

NOTA: De forma a não alongar mais esta descrição, ao longo deste capítulo não se apresentam os métodos que o *front-end* deve invocar para realizar as funções pretendidas, mas apenas uma breve descrição. Caso seja necessário identificar alguns dos métodos, aconselha-se a consulta do protótipo da respectiva classe no Apêndice D.

Programa

Um dos primeiros passos que o *front-end* deve fazer para construir a RI do programa que está a compilar, é criar a estrutura de suporte à própria representação, a qual servirá para armazenar toda a informação referente ao programa, biblioteca, ou conjunto de funções. Assim, o *front-end* deve ser dotado de meios para instanciar um objecto da classe *Program* e implicitamente da classe *IdentifierTable* (instanciar e registar). Estes dois objectos vão comportar todos os elementos globais, quer estes sejam funções, variáveis, ou código de nível global, os quais de princípio são tratados de forma semelhante aos elementos correspondentes das funções, procedimentos ou blocos de código que se descrevem mais adiante.

NOTA: De forma a simplificar a representação do texto, no resto desta secção, designar-se-ão funções, procedimentos ou blocos de código, simplesmente por funções.

Funções, Procedimentos ou Blocos de Código

O passo seguinte é dotar o *front-end* dos mecanismos necessários à construção da estrutura de suporte a cada uma das funções. Há para tal que instanciar um objecto da classe *Function*, que também necessita de um *IdentifierTable*.

Existem no entanto mais alguns passos fundamentais, assim na instanciação de *Function* é necessário fornecer a identificação da função (nome); e indicar o tipo de *Function*, ou seja, se se trata de uma *FUNCTION*, *PROCEDUR* ou *COMPSTATE*, em que o primeiro identifica funções no verdadeiro termo da palavra, o segundo procedimentos (que não retornam valores) e o terceiro blocos de código que não recebem nem devolvem qualquer tipo de valor. É ainda necessário dotar o *front-end* de meios que assinalem qual a função hierarquicamente acima, para que a função actual seja aí registada (acrescentando-a à tabela de identificadores). No caso de se tratar da função inicial é ainda conveniente dotar de meios para assinalar tal facto em *Program*.

O *front-end* deve ser ainda capaz de definir o nível de encadeamento da função e se se pretender utilizar os mecanismos de gestão do espaço de endereços, instanciar um objecto da classe *BlockMemory* e registá-lo (na função).

Tabela de Identificadores e Identificadores

Por cada identificador que o *front-end* detecte deve instanciar um objecto do tipo *CellIDTable*, indicando o *scope* do identificador (global, local ou parâmetro), a classe (*AUTO*, *REGISTER*, *STATIC*, *EXTERN*, *FUNCTION*, *TEMP*, *CONSTANT* ou *LABEL*); e o respectivo tipo (*char*, *int*, *float*, *unsigned*, ...). No caso das funções, este último campo serve para indicar o tipo de retorno.

No caso das constantes e das funções, faz ainda falta associar a *CellIDTable* a respectiva expressão (*Constant*) ou *Function* das quais derivam, e no caso das variáveis é ainda necessário acrescentar todas as suas definições (atribuições), indicando os *DataTransfers* onde estas ocorrem e o endereço e tamanho da variável. Para tal, pode-se utilizar o *BlockMemory* da função em causa, que permite realizar a gestão do espaço de endereçamento. Por fim o *front-end* deve registar o *CellIDTable* no *IdentifierTable* através do nome do identificador.

É de lembrar que caso o *front-end* necessite, o *IdentifierTable* pode fornecer mecanismos de atribuição de nomes para as variáveis temporárias e para as constantes.

FlowNodes

Na maior parte das situações o que o *front-end* identifica no código fonte são expressões, que podem ser simples atribuições ou até estruturas condicionais. Neste último caso o *front-end* deve ser dotado de meios para construir através de objectos *FlowNode* a estrutura condicional equivalente.

De forma geral deve ser capaz de registar cada um dos nodos na função e depois, caso se trate do nodo inicial ou final, deve ainda assinalar tal facto, e nunca esquecer que cada *FlowNode* possui uma *Label* que deve ser convenientemente tratada.

Os restantes passos dependem da natureza do *FlowNode,* assim para o caso mais simples, o *JumpNode*, há que tratar da operação de salto (*Jump*); no caso do *ConditionalJump* há que tratar do salto condicional (*CondJump*); e no caso do *ReturnNode* da operação de retorno (*Return*).

DataTransfers

Independentemente de ser ou não uma expressão condicional, o *front-end* deve ser capaz de construir a respectiva representação em MIR. O que passa por instanciar, quer objectos da classe *DataTransfer*, quer da classe *Expressions*. Em relação aos primeiros podem ocorrer alguns problemas de interpretação, é que faz falta perceber em que é que consiste um *DataTransfer* para se saber como os instanciar. Assim e seguindo a ideia

original do RTL, os *DataTransfer* devem assinalar toda e qualquer troca entre dados, decompondo para tal as expressões nas operações mais básicas possíveis. Tudo isto para salientar que é necessário alguns cuidados na implementação, no *front-end*, dos mecanismos que vão permitir instanciar este tipo de objecto.

Todos os *DataTransfers* devem ser registados no nodo ao qual pertencem, ou seja, de serem incluídos no nodo. Requer-se ainda que se defina o tipo concreto de *DataTransfer* (*Assignment*, *AtribAssignment*, etc).

O *Assignment* necessita ainda que se actualize o *source*; o AttribAssignment que se indique o *target* e o *Register* no qual resulta a expressão a atribuir; o *JumpAssignment* que se indique a *Label* para onde se destina o salto; o *CondJumpAssignment* que se indiquem as duas *Labels* destino, mais a expressão de teste; o *ReturnAssignment* necessita da expressão da qual resulta o valor a devolver; o *CallAssignment* necessita de conhecer a função a que se destina a operação de *call* e a lista de argumentos a passar; o *LabelAssignment* dispensa qualquer tratamento interno; e o *PhyAssignment* deve ser inserido e actualizado pelas rotinas próprias para a inserção deste tipo de *DataTransfer*.

Expressions

São quatro os procedimentos fundamentais a realizar pelo *front-end* para um qualquer objecto do tipo *Expression*: primeiro registar a expressão no *DataTransfer* que assinala o resultado da operação; segundo definir os identificadores dos operadores (só no caso de não se utilizar os valores atribuídos por defeito); terceiro definir o tipo do resultado da expressão; e quarto designar os descendentes da expressão caso estes existam. As expressões do tipo *Constant* necessitam que se defina o seu valor e o *Register* e o *Memory* que se indique a variável que representam (*CellIDTable*).

Convém aqui assinalar que o BEDS fornece uma tabela de conversão de tipos, em que dado o tipo dos operandos, devolve o tipo do resultado, segundo as convenções utilizadas na linguagem C.

Dotar o *front-end* de mecanismos que permitam executar todos os procedimentos até aqui descritos como obrigatórios, permite manter actualizadas, aquando da compilação, a maior parte das estruturas de dados, como é o caso do *flowDependents* ou do *flowSupporters*. Há no entanto algumas estruturas que só podem ser preenchidas após a construção da representação intermédia.

8.4 *Ordem de execução das funções do BEDS*

É também função do *front-end* invocar as rotinas de optimização, alocação dos registos, geração do código final, etc, pelo que na construção do compilador é necessário acrescentar ao *front-end* os métodos responsáveis pela execução de cada uma destas rotinas. Mas de forma a conseguir-se obter os resultados desejados, é fundamental conhecer a ordem pela qual devem ser invocadas e como é que tal deve ser feito. São estes e outros aspectos que se pretende abordar ao longo desta secção.

Single Static Assignment

O primeiro passo a realizar é colocar toda a representação em conformidade com a forma SSA. Este processo deve ser feito função a função e através da classe *InsertAllPhyAssign*. Esta é responsável por determinar as *dominance frontiers* recorrendo às classes *DFSet*, *IDomSet* e *IDom*, depois ela própria determina quais as

variáveis que necessitam de funções $\Phi(\ )$ e que argumentos utilizar, através da classe *PlacePhyFunctions*.

Infelizmente, esta componente não foi convenientemente testada para ver se está completamente operacional. Para além do mais verifica-se que as soluções utilizadas consomem demasiado tempo, pelo que faz falta desenvolver mais algum trabalho de forma a melhorar esta componente do BEDS.

Após a representação intermédia estar coerente, pode-se prosseguir com as restantes rotinas. O próximo passo aconselhável, é realizar as operações de nível intermédio, ou seja, de *middle-end*, onde se enquadram algumas optimizações, as rotinas de análise de fluxo de dados e de alocação global.

Data Flow Analyser

O único processo de análise de fluxo de dados a funcionar por completo é a análise do período de vida útil das variáveis, que deve ser executada para cada uma das funções. Para tal, basta utilizar a classe *IterativeDFAnalyse*, que por sua vez recorre à classe *SetUdDuChain*.

Esta forma de análise deve ser executada após as optimizações que dela não dependem e logicamente antes das que dela necessitam, como é o caso das rotinas de alocação global.

Structural Analyser

As rotinas de análise estrutural aplicam-se ao nível de cada função e limitam-se a construir a árvore de controlo de fluxo, devolvendo-a sob a forma de um *IntervalNode*.

Esta rotina é fundamental para qualquer um dos processos que necessitam dos mecanismos mais relacionados com o *back-end*, tais como o *labelling*, a alocação local, a atribuição de registos, etc.

Tree Selector

Este módulo, como já se disse, permite obter a representação linear das expressões do grafo de controlo de fluxo. Para tal, recorre à análise anterior para obter a árvore de controlo de fluxo e depois com base nesta obtém a lista de árvores de expressões. Esta pode ser acedida árvore a árvore ou requisitando a lista completa.

De notar que este processo não destrói a representação intermédia apenas permite uma forma diferente de visualizar a organização das árvores de expressões.

Instruction Labelling

A operação de *labelling* realiza-se ao nível de cada árvore de expressões. Caso se pretenda realizar para uma função, há primeiro que linearizar o respectivo grafo, recorrendo ao *Tree Selector*.

Global Allocator

O processo de alocação global é substancialmente mais complexo, uma vez que implica que se realize de antemão, a análise do período de vida útil e depois o *labelling* de todo o grafo, o que por si implica obter a árvore de controlo de fluxo e a respectiva linearização. Em seguida as rotinas constroem o grafo de interferências e tentam colori-lo. Se tudo correr bem basta realizar a atribuição dos registos, caso contrário há que fazer o *splitting* de uma das variáveis, inserindo as operações necessárias e depois modificar os reticulados obtidos através da análise do período de vida útil, e voltar a

construir o grafo de interferências, até que se consiga obter uma solução estável. No entanto, ao utilizador cabe apenas executar a análise do período de vida útil e depois utilizar a classe *Collor_Allocator*.

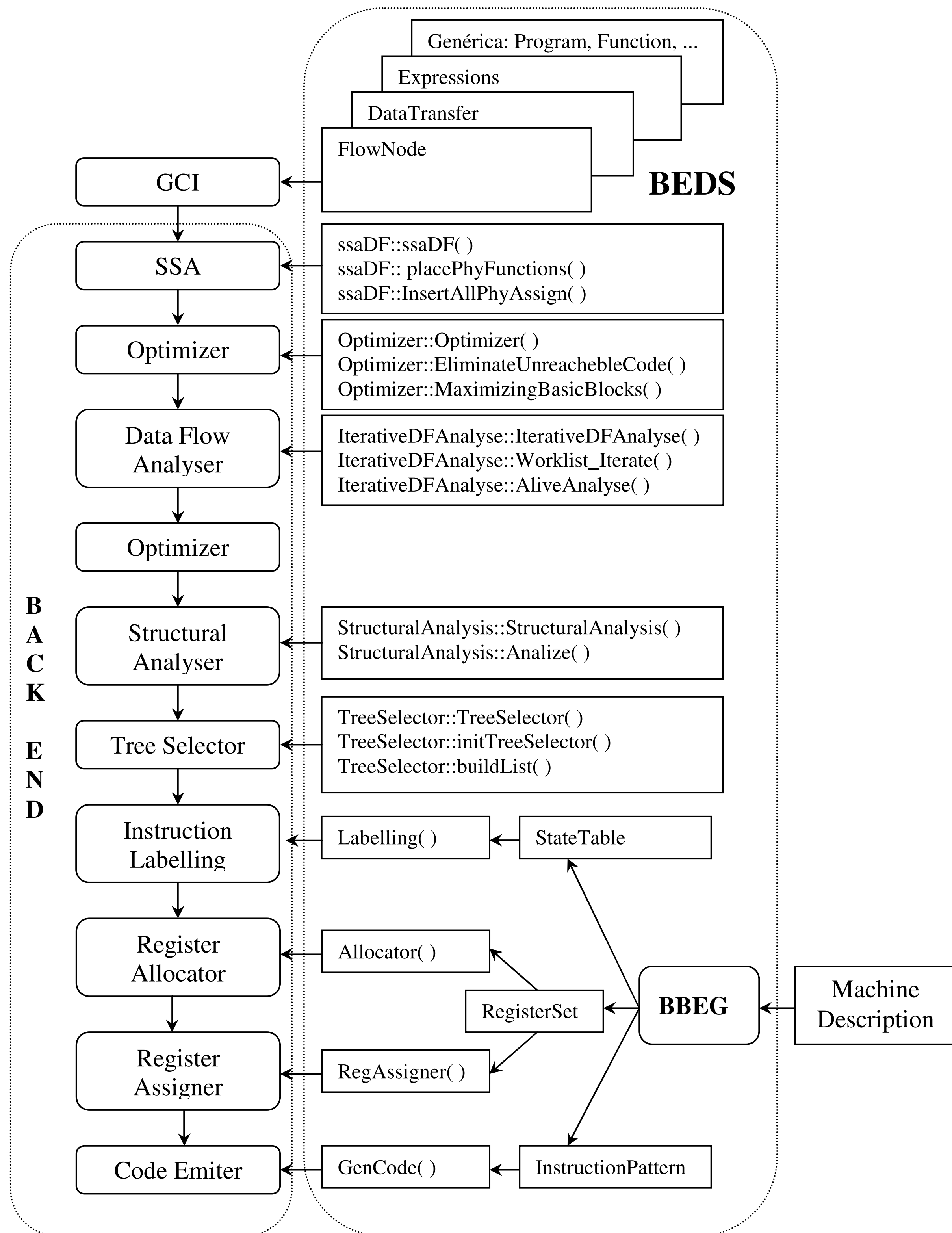


Fig. 8.2 – Sequência das operações para um *back-end* com alocação local.

No fim do processo de alocação pode acontecer que a representação tenha sofrido algumas modificações, mas de princípio tudo deve estar coerente.

Local Allocator

As rotinas de alocação local só se encontram presentes caso se tenha activado, aquando a especificação, a *flag RTLSALLOCATOR*. Estas funcionam em simultâneo com as operações de *labelling*, sendo como tal executadas sempre que esta operação é utilizada. Caso se tenha activado esta *flag*, mas não se pretenda utilizar as rotinas de alocação local, basta colocar a variável *LRAllocator* a *falso*, o que deve ser feito directamente no ficheiro *backend.c*.

Code Emiter

Após a conclusão com êxito da selecção e da alocação, o *front-end* deve, para terminar a cmpilação, invocar o *Code Emiter* de forma a gerar o código em assembly (ou binário).

A Fig. 8.2 representa a sequência das operações a realizar no *back-end* de um compilador com alocação local e os principais métodos que o *front-end* deve invocar para cada operação. De notar que as optimizações propostas são apenas algumas daquelas que se podem utilizar no BEDS.

Infelizmente, uma descrição detalhada da utilização das componentes do BEDS poderia tornar-se demasiado longa para estar contida numa simples secção; espera-se, no entanto, que com o pequeno resumo aqui presente e muitos conhecimentos de C++ se consiga ultrapassar a maior parte das dificuldades. Logo que seja possível, pretende-se apresentar um pequeno manual com exemplos de como se pode explorar as potencialidades do BEDS na implementação de compiladores.

9 Geração de Código Binário

Como se pode constatar pelo capítulo 3, o modelo apresentado para o BEDS ainda não comporta a geração do código binário, ou pelo menos, não disponibiliza os mecanismos necessários ao desenvolvimento desta fase do processo de compilação. De forma a compensar esta lacuna, decidiu-se apresentar uma ferramenta concebida unicamente com o objectivo de auxiliar a implementação dos meios para geração de código binário, o New Jersey Machine-Code Toolkit (NJMCT), fornecendo assim uma solução completa para a concepção de *back-ends*,

Este capítulo começa por uma breve descrição desta ferramenta, após a qual se descrevem os mecanismos que disponibiliza para o desenvolvimento de geradores de código binário. Conclui explicando como é que se interligam as componentes obtidas com as restantes fases do processo de compilação.

9.1 *Descrição do New Jersey Machine Code Toolkit*

O NJMCT foi desenvolvido por N. Ramsey da Universidade de Purdue, e por M. Fernandes da Universidade de Princeton, num projecto que começou por dar os primeiros resultados em 1994, e que se encontra actualmente na versão 0.5 [RF94, RF95, RF95a, RF96]. Serviu também de apoio à realização das respectivas teses de doutoramento [Ramsey92, Fern96], demonstrando com exemplos reais, as potencialidades desta ferramenta.

