TABELA DE SÍMBOLOS

Roberto S. Bigonha e Mariza Andrade Silva Bigonha

UFMG

10 de agosto de 2011

Tabela de Símbolos

Tabelas de símbolos são estruturas de dados usadas pelos compiladores para conter informações sobre as construções do programa fonte.

As informações são coletadas de modo incremental pelas fases de análise de um compilador e usadas pelas fases de síntese para gerar o código objeto.

··· Tabela de Símbolos

- Conteúdo: **Nome**
  Atributos: Offset, Nível, Tipo: integer, real, array, parameter, **etc.**
- Operações principais:
  - 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.

··· Tabela de Símbolos

Quem Cria Entradas na Tabela de Símbolos?

As entradas na tabela de símbolos são criadas e usadas durante a **fase de análise**:

- pelo analisador léxico,
- pelo analisador sintático e
- pelo analisador semântico.

··· Tabela de Símbolos

Observações

- Um reconhecedor sintático normalmente está em melhor posição que o analisador léxico para distinguir entre diferentes declarações de um identificador, devido ao seu conhecimento da estrutura sintática de um programa.
- Em alguns casos, o analisador léxico pode criar uma entrada na TS assim que ler os caracteres que compõem um lexema.

  Normalmente, o analisador léxico só retorna ao analisador sintático um token, por exemplo, id, junto com um apontador para o lexema.

  Porém, somente o reconhecedor sintático pode decidir se usará uma entrada na tabela de símbolos criada anteriormente ou criará uma nova para o identificador.

Tabela de Símbolos por Escopo

"Escopo do identificador x" se refere ao escopo de uma declaração particular de x.

O termo $escopo$ por si só refere-se a uma parte de um programa que é o escopo de uma ou mais declarações.

Importância dos escopos: o mesmo identificador pode ser declarado para diferentes finalidades em diferentes partes de um programa.

Nomes comuns como i e x têm usualmente múltiplos usos.

··· Tabela de Símbolos por Escopo

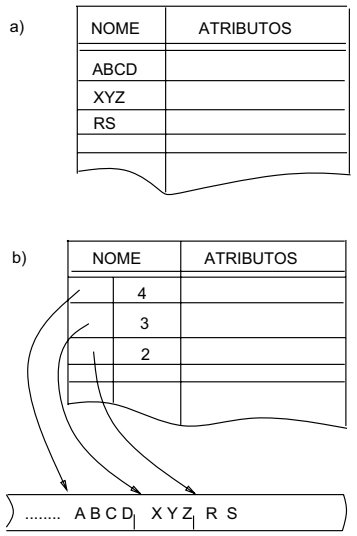
Subclasses podem redeclarar um nome de método que redefine um método de uma superclasse.

Se blocos são aninhados, várias declarações do mesmo id podem aparecer dentro de um único bloco.

Sintaxe de blocos aninhados, quando $stmts$ podem gerar um bloco:

block → " { " decls stmts " }"

Implementações

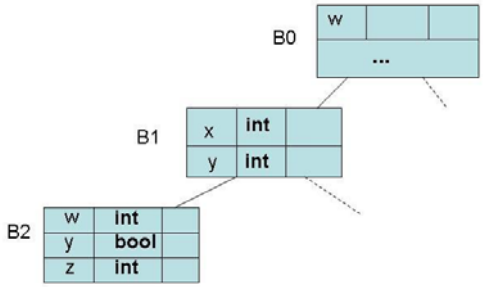


· · · Implementações

O pseudocódigo a seguir usa subscrito para distinguir entre as declarações de um mesmo identificador:

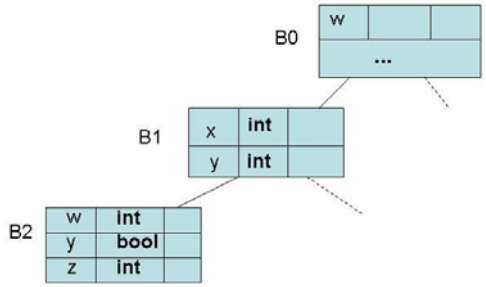
1) {int x_1; int y_1;
2) {int w_2; bool y_2; int z_2;
3) $\cdots\ w_2\ \cdots$; $\cdots\ x_1\ \cdots$; $\cdots\ y_2\ \cdots$;
4) }
5) $\cdots\ w_0\ \cdots$; $\cdots\ x_1\ \cdots$; $\cdots\ y_1\ \cdots$;
6) }

