Alocação Estruturada de registradores
através de coloração de grafos

Este exemplar corresponde à redação final da tese defendida pelo Sr. MAURÍCIO BRETERNITZ JÚNIOR e aprovada pela Comissão Julgadora.

Orientador: prof. Dr. TOMASZ KOWALTOWSKI

Tese apresentada ao Instituto de Matemática, Estatística e Ciências de Computação da Universidade Estadual de Campinas (UNICAMP) como requisito parcial para a obtenção do título de Mestre em Ciências da Computação.

1984

àquele que é capaz de guardar vocês de cair, e de trazê-los sem falha e com alegria diante de sua gloriosa presença, ao único Deus nosso salvador, através de Jesus Cristo nosso Senhor, sejam a glória, majestade, poder e autoridade, desde todas eras do passado, e agora, e para sempre.

Bíblia, Jd. 1:24

RESUMO

Este trabalho descreve a implementação de um mecanismo de alocação de registradores em programas estruturados, através da técnica de coloração de grafos sugerida por Chaitin. O problema da coloração é resolvido colorindo-se individualmente uma série de sub-grafos do grafo de interferências global, escolhidos de acordo com a estrutura do programa. Técnicas de análise de fluxo são utilizadas para construir os grafos de interferência.

ABSTRACT

This work describes an implementation of register allocation in structured programs using graph coloring techniques introduced by Chaitin. The coloring problem is solved by individually coloring a series of sub-graphs of the global interference graph, chosen according to the program's structure. Flow analysis techniques are used to build the interference graphs.

ÍNDICE

1. INTRODUÇÃO:

Durante a geração de código para a máquina objeto, a "alocação de registradores" é um dos problemas básicos que devem ser resolvidos por um compilador: tendo disponível um conjunto finito de registradores rápidos na UCP da máquina, o compilador deve decidir quais valores computados, e quando, devem residir nos registradores, ou ser armazenados em posições de memória.

Este problema vem se tornando particularmente importante com o avanço da tecnologia de fabricação de circuitos integrados VLSI: partindo das arquiteturas mais antigas de máquina com apenas um acumulador, vieram a existir máquinas com vários acumuladores, tipicamente dois a dezesseis, e atualmente são muito comuns, até entre microprocessadores, UCP's contendo trinta registradores ou mais. Mais recentemente, aconteceu o desenvolvimento de UCP's com grande quantidade de registradores, da ordem de centenas deles, onde uma boa solução do problema de alocação de registradores se torna fundamental para o bom desempenho da máquina. A alocação deve ser feita de tal forma que, na maior parte do tempo, todo o processamento da UCP se refira a valores contidos em registradores, minimizando referências à memória, que são mais lentas. O projeto IBM801 [Radin82] e a arquitetura RISC ("reduced instruction set computer") [PatSéq82] são exemplos de UCP's com um grande número de registradores.

A solução proposta por Chaitin [Chait81, Chait82], explorada neste trabalho, utiliza técnicas de coloração de grafos para efetuar esta alocação de registradores. Esta solução consiste em construir um grafo onde cada vértice representa um valor computado pelo programa, e cada aresta representa "interferências" entre estes valores. Dois valores interferem entre si quando, ao serem calculados, eles não podem ser armazenados no mesmo registrador. A seguir é feita uma coloração deste grafo, utilizando o conjunto de

registradores existente na máquina como "cores".

Uma coloração de um grafo é a atribuição de uma cor a cada um de seus vértices de tal maneira que, se dois vértices são adjacentes, então eles tem cores diferentes. Desta forma, caso seja encontrada uma coloração, dois valores que "interfiram" entre si não serão alocados ao mesmo registrador.

A idéia básica explorada neste trabalho é experimentar com a proposta de Chaitin para o caso de compilar linguagens estruturadas utilizando técnicas de análise de fluxo de programas. Tendo em vista a divisão do programa em blocos estruturados, imposta pela própria linguagem de programação, é mais simples estabelecer "escopos" de vida a valores computados pelo programa. É feita então uma alocação de registradores limitada a trechos do programa de acordo com a sua estrutura, visando trabalhar com grafos de interferencia compostos por um número menor de vértices, onde a solução para o problema de coloração de grafos é mais fácil. Técnicas de análise de fluxo permitem conhecer quantos e quais valores devem ser considerados durante cada ponto de programa analisado.

Este trabalho descreve um compilador para um subconjunto de MODULA-2 que implementa as idéias acima e descreve alguns resultados experimentais.

Apresentamos, a seguir, uma descrição da linguagem compilada, a arquitetura para a qual será gerado código, e os mecanismos utilizados para efetuar a análise de fluxo de dados.

A seguir, faremos uma descrição breve de técnicas de alocação em geral, apresentaremos o algoritmo utilizado por Chaitin, seguido por uma descrição do compilador implementado Finalmente, apresentaremos alguns resultados experimentais,

algumas considerações e sugestões para futuras experiências.

1.1 DESCRIÇÃO DA LINGUAGEM COMPILADA:

A linguagem aceita pelo compilador experimental, descrita pela gramática apresentada no Anexo I, é um subconjunto da linguagem MODULA-2, criada por Wirth [Wirth83]. (José Pedro Jr. [JPJ83] descreve uma implementação de um compilador para Modula-2.)

Neste trabalho, as restrições impostas são relativas à simplicidade de implementação, evitando aspectos da linguagem que não dizem respeito ao problema de alocação de registradores em questão.

O subconjunto de MODULA-2 aceito é:

- tipos simples de dados: variáveis e constantes são sempre do tipo inteiro ou booleano, e não foram implementadas operações envolvendo matrizes e indexação. Para verificar como o mecanismo de alocação de registradores se comporta para o cálculo de expressões aritméticas que normalmente ocorrem durante a indexação de matrizes, é possível programar diretamente expressões similares a estas, de maneira que esta limitação não afeta a experimentação. O mesmo ocorre com relação a operações com apontadores e dereferenciação.

- comandos: de atribuição, condicional e apenas o comando repetitivo "LOOP", que abrange as situações correspondentes aos comandos "REPEAT" e "WHILE" como casos particulares.

- declarações: é permitida apenas a declaração de tipos sinônimos aos inteiros e booleanos. Não foram implementados procedimentos, que exigiriam análise de fluxo inter-procedimentos. Sua implementação não é vital para o estudo em questão, e sugerimos que seja alvo de um estudo posterior.

1.2 DESCRIÇÃO DA MÁQUINA ALVO:

Consideramos que a UCP para que o compilador vai gerar código final é uma arquitetura de três endereços, que contém N registradores homogêneos, de maneira que cada instrução de máquina que execute uma operação pode referir-se indistintamente a qualquer dos registradores existentes na UCP. A arquitetura é de "três enderêços", pois estas instruções são da da forma

```
OP      end-1,end-2,end-3
```

significando:

(end-1) ← (end-2) "OP" (end-3)

onde end-i pode designar um registrador ou uma posição de memória. Por exemplo:

```
ADD     R2,R2,R1
        sendo os Ri registradores da máquina
```

significa: R2 recebe R2+R1, e

```
ADD     R2,A,B
```

significa: somar o conteúdo das posições de memória cujos rótulos são A e B, e armazenar o resultado em R2.

A família VAX-11 de computadores [Vax84] é um exemplo de uma UCP com 16 registradores homogêneos que possui instruções com três enderêços.

Para fins de avaliação de desempenho da alocação de

registradores consideramos, também, que cada instrução de máquina tem um custo unitário quando os três argumentos (resultado e dois operandos) a que se refere forem registradores. Cada vez que um destes argumentos é uma posição de memória, esta instrução terá um custo adicional de uma unidade.

Esta é uma aproximação do custo de execução de instruções nas arquiteturas mais comuns, onde estimamos que o tempo necessário para a execução de uma operação internamente à UCP seja comparável ao tempo de acesso à memória para buscar um operando. Para avaliar o tamanho do código gerado, também é possível considerar que cada instrução consistirá em uma palavra para a identificação do código da operação, e uma palavra adicional para especificar cada operando que resida em memória. Este modelo reflete aproximadamente o que se passa nas UCP mais comuns atualmente, como o PDP-10 e o VAX-11, ou os microprocessadores Motorola 68000 e Intel 8085.

A experimentação utilizando uma arquitetura com três endereços fornece resultados que podem ser estendidos para outras arquiteturas, como a arquitetura com dois endereços mais comum atualmente, conforme sugerido no item 5.2, onde descrevemos também como este método de alocação de registradores poderia ser aplicado a algumas arquiteturas não homogêneas.

1.3 ANÁLISE DE FLUXO DE DADOS:

Na implementação do compilador, é necessário resolver dois problemas de análise de fluxo de dados bastante semelhantes: "vida de variáveis" e "propagação de definições", que são resolvidos usando o método para análise de fluxo de dados apresentado a seguir.

Como a linguagem de programação sendo compilada é uma linguagem estruturada, ao examinar o fluxo de execução em um programa, podemos considerar o programa como sendo constituído por "blocos de execução":

1.3.1 Blocos de Execução:

Os blocos de execução podem ser: bloco condicional, bloco repetitivo, sequência de blocos ou blocos básicos:

a - Bloco condicional (B1 // B2):

Representa trechos do programa que são executados alternativamente, significando que, quando o fluxo de execução do programa atingir o início do bloco condicional, apenas um dos blocos "componentes" daquele bloco condicional em questão vai ser executado, atingindo o fim do bloco condicional [Fig. 1.3-1(a)].

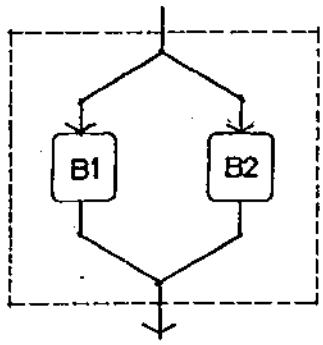
b - Bloco repetitivo B1;(B2;B1)+:

Representa dois trechos do programa que, quando o fluxo de execução do programa atingir o início do bloco repetitivo, o primeiro bloco componente será executado. O fluxo de execução pode então deixar o bloco, ou executar o segundo bloco componente, e atingir de novo o início do bloco repetitivo [Fig 1.3-1(b)].

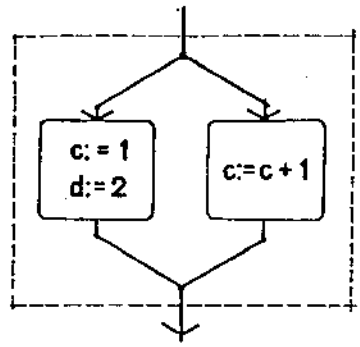
c - Sequência de blocos (B1; B2; ... Bn):

Representa uma lista de trechos do programa que são executados em sequência cada vez que o fluxo de execução do programa atingir o início do bloco "Sequência de blocos" [Fig 1.3-1(c)].