A sua utilização destina-se a auxiliar o desenvolvimento de programas (tais como compiladores, assembladores, depuradores (*debuggers*), interpretadores, etc) que de alguma forma processem código máquina, fornecendo mecanismos para a manipulação dos bits, mas permitindo ao programador abstrair-se dos detalhes deste nível de representação.

A utilização deste tipo de ferramenta, à semelhança dos mecanismos de geração de selectores apresentados no capítulo 7, também pressupõe a existência de uma descrição com as características do processador, mas agora sobre a composição das instruções e dos registos ao nível binário e da relação destes com os elementos da RI.

O NJMCT, com base na descrição do processador, permite obter quatro módulos distintos: o *translator*; uma biblioteca de funções de codificação e gestão do espaço de código; uma biblioteca de funções auxiliares; e ainda um sistema de verificação da especificação (descrição).

O *translator* permite descodificar as instruções de um programa escritas em código binário, produzindo uma representação textual. É, como tal, muito utilizado na construção de depuradores (debuggers).

A biblioteca de funções de codificação e de gestão do espaço de código, é o módulo de maior interesse dentro do contexto desta tese, e destina-se a fornecer meios para codificar as operações da representação intermédia em código binário, ou *assembly*. Além disso fornece mais alguns mecanismos para auxiliar as componentes do próprio processo de geração de código, que de alguma forma são dependentes das características do processador: é o caso da gestão das *labels*, ou dos endereços relativos. A biblioteca, à semelhança do módulo anterior, é construída a partir da especificação e consiste num conjunto de funções e de tipos estruturados em linguagem C.

A biblioteca auxiliar, fornecida com o NJMCT, possui todo um conjunto de funções, independentes das características do processador, que permitem, por exemplo, fazer a gestão dos blocos de código, alocando o espaço para os mesmos, ou gravando o seu conteúdo para ficheiro.

O sistema para verificar a especificação, ao contrário dos anteriores, faz parte do próprio NJMCT, e permite avaliar com base na informação disponibilizada para os dois primeiros módulos se existem inconsistências na especificação.

A biblioteca de funções de codificação tem especial relevo, dentro do modelo concebido para o BEDS, uma vez que basta modificar o selector de instruções de forma a que este passe a invocar as funções de codificação, para se poder gerar directamente código binário, evitando assim qualquer forma de assemblagem, reduzindo como tal o tempo de compilação. É como tal sobre este módulo que incide o resto deste capítulo.

Convém aqui destacar que o nível de abstracção que o NJMCT fornece em relação ao código binário, quando acompanhado por uma especificação bem construída, permite obter uma interface independente das características do processador, facultando assim alguma autonomia às camadas superiores do processo de compilação.

9.2 *Linguagem de Especificação do* NJMCT

O NJMCT utiliza como linguagem de especificação o SLED - *Specification Language for Encoding and Decoding*, que tem por base quatro tipos distintos de componentes: os *tokens,* os *fields*, os *patterns* e os *constructors*. Os *tokens* servem para representar as instruções, segundo o seu tamanho e composição; os *fields* definem os campos que compõem os *token*; os *patterns* permitem atribuir valores aos *fields*, compor padrões, descrever instruções mais complexas, etc; e os *construtors* especificam os protótipos das funções de codificação e as respectivas operações para gerar o código binário.

Ao longo desta secção descreve-se a estrutura da SLED, com base numa gramática EBNF que recorre aos seguintes meta-símbolos:

{ } - Identificador de repetição
[ ] - Identificador de opção
| - Identificador de alternativa

os quais representam respectivamente, repetições, opções e alternativas. As regras são apresentadas a *itálico* e os símbolos reservados a ***bold***.

Todo o axioma válido da SLED é formado por uma sequência de *Espec*'s, de tal forma que:

Axioma → { *Espec*}

9.2.1 Especificação dos terminais e dos campos

Como já se disse o objectivo dos *tokens* (terminais) é representar, ou melhor classificar, as instruções segundo a sua composição. Define-se assim um *token* para um conjunto de instruções, se estas partilham a mesma estrutura base, designadamente em relação às segmentações internas. As quais são representadas com base em *fields* (campos) associados aos *tokens*. Desta forma, passa a existir uma série de identificadores para representar os conjuntos de instruções e as agregações de bits.

A gramática para a especificação dos *tokens* e dos *fields*, é a seguinte:

Espec → ***fields of*** *token-name* **(***width***)** *fields-specs*
fields-specs → { *field-name low-bit***:***high-bit* }

Exemplo 9.1:

<8051.spec> ≡

fields of CodeOp (8)
op 0:7 op1 4:7 op2 3:3 op4 0:3 op5 0:4
op6 1:3 op7 5:7 op8 0:2 op9 0:0

<fieldinfo>

fields of operd16 (16) arg16 0:15 arg16d 0:15 arg11 0:10

<pattern>
<constructor>
<*assembly*_opcode>
<*assembly*_operand>
<*assembly*_syntax>

No exemplo são definidos os seguintes *tokens*: *CodeOp* e *operd16*, com respectivamente 8 e 16 bits de tamanho. Ambos possuem diversos *fields* (*op*, *op1*, ..., *arg16*, ..., *arg11*), onde por exemplo, *op8* representa os 3 (0-2) primeiros bits de *CodeOp*.

É ainda possível associar a cada um dos *fields* alguns atributos pré-definidos que activam, aquando da invocação das funções, a execução de determinados procedimentos. A sintaxe a utilizar obedece à seguinte gramática:

Espec → **fieldinfo** { *fieldinfo_specs* }

fieldinfo_specs → *field-specifier* **is** **[**{*field-item*}**]**

field-specifier → *field-name* | **[**{*field-name*}**]**

field-item → **checked**
| **unchecked**
| **guaranteed**
| **sparse** **[***binding* {,*binding*}**]**
| **names** **[**{*Indent* | *String*}**]**

binding → *(Ident* | *String)* = *integer*

Os atributos permitem activar os seguintes procedimentos:

checked - Faz com que a função de codificação teste se o valor do *field* se encontra dentro dos limites admissíveis. Estes dependem do número de bits atribuído ao *field*.

unchecked - Faz com que a função de codificação crie uma máscara para proteger os bits do *token* que não pertencem ao *field*.

guaranteed - Não existe qualquer tipo de verificação ou protecção por parte da função de codificação.

names - Permite que a função de codificação aceite nomes alternativos (sinónimos) para representar os *fields*.

sparse - Semelhante ao anterior, mas aplica-se às situações em se desconhecem os nomes dos *fields*.

Exemplo 9.2

<fieldinfo> ≡

fieldinfo op8 **is** [names [R0 R1 R2 R3 R4 R5 R6 R7]]

9.2.2 Especificação dos padrões

A principal função dos *patterns* é permitir atribuir valores aos *fields*, o que pode ser feito com base num vasto conjunto de operações, recorrendo inclusivamente à utilização de outros *patterns*. É como tal uma das componentes da linguagem com maior poder descritivo.

A definição dos *patterns* obedece à seguinte gramática:

```
Espec            →    patterns { pattern-binding }

pattern-binding  →    pattern-name is pattern
                      | [{pattern-name}] is pattern
                      | pattern-name is any of [{pattern-name}], which is pattern
```

Os *patterns* são definidos com base em elementos simples, tais como *fields*, *tokens*, ou através dos respectivos valores binários. Para tal utilizam-se as seguintes expressões gramaticais:

```
pattern          →    name                    #Nome do pattern
                      | opcode (arguments)    #Função definida por um construtor
                      | field-binding         #Atribuição de uma expressão a um field
                      | constraint            #Restrição dos valores dos fields
                      | epsilon               #Sequência vazia
                      | some token_name       #Equivalente a uma única classe de token
                      | [{ name}]             #Lista de patterns

field-binding    →    field_name = expr

constraint       →    field_name relational-operator ( integer | generating_expression)

generating_expression →    { lo to hi}
                           | { lo to hi columns n}
                           | [{integer}]

relational-operator → < | <= | = | > | >=
```

É ainda possível definir os *patterns* com base noutros *patterns*, em que para tal utilizam-se as seguintes expressões:

```
pattern          →    pattern ...
                      | ...pattern
                      | pattern & pattern
                      | pattern ; pattern
                      | pattern | pattern
```

Nas duas primeiras opções, o novo *pattern* é definido parcialmente com base no *pattern* do RHS, em que no primeiro caso substitui alguns dos bits iniciais e no segundo os últimos bits. As outras três opções seguintes correspondem à conjunção, sequência e disjunção de *patterns*.

Exemplo 9.3

<pattern> ≡

patterns

	ajc	**is any of** [ajmp acall]**, which is** op5 = [17 1]
	djnz	**is** op1 = 13
	mov	**is some** CodeOp

A primeira declaração do exemplo define *ajc* como sendo um de dois padrões possíveis, o *ajmp* com *op5* igual a 17, ou o *acall* com *op5* igual a 1. A segunda declaração define *djnz* como sendo um *token* do tipo *CodeOp* com *op1* a 13. A última define *mov* como sendo uma qualquer sequência pertencente ao *token* *CodeOp*.

9.2.3 Especificação dos *constructors*

Os *constructors* permitem relacionar a representação binária, com base nos *fields*, *tokens* e *patterns*, com o protótipo a atribuir às funções de codificação. A declaração destes elementos obedece à seguinte gramática:

Espec → ***constructors*** { *constructor*}
constructor → *opcode* { *operand*} *[: type-name] * [branches]*

opcode → *opname* { ^*opname* }
opname → *string* | *pattern-name* | *field-name*

operand → *field-name [!]*
| *constructor-type-name*
| *relocatable-name*
| *unbound-name*
| *literal*

literal → *string* | *integer* | *qualquer um dos caractéres:* <>=[]()+-/&@#%;*$,|

branches → *[*{*equations*}*] [* ***is*** *pattern]*
| { ***when*** {*equations*} ***is*** *pattern* } *[* ***otherwise is*** *pattern]*

equations → *equation* {, *equation*}

equation → *expr relational-operator expr*

expr → *integer* | *identifier [bit-slice] [!]* | *expr binary-operator expr* | *(expr)*

binary-operator → + | - | * | /

bit-slice → @**[** *lo-bit [: hi-bit]* **]**

Exemplo 9.4

<constructor> ≡

constructors

ajc arg11 {arg8=arg11@[0:7]} **is** ajc & op7 = arg11@[8:10] ; arg8
djnz^REr op8 arg8 **is** djnz & op2 = 1 & op8 ; arg8

Como se viu no exemplo anterior, *ajc* tanto pode representar *ajmp* ou *acall*, pelo que a primeira declaração dá origem a duas funções de codificação distintas, a *ajmp(...)* e a *acall(...)*, cada uma com um único parâmetro que é atribuído ao *field arg11*. Ambas codificam a instrução binária através da conjunção de *ajc* com o campo *op7*, concatenando o resultado com *op8*. O valor de *ajc* é obtido a partir de *ajmp* ou de *acall*, enquanto que o valor de *op7* e *op8* é determinado respectivamente com base nos 3 bits mais significativos e nos 11 bits menos significativos de *arg11*.

O segundo construtor define uma função, cujo nome provém da concatenação das *strings* "djnz" com "REr" e que resulta em *djnzREr(x,y)*. Onde *x* e *y* correspondem respectivamente a *op8* e a *arg8*. A instrução a emitir pela função consiste na concatenação de *arg8* com a conjunção dos seguintes padrões *djnz*, *op2* e *op8*. O valor de *op2* é definido na própria declaração do construtor e é igual a 1.

♦

9.2.4 Outras componentes da especificação

Existem alguns aspectos que, não estando directamente relacionados com as instruções do processador, encontram-se por este condicionado. É o caso do incremento do *program counter*, que por omissão é de 8 em 8 bits, mas que em determinadas arquitecturas pode assumir outros valores; ou a ordem da representação binária, em que o bit mais significativo tanto pode ser o que se encontra mais à direita, ou mais à esquerda (little-endian ou big-endian).

Não se pretende aqui apresentar os detalhes da especificação para todas estas situações. Há no entanto uma que devido à sua importância é conveniente descrever, trata-se da gestão dos endereços e das *labels*, para os quais nem sempre é possível conhecer de antemão os seus valores; é, por exemplo, o caso das instruções de salto para posições posteriores do código.

Para ultrapassar estas situações o NJMCT codifica e gera a instrução, preenchendo o espaço do endereço com um valor pré-definido pelo utilizador. Depois armazena sua posição em relação ao bloco de código em que se encontra inserida, para que logo que o valor do endereço seja determinado se actualizem todas as instruções que o referenciaram.

Das duas seguintes expressões gramaticais, a primeira permite definir o valor a atribuir ao endereço enquanto não é conhecido, e a segunda, especificar os *tokens* que identificam os endereços.

Espec → **placeholder for** *token_name* **is** *pattern*

Espec → **relocatable** {*identifier*}

Na secção 9.3 apresentam-se mais alguns detalhes sobre o tratamento destas situações.

9.2.5 Especificação do *assembly*

Para se obter o *translator*, ou emitir código *assembly*, é ainda necessário especificar a sintaxe da representação textual para a qual se pretende converter o código binário e da relação desta com os operandos e operadores das instruções (binárias).

A representação a obter deve ser de um nível semelhante ao *assembly*, uma vez que o *translator* apenas considera as propriedades sintácticas da linguagem máquina, não permitindo como tal construir formas de representação muito adornadas.

NOTA: A partir deste ponto e apenas para facilitar a descrição, considera-se que o *translator* converte as instruções máquina para *assembly*.

A gramática para esta parte da descrição, é a seguinte:

Espec → *assembly-opcode-syntax*
assembly-operand-syntax
assembly-syntax

O *assembly-opcode-syntax*, *assembly-operand-syntax* e *assembly-syntax*, definem respectivamente, os símbolos dos operadores e dos operandos e a sintaxe da linguagem *assembly*, com base nos *fields*, nos *tokens* e nos *patterns*.

A gramática para o *assembly-opcode-syntax* é a seguinte:

assembly-opcode-syntax → **assembly opcode** *glob-pattern* **is** *glob-target*
| **assembly component** *glob-pattern* **is** *glob-target*

glob-pattern → { * | *string* | *glob-alternatives* }

glob-alternatives → { *glob-pattern* {, *glob-pattern*}

glob-taget → { *string* | *$integer* | *$$* }

O *glob-target* é a representação equivalente em *assembly* do padrão identificado através de *glob-pattern*, em que ambos os elementos são processados como *strings*. Caso *glob-pattern* represente parcialmente um ou mais padrões então a declaração é feita com base na segunda expressão (***assembly component*** ...), caso contrário, ou seja se *glob-pattern* identifica completamente um padrão, então utiliza-se a primeira expressão (***assembly opcode*** ...).

Exemplo 9.5

<*assembly*_opcode>≡

assembly component

{o,a}w	is w
*b	is b
{*}A	is $1

A primeira declaração relaciona parte de um padrão formado pelas *strings* "*ow*" ou "*aw*", com o respectivo símbolo em *assembly* ("w"); a segunda relaciona os padrões que terminam em '*b*' com a *string* "*b*"; e a última declaração os padrões que terminam em '*A*' com a própria *string* que representa o padrão.

A relação entre os operandos e os respectivos símbolos em *assembly*, é feita com base nas seguintes expressões:

assembly-operand-syntax → ***assembly operand*** *Ident* ***is*** *operand-syntax*

operand-syntax → *format-string* ***using*** *operand-name*
| *operand-name*

operand-name → ***sparse*** *[binding {, binding}]*
| ***names*** *[{Ident | String }]*
| ***field*** *Ident*

Exemplo 9.6

<*assembly*_operand>≡

assembly operand

[arg8d arg16d]	is "#%d"	
op8	is "%s"	using field op8
op9	is "@R%d"	

A primeira declaração indica que *arg8d* e *arg16d* são representados em *assembly* pelo carácter '#' seguido de um inteiro correspondente ao valor dos campos. A segunda declaração indica que *op8* é representado em *assembly* por uma *string* proveniente da informação associada a este campo aquando da respectiva declaração do *fieldinfo*. A última declaração, indica que *op9* é representado pela *string* "@R" seguida do próprio valor atribuído a *op9*.

De notar que o NJMCT, utiliza a mesma convenção da linguagem C para a formatação e representação dos operandos.

A representação da composição dos padrões em *assembly* obedece à seguinte regra gramatical:

assembly-syntax -> **assembly syntax** *opcode operands*

Exemplo 9.7

<*assembly*_syntax>≡

assembly syntax

onearg^"A"	"A"	
xch^"AR"	"A",	op8

A primeira declaração descreve o padrão da instrução *assembly* a utilizar para os construtores da família *onearg* que terminam em "A". O padrão é composto pela concatenação do nome do construtor com a *string* "A".

A segunda declaração descreve o padrão para as instruções provenientes dos construtores da família *xch* que terminam em "AR". O qual é composto pela concatenação do nome do construtor com a *string* "A", seguido do valor do campo *op8*.

