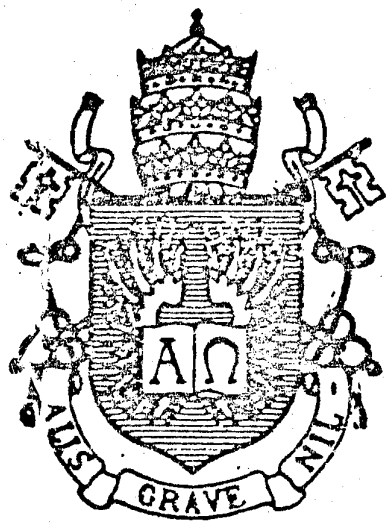


PUC

SÉRIES: Monografias em Ciência da computação, 19/88

OTIMIZAÇÃO DE CÓDIGO UTILIZANDO GRAMÁTICAS DE ATRIBUTOS

Clovis Torres Fernandes
José L. M. Rangel Netto

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
MARQUÊS DE SÃO VICENTE, 225 – CEP 22453
RIO DE JANEIRO – BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

SÉRIES: Monografias em Ciência da computação, 19/88

Editor: Paulo A. S. Veloso novembro,1988

OTIMIZAÇÃO DE CÓDIGO UTILIZANDO GRAMÁTICAS DE ATRIBUTOS

Clovis Torres Fernandes*
ITA - PUC/RJ

José L. M. Rangel Netto**
PUC/RJ

* Parcialmente financiado pela CAPES/PICD.

** Parcialmente financiado pelo MCT

ABSTRACT

Solutions for two data flow problems are presented - reaching definitions and available expressions - by means of attribute grammars. Abstract trees are used as intermediate representation instead of the usual syntax trees. An algorithm performing the common subexpression elimination in linear time on the number of the subexpression occurrences is also presented. This algorithm is based on the available expression information. Some examples illustrate the technique used.

Keywords: code optmization, attribute grammars, abstract trees, data flow problems, common subexpression elimination.

RESUMO

Apresentam-se soluções para dois problemas de fluxos de dados - definições atuantes e expressões disponíveis - utilizando-se gramáticas de atributos. Ao invés das usuais árvores sintáticas, tem-se árvores abstratas como representação intermediária. Apresenta-se também um algoritmo que realiza a eliminação de sub-expressões comuns em tempo linear no número de ocorrências de sub-expreessões. Este algoritmo baseia-se na informação de expressões disponíveis. Alguns exemplos ilustram a técnica empregada.

Palavras-chaves: otimização de código, gramáticas de atributos, árvores abstratas, problemas de fluxos de dados, eliminação de sub-expressões comuns.

SUMÁRIO

Introdução

Serão apresentados dois problemas de fluxos de dados - definições atuantes ("reaching definitions") e expressões disponíveis - e uma transformação - eliminação de sub-expressões comuns.

Toda análise de fluxo de dados para tais problemas é feita utilizando-se gramáticas de atributos (Knuth, 1968; Babich and Jazayeri, 1978) adaptadas para operarem sobre árvores abstratas ao invés de árvores sintáticas. Uma das vantagens de se trabalhar com árvores como representação intermediária é que torna-se desnecessário realizar a análise do fluxo de controle, pois este se encontra subjacente à estrutura da própria árvore.

A solução do problema de definições atuantes permite descobrir computações invariantes dentro de laços tipo While ou Repeat e realizar movimentação de código. A solução do problema de expressões disponíveis permite determinar se uma sub-expressão é comum, podendo ser eliminada ou não. Apresentam-se um algoritmo que realiza a eliminação de sub-expressões comuns em tempo linear ao número de ocorrências de subexpressões bem como passos auxiliares à realização desta transformação.

Todos exemplos apresentados são resultados obtidos com a implementação de um avaliador para a gramática de atributos e do algoritmo de eliminação de sub-expressões propostos neste trabalho.

Técnicas usuais de otimização

Entre as técnicas de otimização incluem-se eliminação de sub-expressões comuns, movimentação de código, eliminação de código morto etc. Esta fase de melhoramento de código consiste em realizar análise do fluxo de controle e do fluxo de dados seguida pela aplicação de transformações (Hecht, 1977;Kennedy, 1981).

A finalidade da análise do fluxo de controle é codificar o fluxo de controle de um programa para uso na análise do fluxo de dados. Tendo como forma intermediária, em geral, quádruplas, constroem-se o grafo de chamadas e o grafo do fluxo.

O grafo de chamadas é um grafo direcionado que representa o relacionamento entre os procedimentos de um programa. Cada nó de um grafo de chamadas corresponde a exatamente um procedimento de um programa e cada arco (x,y) representa uma ou mais referências do procedimento representado pelo nó y no procedimento representado pelo nó x.

O grafo do fluxo é um grafo direcionado que descreve o fluxo de controle entre vários trechos do programa. Cada nó deste grafo corresponde a um bloco básico do procedimento e há um arco (x,y) do bloco x para o bloco y se o controle pode, potencialmente, transferir-se do bloco x para o bloco y em tempo de execução. Um bloco básico é uma seção de programa sem ramificações, exceto em seu fim, ou rótulos, exceto em seu início.

Análise do fluxo de dados é o processo de apurar e coletar informação (antes da execução) sobre possível modificação, preservação e uso de certas entidades de um programa, tais como valores ou outros atributos das variáveis. É usada por compiladores de otimização para assegurar a correção das otimizações.

Distinguem-se vários níveis de análise do fluxo de dados:

a. a nível de comando ou quádrupla :: local;
b. a nível de bloco básico :: local;
c. a nível de procedimento :: global;
d. a nível de programa :: global.

Para análise do fluxo de dados nos níveis a., b. e c. acima necessita-se do grafo do fluxo de controle com certas informações agregadas aos seus nós; para o nível d. necessita-se, também, do

grafo de chamadas. Neste trabalho, análise do fluxo de dados global refere-se apenas ao nível de procedimento.

Uma otimização descoberta por esta fase pode envolver, por exemplo, mover um ou mais comandos (quádruplas, sub-árvores) ou combinar dois ou mais comandos (quádruplas, sub-árvores) em um único comando. Pode envolver, também, manipular expressões e seus valores.

Análise do fluxo de dados global de alto nível

Esta análise do fluxo de dados que utiliza grafos de fluxo de controle é conhecida como de **baixo nível**. Por outro lado, o método de atributos é uma técnica de **alto nível** pois opera sobre uma representação intermediária do programa, a saber uma árvore sintática (em príncipio uma árvore de derivação), que inclui todas as estruturas do fluxo de controle de alto nível presentes no programa fonte (Babich and Jazayeri, 1978), tornando desnecessária a construção de grafos de fluxo de controle. A escolha da representação em árvore é natural neste caso, devido à relativa facilidade de aplicação de transformações a essas estruturas. O termo análise do fluxo de dados de alto nível foi cunhado por Rosen (1977), que faz uso, se bem que de modo distinto do apresentado aqui, da semântica das construções de controle.

O método de atributos é uma técnica para realizar análise do fluxo de dados global de alto nível usando uma representação em árvore do programa. A abordagem geral do método é esta: quando da definição da linguagem fonte, escrevem-se regras de atributos para cada estrutura de controle. Estas regras sumarizam o fluxo de controle em tempo de execução induzido pela estrutura. A grosso modo, pode-se dizer que um conjunto de regras é aplicado sempre que a sua correspondente estrutura de controle seja encontrada na árvore do programa.

A representação intermediária utilizada neste trabalho é a árvore abstrata. Uma árvore abstrata consiste de nós operadores e nós operandos. Os nós operadores são sempre nós internos e os operandos sempre folhas. Cada nó operador S que não seja operador de expressões corresponde a um conjunto de nós (ou comandos) aninhados e é denominado um nó básico. Ao conjunto de nós aninhados (a sub-árvore com raiz em S) denominamos o escopo de S.

Neste trabalho consideram-se os seguintes tipos de nós básicos:

- simples;
- de sequência de comandos;
- iterativos tipo RepeatStm e WhileStm;
- condicionais tipo IfThen e IfThenElse.

São nós básicos simples: de atribuição, de leitura, de escrita ou relacionais. O escopo de um nó básico simples será sempre constituído apenas pelo próprio nó. O apêndice mostra a representação de árvore abstrata utilizada, descrevendo todos os tipos de nós.

Definições atuantes

Uma definição de uma variável *A* é um nó básico simples no qual o valor de *A* é recalculado, ou lido, ou de alguma forma criado. O valor de *A* será ou não alterado dependendo do valor anterior disponível. No caso restrito aqui considerado, não estão incluídas "definições ambíguas", em que a definição de valor é feita por um efeito colateral de uma chamada de unidade de programa, ou através de uma referência (variável tipo apontador) ou ainda uma atribuição a uma componente não especificada de um vetor. Quando uma variável *A* é referenciada em uma expressão tem-se um uso dessa variável. Um ponto em um programa significa uma posição no fluxo de controle subjacente imediatamente antes ou após qualquer nó da árvore abstrata.

Diz-se que uma definição de uma variável *A* atua (alcança) um ponto *P* se há um caminho no fluxo de controle subjacente à estrutura em árvore, daquela definição de *A* até *P*, tal que nenhuma outra definição de *A* apareça naquele caminho.

O problema de definições atuantes consiste em se determinar o conjunto denominado IN[S] de definições de variáveis que podem atuar no ponto imediatamente anterior a cada nó S da árvore. É frequentemente conveniente armazenar a informação de definições atuantes como cadeias de uso-definição ou cadeias ud, que são listas, para cada uso de uma variável, de todas as definições que alcançam este uso. As cadeias ud serão usadas para se descobrir aquelas computações em um laço tipo While ou Repeat que são invariantes, ou seja, cujos valores não mudam enquanto o controle permanece dentro do laço e para realizar movimentação de código (vide algoritmos 10.7 e 10.8 em Aho et al. (1986)).

Para se determinar as definições atuantes deve-se atribuir um número distinto a cada definição: d_1, d_2, . . ,d_n, onde n é o número total de definições encontradas num programa particular. Dado que os conjuntos aqui considerados são subconjuntos de todas as definições encontradas, pode-se usar uma representação de vetores de bits para eles. Convenciona-se que, para a e b conjuntos, $a + b$, $a \cdot b$ e $a - b$ correspondem à união, interseção e diferença de conjuntos, respectivamente. Convenciona-se também que $\{d_i\}$ corresponde a um vetor de bits que indica um conjunto com apenas a definição d_i, através do valor 1 na i-ésima posição e zero nas demais posições e {} a um vetor de bits que indica um conjunto vazio, através de zero nas n posições do mesmo.

