Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Tabela de Símbolos

Roberto da Silva Bigonha
Mariza Andrade da Silva Bigonha

Relatório Técnico RT /95

Caixa Postal, 702
30.161 - Belo Horizonte - MG
Julho de 1995

Tabela de Símbolos

Relatório Técnico

Roberto S. Bigonha
Mariza A. S. Bigonha

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte – MG

julho de 1995

Abstract

This paper presents the main and most popular techniques for organizing symbol tables used to implement compilers for block structured high level programming languages.

Resumo

Este artigo apresenta as principais técnicas mais populares para organizar tabelas de símbolos usadas na implementação de compiladores de linguagens de alto nível com estruturas de bloco.

Palavras-chave: função de custo, custo médio, custo no pior caso, tabela linear, árvore binárias, florestas, hash.

Conteúdo

1 Organização da Tabela de Símbolos

Uma tabela de símbolos é uma estrutura de dados cujo propósito é armazenar informações sobre os símbolos encontrados por um compilador durante o processo de compilação de um programa. Cada entrada na tabela de símbolos pode ser separada em duas partes básicas: o nome do símbolo e seus atributos. O nome é, geralmente, um *string* de caracteres alfanuméricos. Os atributos de um nome podem ser *tipo*, *offset*, *nível* etc. O *tipo* refere-se, por exemplo, a integer, real, arranjos, parâmetros, etc. O *nível* refere-se ao local onde o nome aparece no programa e consequentemente ao conjunto de blocos do programa onde o nome é válido.

As informações contidas na tabela de símbolos são usadas em várias fases da compilação a saber:

1. Durante a análise semântica verifica-se se o uso dos nomes estão consistentes com suas declarações implícita ou explícita.
2. Durante a geração de código é usada para saber o quanto e que tipo de armazenamento deve ser alocado para o nome durante a execução.
3. Na recuperação de erro é usada para evitar repetir mensagens de erro do tipo "variável A indefinida"mais de uma vez.
4. Na otimização de código para descobrir temporários usados mais de uma vez, etc.

2 Operações de uma Tabela de Símbolos

As principais operações que uma tabela de símbolos deve realizar são:

- Determinar se um nome está na tabela de símbolos.

- Adicionar nomes à tabela.
- Ter acesso aos atributos de um nome.
- Adicionar novas informações a um nome.
- Remover nomes da tabela.

3 Implementação

A forma mais simples de implementar uma tabela de símbolos é usando um *arranjo linear de records* com um *record* para cada nome. Este método é apropriado se houver um modesto limite superior no comprimento do identificador.

Por exemplo, no Fortran da IBM, o identificador tem no máximo 8 caracteres, o que corresponde ao número de caracteres que cabe em duas palavras do IBM 370. A Figura 1 (a) mostra o formato apropriado para este caso, com o *nome* preenchido com brancos para completar os oito caracteres.

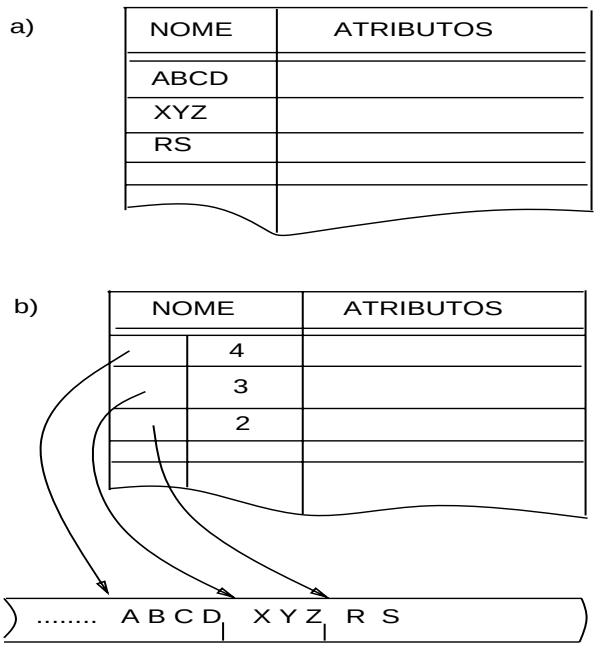

Figura 1: Estrutura da Tabela de Símbolos

Nas linguagens Algol, C e Pascal não existe um limite máximo para o identificador. Na linguagem PL/1, o limite é 31 caracteres. No caso do Algol, C e Pascal, o tamanho é limitado de acordo com a implementação.