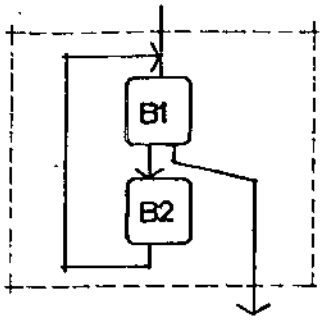
a) bloco condicional



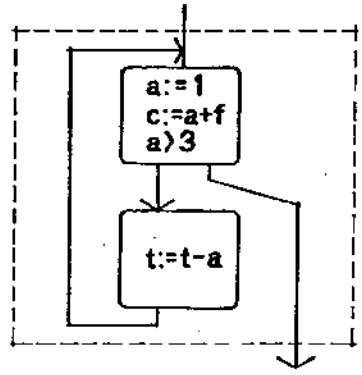

```
if a>1
    then
        c:=1;
        d:=2
    else
        c:=c+1
    end;
```



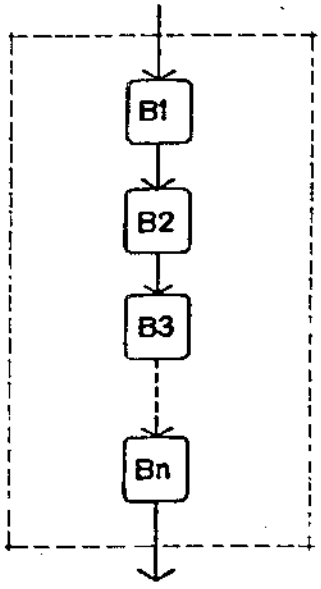
b) bloco repetitivo




```
loop
    a:=1;
    c:=a+f
exit if a>3;
    t:=t-a
end {loop}
```



c) sequencia de blocos




```
c:=1
if a>b then S1
       else S2;
loop
    a:=a+1
exit if c>Ø;
    b:=a+b
end;
```

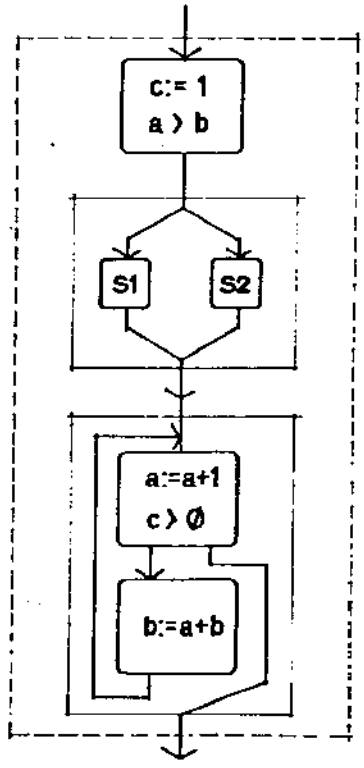


fig. 1.3-1

d - Blocos básicos:

Considerando o programa objeto final constituído por uma sequência linear de operações ou instruções, um trecho t da sequência de instruções constitui um bloco básico quando:

- não existem no programa instruções de desvio para os elementos de t, exceto possívelmente para a primeira instrução em t;

- nenhuma instrução de t, exceto possívelmente a última, é uma instrução de desvio.

De acordo com a definição acima, uma sequência vazia ou uma sequência constituída por uma única instrução constituem sempre blocos básicos. Na implementação aqui descrita procuramos sempre utilizar blocos básicos maximais, embora para fins de experimentação os algoritmos e estruturas de dados desenvolvidos possam trabalhar com blocos básicos neste sentido mais geral. Um bloco básico é considerado maximal se com a inclusão da próxima instrução que o precede ou segue no programa deixaria de ter as propriedades de um bloco básico.

1.3.2 Grafo de fluxo:

Supondo que o programa foi dividido em um conjunto de blocos básicos maximais, o grafo de fluxo do programa é o grafo orientado obtido quando cada vértice representa um bloco básico, e quando existe uma aresta ligando os vértices correspondentes aos blocos Bi a Bj quando a execução do bloco Bi pode ser seguida pela execução imediata do bloco Bj. As arestas partindo do vértice correspondente a um bloco B conduzem aos blocos sucessores de B, enquanto as arestas chegando em B partiram dos blocos predecessores de B [Fig. 1.3-2].

```
if a then
      B1
else
    loop
      B2
    exit if c;
      B3
    end (* loop *)
end; (* if *)
```

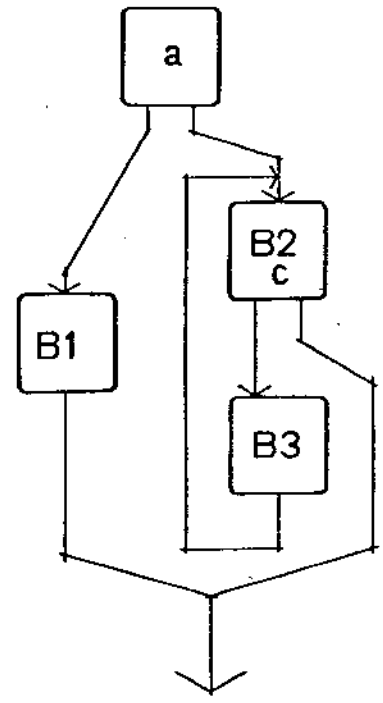


fig 1.3-2

1.3.3 Exemplo de problema de análise de fluxo: vida de variáveis:

A análise de fluxo de dados num programa é efetuada de maneira global, observando todo o programa para obter informações sobre como cada determinada parte do programa efetua transformações no "estado" global do programa, descrito pelo conjunto de suas variáveis, e como estas transformações vão afetar o fluxo de execução em outras regiões do programa.

Um exemplo clássico de análise de fluxo de dados em um programa é o de "vida" de variáveis: uma variável V está viva em um ponto p do programa, se existe um caminho

começando em p até um uso de V, sem que haja atribuição de valor a V neste caminho. Entende-se por uso da variável V uma instrução em que o valor de V aparece como um dos operandos.

Denotando por Xi e Yi, respectivamente, os conjuntos de variáveis vivas no início e no fim de um bloco básico Bi,

por Ci o conjunto de variáveis V cujo uso em Bi não é precedido de nenhuma atribuição de valor a V,

por Ki o conjunto de variáveis cujos valores são preservados por Bi, isto é, variáveis que não são alvo de atribuições em Bi,

e por Si o conjunto de blocos básicos sucessores de Bi no grafo de fluxo teremos para cada Bi:

$$Xi = Ki \wedge Yi \cup Ci$$

e

$$Yi = \bigcup_{j \text{ em } Si} Xj$$

Este é um sistema de equações com as incógnitas Xi e Yi, e com os coeficientes Ci e Ki. Estes coeficientes são determinados a partir das instruções que compõem cada bloco básico, e o problema de análise de fluxo de dados consiste em resolver este sistema de equações.

O problema de vida de variáveis é um problema de análise denominado regressivo e disjuntivo. Disjuntivo devido aos operadores de disjunção (i.e, união) entre os conjuntos Xj que aparecem no sistema de equações a resolver, e regressivo pelo fato de cada Yi depender de todos Xj, sendo os Xj sucessores de Yi. Existe uma condição de contorno inicial Ys na saída, pois ao final do programa não há mais nenhuma variável viva, condição que é propagada para trás, em sentido inverso ao fluxo de execução do programa.

Existem outros problemas de análise que podem ser também descritos por um sistema de equações da forma

$$Xi = Ki \wedge Yi \cup Ci$$

e

$$Yi = \underset{j \text{ em } \emptyset i}{\text{"op"}} Xj$$

sendo os Øi os conjuntos de sucessores ou predecessores de Bi, e "op" a operação de união ou a operação de intersecção de conjuntos.

Em função das operações "op" entre os conjuntos envolvidos, estes problemas são chamados de disjuntivos ou conjuntivos, e em função da dependência Øi ser dos sucessores ou predecessores são chamados de regressivos ou progressivos.

1.3.4 Algoritmo de Análise de Fluxo de Dados:

A linguagem de programação compilada leva sempre a programas estruturados, aos quais corresponde uma classe restrita de grafos de fluxo. Kowaltowski [Kowal84] estuda a solução de problemas de análise de fluxo para o caso de programas estruturados, e introduz os algoritmos utilizados neste trabalho. Rosen [Ros77] também estudou estes mecanismos de análise de fluxo. Silva [Silva84] estuda e compara alguns dos mecanismos de análise de fluxo mais gerais.

A figura 1.3-3 descreve o algoritmo utilizado para efetuar a análise de fluxo para o caso regressivo e disjuntivo, que é o caso do problema de vida de variáveis examinado anteriormente.

```
procedimento Análise_Regressiva_Crescente( B, y );

    procedimento Φ(B);
    caso:
        B é básico:  Φ ← [B.k, B.c];

        B = (B1 ; B2): { k1, c1 ← Φ(B1); k2, c2 ← Φ(B2);
                         B.k, B.c ← [k1 ∧ k2 ; k1 ∧ c2 ∨ c1];
                         Φ ← [B.k, B.c] };

        B = (B1 // B2): { k1, c1 ← Φ(B1); k2, c2 ← Φ(B2);
                          B.k, B.c ← [k1 ∨ k2 ; c1 ∨ c2];
                          Φ ← [B.k, B.c] };

        B é repetitivo { (B1 ; (B2 ; B1)*):
                         { k1, c1 ← Φ(B1); k2, c2 ← Φ(B2);
                           B.k, B.c ← [k1 ; (k1 ∧ c2) ∨ c1];
                           Φ ← [B.k, B.c] } };
    fim Φ;

    procedimento ARC (B, y);
    B.y ← y ;   B.x ← y ∧ B.k ∨ B.c ;
    caso :
        B é básico: { nada ;
        B = (B1 ; B2): { ARC (B2, y);  ARC (B1, B2.x) };

        B = (B1 // B2): { ARC (B1, y);  ARC (B2, y) };

        B é repetitivo { (B1 ; (B2 ; B1)*):
                         { ARC (B2, B.x);  ARC (B1, B2.x ∨ y) };
    fim ARC;

k, c ← Φ (B);
ARC (B, y)
fim Análise_Regressiva_Crescente.
```

fig. 1.3-3

2. OTIMIZAÇÃO DA ALOCAÇÃO DE REGISTRADORES:

Em função da extensão da região do programa em que uma alocação de registradores é efetuada, a alocação de registradores pode ser feita de duas maneiras principais: alocação global e alocação local.

Uma alocação local é aquela feita dentro de um bloco básico, enquanto a alocação global de registradores é feita dentro de uma região do programa envolvendo mais que um bloco básico.

A seguir, vamos descrever alguns mecanismos para a alocação de registradores, voltados para o caso de registradores homogêneos. Conjuntos de registradores não homogêneos impõem restrições adicionais na escolha de registradores, e dificultam a utilização de mecanismos sistemáticos para a alocação.

2.1 ALOCAÇÃO LOCAL:

A alocação local é atrativa pois um compilador pode particionar um programa em blocos básicos com relativa facilidade, e verificar as relações de interferência entre valores utilizados ou definidos dentro de cada bloco básico. Entretanto, a alocação local não mantém a história da alocação de registradores através das fronteiras dos blocos básicos, e desta maneira aqueles valores que estão vivos ao final de cada bloco básico devem ser transferidos para posições de memória a cada transferência de controle entre blocos.

Ha vários mecanismos de alocação local de registradores:

a)- "alocação sob demanda":

O compilador, à medida que gera código, mantém um histórico de quais registradores estão disponíveis a cada ponto, supondo que no início do bloco básico todos os registradores estarão disponíveis, e "requisita" um registrador cada vez que é necessário. Este registrador é considerado "ocupado" até o ponto, no bloco básico, em que seu conteúdo será utilizado pela última vez. A "requisição" de um registrador pode ser feita seja escolhendo um registrador disponível caso haja algum, seja liberando um registrador "ocupado" e armazenando seu conteúdo em memória. A segunda opção causa a necessidade, a partir daí, recarregar o valor que foi armazenado em memória cada vez que ele for necessário naquele bloco básico.