Em seguida, para cada variável simples *A*, determina-se o conjunto de todas definições de *A* que ocorrem no programa, representando-o por $\mathbb{D}_A$.

O próximo passo consiste em calcular dois atributos (conjuntos) para cada nó básico S:

GEN[S] - é o conjunto de definições geradas, aquelas definições dentro do escopo de S que alcançam o fim do escopo de S, independente se alcançam o início.

KILL[S] - é o conjunto de definições que nunca alcançam o fim do escopo de S, mesmo se alcançam o início.

Em seguida, devem-se computar os atributos:

IN[S] - é o conjunto de definições que alcançam o ponto imediatamente anterior ao nó básico S.

OUT[S] - é o conjunto de definições que alcançam o ponto imediatamente posterior ao nó básico S.

Seguindo a terminologia usual em gramáticas de atributos, um atributo é dito **sintetizado** se o seu valor é obtido dos valores de atributos dos seus nós filhos. Um atributo é dito **herdado** se o seu valor é obtido dos valores de atributos do seu nó pai. Os atributos GEN, KILL e OUT são sintetizados e IN é herdado.

Os conjuntos GEN, KILL, IN e OUT serão calculados de acordo com a gramática de atributos associada. As equações de cálculo para os atributos GEN, KILL, IN e OUT satisfazem a condição de Bochmann (l-attributed grammars) e podem ser avaliadas em dois passos da esquerda para direita (Bochmann, 1976; Aho et al., 1986). Os conjuntos GEN e KILL serão avaliados no primeiro passo enquanto IN e OUT serão no segundo.

Nestas condições, utiliza-se, para cada passo, um algoritmo que realiza a avaliação dos atributos através de um percorrimento em profundidade da árvore tendo o aspecto geral apresentado na figura 1. Este algoritmo tem complexidade de tempo **linear** ao número de nós da árvore abstrata.

```
procedure eval( u: ponteiro_para_nó);
begin
| seja t o tipo do nó u;
| seja n = nr. de filhos do nó u
| seja u.i = nó do i-ésimo filho de u
| for i := 1 to n
| do begin
|     | avalia atributos herdados associados a nós do tipo t
|     |          (quando houver);
|     | eval( u.i)
|     end;
| avalia atributos sintetizados associados a nós do tipo t
end;
```

Fig. 1 - algoritmo avaliador de atributos

Primeiro passo: Equações para GEN e KILL

Para cada tipo de nó básico, têm-se as seguintes equações de cálculo dos atributos associados:

1. nó básico simples

   $GEN[S] = \{d_i\}$

   $KILL[S] = \mathbb{D}_A - \{d_i\}$, onde $\mathbb{D}_A$ é o conjunto de todas definições no programa para a variável A e d_i é a i-ésima definição da variável A quando S for de atribuição ou de leitura.

2. nó sequência de comandos:
   a. com apenas 2 comandos:

   $GEN[S] = GEN[S_2] + (GEN[S_1] - KILL[S_2])$

   $KILL[S] = KILL[S_2] + (KILL[S_1] - GEN[S_2])$

b. de maneira geral:

```
begin
| eval(S1);          {calcula GEN[S1] e KILL[S1]}
| G := GEN[S1];
| K := KILL[S1];
| while houver novo irmão Si {1<i≤n}
| do begin
|    | eval(Si);  {calcula GEN[Si] e KILL[Si]}
|    | G := GEN[Si] + (G - KILL[Si]);
|    | K := KILL[Si] + (K - GEN[Si])
|    end;
| GEN[S] := G;
| KILL[S] := K
end
```

Não se consegue especificar adequadamente esta situação com a usual notação declarativa de gramáticas de atributos! O procedimento eval da figura 1 é modificado no seguinte aspecto. Ao encontrar nós do tipo sequência de comandos, ao invés de executar o corpo normal de eval, computa-se o algoritmo iterativo acima, o qual invoca recursivamente eval nos pontos adequados. Esta observação é válida para toda avaliação de nós sequência de comandos apresentada neste trabalho, computando-se o algoritmo iterativo então apresentado.

3. nó IfThenElse

$$GEN[S] := GEN[S_1] + GEN[S_2]$$
$$KILL[S] := KILL[S_1] \circ KILL[S_2]$$

4. nó IfThen

$$GEN[S] := GEN[S_1]$$
$$KILL[S] = \{\}$$

5. nó WhileStm

GEN[S] = GEN[S_1]
KILL[S] = KILL[S_1]

6. nó RepeatStm

GEN[S] = GEN[S_1]
KILL[S] = KILL[S_1]

Segundo passo: Equações para IN e OUT

Em cada nó básico S o valor de OUT obtido deve corresponder a

OUT[S] = GEN[S] + (IN[S] - KILL[S]).

E para o nó básico raiz faz-se a seguinte suposição inicial:

IN[raiz] = (), ou seja, nenhuma definição está atuante no início do programa.

Para cada tipo de nó, têm-se as seguintes equações de cálculo dos atributos associados:

1. nó básico simples:

IN[S] herdado do pai de S;
OUT[S] = GEN[S] + (IN[S] - KILL[S]).

2. nó sequência de comandos:

```
begin
  IN[S1] := IN[S];
  eval(S1);        {calcula OUT[S1]}
  AUT := OUT[S1];
  while houver novo irmão Si (1<i≤n)
  do begin
       IN[Si] := AUT;
       eval(Si);  {calcula OUT[Si]}
       AUT := OUT[Si]
     end;
  OUT[S] := AUT
end
```

3. nó IfThenElse

$IN[S_1] = IN[S]$

$IN[S_2] = IN[S]$ ——— herdados

$OUT[S] = OUT[S_1] + OUT[S_2]$ sintetizado

4. nó IfThen

$IN[S_1] = IN[S]$

$OUT[S] = OUT[S_1] + IN[S]$

5. nó WhileStm

$IN[S_1] = IN[S] + OUT[S_1]$ (*)

$OUT[S] = IN[S] + OUT[S_1]$

onde pode-se provar (Aho et al., 1986-pp. 617) que (*) é igual a

$IN[S_1] = IN[S] + GEN[S]$.

6. nó RepeatStm

$IN[S_1] = IN[S] + GEN[S]$

$OUT[S] = OUT[S_1]$

Exemplo do cálculo de GEN, KILL, IN e OUT

A figura 2 apresenta a árvore abstrata correspondente ao programa fonte abaixo, decorada com os valores de atributos GEN, KILL, IN e OUT calculados de acordo com os passos 1 e 2 acima. Existem 5 pontos de definição de variáveis e apenas as definições d_3, d_4 e d_5 estão atuantes no fim do programa. Todas as definições estão atuantes na entrada da parte de comando do nó RepeatStm, d_1 e d_2 por herança e d_3, d_4 e d_5 devido ao laço do fluxo de controle do nó RepeatStm. A definição d_5 está atuante na saída do nó IfThen porque está atuante na entrada do nó IfThen, embora não esteja atuante na saída do nó da parte then do nó IfThen.

Programa fonte exemplo

```
begin
| i := 2;                    {d1}
| j := i + 1;                {d2}
| repeat
| | i := j - 1;              {d3}
| | j := j + 1;              {d4}
| | if j > 0
| | then j := j - 4         {d5}
| until (j < 0)
end
```

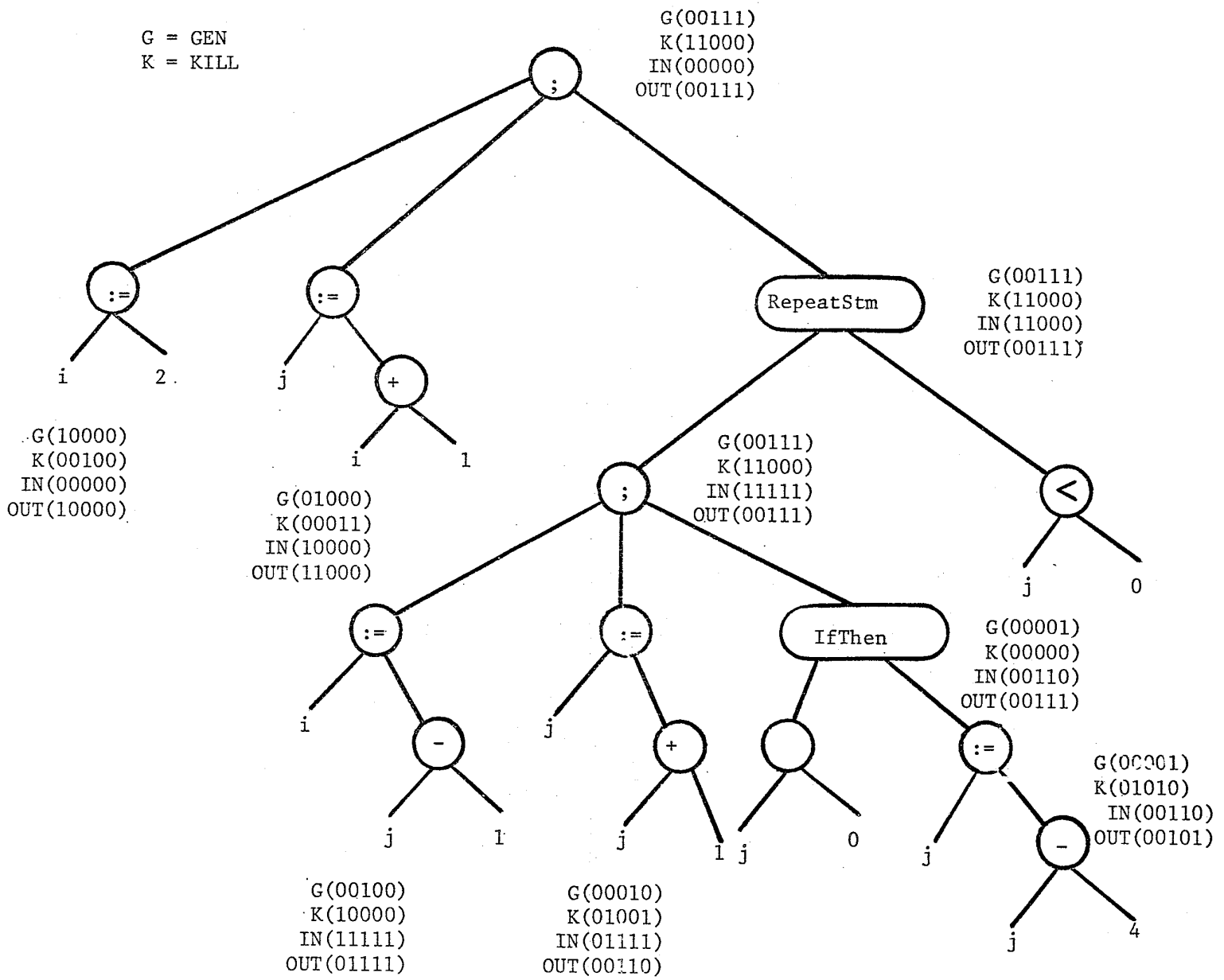


Fig. 2 - Exemplo do cálculo de GEN, KILL, IN e OUT

Expressões disponíveis

Uma expressão x *op* y está disponível num ponto *P* se todo caminho do fluxo subjacente (possivelmente com ciclos), do nó inicial até *P* avalia x *op* y, e após a última dessas avaliações antes de alcançar *P*, x ou y são variáveis e não há subsequentes definições das mesmas ou x ou y são sub-expressões e estão disponíveis.