Assim, o esquema da Figura 1 (a) não é apropriado para estas linguagens. Para as mesmas deve se usar um esquema de indireção como o mostrado na Figura 1 (b).

Nesta figura (b), na entrada para o identificador $ABCD$ existe um apontador para um outro arranjo de caracteres, a tabela de nomes propriamente dita e um contador, neste exemplo, 4, dando o comprimento do mesmo. **Esta indireção** permite que o tamanho do campo *nome* na tabela de símbolos permaneça uma constante. Não se deve esquecer que o identificador denotando um nome *deve ser armazenado* para garantir que todos os usos do mesmo nome possam ser associados com o mesmo **record** na tabela de símbolos.

A vantagem mais significativa de (b), a indireção, aparece quando temos um tipo uniforme que é aplicado para poucas entradas na tabela.

4 Organizações de Tabelas de Símbolos

Existem pelo menos 4 alternativas principais para implementar uma tabela de símbolos para compilação de Linguagens com Estrutura de Bloco: linear, árvore binária, floresta de árvore binária e hash.

O ponto principal que determina a escolha de uma destas alternativas é a rapidez com que o nome pode ser adicionado e acessado na tabela. Levando em conta este aspecto avaliaremos os quatro esquemas tendo como base o tempo gasto para adicionar `"n"` en-

tradas e fazer "m" pesquisas.

Usaremos o pequeno programa mostrado na Figura 2 para ilustrar as idéias apresentadas.

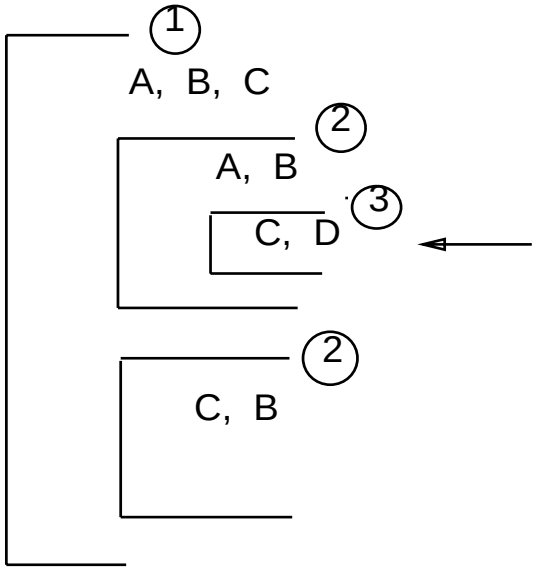

Figura 2: Programa com Estrutura de Blocos

5 Organização: *Tabela Linear*

5.1 Algoritmo para entrar em um bloco

- **Entrada de Bloco:**

  NíVEL := NíVEL + 1

  if NíVEL > NMAX **then** *erro*

  ESCOPO[NíVEL] := L

5.2 Algoritmo para sair de um bloco

- **Saída de Bloco:**

  L := ESCOPO[NíVEL]

  NíVEL := NíVEL − 1

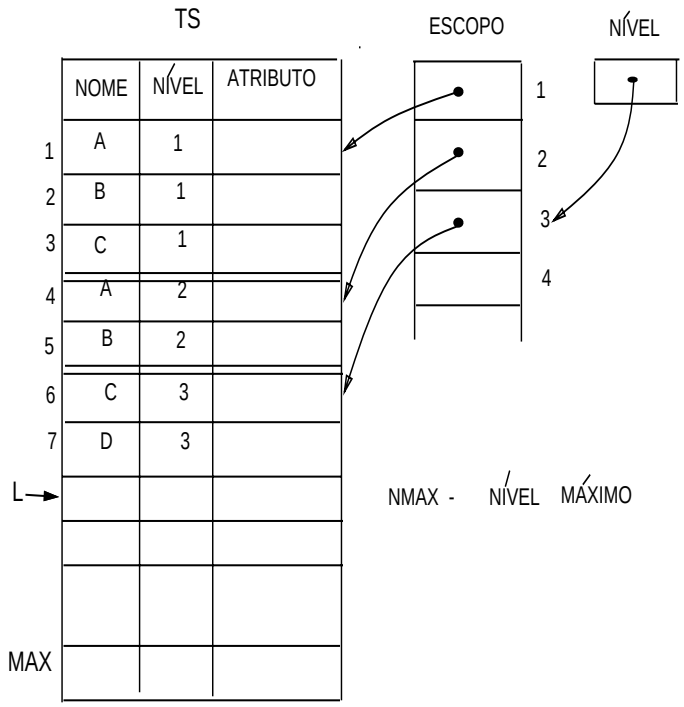

Figura 3: Tabela de Símbolos usando esquema Linear

5.3 Algoritmo para pesquisar por um nome

A pesquisa é efetuada do fim da tabela para seu início.