Há vários mecanismos heurísticos para determinar a escolha do registrador a ser "requisitado", de maneira a escolher aquele registrador cujo conteúdo venha a ser referido poucas vezes subsequentemente à sua armazenagem em memória.

b) - rotulação:

Ignorando a existência de sub-expressões comuns a várias expressões, é possível representar cada expressão como uma árvore onde os nós interiores representam os operadores, e as folhas representam as variáveis (ou constantes) utilizadas para o cálculo da expressão. Existe um algoritmo simples ([AhoUll77], pag.540) que determina em tempo proporcional ao número de nós da árvore a ordem ótima (i.e, aquela que utiliza o menor número de registradores) para o cálculo desta expressão. No caso de expressões com subexpressões comuns (DAG's), as subexpressões serão recalculadas [Fig. 2.1-1].

O algoritmo tem duas partes: a primeira delas rotula cada nó da árvore com um inteiro que denota o menor número de registradores necessário para calcular a expressão representada pela subárvore que começa naquele nó sem armazenar resultados intermediários em memória, e a segunda parte percorre a árvore gerando código e atribuindo registradores à medida que são necessários, numa ordem de acordo com os rótulos atribuídos a cada nó.

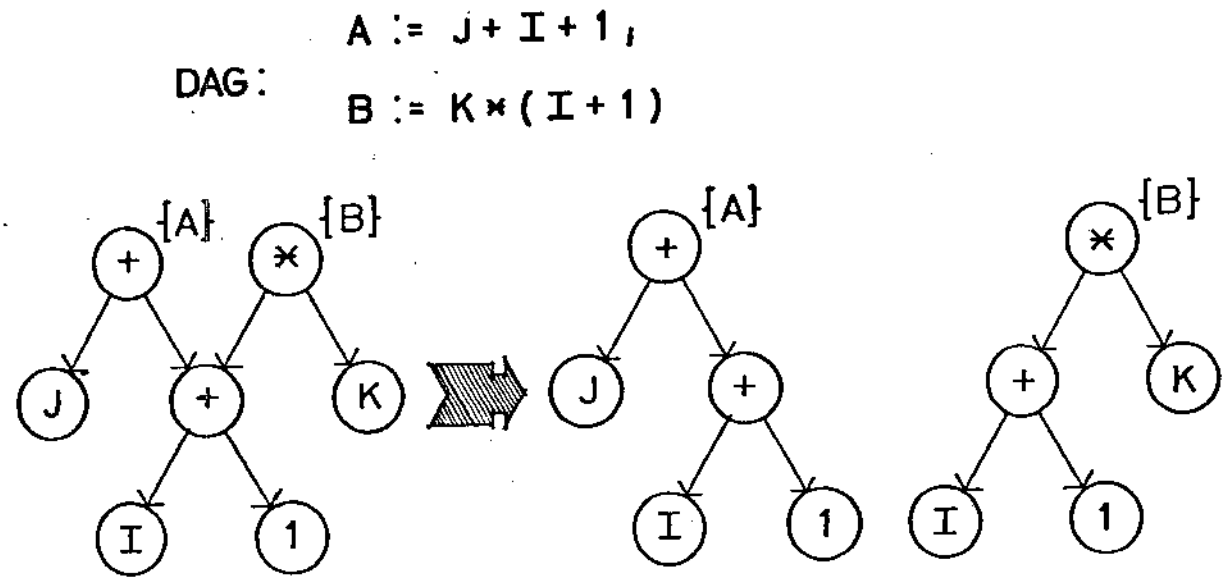


fig. 2.1-1

2.2 ALOCAÇÃO GLOBAL:

São conhecidos vários mecanismos para alocação global de registradores. Um mecanismo bastante comum utiliza a contagem da quantidade de usos como critério de prioridade de alocação de uma variável a um registrador (em inglês, "usage counts") [Freib74]. A idéia consiste em utilizar a quantidade de referências (estáticas) ao resultado de cada computação como um indicador de prioridade para alocar aquele resultado a um registrador.

Day [Day70] apresenta e discute a eficiência de um algoritmo para a alocação global de registradores utilizando técnicas de programação inteira, e fornece referências bibliográficas sobre o assunto. Segundo Day, associa-se um "lucro" a cada computação no programa. Para cada computação, este "lucro" estima a melhoria no tempo de execução do

programa que será obtida caso aquela computação seja armazenada em um registrador. A seguir, maximiza-se a função de cálculo do lucro total, sujeita à restrição do número máximo de registradores que podem estar associados com algum valor em cada ponto do programa. Day propõe e analisa a eficiência de vários algoritmos, apresenta um algoritmo (custoso em tempo de execução) que descobre sempre a solução ótima, e apresenta alguns algoritmos mais eficientes que obtém soluções próximas à solução ótima.

O algoritmo utilizado por Chaitin efetua uma alocação global de registradores, e está descrito a seguir.

2.3 DESCRIÇÃO DO ALGORITMO UTILIZADO POR CHAITIN:

Chaitin implementou a fase de alocação de registradores de um compilador PL-I para a máquina experimental IBM-801 [Chait81, Chait82], que é uma máquina com 32 registradores homogêneos.

Esta fase trabalha sobre uma representação do programa em linguagem intermediária (LI) semelhante à linguagem de máquina, supondo a existência de um número ilimitado de registradores ("registradores simbólicos").

A alocação de registradores consiste em mapear o conjunto com um grande número de registradores suposto na descrição do programa em LI no conjunto real de 32 registradores existentes na máquina alvo. Para fazer isto, às vezes torna-se necessário adicionar, ao programa descrito em LI, código para armazenar em memória alguns valores computados em registradores simbólicos e código para mais tarde recarregá-los.

O primeiro passo no processamento do programa consiste em utilizar técnicas de análise global de fluxo para conhecer, a cada ponto no programa descrito em LI, qual o conjunto de registradores simbólicos vivos naquele ponto.

A seguir, é construído o grafo de interferências entre registradores. Este grafo contém um vértice para cada registrador simbólico utilizado na descrição do programa em LI. Dois vértices são adjacentes, i.e., existe uma aresta entre eles, quando os dois registradores simbólicos que eles representam interferem entre si. Dois registradores interferem entre si quando um deles está vivo em algum ponto de definição do outro, ou seja, em algum ponto onde o outro é carregado com algum valor.

Nesse ponto, conforme citamos, a alocação de

registradores simbólicos aos registradores reais é feita através da coloração do grafo de interferências, considerando como "cores" os registradores reais existentes na máquina alvo.

Conforme dissemos, uma coloração de um grafo é a atribuição de uma cor a cada um de seus vértices de tal maneira que, se dois vértices são adjacentes, então eles tem cores diferentes. Uma coloração de um grafo é uma N-coloração se ela não utiliza mais do que N cores diferentes. O número cromático de um grafo é o menor N para o qual existe uma N-coloração daquele grafo.

Após a coloração do grafo de interferências, se um dado registrador simbólico foi colorido com a cor correspondente a um registrador real da máquina isto significa que o registrador simbólico foi atribuído àquele registrador.

Então, para Chaitin, uma 32-coloração do grafo de interferências corresponde a uma alocação de registradores aceitável. Quando o número cromático do grafo de interferências for maior que 32, é necessário adicionar código ao programa em LI para armazenar em memória certos valores computados, e para recarregar estes valores em algum registrador sempre que estes valores sejam utilizados como operando em alguma instrução.

O problema de coloração de grafos quaisquer é computacionalmente difícil (NP-completo) [Karp72, LSSSK79] e, em seu artigo, Chaitin demonstra que, para qualquer grafo G, podemos sintetizar um programa cujo grafo de interferências seja idêntico a G.

O algoritmo heurístico utilizado por Chaitin para fazer a coloração do grafo de interferências baseia-se na seguinte observação: suponhamos que se deseje efetuar uma

N-coloração de um grafo G que contém um vértice V cujo grau é menor que N. Então, G é N-colorível se e sómente se o grafo G', que é o grafo obtido a partir de G eliminando-se V e todas suas arestas, for N-colorível. O algoritmo consiste então em eliminar sucessivamente do grafo, vértices com grau menor que N. Na maioria dos casos práticos, isto acontece de tal forma que todos os vértices do grafo são eliminados, i. é, até que o problema original de N-colorir o grafo de interferências reduz-se ao problema trivial de N-colorir o grafo vazio. Então, os vértices são adicionados ao grafo na ordem inversa em que foram retirados, e á medida que são colocados de volta ao grafo, é escolhida uma cor para eles. Ao colocar um vértice de volta ao grafo, sabendo que seu grau será menor que N, basta escolher para ele uma cor que não tenha sido já atribuída a nenhum dos vértices adjacentes a ele. Como sabemos que o grau deste vértice é menor que N, no máximo haverá N-1 cores distintas já atribuídas aos vértices adjacentes, e ainda restará uma cor a atribuir àquele vértice, dentre as N cores disponíveis.

Este algoritmo pode ser implementado de forma a ter um tempo de execução proporcional ao tamanho do grafo. Experimentalmente, para Chaitin, este algoritmo produziu excelentes resultados, já que a grande maioria dos grafos de interferência obtidos ao compilar programas reais era fácilmente 32-colorível utilizando este algoritmo.

O algoritmo utilizado por Chaitin pode falhar em obter uma N-coloração apenas quando, em algum ponto, ele alcança um grafo G' onde todos os vértices possuem grau N ou maior. Neste caso, o procedimento adotado por Chaitin é escolher um vértice, retirá-lo do grafo, e continuar a coloração. Nesse momento, Chaitin escolhe para retirar do grafo aquele vértice que possua o maior grau, visando desta maneira diminuir o grau de um grande número de outros vértices e assim diminuir a probabilidade do algoritmo falhar novamente. Ao recolocar o vértice escolhido no grafo, caso

seja constatado que o número de cores atribuídas aos vértices adjacentes seja igual a N, isto significa que aquele vértice corresponde a um valor que será sempre armazenado em uma posição de memória cada vez que é calculado, e re-carregado em algum registrador simbólico antes de ser utilizado. A descrição do programa em LI é alterada de maneira a refletir este fato. Entretanto, como foi necessário "criar" um novo registrador simbólico, torna-se necessário repetir o processo de alocação dos registradores, para levar em conta a existência deste novo registrador.

O procedimento todo é então repetido, desde a reconstrução do grafo de interferências até a coloração do grafo. Este procedimento é repetido sucessivamente até que na coloração do grafo não tenha sido necessário escolher valores a serem armazenados em memória. Segundo Chaitin, não é preciso um número muito grande de iterações até que isto aconteça. Na maioria dos casos, duas ou três iterações serão suficientes.

2.4 OUTRAS EXPERIÊNCIAS COM COLORAÇÃO:

Fabri [Fab79] estende a idéia proposta por Chaitin para otimizar a utilização de memória. Sua intenção é fazer várias variáveis ou vetores compartilharem as mesmas posições de memória. Isto é possível quando estas variáveis não "interferem" entre si. Para vetores, Fabri estende o conceito de interferências de maneira a levar em conta que, em um programa, podem acontecer referências localizadas a trechos de vetores e que é possível dividir um vetor em trechos que podem compartilhar posições de memória com outras variáveis ou outros trechos de vetores que não interfiram com ele.

Chow e Hennessy [ChowHenn84] alteram o algoritmo proposto por Chaitin associando uma prioridade para a coloração de cada vértice baseando-se em estimativas do benefício que poderá ser obtido ao alocar a variável associada ao vértice em um registrador. Ao alocar uma variável a um registrador, é suposto inicialmente que cada variável esteja associada com uma posição de memória. Condiderando o programa dividido em trechos de código cada um não maior que um bloco básico, será necessário carregar o valor da variável em um registrador antes de referir-se à variável como estando associada àquele registrador no trecho de código subsequente. Caso a variável seja alterada enquanto "reside" em um registrador, a posição de memória associada àquela variável deve ser atualizada à saída daquele trecho de programa, a menos que esta não esteja mais viva.