Um nó **mata** a expressão x *op* y se define x ou y e subsequentemente não reavalia x *op* y. Um nó **gera** a expressão x *op* y se avalia x *op* y e subsequentemente não redefine x ou y. A definição de x pode ocorrer de duas formas: ou x é uma variável sendo definida ou x é uma sub-expressão e pelo menos uma variável componente de x está sendo definida.

O problema de expressões disponíveis consiste em se determinar o conjunto E_IN[S] de expressões disponíveis à entrada de cada nó S da árvore. O principal uso da informação de expressões disponíveis é para descobrir e, possivelmente, eliminar subexpressões comuns. Seja EP o conjunto "universo" de todas as expressões aparecendo no programa, numeradas $e_1, e_2, \ldots, e_m$, onde m é a cardinalidade de EP. Para se resolver o problema de expressões disponíveis, determinam-se os valores dos seguintes 4 atributos para cada nó básico S:

E_GEN[S] - o conjunto de expressões geradas no nó básico S;
E_KILL[S] - o conjunto de expressões em EP mortas em S;
E_IN[S] - o conjunto de expressões em EP que stão disponíveis no ponto imediatamente anterior a S;
E_OUT[S] - o conjunto de expressões em EP que estão disponíveis no ponto imediatamente posterior a S.

De forma análoga ao problema de definições atuantes, os conjuntos E_GEN e E_KILL serão avaliados num primeiro passo e os conjuntos E_IN e E_OUT num segundo.

Primeiro passo: Equações para E_GEN E E_KILL

Para cada tipo de nó básico têm-se as seguintes equações de cálculo dos atributos associados:

1. nó básico simples

   a. atribuição

      E_GEN[S] = conjunto de expressões da forma *x op y* que ocorrem no nó básico S e nem x nem y são redefinidos no escopo de S.

      E_KILL[S] = D$, onde D$ é o conjunto de expressões que contenham a variável x como operando e x é redefinida no escopo de S.

   b. relação (expbool)

      E_GEN[S] = idem atribuição.

      Neste trabalho, supomos que o nó expbool não mata nenhuma expressão.

2. nó sequência de comandos

   a. com apenas 2 comandos

$$E\_GEN[S] = E\_GEN[S_2] + (E\_GEN[S_1] - E\_KILL[S_2])$$
$$E\_KILL[S] = E\_KILL[S_2] + (E\_KILL[S_1] - E\_GEN[S_2])$$

b. de maneira geral

```
Algoritmo de Cálculo:
begin
| eval(S_1);        {calcula E_GEN[S_1] e E_KILL[S_1]}
| E_G := E_GEN[S_1];
| E_K := E_KILL[S_1];
| while houver novo irmão S_i (1<i≤n)
| do begin
|     | eval(S_i); {calcula E_GEN[S_i] e E_KILL[S_i]}
|     | E-G := E-GEN[S_i] + (E_G - E_KILL[S_i]);
|     | E_K := E_KILL[S_i] + (E_K - E_GEN[S_i])
|   end;
| E_GEN[S] := E_G;
| E_KILL[S] := E_K
end
```

3. nó IfThenElse

$$E\_GEN[S] = (E\_GEN[S_1] \circ E\_GEN[S_2]) + (E\_GEN[expbool] - (E\_KILL[S_1] + E\_KILL[S_2]))$$

$$E\_KILL[S] = E\_KILL[S_1] + E\_KILL[S_2]$$

4. nó IfThen

$$E\_GEN[S] = E\_GEN[caminho_1] \circ E\_GEN[caminho_2]$$

De acordo com a representação do IfThen apresentada no apêndice, temos:

$$caminho1 = (E\_GEN[S_1] + (E\_GEN[expbool] - E\_KILL[S_1]))$$

$$caminho2 = E\_GEN[expbool]$$

Logo

$$E\_GEN[S] = [(E\_GEN[S_1] + (E\_GEN[expbool] - E\_KILL[S_1]))] \circ E\_GEN[expbool]$$

$$= (E\_GEN[S_1] \circ E\_GEN[expbool]) + [(E\_GEN[expbool] - E\_KILL[S_1]) \circ E\_GEN[expbool]]$$

$$= (E\_GEN[S_1] \circ E\_GEN[expbool]) + (E\_GEN[expbool] - E\_KILL[S_1])$$

$$E\_KILL[S] = E\_KILL[caminho1] + E\_KILL[caminho2]$$

onde

$caminho1 = \langle\rangle + E\_KILL[S_1] = E\_KILL[S_1]$

$caminho2 = \langle\rangle$

Logo

$$E\_KILL[S] = E\_KILL[S_1]$$

5. nó WhileStm

$$E\_GEN[S] = E\_GEN[caminho_1] \circ E\_GEN[caminho_2]$$

De acordo com a representação do WhileStm apresentada no apêndice, temos:

$caminho1 = E\_GEN[expbool]$

$caminho2 = (E\_GEN[expbool] - E\_KILL[S_1]) + E\_GEN[S_1] + E\_GEN[expbool]$

Logo

$$E\_GEN[S] = E\_GEN[expbool]$$

$$E\_KILL[S] = E\_KILL[caminho1] + E\_KILL[caminho2]$$

onde

$caminho1 = \langle\rangle$

$caminho2 = (\langle\rangle - E\_GEN[S_1]) + E\_KILL[S_1] + \langle\rangle = E\_KILL[S_1]$

Logo

$$E\_KILL[S] = E\_KILL[S_1]$$

6. nó RepeatStm

$$E\_GEN[S] = E\_GEN[S_1] + E\_GEN[expbool]$$

$$E\_KILL[S] = E\_KILL[S_1]$$

Segundo passo: Equações para E_IN e E_OUT

Em cada nó básico S o valor de E_OUT obtido corresponde a

E_OUT[S] = E_GEN[S] + (E_IN[S] - E_KILL[S]).

Para o nó básico raiz faz-se a seguinte suposição inicial:

IN[raiz] = {}, ou seja, nenhuma expressão está disponível no início do programa.

Para cada tipo de nó têm-se as seguintes equações de cálculo dos atributos associados:

1. nó básico simples

E_OUT[S] = E_GEN[S] + (E_IN[S] - E_KILL[S])

2. nó sequência de comandos

```
begin
| E_IN[S1] := E_IN[S];
| eval(S1);          (calcula E_OUT[S1])
| AUT := E_OUT[S1];
| while houver novo irmão Si (1<i≤n)
| do begin
|     | E_IN[Si] := AUT;
|     | eval(Si);   (calcula E_OUT[Si])
|     | AUT := E_OUT[Si]
|   end;
| E_OUT[S] := AUT
end
```

3. nó IfThenElse

$$\frac{E\_IN[S_1] = E\_IN[S] + E\_GEN[expbool] \qquad E\_IN[S_2] = E\_IN[S] + E\_GEN[expbool]}{E\_OUT[S] = E\_OUT[S_1] \circ E\_OUT[S_2]}$$

4. nó IfThen

$$\frac{E\_IN[S_1] = E\_IN[S] + E\_GEN[expbool]}{E\_OUT[S] = E\_OUT[S_1] \circ E\_IN[S_1]}$$

5 nó WhileStm

$$\frac{E\_IN[S_1] = E\_GEN[expbool] + (E\_IN[S] \circ E\_OUT[S_1]) \qquad (*)}{E\_OUT[S] = E\_GEN[expbool] + (E\_IN[S] \circ E\_OUT[S_1])}$$

Observe-se que $E\_IN[S_1]$ e $E\_OUT[S]$ possuem equações idênticas e que $E\_IN[S_1]$ não pode ser computado até que se tenha computado $E\_OUT[S_1]$. Temos duas maneiras de resolver este problema de circularidade. Em ambas é suposto que $E\_OUT[S_1]$ seja o conjunto universo (EP) inicialmente.

Na primeira, o avaliador deve ser iterado até que o valor atual de $E\_OUT[S_1]$ seja igual ao seu valor anterior. Para comandos bem comportados como WhileStm e RepeatStm, a iteração converge em dois passos apenas e pode ser realizada apenas no escopo do comando. O primeiro passo origina o valor final de $E\_OUT[S_1]$ e o segundo constata que o valor não se altera e a iteração pode se encerrar. Para comandos do tipo GOTO e rotulados, o avaliador deve ser iterado o número necessário de vezes, até que os valores se estabilizem (Babich and Jazayeri, 1978; Farrow, R., 1986), assemelhando-se aos algoritmos iterativos para grafos de fluxos de controle gerais, apresentados em Aho et al. (1986).