9.3 *Utilização das funções de codificação*

Uma vez terminada e submetida a especificação ao gerador, indicando o tipo de *output* pretendido, código máquina ou *assembly*, obtém-se a biblioteca com as funções de codificação.

As aplicações que pretendam recorrer a esta biblioteca não se podem limitar a invocar as funções de codificação, necessitam também de executar alguns procedimentos antes e após a geração das instruções utilizando as funções e as estruturas da biblioteca que acompanha o NJMCT, a *mclib.h*.

Um dos primeiros procedimentos consiste em obter um recipiente para as instruções a emitir. Para tal, o NJMCT possui o que designa por *relocatable block*, ou seja, uma estrutura de dados que permite armazenar e gerir uma sequência de instruções, que podem corresponder, por exemplo, a um bloco de código simples, ou de uma função. Um *relocatable block* (*RBlock*), aqui representado por *rb*, possui os seguintes campos de informação:

rb.lc	- Indica a posição do *program counter* dentro de rb;
rb.size	- O tamanho do *rb*;
rb.low	- O menor valor de *rb.lc* para o qual o conteúdo é conhecido;
rb.address	- O endereço absoluto de *rb*;
rb.contents	- Buffer de tamanho *rb.size* que começa em 0 e termina em *rb.size-1* e serve para armazenar as instruções;
rb.label	- *label* da posição inicial de *rb.contents*.

A biblioteca que acompanha o NJMCT fornece um conjunto de funções e macros para inicializar e aceder a cada um destes campos. A única excepção é o *rb.contents*

cujo conteúdo não pode ser acedido directamente mas apenas copiado para um ficheiro ou *buffer*. O acesso para escrita faz-se através das funções que emitem as instruções.

De notar, que podem existir vários *relocatable blocks* em utilização, mas só um é que se encontra activo.

O campo *rb.label* é uma estrutura (*RLabel*) que permite armazenar as características mais relevantes de uma *label*, a qual é formada pelos seguintes campos:

block	- Identifica o *relocatable block* a que pertence a *label*;
offset	- Indica o offset da posição da *label* em relação ao início de *block*;
name	- Define a *string* associada à *label*.

Um *relocatable block*, para além de *rb.label*, pode conter tantas *labels* quantas as necessárias.

O *mclib.h* disponibiliza também diversas funções que permitem emitir as instruções para *rb.contents*. De notar que estas são utilizadas pelas próprias funções de codificação. A emissão de uma instrução incrementa automaticamente o *rb.lc* do bloco activo e se necessário aumenta o *rb.contents*.

Como já se viu, se a instrução a emitir utiliza um endereço que ainda não foi determinado, então na posição por ele a ocupar é colocado um *placeholder*, que tem por finalidade reservar o espaço necessário ao endereço. A nível dos mecanismos internos do NJMCT é declarada uma estrutura para reter a informação necessária à manutenção do *placeholder* (*RClosure*), a qual é composta pelos seguintes campos:

dest_block	- Identifica o *relocatable block* onde se encontra inserido o *placeholder*;
dest_lc	- Indica a posição do *placeholder* dentro do *relocatable block*;
depends_on	- É o conjunto de endereços relocatáveis dos quais depende o *placeholder*;
apply	- Função que permite reescrever o *placeholder*.

O *mclib.h* contém várias outras funções e estruturas, para além das atrás referidas. Optou-se, no entanto, por apresentar apenas aquelas que são mais relevantes. Convém ainda referir que o utilizador é responsável por definir e implementar algumas estruturas e funções, como é o caso dos procedimentos de alocação do espaço para o *rb.contents*.

9.4 *Resumo*

Apesar do NJMCT não pertencer ao sistema apresentado ao longo desta tese e da descrição ter sido feita essencialmente do ponto de vista do utilizador, achou-se mesmo assim, que era importante apresentar uma solução que permitisse dar continuidade ao trabalho possível de desenvolver através do BEDS, fornecendo uma solução completa para o desenvolvimento de *back-ends*.

Para além disso, serviu para descrever o tipo de informação necessária à construção do gerador de código máquina e ver como essa informação pode ser representada. Permitiu também expor alguns dos problemas inerentes a esta fase do processo e em parte ajudou a consolidar as ideias de como deve ser um sistema de geração de geradores de código binário.

Infelizmente, a experiência obtida através do estudo desta ferramenta ficou-se pela utilização não tendo sido possível, por falta de tempo, um estudo sobre a sua implementação e eventual adaptação, ou construção de raiz, de algo semelhante para integrar no BEDS.

Convém ainda referir que neste capítulo focou-se essencialmente um dos módulos desta ferramenta e apenas com a profundidade necessária para se perceber os conceitos envolvidos e entender o modo de funcionamento.

No Apêndice C encontra-se, a título de exemplo, a especificação em SLED de um microcontrolador, a qual foi desenvolvida para ajudar a compreender esta ferramenta, não muito fácil de usar.

10 Conclusão

Durante o longo período em que decorreu o trabalho desenvolvido para esta tese, houve uma grande reformulação das noções sobre a concepção e estrutura de um compilador.

Começou-se com uma abordagem extremamente simplista, resumindo um compilador a um conjunto de processos, formado pela análise léxica, sintáctica e semântica, aos quais se seguia uma pequena fase de geração de código, que se pensava ter a ver com pouco mais do que a selecção de instruções e atribuição de registos.

Apesar do conhecimento inicial ser algo limitado, o mesmo não se podia dizer dos objectivos, os quais se propunham a obter soluções completas que permitissem a automatização das fases do compilador inerentes à geração de código, à semelhança do que existia para a análise léxica, sintáctica e semântica.

Assim, começou-se por estudar as soluções existentes, analisando o seu modo de funcionamento, avaliando as potencialidades e medindo a eficiência. Foram alvo deste processo ferramentas como o BURG, IBURG, BEG, NJMCT, algumas componentes do GCC, etc.

Após esta fase constatou-se que o modelo de compilador até aí utilizado, não permitia distrinçar de forma clara alguns processos fundamentais, tais como alocação (distribuição) dos registos e optimização de código.

Foi como tal necessário rever os conceitos iniciais, estudar outras soluções (RTLS, GCC, COSY, etc) e formalizar, não só um modelo para o compilador, como também algo que era intrínseco a todas as fases, que é a forma da representação do código. Surgiu assim a *My Intermediate Representation* e mais genericamente o *Back-End Development System*, à volta dos quais está todo o trabalho desenvolvido para esta tese de mestrado.

10.1 *Estado actual do BEDS*

Como foi possível constatar ao longo dos últimos sete capítulos, o BEDS propõe-se a fornecer, através de bibliotecas e de módulos gerados, tudo o que é necessário para

suportar o desenvolvimento dos processos de um *back-end*. Aliás, pode mesmo dizer-se que o BEDS vai muito para além disso, ao fornecer também uma componente bastante elaborada, para o que se designou por *middle-end*, ou seja, as fases do processo de compilação que não dependem da linguagem fonte, nem das características do processador, como é o caso da maior parte das optimizações e dos processos de análise inerentes a estas.

Para além da concepção de um modelo para um compilador portável (capítulo 3), o trabalho desenvolvido abrangeu o estudo, desenvolvimento e implementação de formas de representação (capítulo 4); dos processos de análise do fluxo de dados e de controlo, fundamentais a um grande número de rotinas, nas quais se incluem muitas das optimizações (capítulo 5); dos mecanismos de alocação de registos, donde convém realçar que o BEDS comporta alocação global, através do algoritmo de coloração de grafos de Chaitin (capítulo 6); e dos mecanismos de selecção de instruções e de como os conseguir gerar com base na especificação das características do processador (capítulo 7). E para concluir, concebeu-se e desenvolveu-se uma linguagem para descrever as características do processador e parametrizar algumas das rotinas provenientes dos restantes pontos (capítulo 8). Para além disso, apresentou-se uma ferramenta cuja utilização conjunta com o BEDS, permite a construção completa do *back-end* de um compilador (capítulo 9).

É de realçar que todas as soluções empregues nos diversos módulos que compõem o BEDS, foram escolhidas e concebidas, só após o estudo do que já existia dentro de cada área e sempre que possível testando ferramentas que utilizassem o mesmo tipo de tecnologia.

Julgo no entanto que o grande mérito desta tese, advém essencialmente da integração de todos os mecanismos necessários ao completo desenvolvimento de um *back-end*, algo que é raro e onde os poucos casos conhecidos são de domínio comercial. Para além do mais, o BEDS mesmo inacabado, não só cumpre o necessário à geração de código, como fornece soluções concretas e executáveis, quer sob o ponto de vista do desenvolvimento de compiladores, quer dos sistemas de apoio à construção destes.

De forma mais detalhada, convém salientar que do trabalho elaborado ao longo desta tese, resultaram as seguintes contribuições: introdução do conceito de *middle-end*, como fase do processo de compilação independente da linguagem fonte e do processador; especificação de um modelo de representação e suporte ao tratamento de código; implementação de um sistema que com base na descrição do conjunto de registos de um processador, das respectivas instruções e da relação destas com as expressões da representação intermédia, permite gerar/moldar todas as fases do *back-end* (excepto geração de código binário); implementação de um sistema de alocação global, com base num algoritmo conhecido e com provas dadas, mas adaptado de forma a realizar a alocação para conjuntos não uniformes de registos; realçar ainda a utilização conjunta dos mecanismos de *labelling* da selecção de instruções, como importante auxiliar dos processos de alocação, quer estes sejam locais ou globais; e para concluir, a implementação de um sistema de selecção de instruções, que permite avaliar os custos destas e o contexto de execução em *run-time*, sem que para tal tenha perdas significativas de performance e que se encontra em pleno funcionamento.

10.2 *Trabalho Futuro*

Como já atrás foi dito, o BEDS está longe de estar concluído, quanto mais não seja porque ainda não houve qualquer fase de teste e avaliação dos resultados, excepção feita a alguns casos pontuais utilizados para detectar e corrigir erros de implementação. Para além disso, existem situações que ainda não foram contempladas e que apesar de não serem fundamentais aos processos já existentes, é de todo conveniente a sua concretização, quanto mais não seja para que o BEDS deixe o estado embrionário em que se encontra e passe a ocupar uma posição no universo das soluções que existem nesta área.

Das situações fundamentais a iniciar, ou a concluir, faz parte o desenvolvimento de um conjunto de *front-ends* para as linguagens mais utilizadas, não só como forma de incentivar a utilização do BEDS, como também para avaliar até que ponto a MIR garante os mecanismos necessários à representação de código intermédio, isto para além daqueles que já se sabem faltar, como é o caso da representação de *arrays*, estruturas e o tratamento de tipos. Convém ainda verificar o comportamento da MIR em relação aos apontadores.

É ainda necessário verificar os processos de análise já implementados e eventualmente construir outros processos, tais como a análise de dependência de dados e de *aliases*. Depois, e como forma de rentabilizar essas formas de análise, faz falta desenvolver todo um conjunto de optimizações que são consideradas como fundamentais à grande maioria dos compiladores modernos.

A nível dos processos de alocação é imprescindível testar as soluções fornecidas, utilizando outras arquitecturas, de forma a verificar a eficiência da representação dos registos e das próprias rotinas de alocação.

Na selecção de instruções são potencialmente poucas as melhorias que se possam realizar, uma vez que as soluções utilizadas foram largamente testadas, quer no BEDS, quer em vários outros selectores de instruções. O mesmo já não se pode dizer em relação à linguagem de especificação, a BBEGL. Esta não só exige que se avalie a sua capacidade descritiva, como eventualmente se a modifique de forma a fornecer ao utilizador maior intervenção e simultaneamente que permita simplificar o processo de descrição.

É ainda fundamental expandir a BBEGL (e simultaneamente o BEDS) para comportar a descrição do código binário e da relação deste com as expressões da representação intermédia, e posteriormente gerar o próprio gerador de código binário.

Para além dos aspectos anteriores, existem outros que só e simplesmente não foram explorados, tais como: optimização de acessos à memória cache; rentabilização de arquitecturas *multi-pipelining* e super-escalares; utilização de técnicas para processamento paralelo; obtenção da informação necessária à construção do *back-end* a partir da descrição VHDL do processador; etc.

Julgo que auge do BEDS será eventualmente alcançado, quando este permitir gerar as rotinas necessárias a todas as fases atrás descritas, incluindo a optimização, com base numa simples especificação do que estas devem realizar e em que fase do processo de compilação o devem fazer.

Esperando que o trabalho realizado possa vir a ser útil a alguém e que consiga cativar alguns adeptos, deixo desde já expressa a minha disponibilidade para prestar qualquer esclarecimento e porque não contribuição, dentro das áreas abrangidas pelo trabalho desenvolvido nesta tese de mestrado.

Bibliografia

[AC75] Aho, A.V. and Corasick, M.J. 1975. Efficient *string* matching: An aid to bibliographic search. Communication ACM 18, June, pp. 333-340.

[AG86] Aho, A.V., and Ganapathi, M. (1986). Efficient tree pattern matching: An aid code generation. In Proceedings of the (13)th ACM Symposium on Principles of Programming Languages. ACM, New York, pp. 334-340.

[AGT89] Aho, A.V., Ganapathi, M. e Tjiang, S.W. 1989. Code generation using tree matching and dynamic programming. ACM Transactions on Programming Languages and Systems, 4(Oct), pp. 491-516.

[AJ76] Aho, A.V. and Johnson, S.C. 1976. Optimal code generation for expression trees. Journal ACM 23:3, pp. 488-501.

[APCEB96] Auslander, J., Philipose, M., Chambers, C., Eggers, S.J. and Bershad, B.N. 1996. Fast, effective dynamic compilation. Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, pp. 149-159.

[AS87] Appel, A.W. and Supowit, K.J. 1987. Generalizations of the Sethi-Ullman algorithm for register allocation. Software-Practice and Experience, Vol. 17, 6 (June), pp. 417-421.

[ASU86] Aho, A.V., Sethi, R., and Ullman, J.D. 1986. Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading, MA.

[BCKT89] Briggs, P., Cooper, K.D., Kennedy, K. and Torczon, L. 1989. Colouring heuristics for register allocation. ACM, Department of Computer Science, Rice University, Houston.

[BCT92] Briggs, P., Cooper, K.D. and Torczon, L. 1992, Colouring register pairs. ACM Letters on Programming Languages and Systems, Vol. 1, 1 (March), pp. 3-13.

[BDB90] Balachandran, A., Dhamdhere, D.M. e Biswas, S. 1990. Efficient retargetable code generation using bottom-up tree pattern matching. Computer Languages, 15, 3, pp. 127-140.

[BG89] Bernstein, D. and Gertner, I. 1989. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. ACM Transactions on Programming Languages and Systems, Vol. 11, 1(Jan), pp. 57-66.

[Bernstein, et al] Bernstein, D., Goldin, D., Golumbic, M., Krawczyk, H., Mansour, Y., Nahshon, I. and Pinter, R. 1989. Spill code minimization techniques for optimizing compilers. ACM, IBM Israel Science and Technology, Haifa, Israel.

[BGS94] Bacon, D.F., Graham, S.L. and Sharp, O.J. 1994. Compiler transformations for high-performance computing. ACM Computer Surveys, Vol. 26, 4 (Dec), pp. 345-420.

[BM??] Brandis, M.M. and Mossenbock, H. Single Pass Generation of Static Single Assignment Form for Structured Languages. Extraído da Internet.

[BMW91] Borstler, J., Moncke, U. and Wilhelm, R. 1991. Table Compression for tree automata. ACM Transactions on Programming Languages and Systems, Vol. 13, 3(July), pp. 295-314.

[BR91] Bernstein, D. and Rodeh, M. 1991. Global instruction scheduling for superscalar machines. Proceedings of the ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, pp. 241-255.

[BSM87] Biljon, W.R., Sewry, D.A. and Mulders, M.A. 1987. Register allocation in a pattern matching code generator. Software-Practice and Experience, Vol. 17, 8 (Aug), pp. 521-531.

[Buck94] Buck, J.T. 1994. Static scheduling and code generation from dynamic data flow graphs with integer-valued control streams. In Proceedings of 28th Asilomar Conference on Signals, Systems, and Computers.

[CCMH91] Chang, P.P., Chen, W.Y., Mahlke, S.A. and Hwu, W.W. 1991. Comparing static and dynamic code scheduling for multiple-instruction-issue processors. ACM.

[CACCHM81] Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., and Markstein, P.W. 1981. Register allocation via colouring. Computer Languages 6, 1(Jan.), pp.47-57.

[CFRWZ91] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems, 13(April), pp. 451-490.

[CH90] Chow, F.C. and Hennessy, J.L. 1990. The priority-based colouring approach to register allocation. ACM Transactions on Programming Languages and Systems, Vol. 12, 4(Oct), pp. 501-536.

[Chait82] Chaitin, G.J. 1982. Register allocation and spilling via graph colouring. In Proceedings of the SIGPLAN Symposium on Compiler Construction. Boston, Mass. ACM SIGPLAN Notices 17, 6(June), pp. 98-105.

[Chase87] Chase, D.R. 1987. An improvement to bottom-up tree pattern matching. In Proceedings of the 14th Annual Symposium on Principles of Programming Languages. ACM, New York, pp. 168-177.

[CK91] Callahan, D. and Koblenz, B. 1991. Register allocation via hierarchical graph colouring. Proceedings of the ACM SIGPLAN ´91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, pp. 192-203.