Levando em conta os custos de:

- carregar o conteúdo de uma posição de memória em um registrador;

- diferença de tempo de execução de uma operação quando seu operando está em memória e quando está em um registrador;

- diferença de tempo de execução na definição de cada variável quando reside em registrador e quando reside em memória

é computado o benefício obtido pela alocação de cada variável a um registrador, em função da quantidade de definições e de usos daquela variável no trecho considerado. Chow e Hennessy utilizam este critério de benefícios, também, para fazer previamente uma alocação local de cada variável a um registrador, quando isto é vantajoso. A seguir, é feita a alocação global através da coloração do grafo de interferências, utilizando o critério de prioridade para escolher a ordem de retirada de cada vértice do grafo, na coloração utilizando o algoritmo proposto por Chaitin.

3. DESCRIÇÃO DO COMPILADOR IMPLEMENTADO:

Nossa intenção é particionar o problema de coloração do grafo de interferências, utilizado para fazer a alocação global de registradores, numa série de sub-problemas de coloração, visando obter grafos menores a colorir. Desta maneira, pode tornar-se prática até a utilização eventual de algum algoritmo de busca exaustiva para fazer a coloração.

Cada bloco (condicional, repetitivo, sequência de blocos ou bloco básico) terá associado um grafo de interferências válido quando o fluxo de execução do programa estiver em algum ponto naquele bloco. Os vértices desse grafo de interferências correspondem aos valores que estão vivos à entrada ou à saída do bloco.

A coloração é iniciada pelos grafos associados aos blocos mais englobantes, considerando-se todo o programa como um bloco do tipo sequência de blocos, a seguir colorindo os grafos associados a seus blocos componentes, e assim sucessivamente.

3.1 ORGANIZAÇÃO DO COMPILADOR:

O compilador é dividido em várias fases que se sucedem tratando o programa que está sendo compilado. Estas fases se comunicam modificando ou "decorando" com informações uma estrutura de dados em árvore que representa o programa, chamada de "árvore do programa". Cada fase, que exige um ou mais passos pelo programa, executa uma função específica, seja análise de fluxo, seja transformação do programa original visando um código objeto mais curto, etc.

Este arranjo permite a experimentação com vários mecanismos para a melhoria de código objeto. Por exemplo, a deteção global de sub-expressões comuns, não efetuada, pode ser acrescentada como um passo adicional que transforma a árvore do programa de maneira a utilizar um único nó da árvore para representar aquelas expressões idênticas que são utilizadas em várias computações. Isto pode ser feito de maneira a não afetar os demais passos.

Em um compilador para uso frequente, é possível e desejável fundir-se vários passos, executando ao mesmo tempo várias destas funções, obtendo assim um compilador mais eficiente. Esta é a técnica utilizada no compilador para BLISS-11 [Wul75], para o PDP-11. Em nosso compilador, mantivemos estanque e explícita a independência entre cada uma das fases, de maneira a tornar mais fácil a experimentação.

As fases, descritas a seguir, são:

a - Análise Léxica e Sintática

b - Montagem da Árvore do Programa

c - Análise de Fluxo - propagação de definições

d - Transformação de variáveis em nomes

e - Análise de Fluxo - nomes vivos

f - Escolha da Ordem de Execução

g - Construção dos Grafos de Interferência

h - Coloração dos Grafos de Interferência

3.2 ANÁLISE LÉXICA E SINTÁTICA:

Esta fase produz, a partir da leitura do texto fonte, uma estrutura de dados que representa a árvore sintática do programa. A análise sintática é feita de forma descendente recursiva.

Durante a leitura do texto fonte é emitida a listagem e é feita também a análise da correcão do programa.

A figura 3.2-1 ilustra a árvore sintática montada a partir de um trecho de programa fonte.

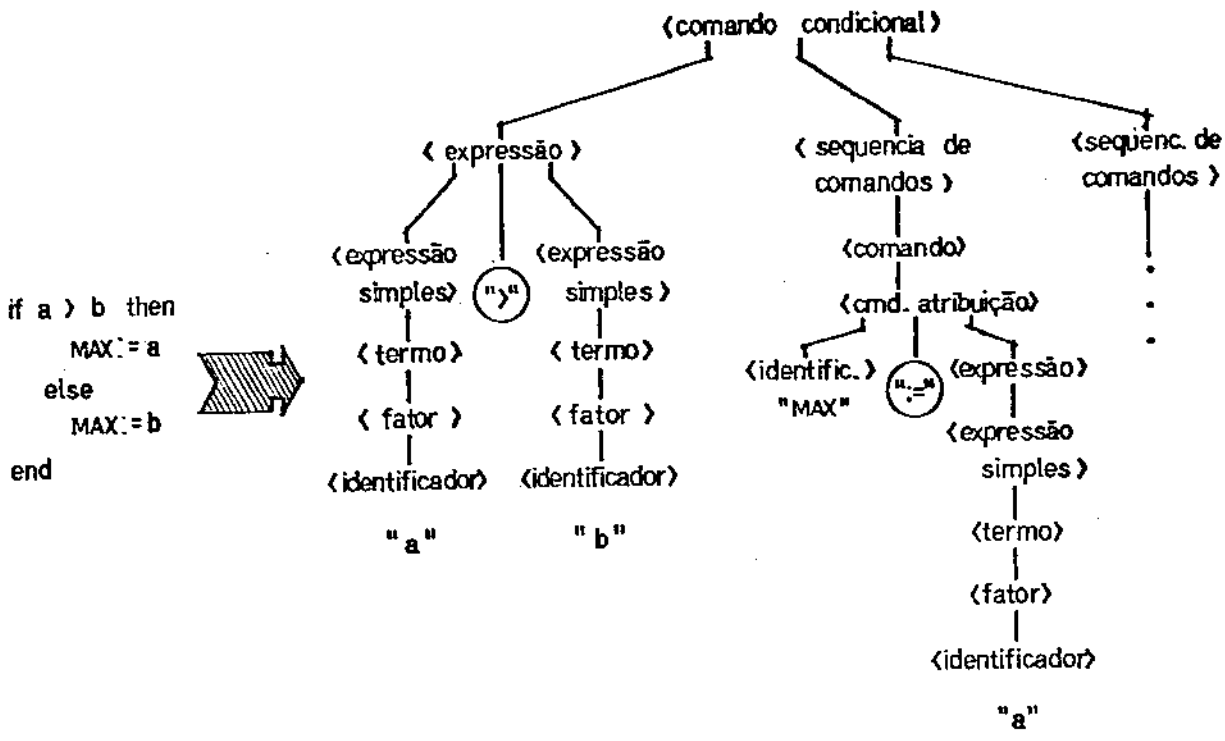


fig. 3.2-1_Análise Sintática

3.3 MONTAGEM DA ÁRVORE DO PROGRAMA:

A partir da árvore sintática, produz-se a árvore do programa, cujos nós são blocos de execução: bloco condicional, bloco repetitivo, bloco "sequência linear de blocos" ou bloco básico.

Esta árvore é consequência do fato do programa ser estruturado, gerando um sistema de "blocos encaixados". Por exemplo, um bloco condicional é utilizado para representar a execução alternativa de dois outros blocos: um bloco repetitivo e um bloco básico. Estes dois outros blocos são representados na árvore como "filhos" daquele nó que representa o bloco condicional. Blocos básicos correspondem às "folhas" da árvore de programa, pois eles representam trechos lineares onde todas instruções no bloco são sempre executadas sequencialmente, iniciando pela primeira instrução no bloco, e terminando pela última [Fig 3.3-1].

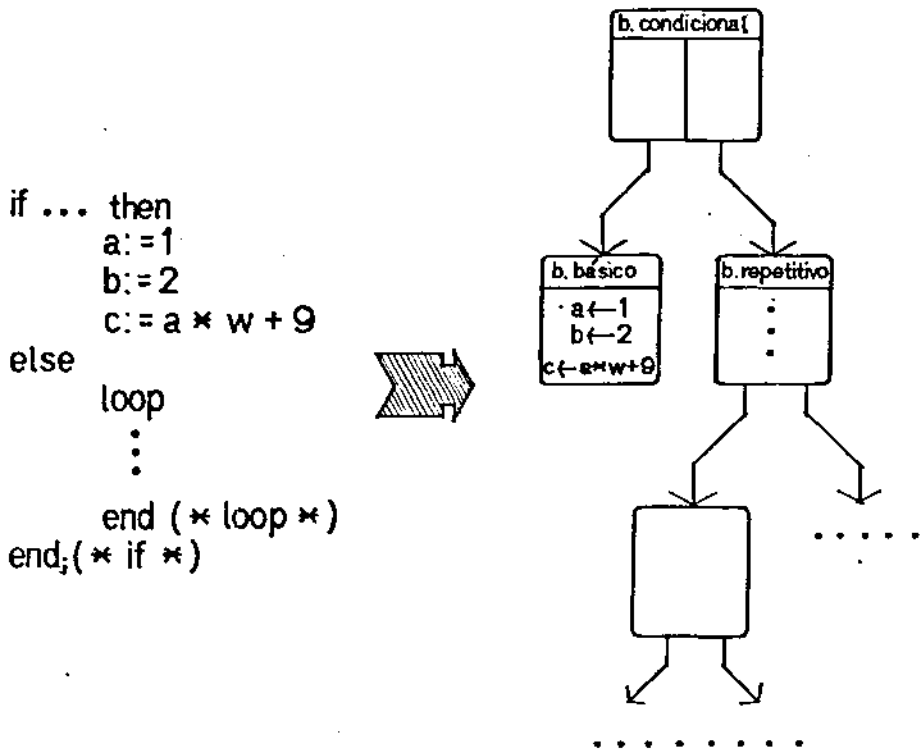


fig 3.3-1

A árvore do programa representa a estrutura de execução do programa (grafo de fluxo) através dos blocos de execução.

3.3.1 Descrição dos Blocos de Execução:

Conforme dissemos em 1.3, os blocos de execução podem ser: bloco condicional, bloco repetitivo, sequência de blocos ou blocos básicos.

a - Bloco condicional e Bloco repetitivo:

Estes blocos são representados por nós da árvore que contém apontadores para dois outros blocos "componentes".

b - Sequência de blocos:

Uma sequência de blocos é representada por uma lista ligada de blocos.

Note-se que um bloco do tipo "sequência de blocos" sempre pode ser considerado como sendo composto por um par de blocos, seja um par de blocos dos tipos básico, condicional ou repetitivo, ou como um bloco de um destes tipos seguido por um bloco do tipo "sequência de blocos".

c - Blocos básicos:

Nesta implementação, os blocos básicos são representados por grafos dirigidos acíclicos (em inglês DAG's, "directed acyclic graphs"), que são estruturas que fornecem uma "fotografia" de como valores são computados dentro do bloco, e subsequentemente reusados naquele bloco.

No DAG que representa as computações efetuadas num bloco básico:

- as folhas são rotuladas com identificadores únicos, correspondentes a variáveis ou a constantes;

- os nós interiores são rotulados com operadores: adição, multiplicação, desreferenciação, etc.

Além disto, os nós interiores podem ter associados um conjunto de identificadores ("nomes").

Nós interiores de um DAG representam valores computados, e os identificadores associados a um dado nó possuem aquele valor. [Fig. 3.3-2]

Representando os blocos básicos por DAG's, é simples determinar que valores definidos fora do bloco básico são utilizados naquele bloco básico (correspondem às "fôlhas"), e determinar que operações naquele bloco básico geram valores que poderiam ser utilizados fora do bloco (são os identificadores associados aos nós interiores).