Na segunda maneira, sabe-se que

$$E\_OUT[S_1] = E\_GEN[S_1] + (E\_IN[S_1] - E\_KILL[S_1] \qquad (**)$$

expressa de maneira direta $E\_OUT[S_1]$ em termos de $E\_IN[S_1]$. Supondo (*) e (**) como verdadeiras, estas duas equações definem uma recorrência para $E\_IN[S_1]$ e $E\_OUT[S_1]$ simultaneamente, da qual se deriva, em dois passos, uma equação para $E\_IN[S_1]$ independente de $E\_OUT[S_1]$. Deste modo, o avaliador poderá realizar o cálculo de $E\_IN[S_1]$ em apenas um passo. A solução dada aqui é análoga à apresentada em Aho et al. (1986) para o problema de definições atuantes e é mostrada a seguir.

Assumindo $E\_OUT[S_1] = EP$ (universo), computamos uma estimativa para $E\_IN[S_1]$:

$$E\_IN[S_1]' = E\_GEN[expbool] + E\_IN[S].$$

A seguir, obtém-se uma melhor estimativa para $E\_OUT[S_1]$:

$$E\_OUT[S_1]' = E\_GEN[S_1] + ((E\_GEN[expbool] + E\_IN[S]) - E\_KILL[S_1]).$$

Aplicando esta estimativa à primeira equação, tem-se:

$$\begin{aligned} E\_IN[S_1]'' &= E\_GEN[expbool] + [(E\_GEN[S_1] + \\ &\quad ((E\_GEN[expbool] + E\_IN[S]) - E\_KILL[S_1])) \circ E\_IN[S]] \\ &= E\_GEN[expbool] + [(E\_IN[S] \circ E\_GEN[S_1]) + \\ &\quad (E\_IN[S] - E\_KILL[S_1])]. \end{aligned}$$

Reaplicando $E\_IN[S_1]''$ à segunda equação obtem-se:

$$\begin{aligned} E\_OUT[S_1]'' &= E\_GEN[S_1] + [(E\_GEN[expbool] + [(E\_IN[S] \circ \\ &\quad E\_GEN[S_1]) + (E\_IN[S] - E\_KILL[S_1])]) - E\_KILL[S_1]] \\ &= E\_GEN[S_1] + ((E\_GEN[expbool] + E\_IN[S]) - \\ &\quad E\_KILL[S_1]) = E\_OUT[S_1]'. \end{aligned}$$

E portanto os valores limitantes para $E\_IN[S_1]$ e $E\_OUT[S_1]$ são $E\_IN[S_1]'$ e $E\_OUT[S_1]'$, respectivamente. Dado que $E\_KILL[S] = E\_KILL[S_1]$, tem-se então derivado uma equação para (*), independente de $E\_OUT[S_1]$:

$$E\_IN[S_1] = E\_GEN[expbool] + (E\_IN[S] \circ E\_GEN[S_1]) + (E\_IN[S] - E\_KILL[S]).$$

6. nó RepeatStm

$$E\_IN[S_1] = E\_IN[S] \circ (E\_OUT[S_1] + E\_GEN[expbool])$$

$$E\_OUT[S] = E\_OUT[S_1] + E\_GEN[expbool]$$

Observe-se que $E\_IN[S_1]$ não pode ser calculado até que se tenha computado $E\_OUT[S_1]$. O procedimento a ser seguido é análogo ao realizado para o nó WhileStm. Supondo que seja verdade que

$$E\_OUT[S_1] = E\_GEN[S_1] + (E\_IN[S_1] - E\_KILL[S_1])$$

e fazendo $E\_OUT[S_1] = \emptyset P$, inicialmente, obtém-se a seguinte estimativa para $E\_IN[S_1]$:

$$E\_IN[S_1]' = E\_IN[S]$$

A seguir, obtém-se uma melhor estimativa para $E\_OUT[S_1]$:

$$E\_OUT[S_1]' = E\_GEN[S_1] + (E\_IN[S] - E\_KILL[S_1])$$

Aplicando esta estimativa à primeira equação, tem-se:

$$E\_IN[S_1]'' = E\_IN[S] \circ [E\_GEN[S_1] + E\_IN[S] - E\_KILL[S_1]]$$
$$= E\_IN[S] \circ E\_GEN[S_1] + E\_IN[S] - E\_KILL[S_1]$$

Reaplicando $E\_IN[S_1]''$ à segunda equação obtem-se:

$$E\_OUT[S_1]'' = E\_GEN[S_1] + [E\_IN[S] \circ E\_GEN[S_1] + E\_IN[S] - E\_KILL[S_1]]$$
$$= E\_GEN[S_1] + (E\_IN[S] - E\_KILL[S_1]) = E\_OUT[S_1]'$$

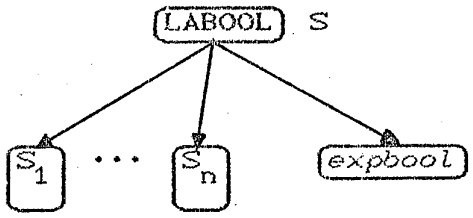
Portanto os valores limitantes para $E\_IN[S_1]$ e $E\_OUT[S_1]$ são $E\_IN[S_1]'$ e $E\_OUT[S_1]'$. Dado que $E\_KILL[S] = E\_KILL[S_1]$, tem-se derivado uma equação para $E\_IN[S_1]$ independente de $E\_OUT[S_1]$:

$$E\_IN[S_1] = (E\_IN[S] \circ E\_GEN[S_1]) + (E\_IN[S] - E\_KILL[S])$$

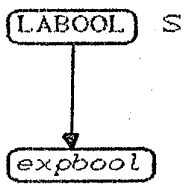
que é muito parecida com a equação derivada para o nó WhileStm.

O nó LABOOL

O nó **LABOOL** constitui uma generalização do nó relacional, visando facilitar a eliminação de subexpressões comuns. Esta generalização será desnecessária caso se adote o esquema de embutir nós de atribuição em sub-árvores que correspondam a expressões. A sua forma geral é:



após possíveis otimizações, onde $S_1,\ldots,S_n$ são sempre nós de atribuição. A sua forma inicial, antes de qualquer otimização, é:



Ao se considerar o nó **LABOOL**, deve-se substituir todo nó *expbool* por **LABOOL** nas equações para cálculo de E_GEN, e E_KILL no

primeiro passo, além de se acrescentar ao mesmo a seguinte para o nó **LABOOL**:

$$E\_GEN[S] = E\_GEN[expbool]$$

Ao contrário do que se fazia com o nó *expbool*, associam-se os atributos IN e OUT ao nó **LABOOL**. Com esta medida, muitas das equações do segundo passo são simplificadas e tornadas mais claras. Assim, exceto para os nós básicos simples e sequências de comandos que não se alteram, o segundo passo passa a contar com as seguintes equações:

nó **LABOOL**:

$$E\_OUT[S] = E\_OUT[expbool]$$

nó IfThenElse:

$$E\_IN[LABOOL] = E\_IN[S]$$
$$E\_IN[S_1] = E\_OUT[LABOOL]$$
$$E\_IN[S_2] = E\_OUT[LABOOL]$$
$$E\_OUT[S] = E\_OUT[S_1] \circ E\_OUT[S_2]$$

nó IfThen:

$$E\_IN[LABOOL] = E\_IN[S]$$
$$E\_IN[S_1] = E\_OUT[LABOOL]$$
$$E\_OUT[S] = E\_OUT[LABOOL] \circ E\_OUT[S_1]$$

nó WhileStm:

$$E\_IN[LABOOL] = E\_IN[S] \circ E\_OUT[S_1]$$
$$= (E\_IN[S] \circ E\_GEN[S_1]) + (E\_IN[S] - E\_KILL[S])$$
$$E\_IN[S_1] = E\_OUT[LABOOL]$$
$$E\_OUT[S] = E\_OUT[LABOOL]$$

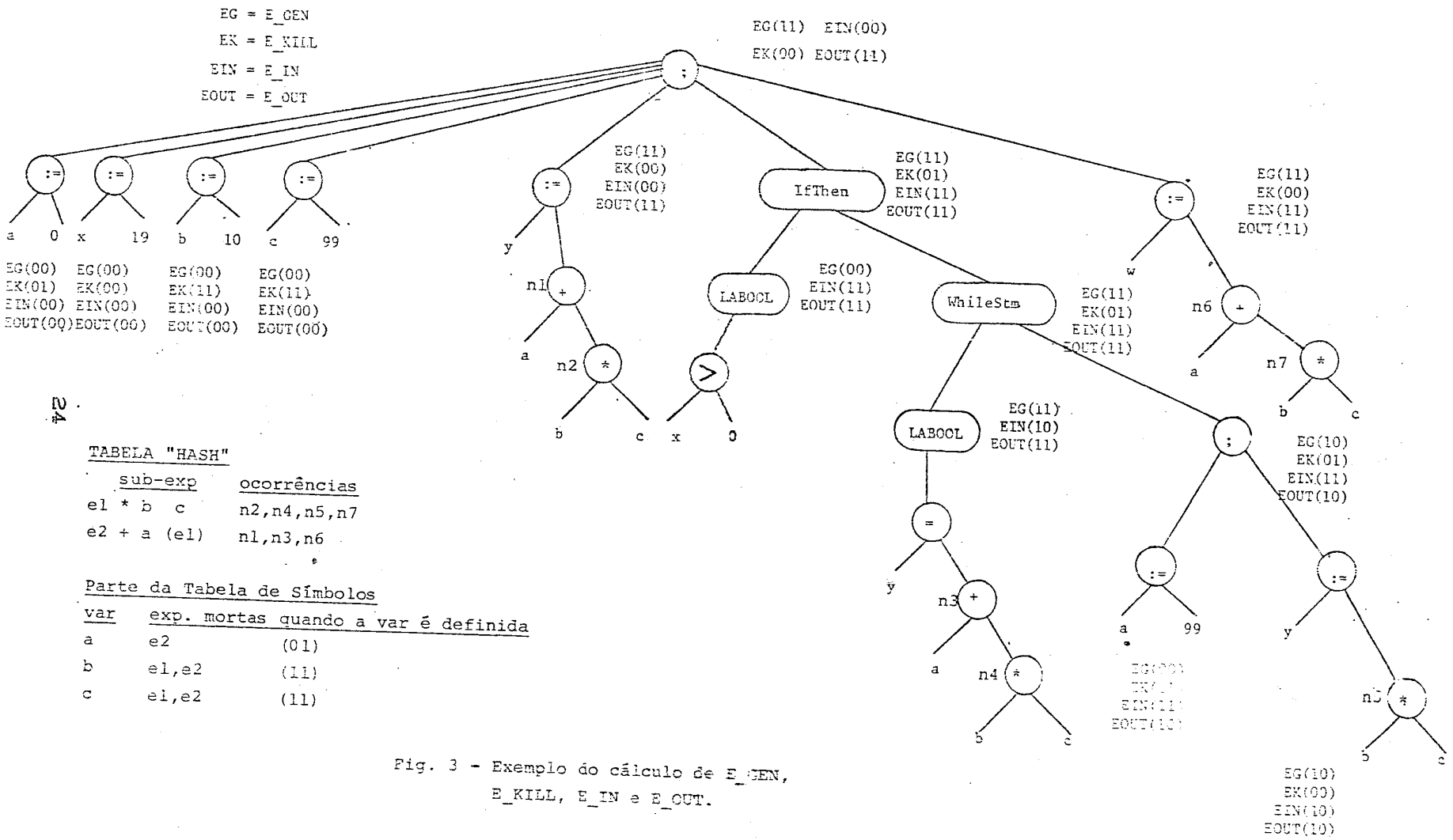
nó RepeatStm:

$$E\_IN[S_1] = E\_IN[S] \circ E\_OUT[LABOOL]$$
$$= E\_IN[S] \circ (E\_OUT[S_1] + E\_GEN[LABOOL])$$
$$= (E\_IN[S] \circ E\_GEN[S_1]) + (E\_IN[S] - E\_KILL[S])$$
$$E\_IN[LABOOL] = E\_OUT[S_1]$$
$$E\_OUT[S] = E\_OUT[LABOOL]$$

Exemplo do cálculo de E_GEN, E_KILL, E_IN e E_OUT

A figura 3 apresenta a árvore abstrata decorada com os valores de atributos E_GEN, E_KILL, E_IN e E_OUT calculados de acordo com os passos 1 e 2 acima. O acesso à cada sub-expressão é realizado através de uma tabela "hash", onde cada entrada apresenta os endereços de ocorrências da sub-expressão correspondente. No exemplo temos duas sub-expressões distintas: e_1, que ocorre na árvore nos nós n2,n4,n5,n7 e_2, que ocorre nos nós n1,n3 e n6. Para cada variável, associa-se um vetor de bits (D$) à sua entrada na Tabela de Símbolos o qual indica que sub-expressões são mortas quando a variável correspondente é definida. Quando uma variável x é definida no escopo de uma atribuição, o atributo E_KILL[atribuição] corresponde ao vetor de bits D$ associado a x. Para facilitar o cálculo, as variáveis x, y e w não estão sendo usadas como operando de sub-expressões.

Pela árvore pode-se observar que o cálculo de E_GEN e EOUT para o nó WhileStm corresponde ao E_GEN e E_OUT do seu nó LABOOL respectivamente. Observa-se, também, que E_OUT[WhileStm] não sofre nenhuma influência da redefinição da variável a, ocorrida na parte de comando do WhileStm. Neste caso particular, no entanto, essa redefinição dá origem ao E_KILL do WhileStm e do IfThen, o que poderia dar origem a repercussões posteriores.



24

TABELA "HASH"

sub-exp	ocorrências
e1 * b c	n2,n4,n5,n7
e2 + a (e1)	n1,n3,n6

Parte da Tabela de Símbolos

var	exp. mortas quando a var é definida	
a	e2	(01)
b	e1,e2	(11)
c	e1,e2	(11)

Fig. 3 - Exemplo do cálculo de E_GEN, E_KILL, E_IN e E_OUT.

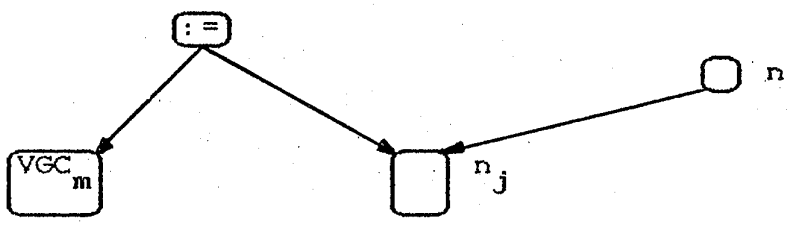
Eliminação de sub-expressões comuns

A solução do problema de expressões disponíveis discutido anteriormente permite determinar se uma expressão no ponto *P* (no grafo do fluxo subjacente) é uma sub-expressão comum (aqui abreviada para *sec*).

Dado uma lista: e_i: n_1, n_2, n_3, ..., n_k

onde $e_i \in EIP(\mathcal{P})$, $1 \leq i \leq n$, n é o número total de sub-expressões distintas ocorrendo no programa $\mathcal{P}$, e n_j corresponde a um endereço do nó raiz de uma ocorrência de e_i, $1 \leq j \leq k$, k número total de ocorrências de e_i; o problema de se determinar se uma ocorrência de sub-expressão é uma sec, e portanto passível de eliminação, se resume em se determinar se e_i está ou não disponível à entrada do nó básico que inclui n_j. Se não estiver disponível, faz-se a seguinte transformação:

1. cria-se uma nova Variável Gerada pelo Compilador: VGC_m, onde m é o número de VGC's já geradas, inicialmente igual a zero; VGC_m corresponde a um nó folha do tipo variável;
2. insere-se um novo nó de atribuição num ponto adequado imediatamente anterior ao nó básico que imediatamente inclui o nó de endereço n_j:



3. faz-se com que o nó n, que já apontava para a sub-árvore com raiz n_j antes da criação da nova atribuição a VGC_m, aponte para um novo nó criado para conter a variável VGC_m.

Se estiver disponível, realiza apenas o passo 3 acima.

O exemplo a seguir é apresentado com o texto linear do programa, mas é equivalente ao resultado obtido com árvore abstrata:

nr. nó	atribuição	exp. disp.	transformações
		$E\_IN[2]=\{\}$	
2 n_1	l := i + 1		VGC_1 := i + 1
		$E\_OUT[2]=\{e_1\}$	l := VGC_1
		$E\_IN[6]=\{e_1\}$	
6 n_2	j := i + 1		j := VGC_1
		$E\_OUT[6]=\{e_1\}$	
	if j > 0 then		
	begin		
		$E\_IN[7]=\{e_1\}$	
7	i := x		
		$E\_OUT[7]=\{\}$	
		$E\_IN[8]=\{\}$	
8 n_3	k := i + 1		VGC_1 := i + 1
		$E\_OUT[8]=\{e_1\}$	k := VGC_1
	end		
		$E\_IN[10]=\{e_1\}$	
10 n_4	m := i + 1		m := VGC_1
		$E\_OUT[10]=\{e_1\}$	
		$E\_IN[12]=\{e_1\}$	
12 n_5	n := i + 1		n := VGC_1
		$E\_OU[12]=\{e_1\}$	

Tem-se apenas a expressão e_1 que ocorre em 5 nós - $n_1(2), n_2(6)$ $n_3(8), n_4(10), n_5(12)$ - de onde são selecionados os seguintes:

por quê?

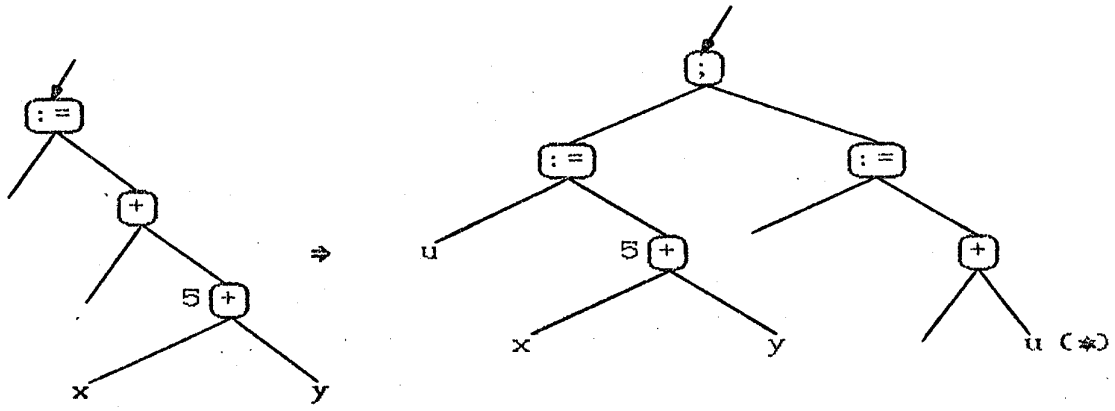
n_1, n_3 → e_1 não está disponível à entrada do nó 8.

→ é a primeira ocorrência e, obviamente, nada está disponível à sua entrada.

Esta abordagem apresenta o seguinte problema. Dado a lista

e_1: 2 3 5 10 12 13 , onde e_1 corresponde a x + y

e supondo que a expressão comum x + y não esteja disponível à entrada do nó de endereço 5, geram-se transformações do tipo seguinte:



Neste caso, se a VGC u não estiver viva imediatamente após o ponto único de substituição (*), terá sido efetuada uma transformação desnecessária pois o valor de u calculado neste ponto não será mais usado. Haveria prejuízo, em termos de espaço, em se realizar a transformação. Diz-se que u está viva no ponto *P* se o valor de u em *P* pode ser usado ao longo de algum caminho no fluxo de controle subjacente após *P*, caso contrário u está morta em *P*.

Como resolver este problema? Neste trabalho o algoritmo proposto não tenta resolvê-lo. Mas pode-se resolvê-lo, tendo-se a informação de análise de variável viva, basicamente através das duas maneiras seguintes: 1.) diretamente na geração de código objeto ou 2.) aperfeiçoando-se o algoritmo de eliminação de sec's. O custo é o maior tempo de processamento dispendido na análise de variável viva. Na linha da segunda alternativa, Mintz et al. (1979) propõem que a análise de variável viva determine que transformações, ao nível da estrutura em árvore, serão necessárias ao se realizar eliminação de sec's. Na conclusão da análise, todas as sec's e estarão substituídas por sua respectiva VGC e esta será

explicitamente atualizada na computação de e sempre que isto seja estritamente necessário. Ele propõe que as atribuições vivas sejam explicitamente colocadas no meio da estrutura das expressões e as mortas sejam eliminadas. Uma atribuição é viva se a VGC u em P que recebe o valor da atribuição for viva .