Tabelas de Símbolos para o pseudo-código dado:



· · · Implementações

Tabelas de Símbolos para o pseudo-código dado:



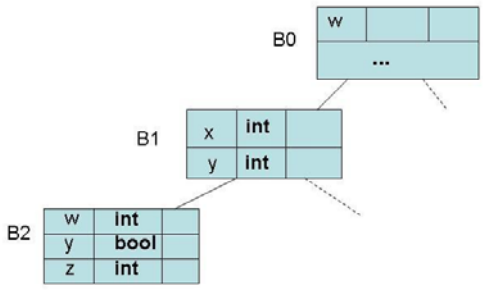
B_1 é para o bloco começando na linha 1 e B_2 é para o bloco começando na linha 2.

No topo da figura há uma tabela de símbolos adicional B_0 para quaisquer declarações globais ou default fornecidas pela linguagem.

Enquanto as linhas 2 a 4 são analisadas, o ambiente é representado por uma referência à tabela de símbolos mais interna - aquela para B_2.

· · · Implementações

Tabelas de Símbolos para o pseudo-código dado:



Na linha 5, a tabela de símbolos para B_2 torna-se inacessível.

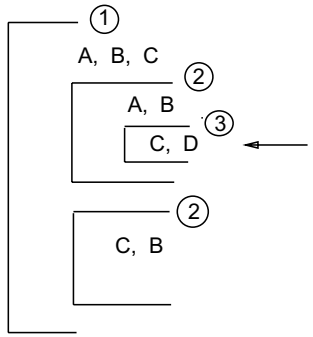
O ambiente se refere à tabela de símbolos para B_1, da qual é possível alcançar a tabela de símbolos global,

mas não a tabela para B_2.

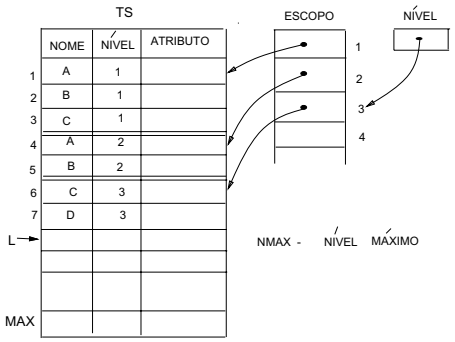
Organização de Tabelas de Símbolos

- Linguagens com Estrutura de Bloco

1. LINEAR
2. ÁRVORE BINÁRIA
3. FLORESTA DE ÁRVORES BINÁRIAS
4. HASH



1. Organização na forma LINEAR



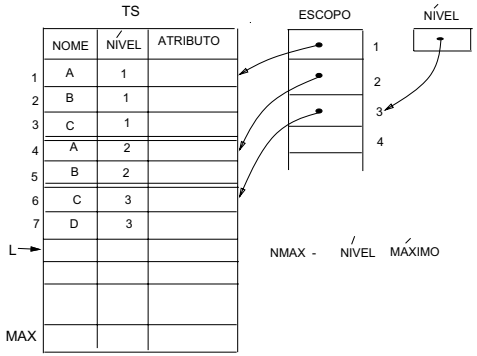
- **1. Entrada de Bloco:**

```
NíVEL  :=  NíVEL + 1
if NíVEL > NMAX  then erro( )
ESCOPO[NíVEL] := L
```