[DDH84] Dencker, P., Durre, K. and Heuft, J. 1984. Optimization of parser tables for portable compilers. ACM Transactions on Programming Languages and Systems, Vol. 6, 4(Oct), pp. 546-572.

[DF80] Davidson, J.W. and Fraser, C.W. 1980. The Design and Application of a Retargetable Peephole Optimizer. ACM Transactions on Programming Languages and Systems, Vol. 2, 2(April), pp. 191-202.

[DF84a] Davidson, J.W. and Fraser, C.W. 1984. Code selection through object code optimization. ACM Transactions on Programming Languages and Systems, Vol. 6, 4(Oct), pp. 505-526.

[DF84b] Davidson, J.W. and Fraser, C.W. 1984. Register allocation and exhaustive peephole optimization. Software-Practice and Experience, Vol. 14, 9(Sept), pp. 857-865.

[Dham88] Dhamdhere, D.M. 1988. Register assignment using code placement techniques. Computer Languages, Vol. 13, 2, pp. 75-93.

[ED96] Eichenberger, A.E. and Davidson, E. 1996. A reduced multipipeline machine description that preserves scheduling constraints. Proceedings of the ACM SIGPLAN ´96 Conference on Programming Language Design and Implementation, PA, USA, pp. 12-22.

[Emmel89] Emmelmann, H. 1989. BEG - a *Back end* Generator, user manual. Universitat Karlsruhe.

[EP??] Engler, D.R. and Proebsting, T.A. DCG: An efficient, retargetable dynamic code generation system. Extraído da Internet.

[EHK96] Engler, D.R., Hsieh, W.C. and Kaashoek, M.F. 1996. C: A language for high-level, efficient, and machine-independent dynamic code generation. In Proceedings of the 23th Annual Symposium on Principles of Programming Languages. ACM, St. Petersburg, USA, pp. 131-144.

[EV94] Emmelmann, H. and Vollmer, J. 1994. GMD Modula System MOCKA – User manual. Universitat Karlsruhe.

[Fern96] Fernández, M.F. 1996. A retargetable optimizing linker. Ph. D. Department of Computer Science, Princeton University.

[FH90] Fraser, C. And Hanson, D. 1990. A code generation interface for ANSI C. Research Report CS-TR-270-90.

[FH91] Fraser, C. And Hanson, D. 1991. A retargetable compiler for ANSI C. Research Report CS-TR-303-91.

[FH92] Fraser, C. and Hanson, D. 1992. Simple register spilling in a retargetable compiler. Software-Pratice and Experience, Vol. 22, 1(Jan), pp. 85-99.

[FH95] Fraser, C. and Hanson, D. 1995. A retargetable C compiler: design and implementation. Addison-Wesley Publishing Company.

[FHP91] Fraser, C.W., Henry, R.R. and Proebsting, T.A. 1991. BURG – Fast optimal instruction selection and tree parsing. ACM SIGPLAN Notices, Vol. 27, 4 (April), pp. 68-76.

[FHP92] Fraser, C.W., Hanson, D.R. and Proebsting, T.A. 1992. Engineering efficient code generators using tree matching and dynamic programming. Research Report CS-TR-386-92.

[FOW87] Ferrante, J., Ottenstein, K.J. and Warren, J.D. 1987. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, Vol. 9, 3 (July), pp. 319-349.

[Fras77] Fraser, C.W. 1977. Automatic generation of code generators. Ph. Dissertation, Yale University, New Haven, Vonn..

[Fras89] Fraser, C.W. 1989. A language for writing code generators. ACM, AT&T Bell Laboratories.

[FSW94] Ferdinand, C., Seidl, H. and Wilhelm, R. 1994. Tree automata for code selection. Acta Informática, 31, pp. 741-760.

[FW88] Fraser, C.W. and Wendt, A.L. 1988. Automatic generation of fast optimizing code generators. Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, pp. 79-84.

[GE90] Grosch, J. and Emmelmann, H. 1990. A tool box for compiler construction. Project Compiler Generation, Report nº20, University Karlsruhe.

[GE82] Ganapathi, M. e Fischer, C.N. 1982. Description driver code generation using attribute grammars. In Proceedings of the 5th Annual Symposium on Principles of Programming Languages. ACM, New York, pp. 108-119.

[GE84] Ganapathi, M. e Fischer, C.N. 1984. Attributed linear intermediate representations for retargetable code generators. Software-Pratice and Experience, vol. 14, 4(April), pp. 347-364.

[GE85] Ganapathi, M. e Fischer, C.N. 1985. Affix grammar driven code generation. ACM Transactions on Programming Languages and Systems, vol. 7, 4 (Oct), 560-599.

[GF88] Ganapathi, M. and Fischer, C.N. 1988. Integrating code generation and peephole optimization. Acta Informática, 25, pp. 85-109.

[GG78] Glanville, R.S. and Graham, S.L. 1978. A new method for compiler code generation. In Proceedings of the 5th Annual Symposium on Principles of Programming Languages. ACM, New York, pp. 231-240.

[GHAMP84] Glanville, S.L., Henry, R.R., Aigrain, P., Mckusick, M., and Pelegri-Llopart, E. 1984. Experience with a Graham-Glanville style code generator. Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, ACM SIGPLAN Notices, vol.19, nº 6 (June), pp. 13-24.

[GHKSW90] Gray, R., Heuring,V., Kram,S., Sloam, A. and Waite,W. 1990. Eli: A complete, flexible compiler construction system. Research report, University of Colorado, Boulder.

[GHS82] Graham, S.L., Henry, R.R. and Schulman, R.A. 1982. An experiment in table driven code generation. ACM SIGPLAN Noticves 17, 6 (June), pp. 32-43.

[Gilm95] Gilmore, C.M. 1995. Microprocessors, principles and applications. McGraw-Hill International Editions, Electrical and Electronic Technology Series.

[Glan77] Glanville, R.S. 1977. A machine independent algorithm for code generation and its use in retargetable compilers. Ph. D. Dissertation, University of California, Berkeley.

[GM94] Golden, M. and Mudge, T. 1994. A comparison of two pipeline organizations. Proceedings on MICRO-27, San Jose CA, USA, pp. 153-161.

[Graham80] Graham, S. 1980. Table-driven code generation. University of California, Berkeley.

[GR92] Granlund, T. and Kenner, R. 1992. Eliminating branches using a superoptimizer and the GNU C compiler. Proceedings of the ACM

SIGPLAN '92 Conference on Programming Language Design and Implementation, pp. 341-352.

[GSO94] Gupta, R., Soffa, M.L. and Ombres, D. 1994. Efficient register allocation via colouring using clique separators. ACM Transactions on Programming Languages and Systems, Vol. 16, 3 (May), pp. 370-386.

[Hanson83] Hanson, D.R. 1983. Simple code optimizations. *Software*-Practice and Experience, Vol. 13, pp. 745-763.

[HC86] Hatcher, P.J. e Christopher, T.W. 1986. High quality code generation via bottom-up tree pattern matching. In Proceedings of the 13th Annual Symposium of Principles of Programming Languages. ACM, pp. 119-129.

[HD89] Henry, R.R. e Damron, P.C. 1989. Performance of table driven code generators using tree pattern matching. Tech. Rep. 89-02-02, Univ. of Washington, Seatle, Wash.

[Henry84] Henry, R.R. Graham-Glanville code generators. Ph. D. Dissertation, Computer Science Division, Electrical Engineering and Computer Science, University of California, Berkeley, 1984.

[Henry89] Henry, R.R. 1989. Encoding optimal pattern selection in a table driven bottom-up tree pattern matcher, Tech Rep. 89-02-04, Univ. of Washington, Seatle, Wash.

[Henry91] Henry, R.R. 1991. Hard-coding Bottom-up Code Generation Tables to Save Time and Space, Software-Practice and Experience, vol. 21, 1(Jan), pp. 1-12.

[HKC84] Hatcher, P.J., Kukuck, R.C. and Christopher, T.W. 1984. Using dynamic programming to generate optimized code in a Graham-Glanville style code generator. In Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, ACM SIGPLAN Notices 19, 6 (June), pp. 25-36.

[HO82] Hoffmann, C.M. e O'Donnell, M.J. 1982. Pattern matching in trees. J. ACM 29, 1(Jan), pp. 68-95.

[HZ96] Hoover, R. and Zadeck, K. 1996. Generation machine specific optimizing compilers. In Proceedings of the 23th Annual Symposium on Principles of Programming Languages. ACM, St. Petersburg, USA, pp. 219-229.

[JML91] Johnson, R.E., McConnell, C. e Lake, J.M. 1991. The RTL System: A framework for code optimization. Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 5(May), pp. 255-274.

[KH93] Kolte, P. and Harrold, M.J. 1993. Load/Store range analysis for global register allocation. Proceedings of the ACM SIGPLAN '93 Conference on

Programming Language Design and Implementation, Albuquerque, pp. 268-277.

[Kild73] Kildall, G. 1973. A Unified Approach to Global Program Optimization. POPL73, pp. 194-206.

[KMP77] Knuth, D., Morris, J. and Pratt, V. 1977. Fast pattern matching in *strings*. SIAM J. Compt. 6, Feb., pp. 323-350.

[KPF95] Kurlander, S.M., Proebsting, T.A. and Fischer, C.N. 1995. Efficient instruction scheduling for delayed-load architectures. ACM Transactions on Programming Languages and Systems, Vol. 17, 5 (Sep), pp. 740-776.

[KRS94] Knoop, J., Ruthing, O. and Steffen, B. 1994. Optimal code motion: theory and pratice. ACM Transactions on Programming Languages and Systems, Vol. 16, 4 (July), pp. 1117-1155.

[LE95] Lo, J.L. and Eggers, S.J. 1995. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. ACM SIGPLAN, La Jolla, USA.

[LDKT??] Liao, S., Devadas, S., Keutzer, K. and Tjiang, S. Instruction selection using binate covering for code size optimization. ACM. Extraído da Internet.

[LDKTW??] Liao, S., Devadas, S., Keutzer, K., Tjiang, S. and Wang, A. Code optimization techniques for embedded DSP microprocessors. ACM. Extraído da Internet.

[Lopes97] Lopes, J.A. 1997. An architecture for the compilation of persistent polymorphic reflexive higher-order languages. Ph. Dissertation, Department of Computing Science, Glasgow University.

[MHL91] Maydan, D.E., Hennessy, J.L. and Lam, M.S. 1991. Efficient and exact data dependence analyses. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, pp. 1-14.

[Morris91] Morris, W.G. 1991. CCG: A prototype coagulating code generator. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, pp. 45-58.

[MRS90] McConnell, C., Roberts, J.D. and Schoening, C.B. 1990. The RTL System. Departement of Computer Science, University Illinois at Urbana-Champaign.

[MRS??A] McConnell, C., Roberts, J.D. and Schoening, C.B. 1990. Using SSA Form in a Code Optimizer. Department of Computer Science, University Illinois at Urbana-Champaign. Extraído da Internet.

[Much97] Muchnick, S.S. 1997. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers. ISBN 1-55860-320-4.

[NG93] Ning, Q. and Gao, G.R. 1993. A novel framework of register allocation for software pipelining. In Proceedings of the 20th Annual Symposium on Principles of Programming Languages, ACM, pp. 29-42.

[NN95] Novack, S. and Nicolau, A. 1995. A hierarchical approach to instruction-level paralelization. International Journal of Parallel Programming, Vol. 23.

[NP??a] Norris, C. and Pollock, L.L. A scheduler-sensitive global register allocator. Department of Computer and Information Sciences, University of Delaware, Newark. Extraído da Internet.

[NP??b] Norris, C. and Pollock, L.L. Register allocation over the program dependence graph. Department of Computer and Information Sciences, University of Delaware, Newark. Extraído da Internet.

[NP95] Norris, C. and Pollock, L.L. An experimental study of several cooperative register allocation and instruction scheduling strategies. Proceedings of MICRO-28, pp. 169-179.

[Patter95] Patterson, J.R.C. 1995. Accurate Static Branch Prediction by Value Range Propagation. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, La Jolla, San Diego, pp. 67-78.

[Peleg88] Pelegri-Llopart, E. Rewrite systems pattern matching and code generation. Ph. D. Thesis. Technical Report UCB/CSD 88/423, Computer Science Division, Univ. California, Berkeley, 1988.

[Per96] Pereira, M.J.T. 1996. Concepção e especificação de uma linguagem visual. Ms. Dissertation, Departamento de Informática, Universidade do Minho.

[PF92] Proebsting, T.A. and Fischer, C.N. 1992. Probabilistic register allocation. Proceedings of the ACM SIGPLAN ´92 Conference on Programming Language Design and Implementation, pp. 300-310.

[PF94] Proebsting, T.A. and Fraser, C.W. 1994. Detecting pipeline structural hazards quickly. In Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 280-286.

[PF96] Proebsting, T.A. and Fischer, C.N. 1996. Demand-driven register allocation. ACM Transactions on Programming Languages and Systems, Vol. 18, 6 (Oct), pp. 683-710.

[PG88] Pelegri-Llopart, E. e Graham, S.L. 1988 Optimal code generation for expression trees: An application of BURS theory. In Proceedings of the 15th

Annual Symposium on Principles of Programming Languages. ACM, New York, pp. 294-308.

[Pinter93] Pinter, S.S. 1993. Register allocation with instruction scheduling: a new approach. Proceedings of the ACM SIGPLAN ´93 Conference on Programming Language Design and Implementation, Albuquerque, pp. 248-257.

[Proeb92a] Proebsting, T.A. 1992. Code generation techniques. Ph. Dissertation, Wisconsin University, Madison.

[Proeb92b] Proebsting, T.A. 1992. Simple and Efficient BURS Table Generation. In Proceedings of the 19th Annual Symposium on Principles of Programming Languages. ACM, pp. 331-340.

[Proeb95a] Proebsting, T.A: 1995. BURS Automata Generation. ACM Transactions on Programming Languages and Systems, Vol. 17, 3 (May), pp. 461-486.

[Proeb95b] Proebsting, T.A: 1995. Optimizing an ANSI C interpreter with superoperators. In Proceedings of the 22th Annual Symposium on Principles of Programming Languages. ACM, San Francisco, pp. 322-332.

[PTB??] Proebsting, T.A., Townsend, G. and Bridges, P. Toba: Java for applications, a way ahead of time (WAT) compiler. University of Arizona. Extraído da Internet.

[PW??] Proebsting, T.A. and Whaley, B.R. One-pass, optimal tree parsing – with or without trees. Department of Computer Science, University of Arizona, Tucson. Extraído da Internet.

[PWU97] Peleg, A., Wilkie, S. and Weiser, U. 1997. Intel MMX for multimedia PCs. Communications of the ACM, Vol. 40, 1(Jan), pp. 25-38.

[Ramsey92] Ramsey, N. 1992. A Retargetable Debugger. Ph. D. Thesis, Princeton University, Department of Computer Science. Technical Report CS-TR-403-92.

[Ramsey96] Ramsey, N. 1996. Relocating machine instructions by curring. Proceedings of the ACM SIGPLAN ´96 Conference on Programming Language Design and Implementation, pp. 226-236.

[RF94] Ramsey, N., and Fernández, M. 1994. New Jersey Machine-Code Toolkit reference manual. Technical Report TR-471-94. Department of Computer Science, Princeton University.

[RF95] Ramsey, N., and Fernández, M. 1995. The New Jersey Machine-Code Toolkit. In Proceedings of the 1995 USENIX Technical Conference, pp. 289-302, New Orleans, LA.

[RF95a] Ramsey, N., and Fernández, M. 1995. New Jersey Machine-Code Toolkit. Department of Computer Science, Princeton University, 15/12/95.

[RF96] Ramsey, N., and Fernández, M. 1996. New Jersey Machine-Code Toolkit architecture specifications. Department of Computer Science, Princeton University, 7/05/96.

[RLTS92] Rau, B.R., Lee, M., Tirumalai, P.P., Schlansker, M.S. 1992. Register allocation for software pipelined loops. Proceedings of the ACM SIGPLAN ´92 Conference on Programming Language Design and Implementation, pp. 283-299.

[RS??] Sweany, P. and Beaty, S. Post-compaction register assignment in a retargetable compiler. Colorado State University, Fort Collins, Colorado. Extraído da Internet.

[SKAH91] Smotherman, M., Krishnamurthy, S., Aravind, P.S. and Hunnicutt, D. 1991. Efficient DAG construction and heuristic calculation for instruction scheduling. ACM.

[Stallman94] Stallman, R. 1994. Using and porting GNU CC. Free Software Foundation.

[SW94] Smith, J.E. and Weiss, S. 1994. PowerPc 601 and Alpha 21064: a tale of two RISCs. IEEE Computer Journal, June, pp. 46-58

[Tjiang85] Tjiang, S.W.K. Twig reference manual. Computing Science Technical Report 120, AT&T Bell Laboratories, Murray Hill, N.J., 1985.

[VS95] Venugopal, R. and Srikant, Y.N. 1995. Scheduling expression trees with reusable registers on delayed-load architectures. Computer Languages, Vol. 21, 1, pp. 49-65.