O algoritmo da figura 4 realiza a eliminação de sec's preconizada até aqui nesta seção, sem tentar resolver o problema acima mencionado. Este algoritmo apresenta complexidade de tempo linear no número de ocorrências de sub-expressões.

```
begin
| for cada tipo de sub-expressão e
| do if número de ocorrências de e > 1    (**)
|    then (***)
|         begin
|         | cria nova variável VGC;
|         | for cada ocorrência de e no nó n
|         | do begin
|         |    | if o valor de e não está disponível à entrada do
|         |    |    nó que inclui n
|         |    | then insere o nó correspondente à VGC := sub-
|         |    |      árvore apontada por n, num ponto adequado
|         |    |      imediatamente anterior ao nó que
|         |    |      imediatamente inclui n;
|         |    | substitui a sub-árvore apontada por n pela VGC
|         |    end
|         end
end
```

Fig. 4 - Algoritmo de eliminação de sec's

Burkhard (1985) apresenta o seguinte aperfeiçoamento ao processo de eliminação de sec's. Antes de verificar se uma sub-expressão e no nó **n** é comum ou não, verifica-se se o padrão formado pelo nó **n** e sua sub-árvore pertence a uma tabela de padrões de árvores. Esta tabela define um conjunto de padrões de árvores que representam os diferentes modos de endereçamento de uma máquina alvo.

Se o nó **n** e sua sub-árvore casarem com qualquer dos padrões da tabela, então esta computação, se redundante, não necessitaria ser eliminada, porque o nó **n** e sua sub-árvore seriam usados pelo gerador de código na emissão de uma instrução. Quando nenhum casamento existir, então o nó **n** e sua sub-árvore poderiam, se redundantes, serem substituídos pela VGC correspondente. No algoritmo acima apresentado, a implementação desta idéia tomaria lugar no ponto marcado por (***) na figura 4.

Um outro aperfeiçoamento seria considerar apenas os $e_i \in EP$ cujo número de ocorrências seja maior que 1. Com isso, os vetores de bits utilizados para o cálculo de expressões disponíveis tornam-se menores, diminuindo o tempo de computação. Como consequência, pode-se eliminar o teste de verificação se uma sub-expressão e ocorre mais de uma vez, indicado por (**) na figura 4.

Estes aperfeiçoamentos propostos não foram implementados, mas merecem ser considerados em implementações futuras. Considerando estas idéias, a figura 5 apresenta o algoritmo revisado.

A transformação eliminação de sec's

Como foi visto anteriormente, a transformação eliminacão de sec's tem duas formas: 1. se a sub-expressão estiver disponível à entrada do nó básico que a inclui, a transformação é simples e consiste apenas em substituir a sec pela VGC correspondente; 2. senão, consiste basicamente em criar um novo nó de atribuição para a VGC correspondente, inserir este nó num ponto adequado e substituir a sec pela sua VGC. Nesta seção identificam-se os pontos adequados bem como passos necessários à realização da

```
begin
| for cada tipo de sub-expressão e {c/nr. oocorrências > 1}
| do if o padrão da sub-árvore do nó n não se
|       encontra na tabela de padrões de árvores
|    then begin
|         | cria nova variável VGC;
|         | for cada ocorrência de e no nó n
|         | do begin
|         |    | if o valor de e não está disponível à entrada do
|         |    |    nó que inclui n
|         |    | then insere o nó correspondente à VGC := sub-
|         |    |      árvore apontada por n, num ponto adequado
|         |    |      imediatamente anterior ao nó que
|         |    |      imediatamente inclui n;
|         |    | substitui a sub-árvore apontada por n pela VGC
|         |    end
|         end
end
```

Fig. 5 - Algoritmo de eliminação de redundâncias

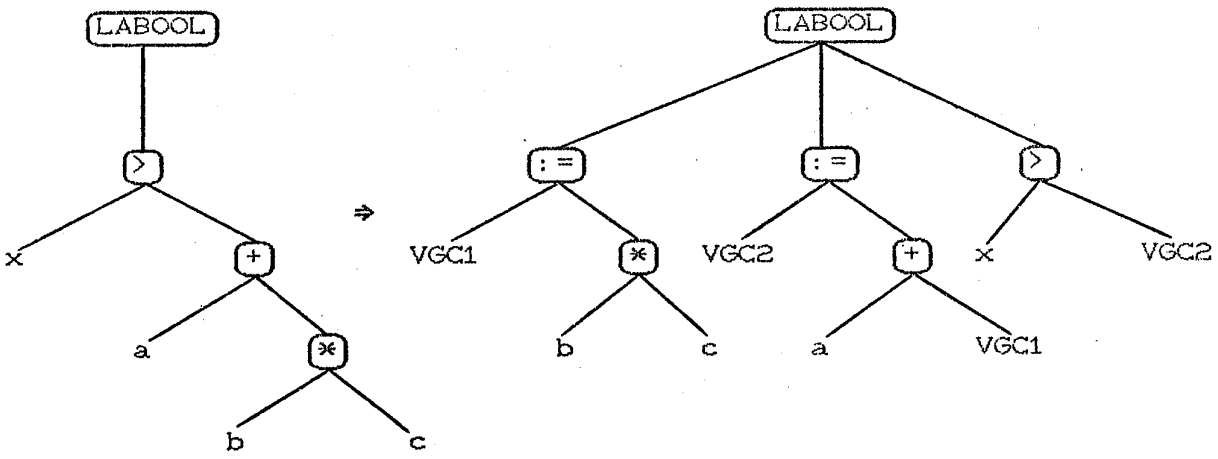
segunda forma de transformação, à qual denominaremos simplesmente de eliminação de sec's..

Basicamente existem três situações onde essa segunda forma de transformação pode tomar lugar:

- dentro de expressões booleanas;
- dentro de nós que têm como pai o nó sequência de comandos e
- dentro de nós isolados, isto é, que não têm como pai o nó sequência de comandos.

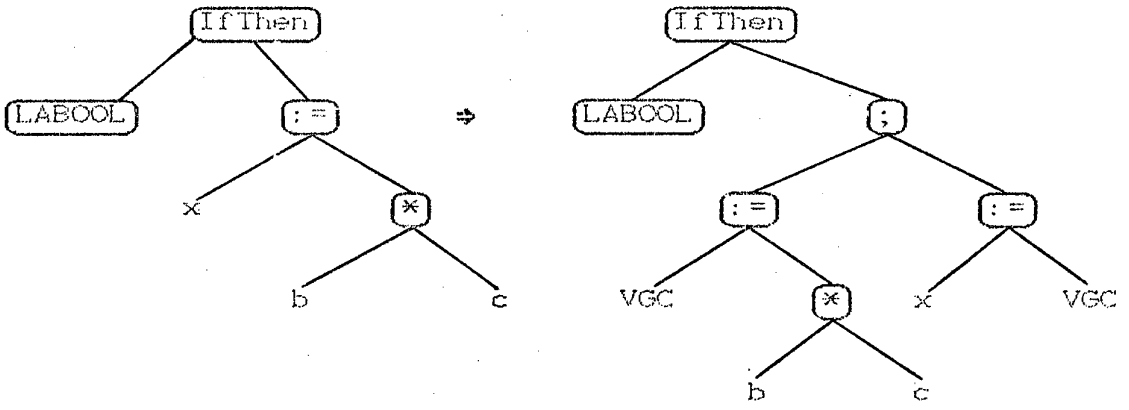
A eliminação de sec's que ocorre dentro de expressões booleanas é facilitada pelo nó LABOOL. No exemplo abaixo estamos supondo que tanto b*c quanto a+b*c não estão disponíveis na entrada de LABOOL.

Ele serve para mostrar como e em que ordem a transformação de sec's é realizada. Primeiramente, coloca-se a atribuição de b*c para VGC1 imediatamente antes do nó *expbool*. A segunda atribuição, de a+VGC1 para VGC2, coloca-se imediatamente antes do nó *expbool*, após a atribuição para VGC1:



A eliminação de sec's que ocorre dentro de nós que têm como pai o nó sequência de comandos é análoga à que ocorre dentro do nó **LABOOL**. Basta enxergar o nó **LABOOL** como um nó sequência de comandos.

A eliminação de sec's que ocorre dentro de um nó isolado simplesmente transforma este nó num nó de sequência de comandos. Imagine que se tem um nó IfThen cujo comando da parte then é um nó de atribuição. Então, ao se realizar uma eliminação de uma sec dentro da parte de expressão da atribuição, cria-se um nó sequência de comandos, no qual se pendura a nova atribuição a uma VGC imediatamente antes da antiga atribuição que já estará com a sua sec substituída pela VGC. A figura abaixo exemplifica esta situação, considerando-se que b*c não está disponível na entrada da atribuição a x.



A figura 6 apresenta a árvore da figura 3 após a execução do algoritmo de eliminação de sec's. Para a sub-expressão e_1 associou-se a VGC1 e para e_2 a VGC2. Criou-se um nó de atribuição da sub-expressão e_1 para VGC1 e eliminaram-se as redundâncias de e_1 em n2,n4,n5 e n7. Criaram-se dois nós de atribuição da sub-expressão e_2 para VGC2 e eliminaram-se as redundâncias de e_2 em n1,n3 e n6.