- **Get-Entry(X):**

K := L (0 − não achou; K − endereço símbolo)
while K > 1 **do**
  K := K − 1
  if X = TS.NOME[k]
  then *return(K)*
end
return(0)

5.4 Algoritmo para inserir um nome

- **INSTALA(X, ATRIBUTO):**

    K := L
  while K > ESCOPO[NíVEL] **do**

```
    K := K − 1
    if X = TS.NOME[k] then erro
  end
  if L = MAX + 1 then erro
  TS.NOME[L] := X
  TS.NíVEL[L] := NíVEL
  TS.ATRIBUTO[L] := ATRIBUTO
  L := L + 1
```

6 Organização: *Árvore Binária*

Um método mais eficiente para organizar uma tabela de símbolos é adicionar dois campos de elo (*links*) Esquerdo e Direito a cada registro. Usamos estes dois elos para ligar os registros como uma árvore binária. *Esta arvore* tem a propriedade que todos os nomes são acessíveis a partir de $name_i$ seguindo o elo à esquerda e então seguindo qualquer seqüência de elos que precede $name_i$ em ordem alfabética.

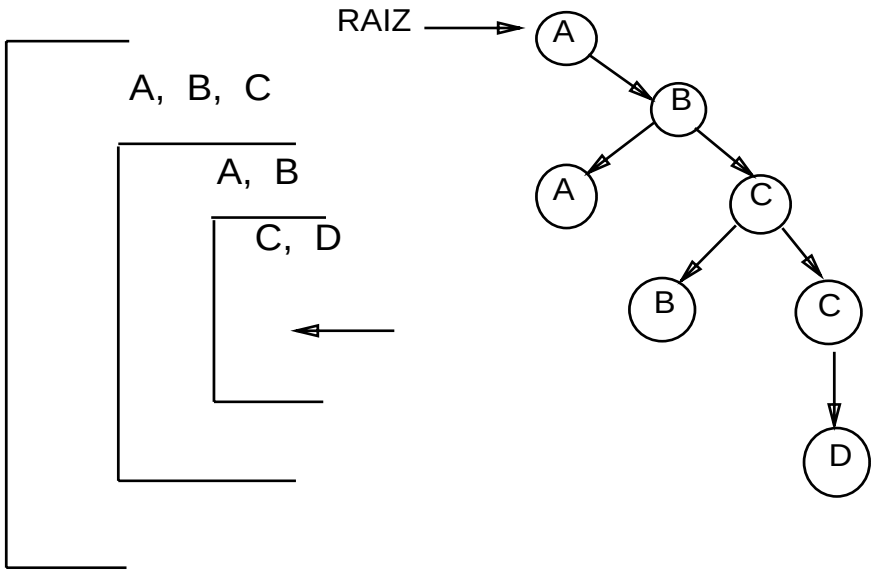

Figura 4: Programa com Estrutura de Blocos Usando Árvore Binária

A Figura 4 mostra um exemplo de um programa com estrutura de blocos e a Figura 5 ilustra como a pesquisa é feita. Se estamos procurando por $name_e$ e encontramos o registro para $name_i$, precisamos somente seguir $Esquerda_i$ $\underline{se}$ $name_e < name_i$ e seguir $Direita_i$ $\underline{se}$ $name_e > name_i$. $\underline{Se}$ $name_e = name_i$ já o encontramos.