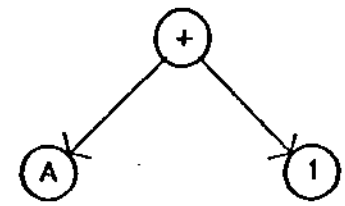
Note-se que a representação de blocos básicos utilizando DAG's já permite também detetar automaticamente a ocorrência de sub-expressões comuns a várias computações dentro daquele bloco básico [Figura 3.3-3]. Isto é feito nesta implementação.

3.3.2 Estruturas de Dados:

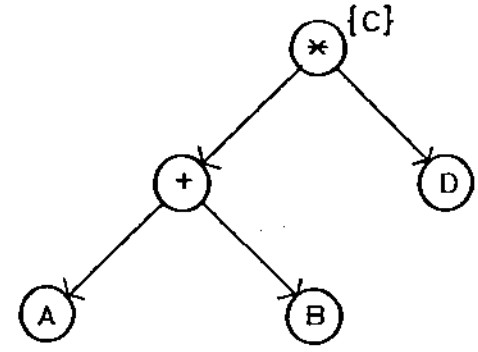
Para ilustração, vamos apresentar no apendice II as declarações em Pascal das estruturas de dados utilizadas para a representação dos blocos.

3.3.3 Árvore do programa:

a) DAG representando A + 1



b) DAG representando C := (A+B) * D



c) DAG representando

```
I := I+1;
J := I * 2;
K := J
```

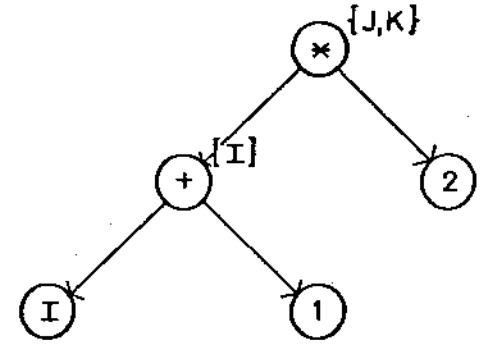


fig. 3.3-2_DAGs

Na estrutura descrita, o programa objeto final é formado pela sequência de blocos básicos obtida quando percorremos as folhas da árvore do programa em pré-ordem [Figura 3.3-4], adicionando instruções de desvio quando necessárias, correspondentes aos nós interiores da árvore do programa (bloco condicional, ou bloco repetitivo).

A "hierarquia" de cada nó na árvore vai informar

```
A := (I + 3) * 2;

B := (I + 3) + C;
```

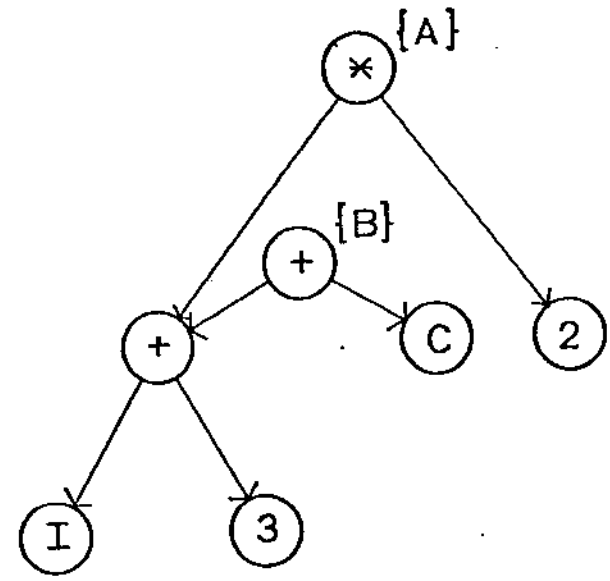
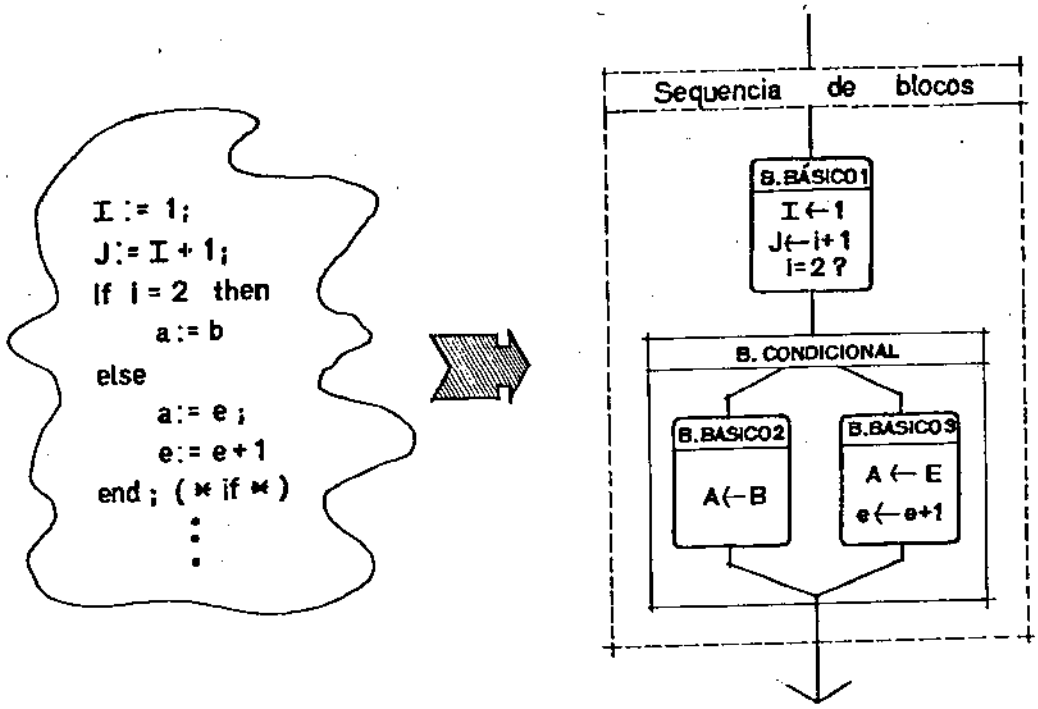


fig. 3.3-3

sobre as relações de encaixamento entre os blocos: informa que blocos são componentes de um determinado bloco, e de que blocos um dado bloco é componente. Esta informação é utilizada, por exemplo, para efetuar a alocação de registradores de maneira a permitir que utilizando informações de fluxo do programa expressas na árvore, se faça a alocação de maneira local a cada bloco, visando obter maior liberdade na atribuição destes registradores, e obter grafos com um menor número de nós a colorir.


I := 1;
J := I + 1;
If i = 2 then
a := b
else
a := e ;
e := e + 1
end ; (* if *)
Sequencia de blocos
B.BÁSICO1
I ← 1
J ← i + 1
i = 2 ?
B. CONDICIONAL
B.BÁSICO2
A ← B
B.BÁSICO3
A ← E
e ← e+1

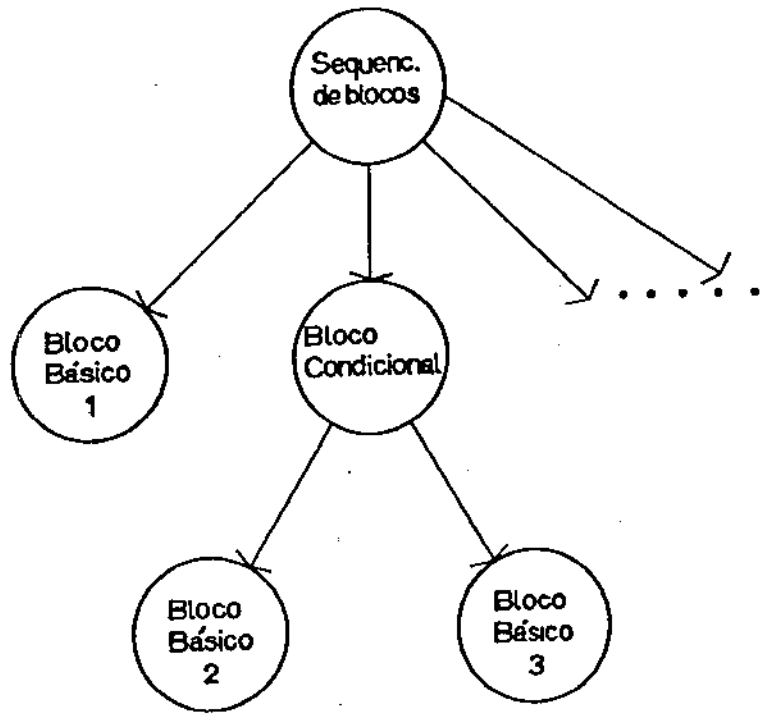

Sequenc. de blocos
Bloco Básico 1
Bloco Condicional
Bloco Básico 2
Bloco Básico 3

fig. 3.3-4

3.4 ANÁLISE DE FLUXO: PROPAGAÇÃO DE DEFINIÇÕES:

Uma definição "d: A:=expressão" de uma variável A é o ponto no programa onde é atribuído um valor àquela variável. Uma "definição" é, em geral, da forma

A := B "op" C

ou

A := "op" B.

Um uso de uma variável A é um ponto no programa onde esta variável é utilizada como argumento de alguma operação.

Dizemos que uma definição d da variável A alcança um ponto p no programa se existe um caminho de d até p tal que não há outras definições da variável A neste caminho.

Esta fase resolve um problema de análise de fluxo, de forma a conhecer, para cada ponto no programa, quais são as definições que alcançam aquele ponto. Os resultados obtidos nesta fase são utilizados na fase seguinte, que vai relacionar entre si todas as definições de uma variável que alcançam um determinado uso daquela variável.

3.4.1 Propagação de Definições:

Considerando o bloco básico B: o conjunto de definições que alcançam o final do bloco B são:

- aquelas definições em B que definem nomes que não são posteriormente redefinidos em B;

- aquelas definições que alcançam o início de B, e que definem nomes que não são definidos em B.

As definições que alcançam o início de B são aquelas definições que alcançam o final de algum dos blocos predecessores de B.

Este problema pode ser expresso como um problema de análise de fluxo disjuntiva e progressiva. Disjuntiva, pois o conjunto de definições que alcançam o início de um bloco é encontrado fazendo-se a união (i.e, a disjunção) dos conjuntos de definições que alcançam o final de algum dos blocos predecessores de B, e progressiva pois o conjunto de definições que alcançam o início de cada bloco B depende do conjunto de definições que alcançam o final dos predecessores de B.

Existe no início do programa um conjunto vazio de definições que alcançam aquele ponto, e esta condição é propagada adiante pelo grafo de fluxo do programa.