[Wall??] Wall, D.W. Global register allocation at link time. Digital Equipment Corporation, Western Research Lab. Extraído da Internet.

[Wasil72] Wasilew, S.G. A compiler writing system with optimization capabilities for complex order structures. Ph.D. dissertation, Northwestern University, Evanston.

[Weing73] Weingart, S.W. An efficient and systematic method of compiler code generation. Ph.D. dissertation, Computer Science Department, Yale University, New Haven.

[Wendt90] Wendt, A.L. 1990. Fast code generation using automatically-generated decision trees. Proceedings of the ACM SIGPLAN ‘90 Conference on Programming Language Design and Implementation, White Plains, New York, pp. 9-15.

[Wilson, et al 94] Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S. and Hennessy, J.L. 1994. The SUIF compiler system: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Department of Electrical Engineering and Computer Science, Stanford University.

[WKEE94] Wang, J., Krall, A., Ertl, M.A. and Eisenbeis, C. 1994. *Software* pipelining with register allocation and spilling. Proceedings on MICRO-27, San Jose CA, USA, pp. 95-99.

[WS91] Whitfield, D. and Soffa, M.L. 1991. Automatic generation of global optimizers. Proceedings of the ACM SIGPLAN ´91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, pp. 120-129.

[Wulf81] Wulf, W. 1981. Compilers and computer architecture. IEEE Computer Journal, July, pp. 41-47.

[YA95] Yeralan, S. and Ahluwalia, A. 1995. Programming and Interface the 8051 Microcontrolor. Addison-Wesley Publishing Company.

Apêndice A

Gramática do BBEGL

```
/*--------------------------- gnr.con-------------------------------------*/

Description        :    descr_name descr_flags start_code body end end_code.
start_code         :    COD_C .
end_code           :    COD_C .
body               :    declaration rules_block .
declaration        :    dcl_op descr_opernd / descr_opernd dcl_op / descr_opernd .

/*----------------------------- **** -------------------------------------*/

descr_name         :    'MACHINE' 'DESCRIPTION' term ';' .
end                :    'END' ';' .

/*----------------------------- **** -------------------------------------*/

descr_flags        :    'FLAGS' ':' lst_of_flags .
lst_of_flags       :    lst_of_flags flag / flag .
flag               :    'SET' IDENT ';' .

/*----------------------------- **** -------------------------------------*/

dcl_op             :    'OPERATORS' ':' lst_of_ops  .
lst_of_ops         :    lst_of_ops  op  / op .
op                 :    term ['=' function] ';' .
function           :    term '(' ')' [',' '"'PTYPE'"'] .

/*----------------------------- **** -------------------------------------*/

descr_opernd       :    'OPERANDS' ':' term_decl [lst_of_sets axiom_decl] .
term_decl          :    'TERM' '='  lst_of_reg ';' .
lst_of_reg         :    lst_of_reg ',' register / register .
register           :    term .
term               :    IDENT .
axiom_decl         :    'AXIOM' '=' term ';' .
lst_of_sets        :    lst_of_sets def_set / def_set .
```

```
def_set          :    term ';' / term '=' lst_of_elem ';' / term '=' function ';' .

lst_of_elem      :    lst_of_elem ',' element / element .

element          :    term / term '<' lst_of_elem '>' .

/*----------------------------- **** --------------------------------------*/

rules_block      :    'RULES' 'PATTERN' ':' lst_of_rules .

lst_of_rules     :    lst_of_rules rule_pat / rule_pat .

rule_pat         :    rule computations .

    rule         :    'RULE' ':' rule_result '::=' rule_pattern ';' .

    rule_pattern :    term '(' rule_pattern ',' rule_pattern ')' /
                      term '(' rule_pattern ')' /
                      term .

    rule_result  :    term '<' regid '>' .

    regid        :    '$'CONST .

computations     :    pre_cond pos_pat cost_pat emit .

    pre_cond     :    'CONDITIONS' ':' COD_C  ';' .

    pos_pat      :    'COMPUTE' ':' COD_C ';'.

    cost_pat     :    'COST' ':' custo ';' .

    custo        :    CONST / IDENT '(' ')' .

    emit         :    'EMIT' ':'  COD_EMIT ';' .

/*----------------------------- **** --------------------------------------*/
```

Apêndice B

Exemplo de uma especificação em BBEGL

Este exemplo apresenta uma especificação, em BBEGL, de um hipotético microprocessador com quatro registos físicos (R0,R1,R2 e R3)e com a possibilidade de utilizar dois registos compostos (D1 e D2), formados respectivamente por (R0,R1) e (R2,R3). As instruções utilizam 5 modos de endereçamento: o *reg* para os registos simples, o *ireg* para o registo R1, o *dreg* para os registos compostos, o *con* para as constantes e o *addr* para os endereços. Em que este último é o não-terminal objectivo.

```
MACHINE DESCRIPTION x86 ;

FLAGS:
        SET RTLSINTERFACE;
        SET RTLSGENERATOR;
        SET RTLSALLOCATOR;
        SET REGTABLES;

%{
/* versão 05.11.98 */
#include "main.h"
#include "Label.h"

#define GETCNST(p) (((Storage *)(p))->getUValue())
#define GETMEM(p) (((Memory *)(p))->getUValue())
#define GETLABEL(p) getLabel((Label *)(p))

UValue getLabel(Label *p){
        UValue val;
        val.i = p->getVersion();
        return val;
}
%}

OPERATORS:

        LABEL = GETLABEL(),"%d";
        CNST = GETCNST(),"%d";
        MEM = GETMEM(),"%d";
        ASGN;
        ADD;
        ADDP;
        INDIR;
        MUL;
        DIV;
        GT;
        JUMP;
        CJUMP;
        ARG;
```

```
OPERANDS:
    TERM    = R0, R1, R2, R3;
    reg     = R0, R1, R2, R3;
    ireg    = R1;
    dreg    = D0<R2,R3>, D1<R0,R1>;
    con;
    addr;
    nop;
    lab;
    AXIOM           = addr;

RULES PATTERN:

    RULE:           reg<$0> ::= INDIR(addr);
    CONDITIONS:     %{%};
    COMPUTE:        %{ %};
    COST:           2;
    EMIT:           %{MOV $0, $2\n%};

    RULE:           ireg<$0> ::= addr;
    CONDITIONS:     %{%};
    COMPUTE:        %{ %};
    COST:           2;
    EMIT:           %{MOV $0, #$1\n%};

    RULE:           reg<$1> ::= ireg;
    CONDITIONS:     %{ %};
    COMPUTE:        %{ %};
    COST:           0;
    EMIT:           %{%};

    RULE:           reg<$0> ::= con;
    CONDITIONS: %{ %};
    COMPUTE:        %{ %};
    COST:           0;
    EMIT:           %{MOV $0, #$1\n%};

    RULE:           ireg<$2> ::= ADDP(ireg,addr);
    CONDITIONS: %{ %};
    COMPUTE:        %{ %};
    COST:           2;
    EMIT:           %{ADD $2, $3\n%};

    RULE:           reg<$0> ::= ADDP(addr,con);
    CONDITIONS:     %{ %};
    COMPUTE:        %{ %};
    COST:           5;
    EMIT:           %{MOV $0, #$2\nADDP $0, #$3\n%};

    RULE:           reg<$0> ::= ADDP(addr,CNST);
    CONDITIONS:     %{ %};
    COMPUTE:        %{ %};
    COST:           5;
    EMIT:           %{MOV $0, #$2\nADDP $0, #$3\n%};

    RULE:           ireg<$2> ::= ADDP(ireg,con);
    CONDITIONS:     %{ %};
    COMPUTE:        %{ %};
    COST:           2;
    EMIT:           %{ADDP $2, #$3\n%};
```

```
RULE:          reg<$2> ::= MUL(reg,con);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{MUL $2, #$3\n%};

RULE:          reg<$2> ::= MUL(reg,reg);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          1;
EMIT:          %{MUL $2, $3\n%};

RULE:          reg<$2> ::= MUL(reg,addr);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{MUL $2, $3\n%};

RULE:          reg<$2> ::= ADD(reg,con);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{ADD $2, #$3\n%};

RULE:          reg<$2> ::= ADD(reg,reg);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          1;
EMIT:          %{ADD $2, $3\n%};

RULE:          reg<$2> ::= ADD(reg,addr);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{ADD $2, $3\n%};

RULE:          addr<$2> ::= ADD(addr,reg);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          1;
EMIT:          %{ADD $2, $3\n%};

RULE:          addr<$2> ::= ASGN(addr,reg);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{MOV $2, $3\n%};

RULE:          addr<$2> ::= ASGN(addr,con);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{MOV $2, #$3\n%};

RULE:          addr<$2> ::= ASGN(addr,addr);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          1;
EMIT:          %{MOV $2, $3\n%};
```

```
RULE:          reg<$2> ::= DIV(reg,reg);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{DIV $2, $3\n%};

RULE:          reg<$2> ::= DIV(reg,con);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{DIV $2, #$3\n%};

RULE:          reg<$2> ::= DIV(reg,addr);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          1;
EMIT:          %{DIV $2, $3\n%};

RULE:          reg<$2> ::= GT(reg,reg);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{GT $2, $3\n%};

RULE:          addr<$2> ::= GT(addr,reg);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{GT $2, $3\n%};

RULE:          addr<$2> ::= GT(addr,con);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{GT $2, #$3\n%};

RULE:          addr<$2> ::= GT(addr,addr);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          1;
EMIT:          %{GT $2, $3\n%};

RULE:          addr<$2> ::= JUMP(LABEL);
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          2;
EMIT:          %{JUMP $2\n%};

RULE:          addr ::= CJUMP(reg, ARG(LABEL, LABEL));
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          5;
EMIT:          %{CJMP $2, $4, $5\n%};

RULE:          addr<$1> ::= lab;
CONDITIONS:    %{ %};
COMPUTE:       %{ %};
COST:          0;
EMIT:          %{%};
```

```
        RULE:          lab<$1> ::= LABEL;
        CONDITIONS:    %{ %};
        COMPUTE:       %{ %};
        COST:          1;
        EMIT:          %{LABEL $1:\n%};

        RULE:          addr<$1> ::= MEM;
        CONDITIONS:    %{ %};
        COMPUTE:       %{ %};
        COST:          0;
        EMIT:          %{%};

        RULE:          con<$1> ::= CNST;
        CONDITIONS:    %{ %};
        COMPUTE:       %{ %};
        COST:          0;
        EMIT:          %{%};

    END ;

    %{
    %}
```

Apêndice C

Especificação em SLED do µC 8051

O µC 8051 é utilizado para explicar a gramática da linguagem de especificação do NJMCT. Trata-se de um microcontrolador, com base numa arquitectura de 8 bits, com: acumulador; 4 bancos de registos; 4 portas bidireccionais de 8 bits; uma porta série programável; 2 contadores de 16 bits; 2 níveis de prioridade de interrupções; possibilidade de endereçar directamente bytes e bits; um *Program Counter* (PC) de 16 bits. O 8051 permite código até 64 kbytes e possui 111 tipos de instruções, num total de 255 instruções diferentes, com tamanhos entre 1 a 3 bytes, classificadas em 5 grupos:

- Operações aritméticas
- Operações lógicas
- Operações de transferência de dados
- Operações de manipulação de variáveis booleanas
- Operações de controlo da máquina e saltos.

Cada instrução pode ter um, dois ou três operandos, de entre os seguintes tipos:

- Rn	- Representa um dos 8 registos (R0 a R7), do banco de registos seleccionado.
- direct	- Endereço de dados de 8 bits.
- @Ri	- Endereçamento indirecto, através dos registos R0 ou R1.
- #data	- Valor constante de 8 bits.
- #data16	- Valor constante de 16 bits.
- addr16	- Endereço de dados de 16 bits.
- addr11	- Endereço de 11 bits.
- rel	- Endereço relativo, de -128 a + 127 bytes em relação a posição da próxima instrução a executar.
- bit	- Endereço de um bit ou *flag*.

Além dos blocos já identificados, fazem parte da arquitectura do 8051 diversos registos: o *Stack Pointer* (SP) que permite utilizar uma *stack* até 128 bytes de tamanho; o *Data PoinTeR* (DPTR) que é um registo de 16 bits utilizado para saltos longos (*Long Jumps*); um registo especial *B* de 8 bits para operações aritméticas de multiplicação e divisão; bem como outros registos menos relevantes para funções específicas.

O SLED permite descrever microprocessadores com arquitecturas RISC ou CISC, no entanto a tarefa é bastante mais complicada para o caso das arquitecturas CISC, como é o caso do 8051, os obstáculos colocam-se por diversas razões, tais como: a variação do tamanho das instruções; existirem muitas formas de endereçamento, algumas das quais implícitas no próprio código da operação; a organização do código das operações não ser uniforme; etc.

No 8051, as instruções podem ocupar um, dois ou três bytes, para identificar a instrução e os operandos. Mas mesmo entre instruções com o mesmo tamanho, as suas

estruturas diferem, pelo que apenas se garante que o operador pode ser identificado através do primeiro byte, o qual pode ainda incluir referencias aos operandos, e que o segundo e terceiro byte, caso existam, são apenas utilizados pelos operandos, os quais são normalmente do tipo constante ou endereço.

Por exemplo, a instrução *add A, Rn* que permite somar o conteúdo do registo *Rn* ao acumulador e a instrução *mov addr, #data*, que move o valor *data* para o endereço *addr*, possuem ambas na representação em *assembly* dois argumentos. No entanto, na representação em binário, a primeira consiste num único byte, em que os três bits menos significativos indicam o registo a utilizar e os restantes sete bits, a operação e implicitamente o primeiro operando; já a segunda instrução, necessita de utilizar um byte para identificar a operação, mais dois bytes para representar respectivamente o endereço (*addr*) e o valor da constante (*data*).

Assembly	Cód. Binário		
	1º Byte	2º Byte	3ºByte
add A, Rn	0 0 1 0 1 r r r		
mov addr, #data	0 1 1 1 0 1 0 1	addr (8bits)	data (8bits)

Os três bit *r* identificam um dos 8 registos do 8051.

Na especificação do 8051, foi necessário considerar que uma instrução é composta por pelo menos um *token*, que identifica o código de operação e eventualmente alguns operandos. Se a instrução necessitar mais do que um byte, faz-se a concatenação com outros *tokens*, definidos especificamente para representarem os operandos.

Deste modo, criaram-se três *tokens*, o *CodeOp*, que corresponde ao primeiro byte, o *operd8*, que representa qualquer tipo de operando de 8 bits, e o *operd16*, que representa operandos com tamanho compreendido entre 8 e 16 bits.

O passo seguinte, consiste em identificar entre os vários *tokens*, características que estes tenham em comum, de forma a agrupar as respectivas instruções, reduzindo assim o número de construtores a definir. O 8051 possui no entanto uma arquitectura relativamente ultrapassada, que denota um processador desenvolvido sem grandes preocupações com organização da representação binária, tendo em maior consideração o aproveitamento máximo das funções e organização dos componentes do microprocessador, o que dificulta a identificação de características comuns. Por exemplo, a instrução *mov* possui mais de 50 versões em código máquina, em que praticamente cada uma, possui o seu próprio código de operação.

Após várias experiências, especificaram-se os seguintes campos, para os três tipos de *tokens*:

```
fields of CodeOp (8)
    op 0:7 op1 4:7 op2 3:3 op4 0:3 op5 0:4
    op6 1:3 op7 5:7 op8 0:2 op9 0:0
fieldinfo op8 is [names [ R0 R1 R2 R3 R4 R5 R6 R7]]

fields of operd8 (8)
    arg8 0:7 arg81 0:7 arg8d 0:7

fields of operd16 (16)
    arg16 0:15 arg16d 0:15 arg11 0:10
```

Para *CodeOp*, definiram-se os seguintes campos:

op	- Código completo da operação
op1	- Código da operação do nibble mais significativo
op2	- Utilizado em operações que envolvem registos
op4	- Código da operação do nibble menos significativo
op5	- Utilizado em operações de saltos com referências absolutas
op6	- Utilizado em operações de endereçamento indirecto
op7	- Utilizado em operações de saltos com referências absolutas
op8	- Utilizado em operações que envolvem registos
op9	- Utilizado em operações de endereçamento indirecto

Associou-se a *op8* os nomes dos registos (R0...R7), o que é útil caso se pretenda obter o *Translator* ou gerar código *assembly*.

Todos os campos definidos para *operd8* servem para identificar o mesmo conjunto de bits, apenas se definiram vários para facilitar a descrição da especificação. Desta forma, *arg8* e *arg8d*, servem para representar operandos de 8 bits, respectivamente do tipo endereço e constante. Caso seja necessário um segundo operando do tipo endereço utiliza-se *arg81*.

O *operd16* possui 3 campos, em que *arg16* e *arg16d* são campos de 16 bits, com funções semelhantes a *arg8* e a *arg8d*, e o campo *arg11* serve apenas para as instruções *acall* e *ajmp*.