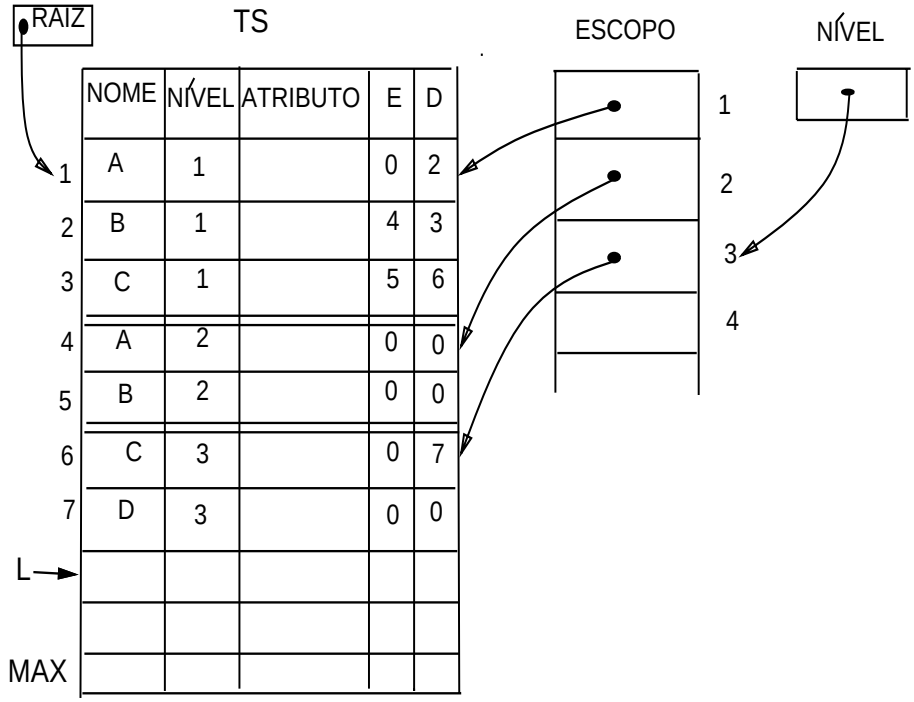

Figura 5: Tabela de Símbolos Usando Árvore Binária

6.1 Algoritmo para entrar em um bloco

• Entrada de Bloco:

NíVEL := NíVEL + 1
if NíVEL > NMAX **then** *erro*
ESCOPO[NíVEL] := L

6.2 Algoritmo para sair de um bloco

- **Saída de Bloco:**

L := ESCOPO[NíVEL] (libera folhas)
if RAIZ $\geq$ L **then** RAIZ := 0
else for I := 1 **to** L $-$ 1
if TS.E[I] $\geq$ L **then** TS.E[I] := 0
if TS.D[I] $\geq$ L **then** TS.D[I] := 0
end
NíVEL := NíVEL $-$ 1

6.3 Algoritmo para pesquisar por um nome

- **GET-ENTRY(X):**

S := RAIZ
K := 0 (0 - não achou; K - endereço de X)
while S $\neq$ 0 **do**
if X = TS.NOME[S] **then** K := S; S := TS.D[S]
else
if X $<$ TS.NOME[S] **then** S := TS.E[S]
else S := TS.D[S]
end
return(K)

6.4 Algoritmo para inserir um nome

Inicialmente pesquisa a árvore pelo identificador X. Dentro do `while` ao fazer K := S, o algoritmo não pára na primeira vez que encontra a variável porque primeiro temos que atingir o nível mais interno. No final da pesquisa **S = O** e **i = posição anterior a S**, ou seja, a posição onde acabou de inserir o elemento.

- **INSTALA(X, ATRIBUTO):**

S := RAIZ; K := 0; i := RAIZ
while S $\neq$ 0 **do**
i := S
if X = TS.NOME[S] **then** K := S; S:= TS.D[S] **else**
if X < TS.NOME[S] **then** S := TS.E[S] **else** S := TS.D[S]
end
if K $\geq$ ESCOPO[NíVEL] **then** *erro*
if L $\geq$ MAX + 1 **then** *erro*
TS.NOME[L] := X; TS.NíVEL[L] := NíVEL
TS.ATRIBUTO[L] := ATRIBUTO; TS.E[L] := TS.D[L] := 0
if RAIZ = 0 **then** RAIZ := L **else**
if X < TS,NOME[i] **then** TS.E[i] := L **else** TS.D[i] := L;
L := L + 1