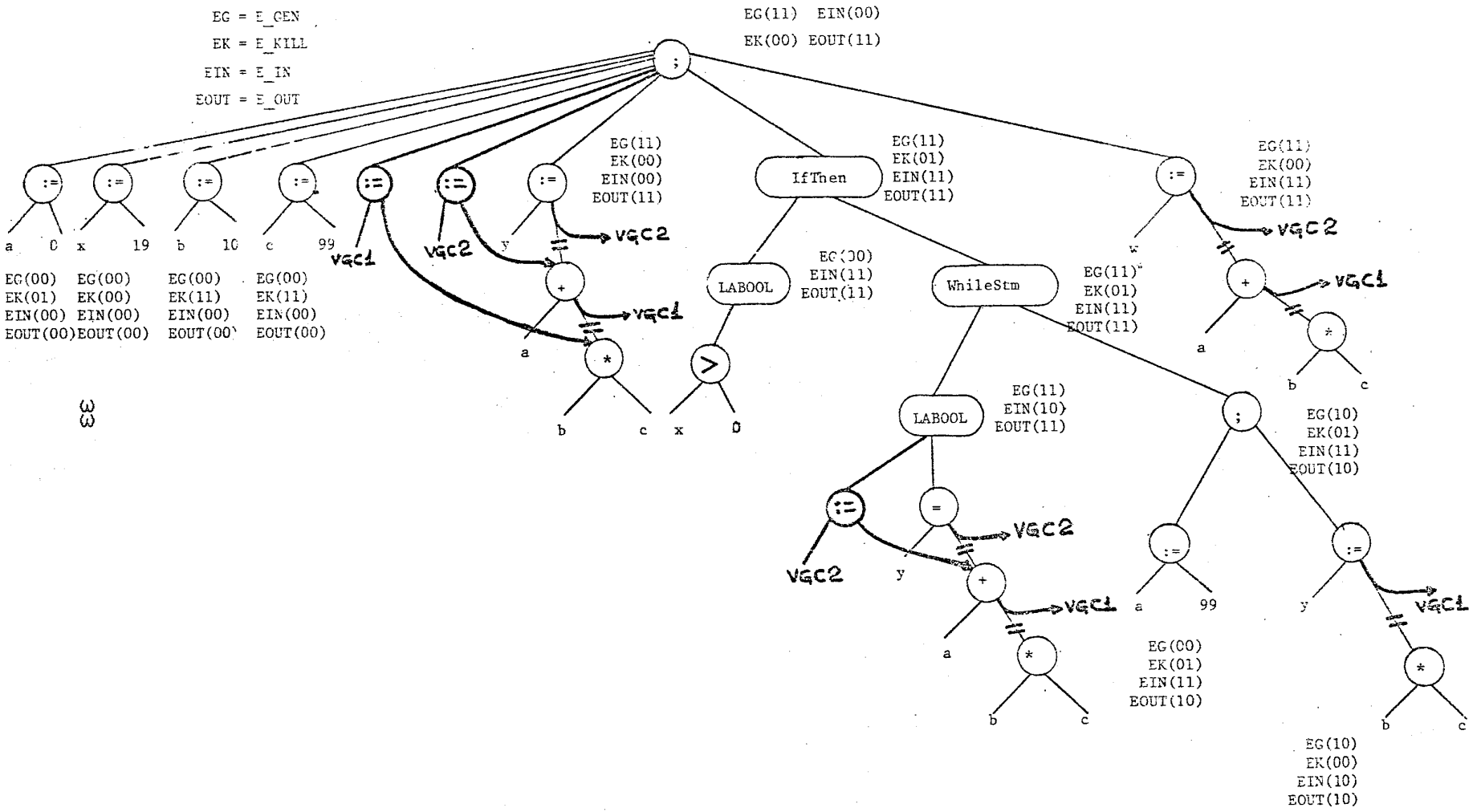


Fig. 6 - Exemplo de eliminação de sec's.

Conclusão

Neste trabalho apresentou-se soluções de dois problemas de fluxos de dados : definições atuantes e expressões disponíveis. O método utilizado foi o de gramáticas de atributos mas, ao invés de se usar árvores sintáticas, como usual, têm-se árvores abstratas como representação intermediária. Isto colocou algumas dificuldades de representação com a usual notação declarativa de gramáticas de atributos. Estes problemas, no entanto, foram superados de maneira ad-hoc.

Apresentou-se um algoritmo de eliminação de sec's que funciona com base apenas na informação de expressões disponíveis. Alguns exemplos foram apresentados para ilustrar a técnica empregada. Discutiu-se , também, possíveis melhorias ao algoritmo proposto.

Agradecimentos: a Paulo Sávio da Silva Costa pelos comentários úteis que ajudaram a esclarecer alguns pontos da versão anterior deste trabalho.

Bibliografia

AHO, A.V.; SETHI, R.; ULLMAN, J.D. Compilers: Principles, Techniques, and Tools. Reading, MA, Addison-Wesley, 1986.

BABICH, W.A.; JAZAYERI, M. The method of atributes for data flow analysis. Acta Informatica, 10:265-272, 1978.

BOCHMANN, G.V. Semantic evaluation from left to right. Communications of the ACM, 19(2):55-62, Feb., 1976.

BURKHARD, N.A. Machine-independent C optimizer. Sigplan Notices, 20(11):23-26, Nov., 1985.

FARROW, R. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. Sigplan Notices, 21(7):85-98, Jul., 1986. Proceedings of the SIGPLAN'86 Symposium on Compiler Construction.

HECHT, M.S. Flow Analysis of Computer Programs. New York, NY, North-Holland, 1977.

KENNEDY, K. A survey of data flow analysis. In: MUCHNICK, S.S.; JONES, N.D., eds. Program flow analysis: theory and applications, Englewood Cliffs, NJ, Prentice-Hall, c1981. cap. 1, p.5-54.

KNUTH, D.E. Semantics of context-free languages. Mathematical Systems Theory, 2(2):127-145, Fev., 1968. Errata 5(1):95-96, Jan., 1971.

KNUTH, D.E. The Art of Computer Programming, Vol. I: Fundamental Algorithms, 3rd edition, Reading, MA, Addison-Wesley, 1976.

MINTZ, R.J.; FISHER, G.A.; SHARIR, M The design of a global optimizer. Sigplan Notices, 14(8):226-234, Aug. 1979. Proceedings of the SIGPLAN'79 Symposium on Compiler

Construction.

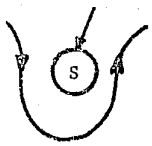
ROSEN, B.K. High-level data flow analysis. Communications of the ACM, 20(10):712-724, Oct , 1977.

Apêndice: Representação da árvore abstrata utilizada

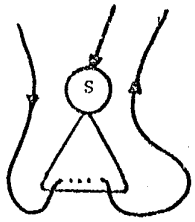
Neste apêndice apresentam-se a sintaxe de cada nó junto com a sua semântica, isto é, com o seu fluxo de controle subjacente, bem como algumas questões de implementação.

Nos comandos WhileStm e IfThen explicitam-se os caminhos possíveis do fluxo de controle. Para efeito de derivação inicial das equações para os problemas de definições atuantes e de expressões disponíveis, deve-se enxergar o nó LABOOL como se fosse o nó *exprbool*.

Quando se apresenta um nó da forma seguinte



quer-se dizer que o fluxo de controle subjacente corresponde ao fluxo da sub-árvore com raiz S:



Para efeito de implementação, a estrutura em árvore n-ária é colocada em forma binária de acordo com Knuth (1976). A figura A1 exemplifica a transformação para árvore binária do nó sequência de comandos e do nó de atribuição. O elo do último filho para o nó pai serve para agilizar o percorrimento na árvore, sem necessidade de recorrer a costuras mais sofisticadas. Desta forma, necessita-se de apenas dois campos em cada nó: elo para filho mais à esquerda e elo para irmão.

Também para efeito de implementação, utiliza-se uma representação compactada da árvore abstrata. Tem-se um nó especial, o "nó quadrado", que basicamente é um nó ponteiro. O objetivo é realizar

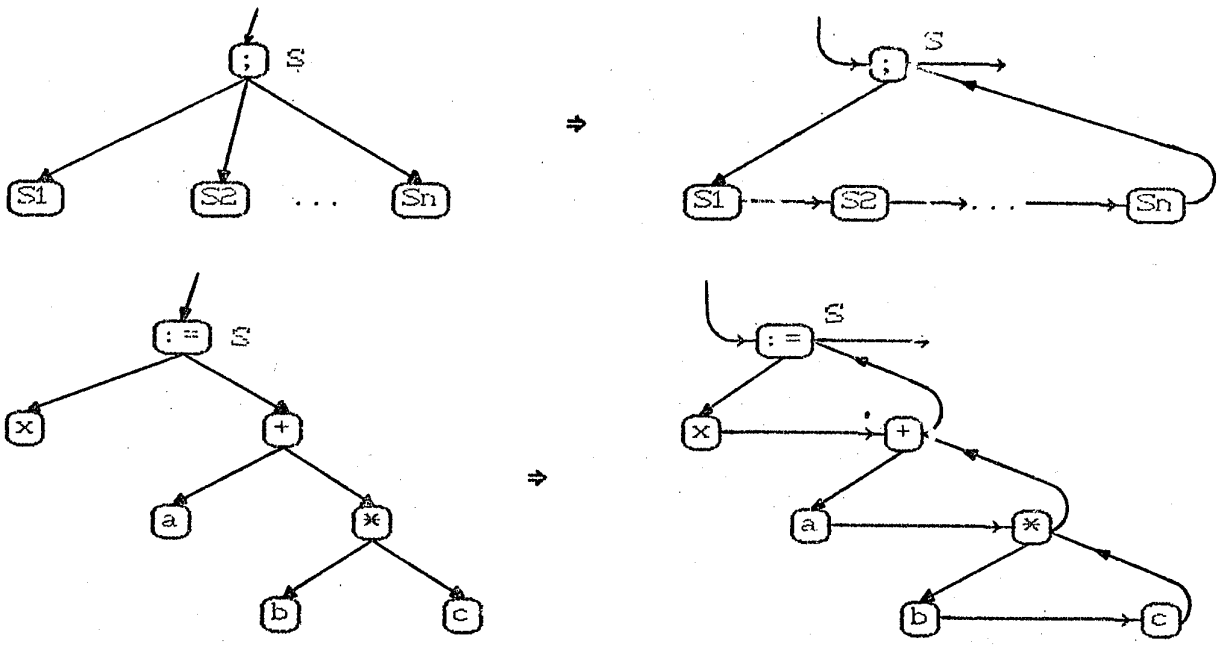


Fig. A1 - Exemplos de árvores n-árias colocadas em forma binária.

uma otimização do espaço ocupado pela representação em árvore do programa. Assim, para cada sub-expressão encontrada em um programa, apenas a sua primeira ocorrência estará explicitamente representada na árvore. Todas as posteriores ocorrências dessa sub-expressãoserão representadas através de um nó quadrado, que aponta para o nó raiz da sua primeira ocorrência. A figura A2 apresenta um exemplo de uma árvore abstrata e a sua representação binária compactada.