O passo seguinte é descrever os padrões, tentando agrupar as instruções com o objectivo de diminuir o tamanho da descrição. O método utilizado consistiu em criar padrões para os casos mais genéricos e tratar os casos específicos individualmente, o que resultou no seguinte:

```
patterns

ajc         is any of [ajmp acall], which is op5 = [17 1]
geral       is any of [nop ret reti swapA daA cplA clrA incDPTR
                 rrA rrcA  rlA rlcA divAB mulAB], which is
                op = [0 34 50 196 212 244 228 163 3 19 35
                51 132 164]
onearg      is any of [inc dec], which is op1 = [0 1]
onearg1     is any of [jc  jnc jz jnz sjmp push pop], which is
                op1 = [4 5 6 7 8  12 13] & op4 = 0
onearg2     is any of [ljmp lcall], which is op = [2 18]
onebitarg   is any of [cpl clr set], which is op1 = [11 12 13]
onebitarg1  is any of [orlCB orlCnB anlCB anlCnB movCB movBC],
                which is op = [114 160 130 176 162 146]
xchd        is op1 = 13
xch         is op1 = 12
djnz        is op1 = 13
twoarg1     is any of [add addc subb], which is op1 = [2 3 9]
twoarg2     is any of [orl anl xrl], which is op1 = [4 5 6]
twoarg3     is any of [jbc jb jnb ], which is op = [16 32 48]
cjne        is op1 = 11
mov         is some CodeOp
movc        is some CodeOp
movx        is some CodeOp
```

O primeiro padrão serve para representar as instruções de salto absoluto (*acall* e *ajmp*), que se caracterizam por possuir um único argumento de 11 bits.

O padrão *geral* corresponde às instruções formadas por um único byte, os padrões *onearg*, *onearg1* e *onearg2* às instruções que utilizam um argumento, mas que por divergirem no valor do código da instrução necessitam de três padrões distintos. Os padrões *onebitarg* e *onebitarg1*, são semelhante aos anteriores mas referem-se a operações sobre bits. Os padrões *xchd*, *xch*, *djnz*, *cjne*, *mov*, *movc* e *movx*, são casos específicos que não podem ser incluídos nos padrões anteriores. E os padrões *twoarg1*, *twoarg2* e *twoarg3* servem para representar instruções com dois argumentos.

De notar, que por vezes se associa ao nome da instrução um ou mais caracteres, cuja finalidade é apenas distinguir instruções que partilham da mesma designação em *assembly*, mas que em código máquina são completamente distintas e como tal pertencem a padrões distintos. Por exemplo, a instrução *cpl* que pode ser representada por *onebitarg* ou por *geral*, mediante o operando seja do tipo bit, ou o acumulador, pelo que se mantivesse a mesma designação para ambas as situações, levaria a que fossem geradas duas funções de codificação com o mesmo nome, mas com parâmetros e procedimentos distintos, o que provocaria alguns problemas uma vez que as funções são geradas em C e esta não é uma linguagem polimórfica. Desta forma para a primeira situação utiliza-se a designação *cpl*, enquanto que para a segunda acrescenta-se um ‘A’, de acumulador, resultando em *cplA*. Existem no entanto algumas situações em que apesar de não existir qualquer necessidade de distinguir as designações, tal é feito apenas para garantir uma representação mais consistente. A concatenação de *strings* é utilizada com os mesmos objectivos nos construtores.

```
constructors

ajc arg11 {arg8=arg11@[0:7]}        is ajc & op7 = arg11@[8:10] ; arg8

geral                               is geral

onearg^A                            is onearg & op4 = 4
onearg^R op8                        is onearg & op2 = 1 & op8
onearg^Ri op9                       is onearg & op6 = 3 & op9
onearg^E arg8                       is onearg & op4 = 5 ; arg8

onearg1^Er arg8                     is onearg1 ; arg8

onearg2 arg16                       is onearg2 ; arg16

onebitarg^C                         is onebitarg & op4 = 3
onebitarg^B arg8                    is onebitarg & op4 = 2 ; arg8
onebitarg1 arg8                     is onebitarg1 ; arg8

xchd^ARi op9                        is xchd & op6 = 3 & op9

xch^ARi op9                         is xch & op6 = 3 & op9
xch^AE arg8                         is xch & op4 = 5 ; arg8
xch^AR op8                          is xch & op2 = 1 & op8

djnz^REr op8 arg8                   is djnz & op2 = 1 & op8 ; arg8
djnz^EEr arg8 arg81                 is djnz & op4 = 5 ; arg8 ; arg81
```

```
twoarg1^ARi op9              is twoarg1 & op6 = 3 & op9
twoarg1^AE arg8              is twoarg1 & op4 = 5 ; arg8
twoarg1^AR op8               is twoarg1 & op2 = 1 & op8
twoarg1^AD arg8d             is twoarg1 & op4 = 4 ; arg8d
twoarg2^ARi op9              is twoarg2 & op6 = 3 & op9
twoarg2^AE arg8              is twoarg2 & op4 = 5 ; arg8
twoarg2^AR op8               is twoarg2 & op2 = 1 & op8
twoarg2^AD arg8d             is twoarg2 & op4 = 4 ; arg8d
twoarg2^EA arg8              is twoarg2 & op4 = 2 ; arg8
twoarg2^ED arg8 arg8d        is twoarg2 & op4 = 3 ; arg8 ; arg8d

twoarg3 arg8 arg81           is twoarg3 ; arg8 ;arg81

cjne^AEEr arg8 arg81         is cjne & op4 = 5 ; arg8 ; arg81
cjne^ADEr arg8d arg8         is cjne & op4 = 4 ; arg8d ; arg8
cjne^RDEr op8 arg8d arg8     is cjne & op2=1 & op8 ; arg8d ; arg8
cjne^RiDEr op9 arg8d arg8    is cjne & op6=3 & op9 ; arg8d ; arg8

mov^AR op8                   is op1 = 14 & op2 = 1 & op8
mov^AE arg8                  is op = 229 ; arg8
mov^ARi op9                  is op1 = 14 & op6 = 3 & op9
mov^AD arg8d                 is op = 116 ; arg8d
mov^RA op8                   is op1= 15 & op2 = 1 & op8
mov^RE op8 arg8              is op1 = 10 & op2 = 1 & op8 ; arg8
mov^RD op8 arg8d             is op1 = 7 & op2 = 1 & op8 ; arg8d
mov^EA arg8                  is op = 245 ; arg8
mov^ER arg8 op8              is op1 = 8 & op2 = 1 & op8 ; arg8
mov^EE arg8 arg81            is op= 133 ; arg8 ; arg81
mov^ERi arg8 op9             is op1 = 8 & op6 = 3 & op9 ; arg8
mov^ED arg8 arg8d            is op = 117 ; arg8 ; arg8d
mov^RiA op9                  is op1 = 15 & op6 = 3 & op9
mov^RiE op1 arg8             is op1 = 10 & op6 = 3 & op9 ; arg8
mov^RiD op9 arg8d            is op1 = 7 & op6 = 3 & op9 ; arg8d
mov^DPTR arg16d              is op = 144 ; arg16d

movc^DPTR                    is op = 147
movc^PC                      is op = 131

movx^ARi op9                 is op1 = 14 & op6 = 1 & op9
movx^A.DPTRi                 is op = 224
movx^RiA op9                 is op1 = 15 & op6 = 1 & op9
movx^DPTRi.A                 is op = 240
```

Esta parte da especificação só é necessária caso se pretenda obter funções de codificação que emitam *assembly* ou gerar o *translator*. O primeiro passo é fazer o mapeamento entre os construtores e a representação em *assembly*:

```
assembly component
    {*}A            is $1
    {*}R            is $1
    {*}Ri           is $1
    {*}E            is $1
    {*}Er           is $1
```

```
    {*}C            is $1
    {*}B            is $1
    {*}DPTR         is $1
    {*}ARi          is $1
    {*}AE           is $1
    {*}AR           is $1
    {*}AD           is $1
    {*}EA           is $1
    {*}ED           is $1
    {*}REr          is $1
    {*}EEr          is $1
    {*}RA           is $1
    {*}RE           is $1
    {*}RD           is $1
    {*}ER           is $1
    {*}EE           is $1
    {*}ERi          is $1
    {*}RiA          is $1
    {*}RiE          is $1
    {*}RiD          is $1
    {*}PC           is $1
    {*}A.DPTRi      is $1
    {*}DPREi.A      is $1
    {*}AB           is $1
    {*}CB           is $1
    {*}CnB          is $1
```

Depois descreve-se a relação entre os operandos e a representação em *assembly*:

```
assembly operand
    [arg8d arg16d]  is "#%d"
    op8             is "%s"
    op9             is "@R%d"
```

E por fim, define-se a sintaxe das instruções em *assembly*:

```
assembly syntax
    onearg^A        "A"
    onebitarg^C     "C"
    xchd^ARi        "A",        op9
    xch^ARi         "A",        op9
    xch^AR"A",      op8
    twoarg1^ARi     "A",        op9
    twoarg1^AE      "A",        arg8
    twoarg1^AR      "A",        op8
    twoarg1^AD      "A",        arg8d
    twoarg2^ARi     "A",        op9
    twoarg2^AE      "A",        arg8
    twoarg2^AR      "A",        op8
    twoarg2^AD      "A",        arg8d
    twoarg2^EA      arg8,       "A"
    cjne^AEEr       "A",        arg8,       arg81
    cjne^ADEr       "A",        arg8d,      arg8
    mov^AR          "A",        op8
```

```
mov^AE          "A",        arg8
mov^ARi         "A",        op9
mov^AD          "A",        arg8d
mov^RA          op8,        "A"
mov^EA          arg8,       "A"
mov^RiA         op9,        "A"
mov^DPTR        "DPTR",     arg16d
movc^DPTR       "A",        "@A" + "DPTR"
movc^PC         "A",        "@A" + "PC"
movx^ARi        "A",        op9
movx^A.DPTRi    "A",        "@DPTR"
movx^RiA        op9,        "A"
movx^DPTRi.A    "@DPTR",    "A"
```

NOTA: Esta especificação do µC 8051 não se encontra completa, nem optmizada.

Apêndice D

Protótipos das classes do BEDS

Classe Program

```
class Program {
private:
      //...
public:
      Set<Function *> listOfFunctions;
      Set<void *> flowNodes;
      Program( );
      void setRootNode(void *croot);
      void *getRootNode( );
      void setIdentTable(IdentifierTable *ctable);
      IdentifierTable *getIdentTable( );
      void setMainFunction(Function *cfunction);
      Function *getMainFunction( );
      virtual ~Program( );
      bool insFunction(Function *cfun);
      bool remFunction(Function *cfun);
      int hasFunction(Function *cfun);
      int howManyFunctions( );
};
```

Classe Function

```
typedef enum { FUNCTION=1,PROCEDUR, COMPSTATE} Ftype;

class Function {
private:
      //...
public:
      Set<void *> fflowNodes;
      IndList<char *> argList;
      Function();
      virtual ~Function( );
      void setName(char *caux);
      char *getName( );
      void setLevel(int clevel);
      int getLevel( );
      void setType(Ftype ctype);
      Ftype getType( );
      void setRootNode(void *croot);
      void *getRootNode( );
      void setParent(Function *cpar);
      Function *getParent( );
```

```
    void setIdentTable(void /*IdentifierTable*/ * ctable);
    void /*IdentifierTable*/ *getIdentTable();
    void setBlockMemory(BlockMemory *cblock);
    BlockMemory *getBlockMemory( );
    bool insFlowNode(void *Cdflownode);
    bool remFlowNode(void *Cdflownode);
    int hasFlowNode(void *Cdflownode);
    int howManyFlowNodes( );
};
```

Classe IdentifierTable

```
class IdentifierTable {
private:
    //...
public:
    StringDict<CellIDTable *> listID;
    IdentifierTable();
    IdentifierTable(char *cname);
    IdentifierTable(char *cname, EType ctype);
    virtual ~IdentifierTable(){}
    char *getAddrName(char *name);
    CellIDTable *insIDSymb(char *cname);
    CellIDTable *insIDSymb(char *cname, EType ctype);
    int hasIDSymb(char *cname);
    EType getTypeIDSymb(char *cname);
    bool setTypeIDSymb(char *cname, EType ctype);
    char *getNextSymbol(char *cname);
    CellIDTable *getAddrCellID(char *cname);
    char *getNewIDTempByName(FlowNode *node, DataTransfer *crt);
    CellIDTable *getNewIDTempByID(FlowNode *node, DataTransfer *crt);
    int getNewIDTempByNumber(FlowNode *node, DataTransfer *crt);
    char *getNewIDConst(FlowNode *node, DataTransfer *crt);
    bool freeIDTemp(char *caux);
    bool freeIDTemp(int caux);
    void freeIDConst(int caux);
    char *getNextVarName(char *cname);
    int howManyVars();
    char *getVarOfNumber(int cnum);
    int getVarNumber(char *cname);
};
```

Classe CellIDTable

```
class CellIDTable {
private:
    //...
public:
    TwoKeySet<FlowNode *, DataTransfer *> IDreg;
    FlowNode *getNextAssignNode(FlowNode *node);
    DataTransfer *getLastAssignInNode(FlowNode *node);
    CellIDTable();
```

```
        CellIDTable(EScope cscope);
        CellIDTable(EType ctype);
        virtual ~CellIDTable();
        void setScope(EScope cscope);
        EScope getScope();
        void setClass(EClass cclass);
        EClass getClass();
        void setType(EType ctype);
        EType getType();
        void addAsgn(DataTransfer *crt);
        bool remAsgn(DataTransfer *crt);
        int hasAsgn(DataTransfer *crt);
        void setIDconst(Constant *cconst);
        Constant *getIDconst();
        void setIDmem(Memory *cmem);
        Memory *getIDmem();
        void setIDfunction(Function *cfun);
        Function *getIDfunction();
        void setIDCRMF(int ccrm, ActualValue cval);
        int getIDCRMF();
        ActualValue getActualValue();
        void setVarNumber(int numb);
        int getVarNumber();
};
```

Classe FlowNode

```
class FlowNode{
private:
        //...
public:
        Set<void *> inEdges;
        IndList<void *> outEdges;
        FlowNode();
        void setFunction(void /*Function*/ *cfun)
        void /*Function*/ *getFunction();
        virtual ~FlowNode();
        void* getInterval();
        void setInterval(void *Cinterval);
        void setStartNode(bool Cst);
        bool getStartNode( );
        void setEndNode(bool Cst);
        bool getEndNode( );
        int howManyOutEdges();
        void sethasBeenVisited();
        void clrhasBeenVisited();
        bool gethasBeenVisited();
        void addLinkIn(void *Clink);
        bool remLinkIn(void *Clink);
        void addLinkOut(int Lpos, void *Clink);
        bool remLinkOut(void * Clink);
```

```
    void settopNumber(int Cnumber);
    int gettopNumber( );
    int howManyInEdges( );
};
```

Classe DataTransfer

```
#define ASSIGN 0
#define PHYASGN 1
#define ATRIBASGN 2
#define LABELASGN 3
#define JMPASGN 4
#define CJMPASGN 5
#define RETASGN 6

typedef struct {
    int rule;
    void *phyTarget;
    int phyType;
} phyContainer;

class DataTransfer {
private:
    //...
public:
    IndList<phyContainer *> list;
    DataTransfer();
    DataTransfer(FlowNode *Cflownode);
    int AsgnType();
    void setAsgnType(int );
    virtual ~DataTransfer();
    void setFlowNode(FlowNode *, bool);
    FlowNode *getFlowNode();
    bool insFlowDependent(DataTransfer *Cdep);
    bool insFlowSupporter(DataTransfer *Csupp);
    bool remFlowDependent(DataTransfer *Cdep);
    bool remFlowSupporter(DataTransfer *Csupp);
    int hasFlowDependent(DataTransfer *Cdep);
    int hasFlowSupporter(DataTransfer *Csupp);
    int howManyFlowDependents();
    int howManyFlowSupporters();
    DataTransfer *getNextFlowDependent(DataTransfer *crt);
    DataTransfer *getNextFlowSupporter(DataTransfer *crt);
    Set<DataTransfer*>*getAddrOfFlowDependents( );
    Set<DataTransfer *> *getAddrOfFlowSupporters( );
    int getTopNumber( );
    void setNumber(int Cnumber);
    int getNumber( );
    void insNxtAtrib(DataTransfer *crt);
    DataTransfer *getNxtAtrib(DataTransfer *crt);
    bool remNxtAtrib(DataTransfer *crt);
```

```
        void setPhyTarget(int rule, void *caux, int t);
        phyContainer *findRuleInList(int rule, int n);
        void *getPhyTarget(int rule, int n);
        int getPhyType(int rule, int n);
        virtual void *getTarget( );
        virtual void *getSource( );
        virtual void setTarget(void *);
        virtual void setSource(void *);
};
```

Classe Expression

```
class Expression {
private:
        //...
public:
        void *state_label;
        Expression( );
        virtual ~Expression(){}
        void setDataTransfer(DataTransfer *Cdt);
        DataTransfer *getDataTransfer( );
        void setType(EType ctype);
        EType getType( );
        int getOp(void);
        void setOp(int cop);
        void *getState( );
        void setState(void *cstate);
        void setLeft(Expression *cleft);
        Expression *getLeft( );
        void setRight(Expression *cright);
        Expression *getRight( );
        void setStatus(int st);
        int getStatus( );
        void *getVal(int r, int n);
        void setVal(int r, void *caux, int t);
        int getValType(int r, int n);
        void DeterminateType();
};
```

Classe TreeSelector

```
typedef struct list {
        Expression *exp;
        struct list *next;
} List;

class TreeSelector{
private:
//...
public:
        TreeSelector(Function *);
        virtual ~TreeSelector( );
```

```
      bool initTreeSelector( );
      void buildList( );
      void initPickTree( );
      Expression *getNextTree( );
```

};Classe Color_Allocator

```
class Color_Allocator {
private:
      //...
public:
      Color_Allocator(Function *cfun, int InitDisp);
      virtual ~Color_Allocator( );
      //...
      void Allocate_Registers( );
};
```

Classe IterativeDFAnalyse

```
class IterativeDFAnalyse {
private:
      //...
public:
      IterativeDFAnalyse(Function *cfun);
      virtual ~IterativeDFAnalyse( );
      void Worklist_Iterate( );
      void AliveAnalyse( );
};
```

Classe ssaDF

```
class ssaDF {
private:
      //...
public:
      ssaDF(MyFunction *);
      virtual ~ssaDF();
      void placeAllPhyFunctions();
      void InsertAllPhyAssign();
};
```

Classe StructuralAnalysis

```
class StructuralAnalysis {
private:
//...
public:
      StructuralAnalysis(Function *cfun);
      virtual ~StructuralAnalysis(){}
      IntervalNode *Analize( );
      void setFunction(Function *cfun);
      Function *getFunction( );
};
```

Apêndice E

Algoritmo de alocação por coloração

Pretende-se neste apêndice ilustrar como se desenrola todo o processo de alocação através do algoritmo de coloração de *Chaitin*, tentando descrever algumas das situações mais relevantes e assinalar alguns dos pontos mais delicados.