Sendo Ci o conjunto de definições em Bi que definem nomes que não são posteriormente redefinidos em Bi,

Ki o conjunto de definições fora de Bi que definem nomes que não são definidos em Bi,

Yi o conjunto de definições que alcançam o início do bloco Bi,

Xi o conjunto de definições que alcançam o final do bloco Bi,

e sendo Pi o conjunto de blocos básicos predecessores de Bi, o problema da propagação de definições é expresso por:

$$Xi = Yi \wedge Ki \cup Ci$$

e

$$Yi = \bigcup_{j \text{ em } Pi} Xj$$

3.4.2 Propagação de Definições em Programas Estruturados:

O problema da propagação de definições é resolvido de maneira semelhante ao algoritmo introduzido em 1.3. Associamos coeficientes K e C "equivalentes" para cada bloco B no programa, a partir dos coeficientes Ki e Ci de seus blocos básicos componentes Bi. Procedendo indutivamente desta maneira, calculamos coeficientes K e C para todos os blocos no programa. Ao considerarmos o programa todo como um bloco tipo "sequência de blocos" e conhecendo seus coeficientes K e C, é possível encontrar a solução Xs, que é o conjunto de definições disponíveis no final do programa, uma vez que a condição de contorno inicial Ys do conjunto (vazio) de definições disponíveis no início do programa é conhecida. Conhecendo X e Y para cada bloco, é possível de maneira análoga calcular Xi e Yi resultantes em cada um de seus blocos componentes Bi, resolvendo o problema de análise de fluxo.

3.5 TRANSFORMAÇÃO DE VARIÁVEIS EM NOMES:

Esta fase faz o equivalente a uma "troca de nomes" de variáveis, utilizando informação obtida na fase de propagação de definições.

Cada variável definida pelo programador é tratada como se fosse separada em várias novas "variáveis" nas diversas instâncias (uso ou definição) onde aparece, de tal forma que, em tempo de execução, as sequências de definições e usos de cada uma destas novas "variáveis" sejam completamente disjuntas entre si.

Isto vai permitir que se faça automaticamente a alocação de uma variável definida pelo programador ora a um registrador, ora a outro, pois cada uma destas "variáveis" vai ser tratada de maneira independente pelo algoritmo de alocação de registradores. Neste trabalho, estas novas "variáveis" são chamadas de "nomes".

Por exemplo, se numa sequência de comandos de atribuição uma determinada variável A recebe algum valor, é a seguir utilizada como argumento no cálculo de uma expressão, recebe então um novo valor e é novamente utilizada no cálculo de outra expressão, podemos separar A em duas novas "variáveis": N1 e N2, sendo N1 utilizada para receber o primeiro valor anteriormente atribuído a A e utilizada como argumento no cálculo da primeira expressão, e N2 utilizada na atribuição e cálculo seguintes [Fig.3.5-1(a)].

Para fazer a separação de variáveis em nomes:

1)- inicialmente separamos em classes de equivalência todas ocorrências de uma variável (definição ou uso), considerando que cada ocorrência faça referência a um nome diferente;

2)- para todas as variáveis V no programa fazemos:

3)- para todos os usos u de V fazemos:

4)- são agrupadas sob a mesma classe de equivalência (um mesmo "nome") o nome associado a u e os nomes associados a todas as definições de V que alcancem u.

Em outras palavras, para cada variável, o processo acima consiste em, considerando inicialmente cada atribuição de valor àquela variável (cada definição daquela variável) como se estivesse criando um nome diferente, "fundir" sob um mesmo nome (i.e, uma classe de equivalência) o conjunto de definições que alcancem cada uso daquela variável [Fig. 3.5-1(b)]. Estas definições da variável também irão alcançar um conjunto de outros usos da mesma variável. A "fusão" de definições que alcancem cada uso é então repetida em cada um destes usos, efetuando uma "propagação" daquele nome por toda a cadeia de definições-usos daquela variável.

Cada nome vai identificar univocamente uma cadeia conexa de definições e usos que apareçam mencionadas no processo acima. Este processo é executado percorrendo-se uma única vez a árvore do programa.

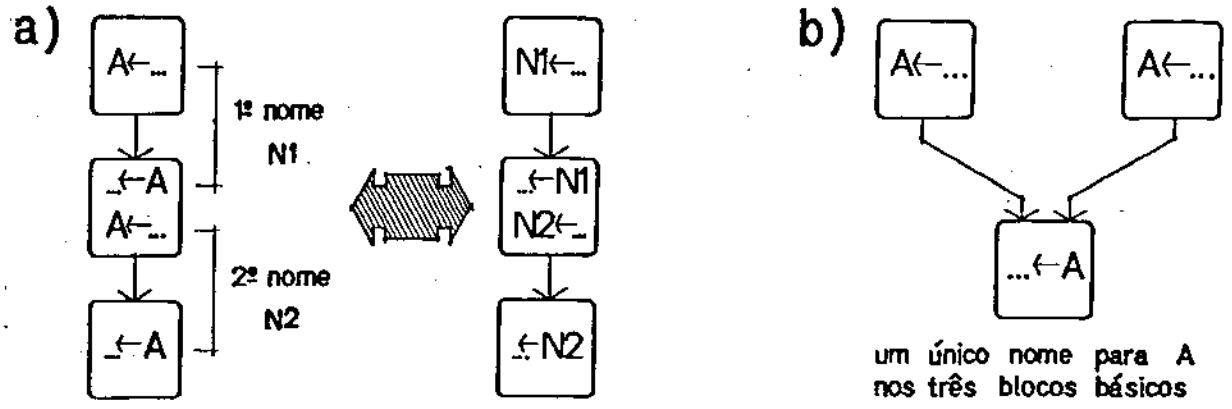

a)
A←...
1º nome
N1
...←A
A←...
2º nome
N2
...←A
N1←...
...←N1
N2←...
...←N2
b)
A←...
A←...
...←A
um único nome para A
nos três blocos básicos

fig. 3.5-1

3.6 ANÁLISE DE FLUXO: NOMES VIVOS:

Analogamente ao caso para variáveis, dizemos que um nome N está vivo em um ponto p do programa se existe um caminho começando em p até um uso de N, sem que haja atribuições de valor (i.e, definições) a N neste caminho.

A alocação (e liberação) de nomes a registradores, é feita conhecendo quais são os nomes que estão vivos em um instante do programa.

O problema de "vida de nomes", já introduzido em 1.3, é resolvido da maneira indicada naquele item.

3.7 ESCOLHA DA ORDEM DE EXECUÇÃO:

Como dissemos, nesta implementação os blocos básicos são representados por DAG's. Para determinar as relações de interferência existentes entre os nomes que estejam vivos à entrada ou à saída de cada bloco básico, é necessário que seja feita previamente uma escolha da ordem de avaliação das expressões representadas pelo DAG. Esta ordem vai dizer também quantos, e quais, resultados intermediários deverão ser armazenados em registradores, para serem posteriormente utilizados no mesmo bloco básico para o cálculo de outras expressões.

Podemos verificar facilmente que a escolha da ordem de execução afeta a alocação de registradores supondo, por exemplo, que exista um determinado valor que esteja vivo à entrada de um bloco básico, seja utilizado uma única vez como argumento naquele bloco, e não esteja vivo à saída. Neste caso, o registrador que inicialmente contém aquele valor poderá ser considerado disponível para armazenar outros valores assim que seu conteúdo for utilizado. Caso a ordem de execução naquele bloco básico seja tal que este valor seja utilizado no calculo da última expressão naquele bloco básico, o registrador permanecerá ocupado durante todo o bloco, ao passo que este registrador poderá ser disponível já logo após o início do bloco caso a expressão onde seu conteúdo é inicialmente necessário seja das primeiras a serem calculadas no bloco básico.

O problema da escolha da ordem ótima de cálculo de um DAG, que contém sub-expressões comuns, é também um problema computacionalmente muito difícil [Bruno-Sethi76]. Wulf [Wul75] apresenta um algoritmo heurístico para a escolha da ordem de cálculo, baseado na solução deste problema para o caso de árvores. Nesta implementação, utilizamos um algoritmo heurístico sugerido por Aho e Ullman [AhoUll77].

Deve-se notar, entretanto, que uma vez definida a ordem de execução dentro de cada bloco básico, o algoritmo para a alocação de registradores sendo descrito pode funcionar de maneira independente. Isto segue do fato que esta escolha da ordem de execução está efetuando implicitamente uma alocação de registradores para conter os valores intermediários que sejam necessários dentro do bloco, e definindo as relações de interferência entre os nomes e os valores temporários.

3.8 CONSTRUÇÃO DOS GRAFOS DE INTERFERÊNCIA:

Existirá, associado a cada bloco no programa, um grafo de relações de interferência entre nomes que pertencem ao conjunto de nomes vivos na entrada ou na saída do bloco. Cada nome é representado por um vértice no grafo, e quando dois nomes "interferem" entre si, existe no grafo uma aresta entre seus vértices correspondentes.

Quando, no grafo correspondente a um bloco B um determinado nome N1 "interfere" com o nome N2, isto significa que: dentro do bloco B, N1 não pode ser alocado ao mesmo registrador que N2, pois existe em B uma situação em que os valores (possivelmente distintos) de N1 e N2 serão necessários.

Cada nome N interfere com todos os nomes que estejam vivos em algum instante de definição de N, ou seja, em um ponto no programa onde é atribuído algum valor a N.

Para construir os grafos de interferência, percorremos cada um dos blocos básicos na ordem de sua execução. Ao encontrar uma definição de um nome N, é criada uma aresta de interferência entre N e cada elemento V do conjunto de nomes vivos naquele instante. Esta aresta é anotada em todos os grafos associados aos blocos onde o par (N, V) estiver presente.

O conjunto dos nomes vivos em um instante qualquer da execução de um bloco básico é obtido partindo-se do conjunto de nomes vivos na entrada do bloco, obtido na fase de "análise de fluxo: nomes vivos". Ao percorrer o bloco básico representado por um DAG, na ordem de sua execução, retira-se

do conjunto de nomes vivos aqueles nomes utilizados como argumento em cada nó do DAG que vai sendo visitado que não serão posteriomente reutilizados no bloco básico nem estão vivos à saída do bloco, e acrescenta-se ao conjunto os nomes definidos por aquele nó do DAG que estejam vivos à saída do bloco básico. Os valores intermediários no cálculo de expressões não são colocados no grafo de interferências. Da mesma maneira, os nomes que não estão vivos nem à entrada nem à saída do bloco são tratados como valores temporários, e não são alocados a registradores utilizando este mecanismo global de alocação.

3.9 COLORAÇÃO DOS GRAFOS DE INTERFERÊNCIA:

Após a fase anterior, a árvore de programa apresenta um sistema de "blocos encaixados", cada bloco com seu grafo de interferências associado. Nos níveis mais altos da árvore estão os blocos mais externos, de escopo mais amplo no programa, e à medida que se caminha em direção às folhas são encontrados blocos cujo escopo se restringe a regiões menores do programa.

Como já foi mencionado, para fazer a alocação em registradores dos nomes vivos à entrada ou saída de um bloco, consideramos cada registrador da máquina objeto como se fosse uma cor, e é feita uma coloração do grafo de interferências associado ao bloco. Assim, cada nome será associado a um registrador, no interior daquele bloco.

São feitas restrições à atribuição de "cores" aos nomes: por exemplo, em dois blocos consecutivos onde um determinado nome está vivo à saída do primeiro e à entrada do segundo, é importante que este nome receba a mesma cor (i.e, registrador) ao se colorir os grafos associados ao primeiro e segundo blocos.

Durante a coloração, a estratégia utilizada é percorrer a árvore do programa colorindo primeiro os grafos associados aos blocos mais externos, daí refletir as informaçoes obtidas nesta coloração para os grafos associados a seus blocos componentes, e assim por diante até atingir os blocos básicos. Ao colorir o grafo associado a um bloco, isto se reflete no fato de se encontrar o grafo já "pré-colorido": alguns nomes já estarão associados a registradores, por serem nomes de escopo mais global no programa.