7 Organização: *Forestas de Árvores Binárias*

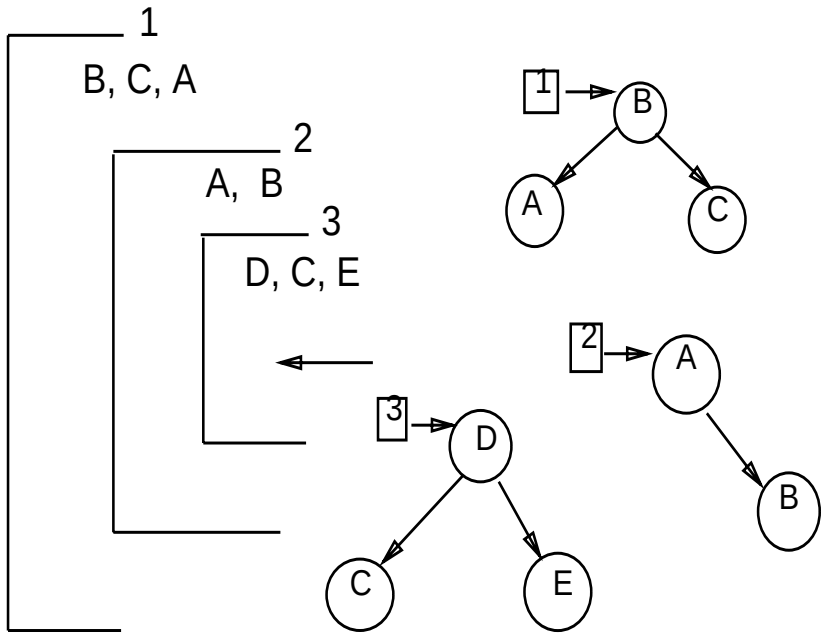

Figura 6: Estrutura de Blocos usando Floresta de Árvore Binária

Esta organização está associada com o fato de que o uso de

variável local ocorre com mais freqüência que variável global, então ao pesquisar primeiro na área local, a chance de encontrar a variável é maior.

Portanto, *floresta de árvores binárias* favorece a pesquisa na área local, só se não encontrar a variável é que vai procurar nos níveis anteriores. Este método está mais de acordo com o *modo* em que os programas são feitos.

Nos outros métodos, por exemplo, no método de árvore binária tem-se que percorrer a árvore toda. Ao encontrar a primeira ocorrência de uma variável não se pode parar a pesquisa porque pode ocorrer uma nova entrada mais interna para a variável.

No exemplo mostrado na Figura 7, se o identificador "A" que estamos procurando é o mais interno, usando o método de árvores binária, teriamos que percorrer a árvore toda.

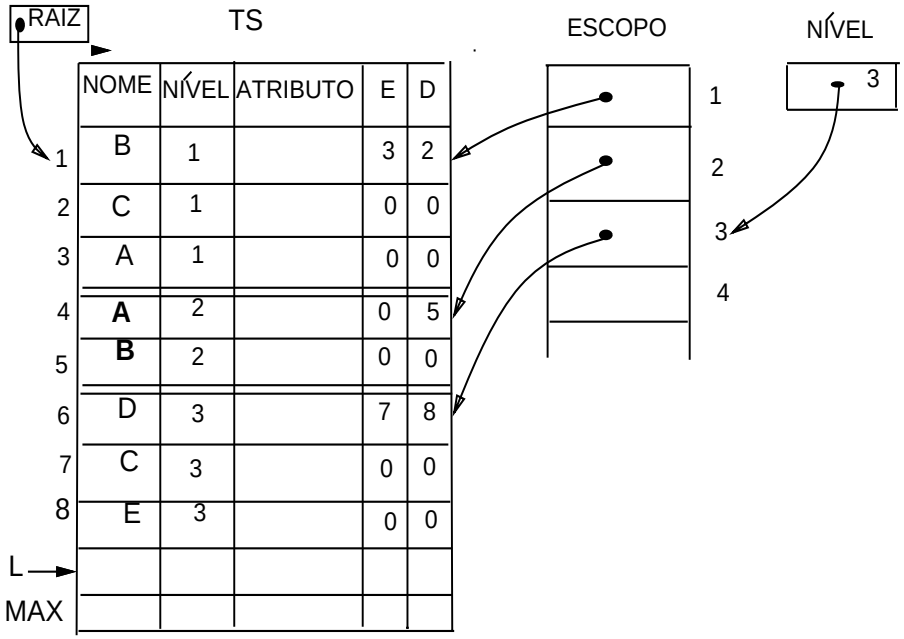

Figura 7: Tabela de Símbolos Usando Floresta de Árvore Binária

Existem pelo menos duas razões para usar floresta de árvores binárias, primeiro porque referência à variáveis locais é supostamente mais frequente que as variáveis globais.