O exemplo decorre sobre o bloco de instruções que se encontra representado do lado esquerdo da Fig E1, que após a conversão para SSA, resulta no bloco que se encontra do lado direito da mesma figura.

A utilização da função *print(...)* é simbólica e tem por objectivo justificar a presença da definição de cada uma das variáveis em relação às posições que estas ocupam no bloco de código. O *input(...)* permite atribuir um valor a uma variável.

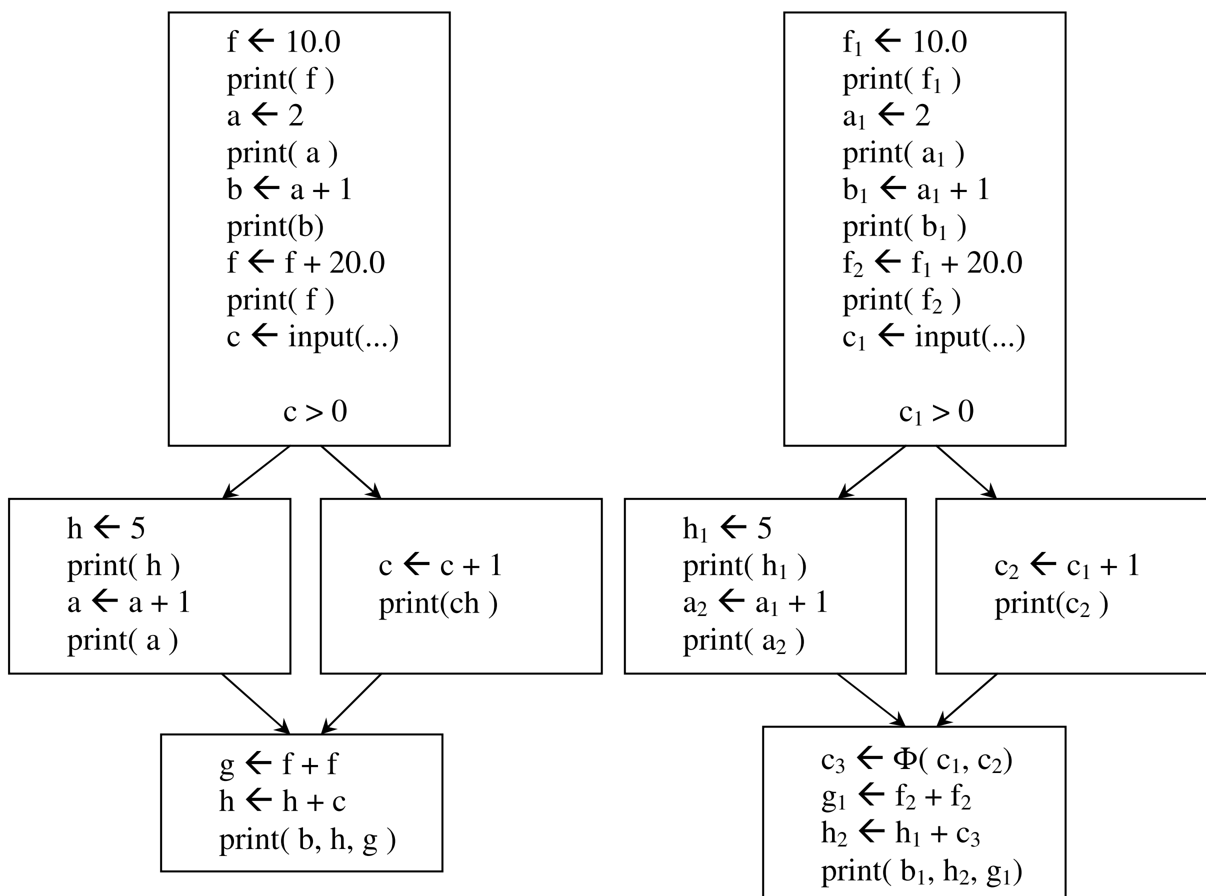


Fig E1 - Programa exemplo.

Considera-se ainda, que existem quatro registos simples (R_0,...,R_3), que são utilizados pelas variáveis *a*, *b*, *c* e *h*. E dois registos compostos (F_0, F_1), utilizados pelas variáveis *f* e *g*, de tal forma que:

$F_0 = (R_0, R_1)$
$F_1 = (R_2, R_3)$

O total de cores disponíveis é, como tal, quatro e o conjunto total de variáveis, segundo a representação SSA, é o seguinte:

$\{ f_1, a_1, b_1, f_2, c_1, h_1, a_2, c_2, c_3, g_1, h_2 \}$

Com esta formulação, não existe qualquer interferência entre as variáveis e os registos. No entanto é indispensável, que os registos utilizados na representação das variáveis *f* e *g* formem um par segundo a composição descrita para F_0 e F_1.

	R_0	R_1	R_2	R_3	f_1	a_1	b_1	f_2	c_1	h_1	a_2	c_2	c_3	g_1
R_1	t													
R_2	t	t												
R_3	t	t	t											
f_1	f	f	f	f										
a_1	f	f	f	f	t									
b_1	f	f	f	f	t	t								
f_2	f	f	f	f	f	t	t							
c_1	f	f	f	f	f	t	t	t						
h_1	f	f	f	f	f	t	t	t	t					
a_2	f	f	f	f	f	f	t	t	t	t				
c_2	f	f	f	f	f	f	t	t	f	f	f			
c_3	f	f	f	f	f	f	t	t	f	t	f	f		
g_1	f	f	f	f	f	f	t	f	f	t	f	f	t	
h_2	f	f	f	f	f	f	t	f	f	f	f	f	f	t

Fig E2 - Matriz de adjacências correspondente ao programa exemplo da Fig E1.

Para se determinar se duas variáveis se encontram simultaneamente "vivas", é necessário, como já foi referido, realizar a análise do período de vida. Os detalhes de execução desta forma de análise não são aqui apresentados (ver capitulo 5), de forma a não alongar este exemplo e uma vez que facilmente se obtém a mesma informação através de uma simples observação do código. Resulta daí a matriz de adjacências da Fig E2.

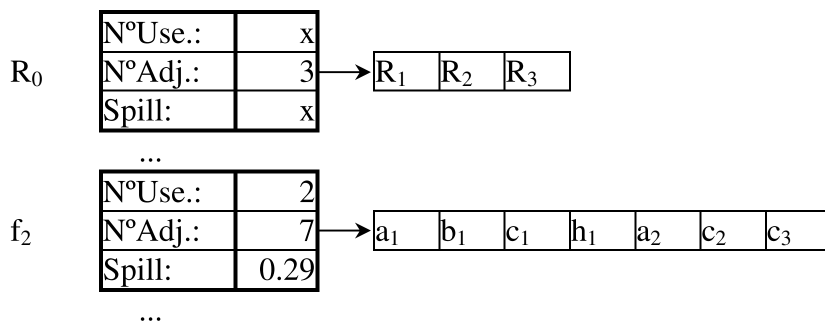


Fig E3 - Lista de adjacências do exemplo.

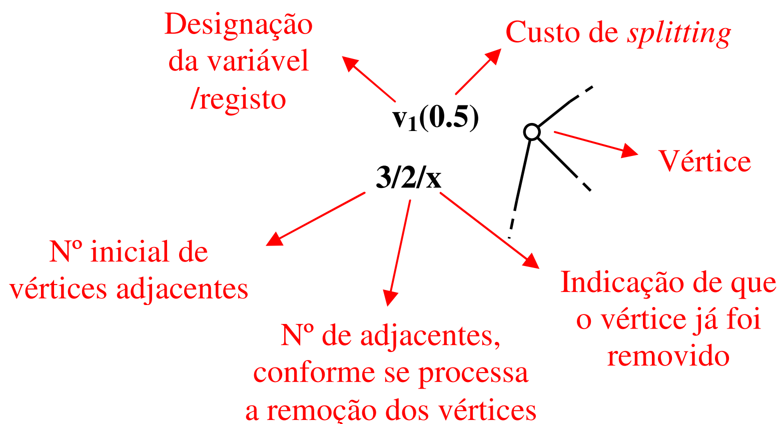


Fig E4 - Legenda para um vértice do grafo de adjacências.

Considera-se ainda que os operandos das funções Φ(...) não entram em conflito entre eles, isto porque na prática as instruções que substituem estas funções, são inseridas nos nodos antecessores, pelo que na realidade as definições de cada um dos operandos nunca alcançam a função Φ(...).

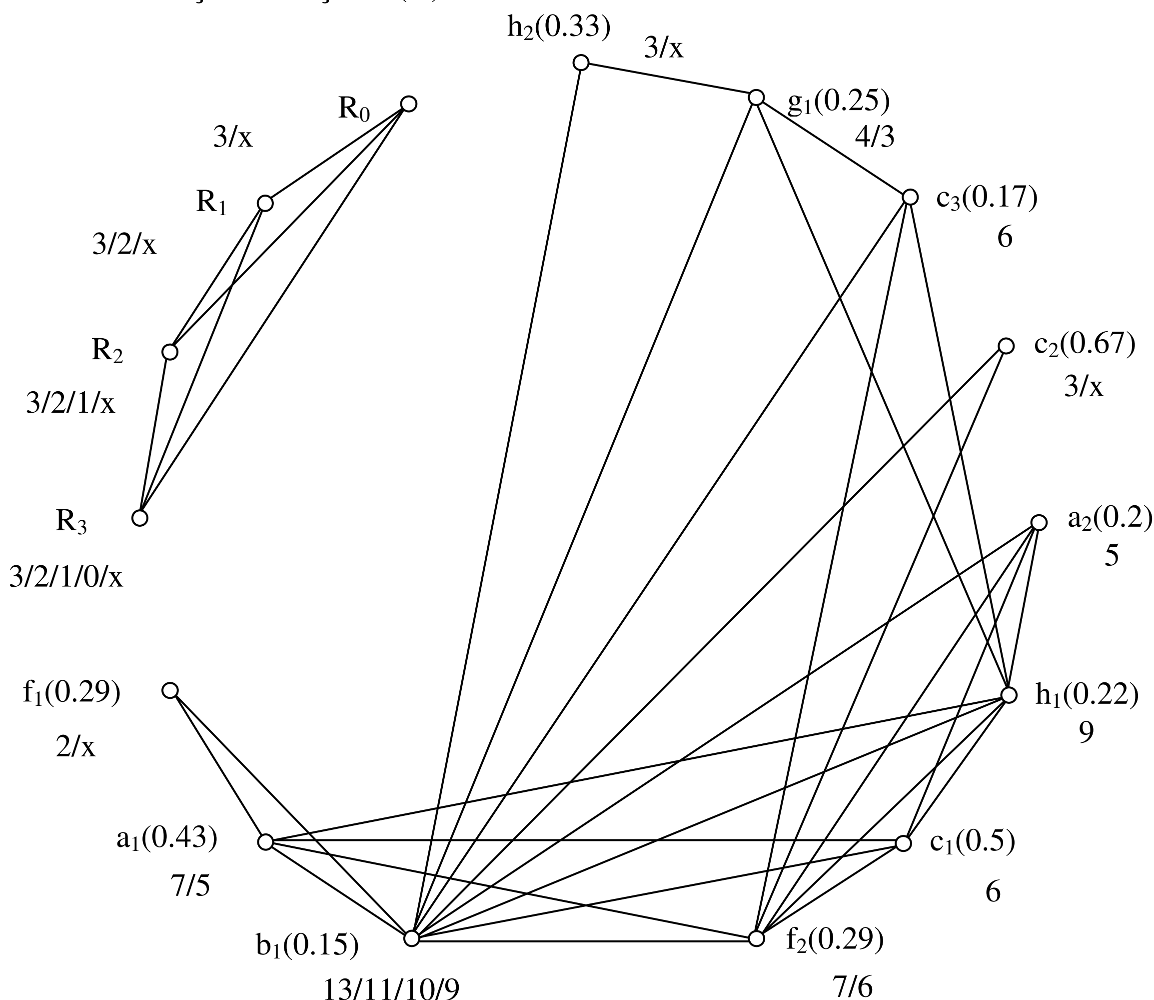


Fig E5 - Grafo de interferências, com os custos de *splitting* e o número "pesado" de adjacentes.

De notar que se considera que numa mesma expressão, os operandos não entram em conflito com a variável que guarda o resultado, excepto no caso de permanecerem "vivos" para além dessa expressão.

Na Fig E3 encontram-se representadas duas entradas da lista de adjacências, uma representando um registo (R_0) e outra uma variável (f_2). Cada entrada possui o número de utilizações da variável (não se aplica aos registos), o número de vértices adjacentes e o custo de *splitting* (também não se aplica aos registos), calculado com base na razão entre os dois valores anteriores.

O respectivo grafo encontra-se representado na Fig E5, onde junto de cada um dos vértices, está a designação do registo ou variável, o custo de *splitting* (entre parêntesis) e o número de adjacentes que o vértice possui conforme se processa o grafo. Este último valor é colocado a *x* quando o vértice é removido do grafo. A legenda da informação associada a cada vértice encontra-se representada na Fig E4.

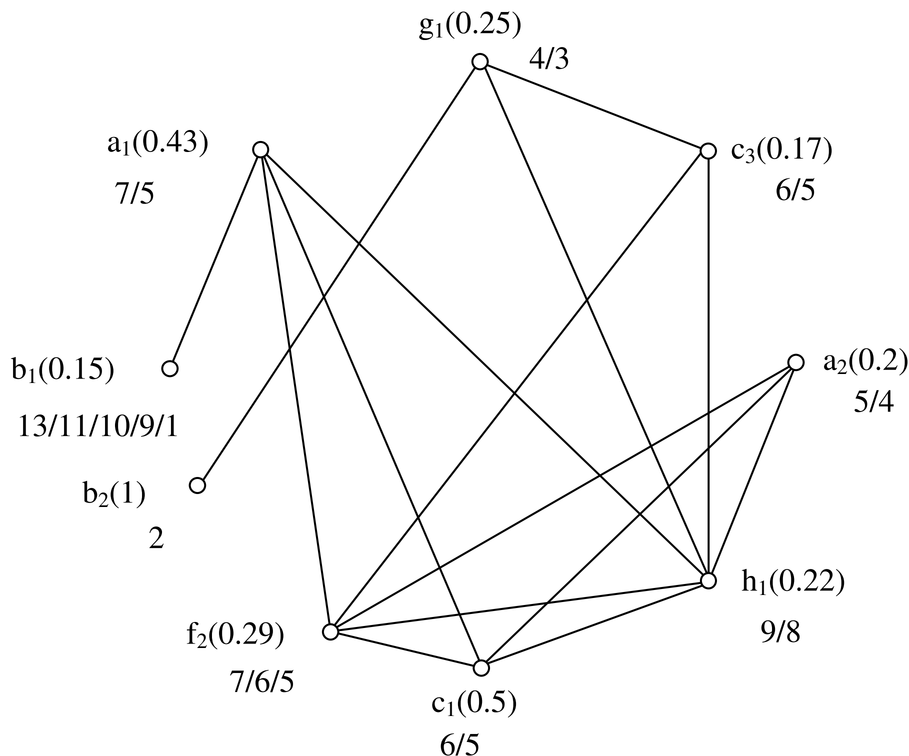


Fig E6 - Grafo de adjacências após o *splitting* de b_1.