4. EXPERIMENTAÇÃO:

O compilador implementado consiste em aproximadamente três mil linhas em Pascal, sendo:

Análise léxica e sintática	430 linhas
Montagem da árvore do programa	380 linhas
Análise de Fluxo	
- propagação de definições	250 linhas
Separação de variáveis em nomes	100 linhas
Análise de Fluxo	
- nomes vivos	160 linhas
Escolha da ordem de execução	175 linhas
Construção dos grafos	
de interferência	315 linhas
Coloração dos grafos	
de interferência	80 linhas

(o restante do compilador são declarações e rotinas auxiliares, incluindo rotinas para a impressão da árvore do programa).

A análise sintática é efetuada, conforme dissemos, de maneira descendente recursiva. O tratamento de erros é bastante rudimentar.

4.1 COMPARAÇÃO COM A IMPLEMENTAÇÃO DE CHAITIN:

Para avaliar o método de alocação de registradores foi efetuada também uma implementação do algoritmo descrito por Chaitin em seu artigo. A intenção é verificar se houve maior possibilidade de economia de registradores efetuando a alocação segundo a maneira estruturada que foi descrita, e verificar a quantidade e o tamanho dos grafos a colorir, para avaliar a aplicabilidade de outros algoritmos para a coloração.

Em nossa implementação, diversamente da implementação de Chaitin, os valores locais a um bloco básico não são alocados a registradores usando o mecanismo geral de coloração. Existem maneiras mais eficientes de fazer a alocação de registradores em trechos lineares de programa. O grafo de inteferências oriundo de um trecho linear de programa pertence a uma classe (grafos de intervalo) para a qual existe um algoritmo que encontra uma coloração em tempo proporcional ao tamanho do grafo.

4.2 RESULTADOS EXPERIMENTAIS:

Os dados experimentais foram obtidos considerando alguns programas em PASCAL que foram traduzidos "à mão" para serem aceitos pelo compilador experimental. O texto fonte dos programas de teste se encontra no Apendice IV.

A tabela 4.2-1 sumariza os resultados obtidos por esta implementação comparando-os com os resultados obtidos pela implementação segundo Chaitin.

A comparação é efetuada considerando a quantidade máxima de registradores necessária e o tamanho dos grafos a colorir.

	COMPIL. EXPERIMENTAL										CHAITIN		
	TAMANHO DOS GRAFOS (nº de vértices)										TAM. GRAFO		
	8	7	6	5	4	3	2	1	Σ	nº total de cores	SÓ NOMES	TOTAL	nº de cores
P1	1								8	8	8	14	8
P2				1				5	10	7	10	20	7
P3				1					5	5	5	12	5
P4				1					5	5	5	22	5
P5						2			6	3	6	8	3
P6						1	2	1	8	5	8	12	5
P7						1	1		5	4	5	14	4
P8							2		4	2	4	4	2
P9							1	1	3	3	3	4	3

Tabela 4.2-1

As colunas "Tamanho dos grafos" apresentam o número de vértices dos grafos obtidos para cada programa de teste, e permitem comparar, observando a coluna "Tamanho dos grafos-Chaitin", a complexidade dos grafos oferecidos ao algoritmo de coloração. Para a implementação de Chaitin, a quantidade de vértices do grafo é fornecida na coluna "Tamanho total", enquanto a coluna "Só nomes" apresenta a quantidade de vértices que representavam nomes (ou seja, excluindo valores temporários) no grafo segundo Chaitin.

Note-se, também, a quantidade de vértices que representam valores temporários que são colocados no grafo na implementação segundo Chaitin. Em nossas experiencias, essa implementação falhou por duas vezes em obter a coloração ótima para estes valores temporários, ocasionando a necessidade de mais um registrador. Conforme citamos em 4.1, é possível fazer esta alocação local de registradores de maneira mais eficiente.

Os resultados apresentados na tabela sugerem que esta maneira de efetuar a alocação usando vários sub-grafos do grafo global obtém resultados equivalentes à implementação segundo Chaitin, e é vantajosa devido a:

- haver grafos de menor número de vértices. Isto permitiria utilizar, caso necessários, algoritmos exaustivos para a coloração;
- economia de memória: nem todos os sub-grafos necessitam estar presentes na memória ao mesmo tempo, durante a compilação.

5. COMENTÁRIOS, CONCLUSÕES E SUGESTÕES

Em nossa implementação, as fases de análise de fluxo (propagação de definições, transformação de variáveis em nomes e análise de nomes vivos) totalizam pouco mais de 500 linhas de texto em Pascal, ou 17% do tamanho do compilador. Esta maneira de implementar a análise de fluxo é vantajosa porque adiciona pouca complexidade ao compilador.

Da mesma maneira, a alocação de registradores através da coloração de grafos (construção e coloração dos grafos de interferência) é implementada por um número equivalente de linhas em Pascal, e apresenta, segundo Chaitin, melhorias significativas na qualidade de código objeto gerado.

5.1 CONSTRUÇÃO DE UM PROGRAMA A PARTIR DE UM GRAFO:

Conforme dissemos, Chaitin demonstra em seu artigo que, tendo um grafo qualquer, é possível sintetizar um programa que apresente o mesmo grafo de interferências. Isto impede a possibilidade de utilização de algoritmos especializados para a coloração de apenas uma certa classe de grafos.

Restringindo a classe de programas a programas estruturados, e efetuando a alocação de registradores e a construção dos grafos de interferência da maneira que foi descrita, resta determinar se ainda é possível construir um programa que apresente um grafo de interferências idêntico a qualquer grafo que escolhamos. Deve-se notar que a construção de Chaitin não é aplicável, pois resulta em programas não estruturados.

5.2 EXTENSÃO PARA OUTRAS ARQUITETURAS:

Este método de alocação de registradores pode ser aplicado também a algumas arquiteturas não homogêneas, e para arquiteturas com dois enderêços.

5.2.1 Arquiteturas de UCP com dois endereços:

Consideramos que, nas arquiteturas de UCP com dois endereços, cada instrução que execute uma operação, seja da forma:

OP end-1,end-2

onde end-1 pode designar apenas registradores da UCP, e end-2 pode designar outro registrador, ou uma posição de memória,

significando:

end-1 ← end-1 "OP" end-2.

Podemos estabelecer uma correspondência (grosseira) entre instruções numa arquitetura com três endereços para grupos de instruções que possuam efeito equivalente numa arquitetura com dois enderêços [Figura 5.2-1].

Instrução na Arq. 3-endereços	correspondente na arq. 1-endereços	Observação
OP R, R_1, R_2	MOV R, R_1 OP R, R_2	$R \leftarrow R_1$ $R \leftarrow R$ "op" R_2
OP R, R_1, M	MOV R, R_1 OP R, M	

OP M, R_1, R_2	MOV T, R_1
	OP T, R_2
	MOV M, T
		reg. adicional necessário (T)
		$M \leftarrow T$

OP M, M_1, R	MOV T, R
	OP T, M_1
	MOV M, T
		"op" comutativa

OP M, M_1, R	MOV T, M_1
	OP T, R
	MOV M, T
		"op" não-comutativa

OP M, M_1, M_2	MOV T, M_1
	OP T, M_2
	MOV M, T	

R_i designam registradores da UCP
M_i designam posições de memória

fig. 5.2-1

A correspondência apresentada pela figura implica, em certos casos, a necessidade de mais um registrador na UCP de dois enderêços, mas pode ser utilizada para estimar o desempenho da estratégia de alocação de registradores em uma arquitetura assim.

5.2.2 Tratamento de arquiteturas não-homogêneas:

Nas arquiteturas que apresentem idiossincrasias, como por exemplo exigir que a a multiplicação seja efetuada em apenas um determinado registrador, é possível construir o grafo de interferências de maneira a refletir algumas destas peculiaridades. A partir da construção do grafo, o algoritmo de coloração pode trabalhar de maneira padrão, sem fazer distinção especial entre qualquer dos registradores.

Chaitin introduz em seu artigo uma maneira de tratar alguns aspectos da arquitetura IBM-370. Fazendo a adição, ao grafo de interferências, de N vértices correspondentes aos N registradores da máquina, é possível expressar a existência de certas relações especiais entre alguns valores computados e estes registradores.

Por exemplo, descrevemos a seguir uma maneira de trabalhar com uma não homogeneidade existente na arquitetura do microprocessador INTEL 8086 [Intel82].

- Resultado da multiplicação efetuada apenas em um registrador específico:

Colocamos uma aresta de interferência entre o vértice no grafo que representa o resultado da multiplicação e entre os vértices que representam todos os outros registradores,

exceto aquele onde a multiplicação deverá resultar. Um artifício análogo pode também ser utilizado no caso de a arquitetura exigir que um dos argumentos da multiplicação esteja em um registrador determinado.

5.3 SUGESTÕES PARA FUTURA EXPERIMENTAÇÃO:

- efetuar a coloração "de baixo para cima":

Ao invés de fazer a coloração dos nós mais próximos à raiz, na árvore do programa, é possível fazer a coloração a partir dos nós mais próximos às folhas, que são os blocos básicos, e prosseguir em direção à raiz.

Usando este método, será necessário compatibilizar os resultados da coloração cada vez que a coloração dos blocos componentes for repassada para o bloco "englobante". Por exemplo, em um bloco condicional, será necessário "trocar os nomes" das cores atribuídas aos nomes em cada um dos blocos componentes de maneira que uma cor identica seja atribuída aos nomes que estejam vivos à entrada ou saída dos dois blocos componentes.

- experimentar com outros algoritmos para a coloração:

Poder-se-á experimentar com outros algoritmos heurísticos para a coloração, e procurar limites, em tamanho de grafo, onde a utilizaão do algoritmo que faz a busca exaustiva de uma coloração seria prática.

BIBLIOGRAFIA

[AhoUll77]
Aho,A.V. and Ullman,J.D, "Principles of Compiler Design", Addison-Wesley, 1977.

[Bruno-Sethi 76]
Bruno,J. and Sethi, R. "Code generation for a one-register machine", J.ACM22,3 (July1976).

[Chait81]
G.J.Chaitin et alii, "Register Allocation via Coloring", Computer Languages, vol. 6 pp 47-57, 1981.

[Chait82]
G.J.Chaitin, "Register Allocation and Spilling via Graph Coloring", J.ACM, vol.6 pp. 98-105, 1981.

[ChowHenn84]
Chowning and Hennessy, Proceedings of the SIGPLAN Symposium on Compiler Construction, 1984.

[Day70]
W.H.E. Day, "Compiler Assignment of Data Items to Registers", IBM Syst. J., no.4, 1970.

[Freib74]
R.A. Freiburghouse, "Register Allocation via Usage Counts", Comm. ACM, v.17 pp 638-642, Nov 1974.

[Intel82]
Intel Co., "8086 Family Users Manual", 1982.

[JPJ83]
José Pedro Jr, "Uma Implementação de Módula-2: Análise e Representação Intermediária", Tese de Mestrado - Depto. de C.Computação, IMECC-UNICAMP,1983.

[Karp72]
R.M.Karp,"Reducibility among combinatorial problems", em R.E.Miller e J.W.Tatcher (eds), "Complexity of computer computations", Plenum Press, New York, pp. 85-104 1972.

[LSSSK79]
C.L.Lucchesi, Imre Simon, Istvan Simon, Janos Simon e T.Kowaltowski, "Aspectos Teóricos da Computação", IMPA-CNPQ, 1979.

[Kowal84]
T. Kowaltowski, "Implementation of Structured Data Flow Analysis", Relatório Interno no. 258, IMECC-UNICAMP, 1984.

[PattSéq82]
D.A.Patterson and C.H.Séquin, "A VLSI RISC", IEEE Computer, pp. 8-20, Sept 1982.