E segundo, porque a pesquisa é feita do fim para o início, não precisa percorrer a árvore toda, como no esquema de árvore binária. Primeiro pesquisa em uma árvore, caso não encontre o nome procurado, então pesquisa em outra árvore.

7.1 Algoritmo para entrar em um bloco

- **Entrada de Bloco:**

  NíVEL := NíVEL + 1
  if NíVEL > NMAX **then** *erro*
  ESCOPO[NíVEL] := 0

7.2 Algoritmo para sair de um bloco

- **Saída de Bloco:**

  if ESCOPO[NíVEL] ≠ 0
  then L := ESCOPO[NíVEL]
  NíVEL := NíVEL − 1

7.3 Algoritmo para pesquisar por um nome

- **GET-ENTRY(X):**

```
n := NíVEL (0 - não achou; K - endereço de X na TS)
while n > 0 do
  K := ESCOPO[n]
  while K ≠ 0 do
    if X = TS.NOME[K] then return(K) else
    if X < TS.NOME[K] then K := TS.E[K] else K := TS.D[K]
  end
  n := n - 1
end; return(0)
```

7.4 Algoritmo para inserir um nome

- **INSTALA(X, ATRIBUTO):**

```
S := ESCOPO[NíVEL]
while S ≠ 0 do
  i := S
  if X = TS.NOME[S] then erro else
  if X < TS.NOME[S] then S := TS.E[S] else S := TS.D[S]
end
if L = MAX + 1 then erro
TS.NOME[L] := X; TS.D[L] := 0; TS.E[L] := 0
if ESCOPO[NíVEL] = 0 then ESCOPO[NíVEL] := L else
if X < TS.NOME[i] then TS.E[i] := L else TS.D[i] := L
L := L + 1
```

8 Organização: *Tabela Hash*

A Organização *hash* consiste de duas partes: a tabela do *hash* e a tabela de símbolos propriamente dita. A tabela *hash* consiste de um arranjo de tamanho fixo com "m" apontadores para as entradas da T.S. Para determinar onde é a entrada do identificador na tabela de símbolos aplicamos a função *hash h(X)*.

Um método de função *hash* que funciona muito bem usa o resto da divisão por M, onde M é o tamanho da tabela: $h(k) = k \ mod \ M$. "K" corresponde ao identificador. Deve-se ter cuidado com a escolha do tamanho de M. Se M é um número par, então *h(k)* é par quando "k" é par, e *h(k)* é impar quando "k" é impar. Portanto, M deve ser um número primo, mas não qualquer primo, devem ser evitados os números primos obtidos a partir de b^i (+/−) j , onde "b" é a base do conjunto de caracteres, 64 para BCD, 128 para ASCII, 256 para EBCDIC ou 100 para alguns códigos decimais.

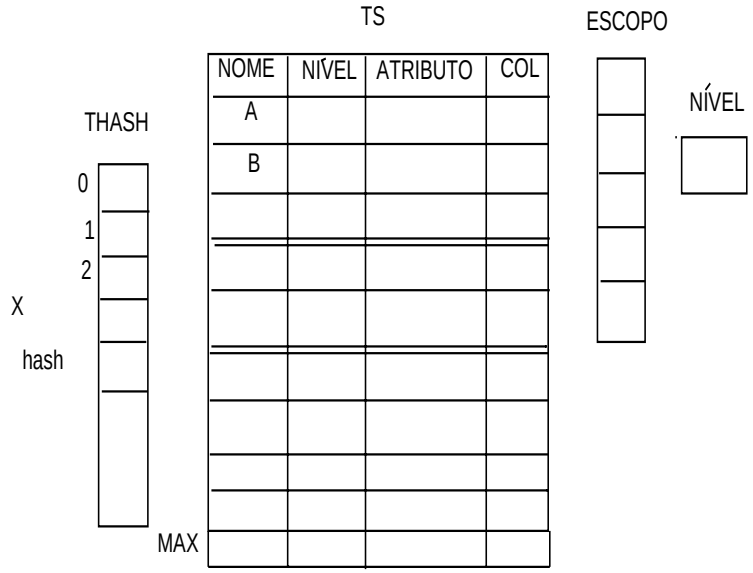

Figura 8: Organização da Tabela de Símbolos Usando *Hash*

`"i"` e `"j"` são pequenos inteiros.