De notar que o custo de *splitting* e o número de vértices adjacentes, é determinado considerando a quantidade de registos necessária a cada variável. Por exemplo, no caso de f_1 interferir com a_1, então f_1 contribui com duas unidades para o número de adjacentes de a_1, uma vez que f_1 necessita de dois registos. Na relação inversa, a_1 apenas contribui com uma unidade para o número de adjacentes de f_1, uma vez que a_1 apenas necessita de um registo. Como o número de adjacentes é necessário para determinar o valor de *splitting*, este também é afectado pelo "peso" de cada variável.

Analisando o grafo, constata-se que é possível remover R_0, R_1, R_2, R_3, f_1, c_2 e h_2, uma vez que para qualquer um destes vértices, a diferença entre o número de cores e os respectivos graus é superior ou igual ao número de registos necessários a cada um.

De forma a sistematizar o processo começa-se por remover os vértices, conforme tal seja possível, a partir da variável R_0 e no sentido anti-horário, colocando-os numa *stack*.

Desta forma, o primeiro candidato é o próprio R_0, que ao ser removido contribui para reduzir em uma unidade, o número de interferências de todos os vértices a ele adjacentes, repetindo-se o processo para R_1, R_2, R_3 e f_1. De notar que este último reduz o número de adjacentes de a_1 e b_1 em duas unidades.

O processo continua, removendo-se h_2, o que contribui para reduzir o número de adjacentes de g_1, mas ao contrário dos casos anteriores, ainda não é possível executar tal operação, uma vez que a diferença entre o número de cores disponíveis e o grau do vértice é inferior ao número de registos necessários à respectiva variável.

Pelo que se chega a uma situação em que nenhum dos vértices, dos que ainda se encontram no grafo, está em condições de ser removido. Há então, que recorrer a mecanismos mais complexos, que no caso do BEDS consiste em realizar o *splitting* de um ou mais vértices, os quais são seleccionados com base nos valores previamente calculados para este tipo de operações. Desta forma o primeiro candidato é a variável b_1.

Como no BEDS, a operação de *splitting* corresponde a dividir uma variável em tantas quantas as novas definições inseridas, quer estas advenham das operações de *load*, quer das funções $\Phi(...)$ entretanto necessárias, acontece, no presente exemplo, que seja necessário introduzir uma nova variável (b_2), ao nível da representação e como tal do grafo de interferências. Neste último, é ainda preciso (re)determinar as interferências de b_1 e b_2 com as restantes variáveis. O grafo resultante encontra-se representado na Fig E6.

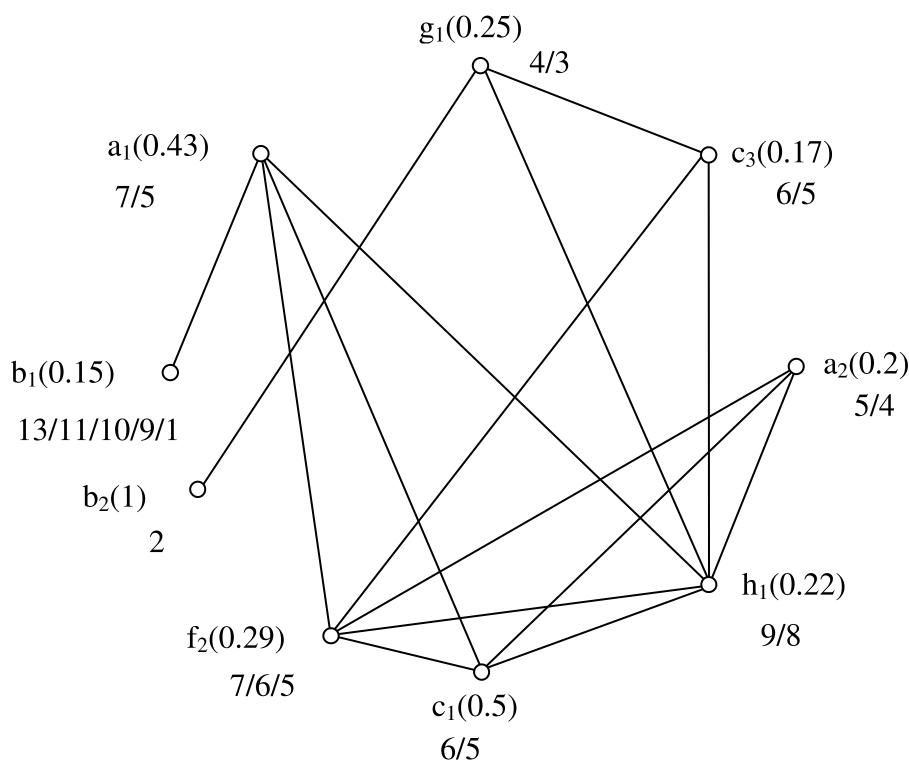


Fig E7 - Grafo de interferências antes da segunda operação de *splitting*.

Após a remoção dos vértices b_1, b_2, g_1, e c_3 volta a colocar-se a mesma situação, ou seja, nenhum dos restantes vértices está em condições de garantir a sua remoção. A situação actual do grafo encontra-se representada na Fig E7.

A escolha do próximo candidato recai sobre a_2, mas facilmente se constata que não existe qualquer benefício em realizar o *splitting* deste vértice. Caso não existam

mecanismos que permitam detectar este tipo de situações, o que acontece é que o vértice é mesmo sujeito a operação de *splitting*, acrescentando assim instruções que apenas degradam a qualidade do código final. É, como tal, aconselhável implementar alguns mecanismos que permitam controlar a ocorrência deste tipo de situações e assinalem o momento em que se deve escolher outro candidato, o que no presente exemplo corresponde a escolher o vértice h_2. Procede-se então à sua decomposição conforme o que se fez para b_1 e que resulta no grafo da Fig E8. Repare-se que a remoção deste vértice, também não acarreta vantagens, mas ao contrário do caso anterior, a situação em que se encontra h_2 não é facilmente detectável, pelo que se aceita como sendo satisfatória a sua ocorrência.

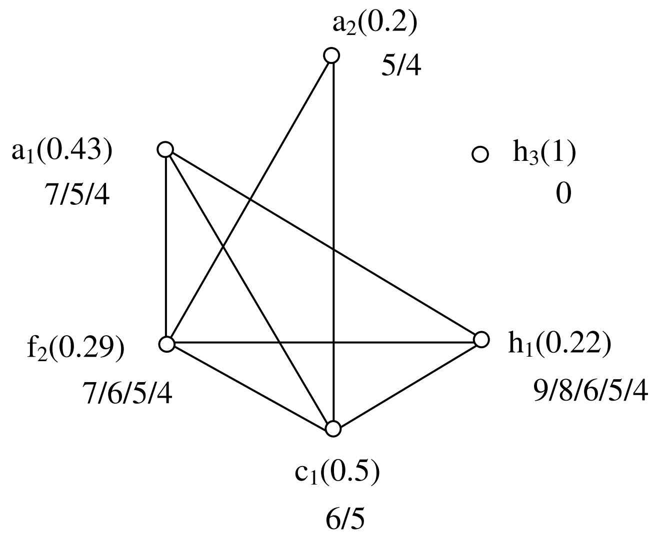


Fig E8 - Grafo após o *splitting* de h_1.

Após a remoção de h_3, volta a ser indispensável realizar nova operação de *splitting*, sendo h_1 novamente o candidato eleito. Mas nas actuais condições do grafo, tal operação é em tudo semelhante à situação que ocorreu para a_2, pelo que é aconselhável seleccionar outro candidato.

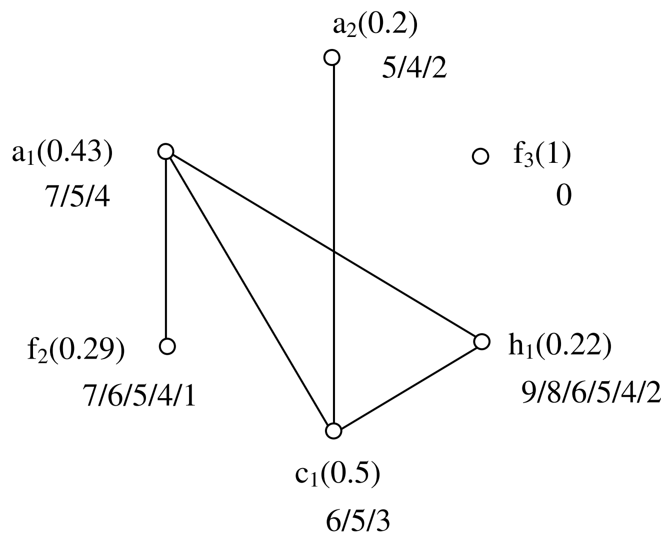


Fig E9 - Grafo de interferências após o *splitting* de f_2.

Desta forma, e uma vez que o *splitting* de h_1 e a_2 não produz qualquer vantagem, a escolha recai sobre f_2, obtendo-se o grafo que se encontra representado na Fig E9, onde se pode constatar a introdução de uma nova variável (f_3).

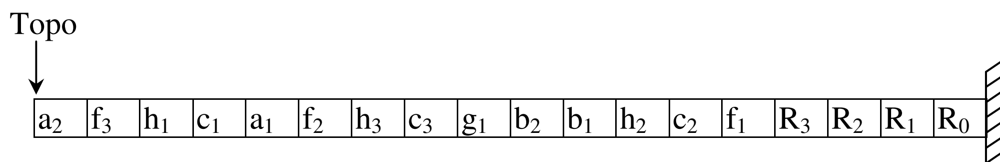


Fig E10 - *Stack* com os vértices do grafo inseridos pela ordem de remoção.

Torna-se assim possível remover f_2, a_1, c_1, h_1, f_3, e conclui-se com a_2, ficando os elementos ordenados na *stack* conforme se encontra representado na Fig E10.

Após a remoção de todos os vértices do grafo para a *stack*, pode-se finalmente começar a colorir o mesmo. As cores disponíveis vão de 1 a 4, sendo a atribuição feita pela mesma ordem e conforme a disponibilidade de cada uma.

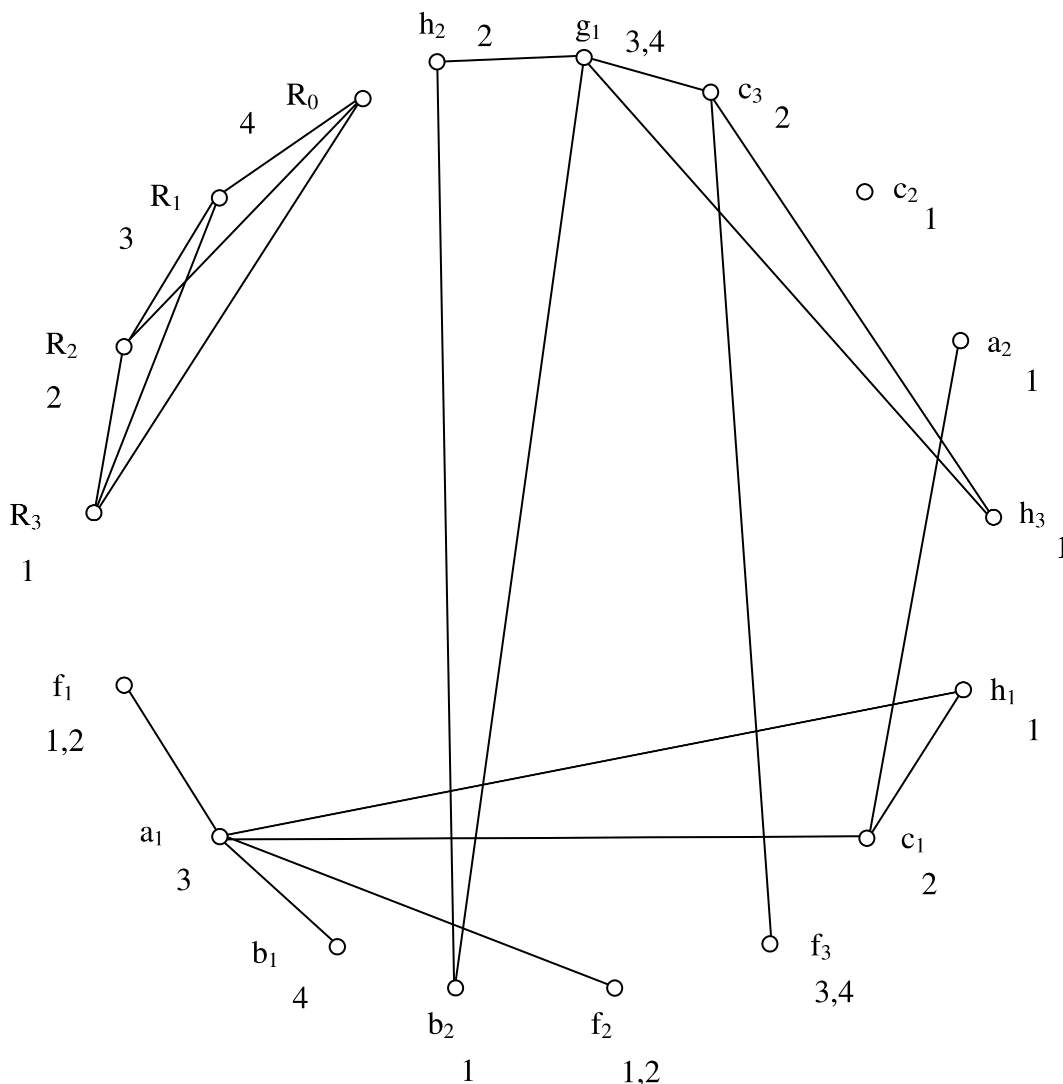


Fig E11 - Grafo de interferências devidamente colorido.

O processo consiste em inserir cada um dos vértices que se encontra na *stack*, novamente no grafo, atribuindo-lhes uma cor (ou conjunto de cores) que seja distinta de das cores dos vértices adjacentes. O resultado final da coloração encontra-se representado na Fig E11.

É de salientar, que para se conseguir estes resultados, se considerou que as cores dos registos compostos seriam formadas pelos pares (3,4) e (1,2), e que a atribuição tem em atenção as cores já atribuídas aos vértices adjacentes de forma a viabilizar a alocação dos registos compostos. O resultado final da alocação encontra-se na Fig E12, onde já estão representadas todas as variáveis entretanto inseridas devido às operações de *splitting*.

A geração final de código pode ser feita através de uma nova selecção das instruções, a qual apenas difere da pré-selecção, devido às instruções entretanto inseridas para as operações de *splitting*.

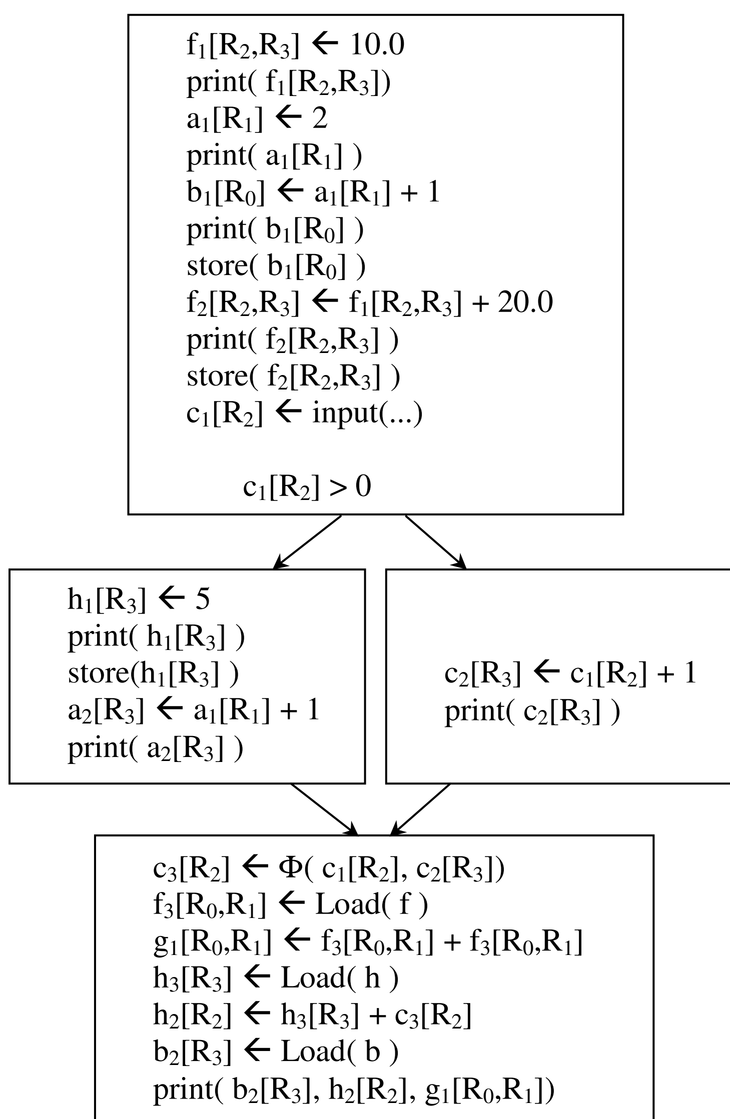


Fig E12 - Representação do programa já com os registos atribuídos.

Em determinados processadores, designadamente em arquitecturas tipo RISC, é ainda possível, realizar a geração do código final, inserindo apenas as novas instruções, desde que se convencione previamente as instruções a utilizar nas operações de *splitting*.