[Radin82]
G.Radin, "The 801 Minicomputer", Proc. Symp. Architectural Support for Programming Languages and Operating Systems, March 1-3 1982.

[Ros77]
B.K.Rosen, "High-Level Data Flow Analysis", Comm. ACM 20, 10, pp. 712-724, 1977.

[Silva84]
Silva, K.L., Tese de Mestrado - Depto. de C.Computação, IMECC-UNICAMP, 1983.

[Vax84] VAX-11 Family, Architecture Handbook, Digital Equipment Corporation, 1984.

[Wirth83]
N.Wirth, "Programming in Modula-2", Springer-Verlag, 1983.

[Wul75]
W. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs and C.M Geschke, "The Design of an Optimizing Compiler", American Elsevier, 1975.

APENDICE I - SINTAXE DA LINGUAGEM COMPILADA:

1. <módulo> = MODULE <identificador> ;

<declarações> ; <bloco>

2. <bloco> = BEGIN <Sequência_de_Comandos> END <identificador> .

3. <Sequência_de_comandos> =

<Comando> {;<Comando>}

4. <Comando> = <Cmd_Atribuição> | <Cmd_Condicional> |

<Cmd_Repetitivo> | <Cmd_EXIT>

5. <Cmd_Atribuição> =

<identificador> := <expressão>

6. <Cmd_Condicional> =

IF <expressão> THEN <Sequência_de_comandos>

[ELSE <Sequência_de_comandos>]

END

7. <Cmd_Repetitivo> =

LOOP <Sequência_de_comandos> END

8. <Cmd_EXIT> = EXIT IF <expressão>

9. <expressão> = <expressão_simples>
[<relação> <expressão_simples>]

10. <relação> = = | ≠ | <> | > | < | >= | <=

11. <expressão_simples> =

<termo> { (+|-|OR) <termo> }

12. <termo> = <fator> [(* | DIV | AND | MOD) <fator>]

13. <fator> = <identificador> | número | (<expressão>)

APÊNDICE II : Estrutura de dados da árvore do programa:

2) Declarações auxiliares:

```
lin_block_cat= (undef_block,basic_block, if_block, loop_block);

DAG_symt_cat= (undef_DG,or_DG, const_DG, var_DG);

ref_linear_block= ^ linear_block;

ref_DAG_node=   DAG_node;

ref_DAG_symt= ^ DAG_symt;

ref_Ass_list_node= ^ Ass_list_node;

name_set = array[0..words_per_name of set of 0..set_el_per_word];


name_graph=array[first_nam..last_nam] of name_set;
```

b) Blocos:

```
br_block=RECORD
          next: ref_linear_block; { linked list }

          k_flow,c_flow,
          x_flow,y_flow,
          K,C,
          X,Y:     name_set;

          graph: name_graph;
          elts_in_graph: name_set; { elements in this graph }
          colors: ARRAY[first_nam..last_nam] of INTEGER;
                                  { colors assigned to nodes;
                                   colors are represented by INTEGERS }
          g_size_bef_coloring:INTEGER;{ graph size before coloring;
                                        used for statistical purposes only }




          case cat: lin_block_cat of

          basic_block: (     { execution order list }
                  nb_node_list:INTEGER; { nbr of DAG nodes
                  node_list:array[1..100] of ref_DAG_node;


                             loop_exec,
                             DAG:     ref_DAG_node;
                             symb_tbl: ref_DAG_symt);

          if_block,
          loop_block: (
                             before_list,
                             after_list:      ref_linear_block)
     END; { record }
```

c) nós do DAG:

```
DAG_node = RECORD
        next_root: ref_DAG_node;
        Assoc_list: ref_AssList_node;
        ass_father: BOOLEAN;

{ these two fiels have both the same meaning,
  but are used in two different tree visits:

  order_num_fathers is used for choosing the execution order,
  and num_fathers is used for building the interference graph. }

        order_num_fathers,
        num_fathers:INTEGER;


        case cat:expr_cat OF

                unaryex,
                binaryex:(                          { linked list of
                        next:ref_DAG_node;          same-operator DAG nodes }
                        oper:op_type;

                        temp_nbr:INTEGER) { if t's a temporary }

                        left,
                        right:ref_DAG_node);

                constex:(
                        value:INTEGER);

                varex:(
                        name:ref_DAG_symt)

                        { points to this VAR's representing entry
                        in the DAG's symbol table }
    END;
```

) tabela de símbolos do DAG:

```
AG_symt = RECORD
        next: ref_DAG_symt;
        use_node,
        def_node: ref_DAG_node;

        case cat:DAG_symt_cat of

        op_DO: (
                operator:  op_type;
                );

                                AssList_node=RECORD
                                        name: ref_DAG_symt;
                                        next: ref_AssList_node
                                END; { record }

        const_DO:(
                );

        var_DO: (
                use_nbr,                { will mean "the definition class"
                                          that reach this use }
                def_nbr: INTEGER) { if def_node <> NIL and is not
                                  a redefinition of the same var
                                  then def_nbr := a new def }

                eq_var : ref_symbol)

    END;
```

APÊNDICE III : Exemplo de percurso na Árvore do Programa:

```
procedure Mak_All_Name_Sets(b: ref_linear_block);
{ Create the "created" and "killed" names sets for
  all blocks in the program }


      procedure Basic_Name_Sets(b:ref_linear_block);
      { Build the sets for live names analysis in
        a single basic block }
      VAR
              st:ref_DAG_sumt;
              newdef:ref_Ass_list_node;
               r:ref_DAG_node;
      begin { Basic_Name_Sets }
      st := b^.symb_tab;
      while st<>NIL do begin
              if st^.cat=var_DD then begin

              (* check for uses of the var: *)

                      if st^.use_node <> NIL then begin

                              add_el_to_set(real_name[st^.use_nord],
                                              b^.c_FLOW);
                      end; { if }

              (* check for definitions of the var *)

                      if st^.def_node^.nam <> st then begin

                              remove_el_from_set(real_name[st^.def_nord],
                                              b^.k_FLOW);
                      end; { if }

              end; { if }
      st := st^.next;
      end; { while }
      end; { Basic_Name_Sets }

begin { Mak_All_Name_Sets }
while b<>NIL do with b^ do begin
      if cat<>basic_block then begin
              Mak_All_Name_Sets(before_list);
              Mak_All_Name_Sets(after_list);
      end ELSE begin
              Basic_Name_Sets(b)
      end; { if }
b:=b^.next;
end; { while }
end; { Mak_All_Name_Sets }
```

Índice IV - Programas de Teste:

P1

```
module x)
var i:integer)
        ;
begin
if 1 then
          i1:=1)
          i2:=2
else if 2 then
          i1:=1)
          i3:=2
else if 3 then
          i2:=1)
          i3:=2
else if 4 then
          i2:=1)
          i4:=2
else if 5 then
          i3:=1)
          i4:=2
else if 6 then
          i4:=1)
          i5:=2
else if 7 then
          i5:=1)
          i6:=2
else if 8 then
          i5:=1)
          i7:=2
else if 9 then
          i6:=1)
          i7:=2
else if 10 then
          i6:=1)
          i5:=2
else if 11 then
          i7:=1)
          i6:=2
end
end
end
end
end
end
end
end
end
end
end
end.
```

P2

```
MODULE Puzzle)

VAR i:INTEGER)

begin
p:=9)
puzzle:=2)
size:=5)
d:=3)
m:=0)
loop
exit if m>size)
        dummy:=puzzle/m
end) (* loop *)

i:=1)
loop
exit if i>5)

        j:=1)
        loop
        exit if j>5)

                k:=1)
                loop
                exit if k>5)

                dummy:=puz e/(i+d*(j+d*k).
                end (* loop *)
        end (* loop *)
end) (* loop *)


i:=0)
loop
exit if i>99 (* typeMax *))

        m:=0)
        loop
        exit if m>size)

                dummy:=p/(i*3)+m
        end (* loop *)
end (* loop *)
end .
```

```
p3
ODULE Wadu;

AR      infile:INTEGER;
* Initialization & sreeting message *)

esin

ros:=33;
de  :=21;
bs  :=39;
eo  :=23;
buf:=37;


oc  :=1;

oop
f cbuf=38 then load:=1

LSE
        IF cbuf = 29 THEN loc := 1
        ELSE
                IF cbuf=cdel THEN
                        loc := loc - 1
                ELSE
                        IF cbuf=ceol THEN List:=9
                        ELSE
                                IF cbuf=93 THEN Execute:=99
                                ELSE
                                        IF cbuf=99 THEN
                                                i:=0;
                                                loop
                                                        a:=1

                                                exit if 434;
                                                        w:=pros/loc;loc:=loc+1
                                                        ceol:=pros/loc;
                                                        loc:=loc+1
                                                end (* loop *)
                                        ELSE
                                                cbuf:=pros/loc;
                                                loc:=loc+1
                                        end (* if 99 *)


                                end (* if 93 *)
                        end (* if cbuf=ceol *)
                end (* if cbuf=cdel *)
        end (* if cbuf=29.. *)
end (* if cbuf=38.. *)

end (* loop *)

END.
```

```
DULE Quick;
* Parte adaptada do Quicksort *)

AR i:INTEGER;

EGIN
op_start:=3;
( top_start<6
ne)
      bubbleSort:=1 (* sort small sections with
                       bubble sort *)


 se


      (* findMedian *)
               start:=5;
               top:=3;
               numbers:=8;

      swap := 1;

      s:=start;
      t:=top;
      n:=start;
      loop
      exit if s>t;

                 (* swap = 1
               then
                       loop
                       exit (* if ( (numbers/s) >= (numbers/n) ) AND
                                  ( s > n) *)
                            s:=s-1
                       end;
                       (* s>n
                       then
                       swap :=
                            x:=(numbers/s)+(numbers/n);

                            a:=s
                       end (* if *)
               else
                       loop
                       exit  if ((numbers/s) <= (numbers/n)) AND
                                ( s < t) )
                            s:=s+1
                       end;
                        (* s<t
                       then
                            swap :=
                            x:=(numbers/s)+(numbers/t);

                            a:=s
                       end;
                       swap:=1
               end (*  *)
      end (* loop *)
o (* if *)
```

P5

```
program t;
(* teste "erande" *)
var a,b:INTEGER;
begin
if 1 then
        i:=1;
        j:=2;
        if 11 then
                i:=3;
                k:=4
        end;
        m:=i+j+k
else
        w:=91;
        x:=92;
        if 99 then
                w:=93;
                y:=94
        end;
        mm:=x+y+w
end
end.
```

P6

```
MODULE Eratosteneš;

VAR
        Size:INTEGER;


begin
iter:=1;
size:=2;


loop
exit if (iter>10);

        count:=10;
        i:=0;
        loop
        exit if i>size;
        flags:=1;
        dummy:=flags+i
        end; (* loop *)

        i:=0;
        loop
        exit if i> size;
        if flags/i then
                prime:=i+i+3;
                k:=i+prime;

                loop
                exit if k> size;

                        flags:=1;
                        dummy:=flags/k;
                        k:=k+prime
                end; (* loop *)

        count:=count+1
        end (* if *)

        end (* loop *)
end (* loop *)
end.
```

P7

```
MODULE Bubble;
(* bubble sort *)

VAR i:INTEGER;
BEGIN
start:=1;
top:=99;
subArray:=5;


loop
sw:=sched:=0;
        i:=start;
        loop
        exit if (index > (top-1));
                if (subArray/index)>(subArray/(index+1))
                then
                        sw:=(subArray/index)+(subArray/(index+1));
                        sw:=sched:=1;

                end
        end
exit if sw=0;

end (* loop *)
end.
```