O *hash linear* sofre de um mal chamado agrupamento. Este fenômeno ocorre na medida em que a tabela começa a ficar cheia, pois a inserção de uma nova chave tende a ocupar uma posição na tabela que esteja contigua a outras posições já ocupadas, o que deteriora o tempo necessário para novas pesquisas.

O *aspecto negativo*: se a função de transformação não consegue espalhar os registros de forma razoável pelas entradas da tabela, então uma longa lista linear pode ser formada, deteriorando o tempo médio da pesquisa.

8.1 Algoritmo para entrar em um bloco

• Entrada de Bloco: (*abloco*)

NíVEL := NíVEL + 1
if NíVEL > NMAX **then** *erro*
ESCOPO[NíVEL] := L

8.2 Algoritmo para sair de um bloco

•Saída de Bloco: (*fbloco*)

S := L
B := ESCOPO[NíVEL]
while S > B **do** % desfaz hash
  S := S − 1
  K := *hash*(TS.NOME[S])
  THASH[K] := TS.COL[S]
end
NíVEL := NíVEL − 1
L := B

Quando está gerando o código intermediário é muito comum gerar um código que tem ponteiros para a tabela de símbolos, o que significa que a tabela de símbolos não pode ser descartada, senão perde-se os ponteiros. No método *hash* uma forma fácil de manter a tabela é *nunca liberar a area*, ou seja, não executar a última instrução $L := B$; apresentado no algoritmo para a *Saída de Bloco*. Com isto, a informação continuaria na tabela de símbolos, mas fora do mecanismo do *hash*. Seria como se tivesse um buraco na tabela do ponto de vista do *hash*, ele não enxergaria, mas a informação continuaria lá.

O mesmo acontece com a *organização usando floresta de árvores binárias*, ao sair de um bloco nesta organização, se o comando $L := escopo[nivel]$, não for executado a área não é liberada. Estes dois métodos são portanto recomendados para uso se é preciso preservar as informações da tabela de símbolos durante todo processo de compilação.

Já no método de organização *linear* o mesmo não acontece, não tem jeito porque este método pressupõe que a tabela toda seja

preenchida de forma contígua.

8.3 Algoritmo para pesquisar por um nome

•GET-ENTRY(X)

n := *hash*(X) (0 - não achou; K - endereço de X)
K := THASH[n]
while K $\neq$ 0 **do**
  if X = TS.NOME[K] **then** *return(K)*
  K := TS.COL[K]
end
return(0)

8.4 Algoritmo para inserir um nome

• INSTALA(X, ATRIBUTO):

n := *hash*(X); K := THASH[n]
while K $\geq$ ESCOPO[NíVEL] **do**
  if X = TS.NOME[K] **then** *erro*
  K := TS.COL[K]
end
if L = MAX + 1 **then** *erro*
TS.NOME[L] := X; TS.NíVEL[L] := NíVEL
TS.ATRIBUTO[L] := ATRIBUTO
TS.COL[L] := THASH[n]; THASH[n] := L
L := L + 1

9 Comparação das Organizações

9.1 Parâmetros:

9.1.1 Tempo

Entrada de Bloco:

Todos os métodos são relativamente baratos, não há muito o que ser feito na entrada.

Saída de Bloco:

No método *linear* é só trazer os ponteiros para cima e liberar a área.

No método de *árvores binárias* é um pouco mais caro, porque tem que eliminar os filhos das subárvores que estão na área a ser liberada. Portanto, tem que percorrer a árvore toda e quando o apontador apontar para a área liberada, tem que zerar os ponteiros.

Se for *hash* têm que extrair da tabela *hash* os elementos que estão sendo eliminados. Esta operação é trabalhosa e cara.

Se for *floresta de árvores binárias* não precisa fazer nada, a liberação é bastante eficiente. Basta jogar a árvore fora.

Instalação de Símbolos:

Os métodos *linear* e *árvore binária* seriam os mais caros, nesta ordem. Seriam os mais ineficientes. A tabela na forma de árvore binária deve ser totalmente percorrida, ainda que em pesquisa binária, para se encontrar uma possível ocorrência de um símbolo antes de instalá-lo. Na tabela usando o método linear, a instalação é também muito demorada porque tem que se percorrer todo o vetor a partir do nível mais externo até o nível mais interno.