No entanto, deve-se atentar para dois fatos importantes. Primeiro, que esta compactação quanto ao espaço físico não faz desaparecer o problema de eliminação de sec's. As árvores continuam as mesmas, apenas estão representadas de forma diferente. Apenas após a transformação eliminação de sec's é que se obtém, possivelmente, nova árvore, também representada em forma compactada. Segundo, que para se conseguir esta compactação, paga-se um preço, ainda que pequeno, no tempo de processamento.

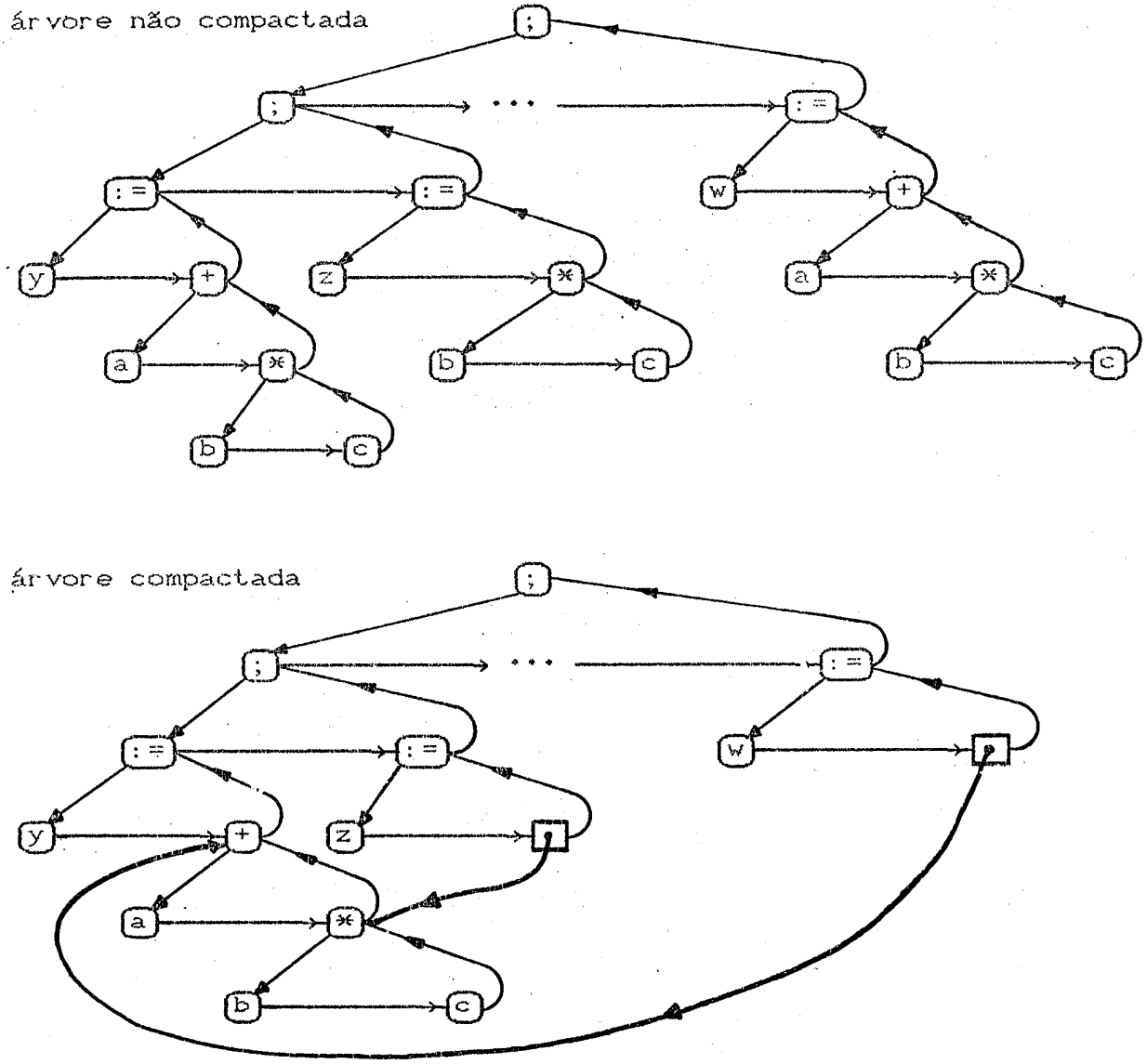


Fig. A2 - Exemplos de árvores não compactada e compactada.

Os fluxos de controle dos nós (comandos estruturados) sequência de comandos, IfThenElse, IfThen, WhileStm e RepeatStm são apresentados a seguir com a seguinte convenção:

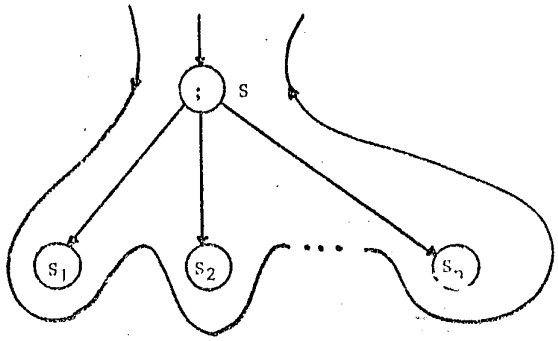
fluxo de controle subjacente

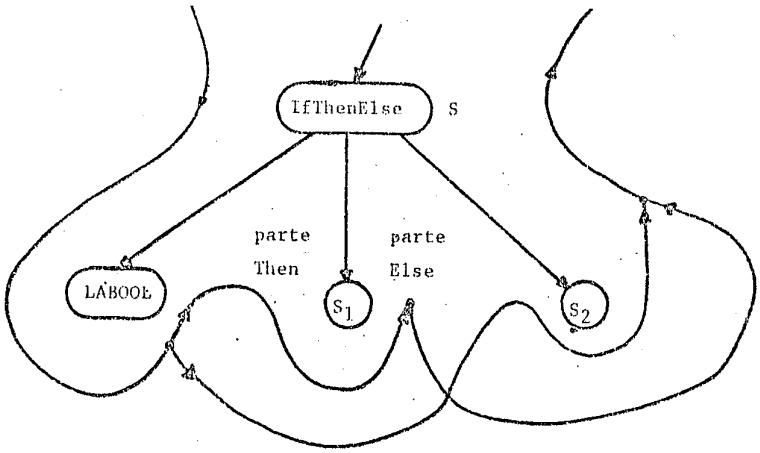
pontos de decisão

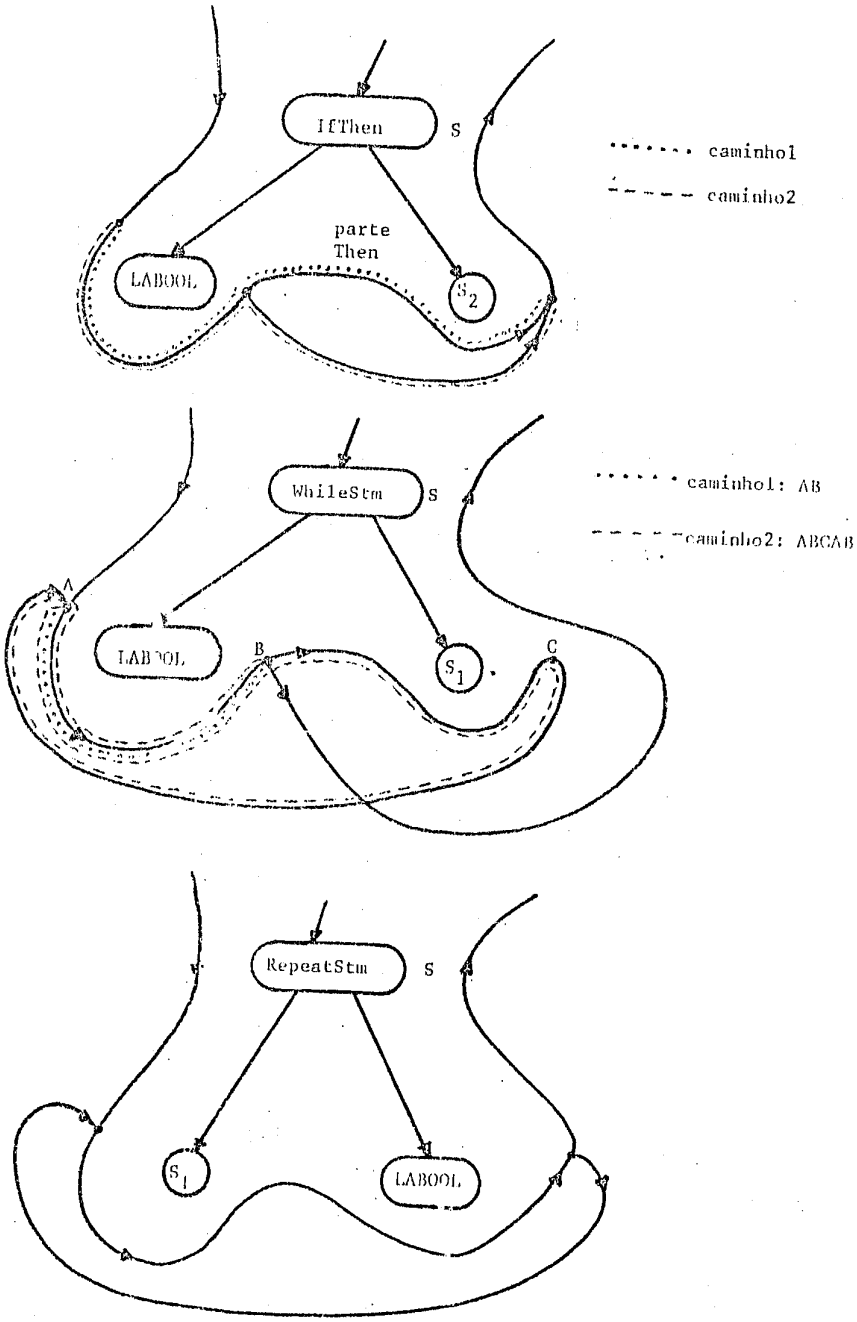
pontos de confluência

Representação utilizada:

nó sequência de comandos






IfThen
S
parte
Then
LABOOL
S_2
caminho1
caminho2
WhileStm
S
LABOOL
A
B
C
S_1
caminho1: AB
caminho2: ABCAB
RepeatStm
S
S_1
LABOOL