Nas tabelas usando floresta de árvores binárias ou o método linear são percorridos apenas os símbolos do nível corrente de escopo que são os que interessam neste caso. Nas tabelas usando florestas de árvores binárias a eficiência é maior pois a pesquisa feita é binária *O(log k)*, ao contrário das tabelas lineares onde a pesquisa é *O(k)*, k representa os símbolos do nível de escopo corrente da tabela.

Na tabela usando *hash*, a idéia é um pouco diferente. Na verdade, o número de símbolos lidos para se determinar se um dado nome está na tabela depende mais da função de espalhamento *hash* do que do número de símbolos do nível corrente. Considerando que a função *hash* é boa pode-se dizer que esta pesquisa é da *O(n/1)* onde `"n"` é o número de símbolos instalados e 1 é o número de elementos da imagem da função *hash*.

Procura de Símbolos:

Do ponto de vista de tempo, a entrada e saída no método *linear* é simples e eficiente contudo a pesquisa é cara.

Do ponto de vista dos métodos de pesquisa binária, a entrada e saída são relativamente eficiente também. A pesquisa têm um tempo médio que é compatível com a pesquisa em árvore que é da ordem de $O(logn)_2$. Na verdade é um valor pior que isto porque tem que percorrer a árvore toda, começando do nível mais baixo para o mais alto. Assim quando se encontra um nome igual na tabela não se tem certeza que aquele é o procurado até que se percorra toda a tabela e verifique que aquele símbolo não foi reinstalado em outro nível mais alto de escopo. Portanto, o tempo médio seria a altura da árvore, $O(logn)$.

As tabelas usando o método de floresta de árvores binárias, como o *hash* apresentam um bom desempenho neste item. Aqui, o símbolo é procurado por nível, começando do mais alto para o mais baixo.

Considerando uma boa função *hash* podemos dizer que as tabelas Hash são mais eficientes neste item pois procuram pelo símbolo apenas entre os n/1 símbolos na lista daqueles para os quais o valor retornado pela função *hash* é o mesmo. A desvantagem desta procura é que ela é linear.

Salvar a Tabela de Símbolos

Como já mencionado antes, as vezes é necessário manter a tabela de símbolos porque o código interna a referencia.

No método *linear* é impossível, porque, tudo tem que ser contíguo na tabela. Nos demais métodos isto é perfeitamente possível porque usa-se uma alocação encadeada.

9.1.2 Espaço

Em termos de espaço, o método *linear* apareceria em primeiro lugar, seguindo da organização *hash*. Em terceiro lugar viria o método de *árvore binária* e finalmente a *floresta de árvore binária*.

A tabela linear aparece em primeiro lugar porque todos os campos da mesma contém informações sobre os símbolos propriamnte ditos.

As tabelas baseadas em árvores binárias e florestas de árvores binárias requerem para cada símbolo na tabela dois campos que apontam para as subárvores da direita e da esquerda.

As tabelas *hash* também consomem espaço adicional para armazenar um apontador para o próximo elemento da lista na qual o símbolo está alocado além do espaço gasto para o vetor *hash* que

dá acesso à tabela.

9.1.3 Facilidade de programação

Neste item a organização linear apresentaria novamente uma vantagem sobre os demais, dado que é muito fácil implementá-la, Contudo, levando em conta que o problema de implementar uma tabela de símbolos é bastante simples e que os algoritmos para os demais métodos estão descritos neste artigo, podemos dizer que, quanto a facilidade de programação, todos os métodos apresentados são equivalentes.

10 Conclusão

Durante o processo de compilação de um programa, o número de vezes que uma tabela é acessada à procura de símbolos é bastante alto. Levando em conta que neste item, as organizações *florestas de árvores binárias* e *hash* são as que apresentam os melhores desempenhos, a opção correta na implementação de um compilador seria por uma destas organizações. Das duas, ainda a melhor escolha recairia para floresta de árvores binárias, dada a sua regularidade em eficiência. O que não acontece com o *hash*, uma vez que este é altamente dependente da função hash escolhida.

Referências

[1] Aho, Alfred V. and Sethi, R. and Ullman, J. D., *Compiler Principles, Techniques and Tools*, Addison Wesley Publishing Company, 1986.