- **2. Saída de Bloco:**

```
L :=   ESCOPO[NíVEL]
NíVEL  :=  NíVEL - 1
```

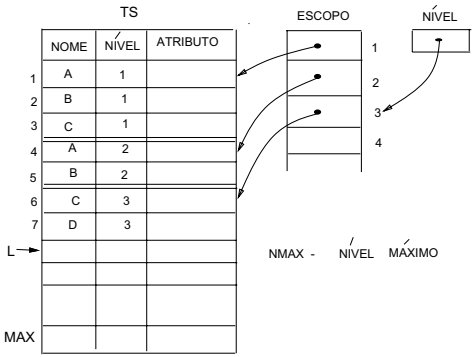

· · · forma LINEAR



3. Get-Entry(X):

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

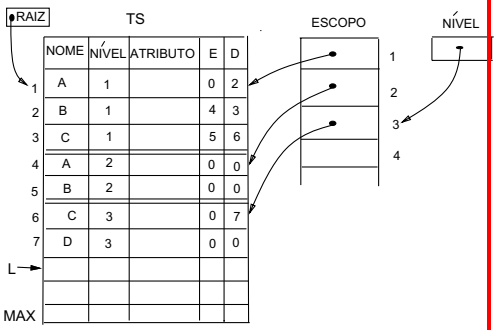

· · · forma LINEAR

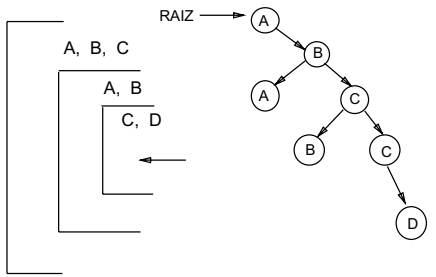


4. INSTALA(X, ATRIBUTO):

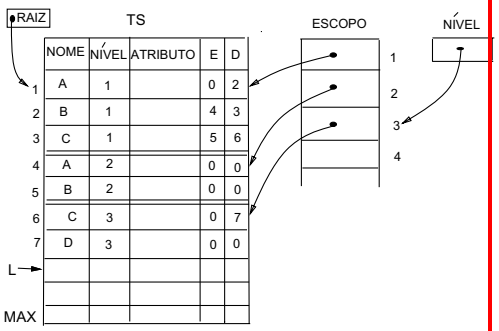
```
K := L
while K > ESCOPO[NíVEL]  do
   K := K - 1
   if X=TS.NOME[k] then erro( )
end
 if L = MAX + 1  then erro( )
TS.NOME[L] := X
TS.NíVEL[L] := NíVEL
TS.ATRIBUTO[L] := ATRIBUTO
L := L + 1
```


2. Organização: ÁRVORE BINÁRIA





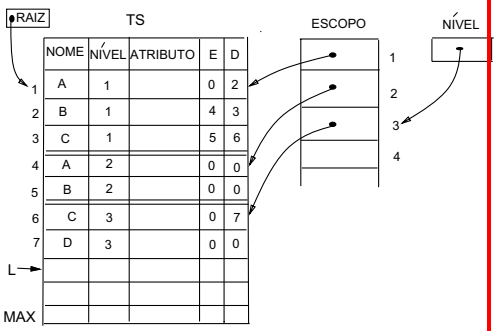
··· ÁRVORE BINÁRIA



1. Entrada de Bloco:

```
NíVEL  :=  NíVEL + 1
if NíVEL > NMAX then erro( )
ESCOPO[NíVEL] := L
```

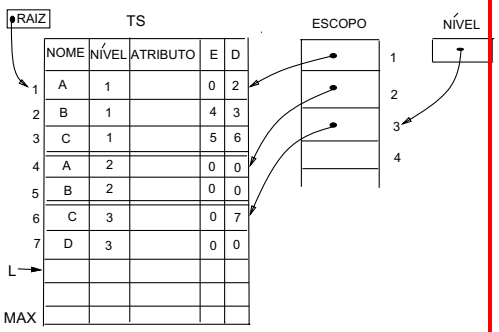

··· ÁRVORE BINÁRIA



- ### 2. Saída de Bloco:

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

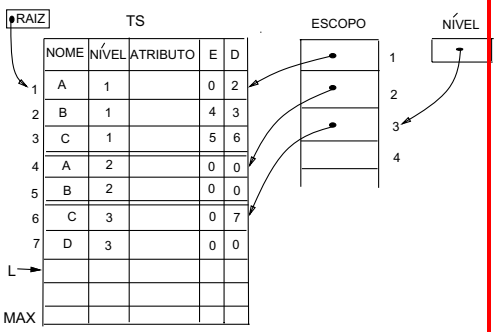

··· ÁRVORE BINÁRIA



3. GET-ENTRY(X):

```
S := RAIZ
K := 0   (0 - não achou;
          K - endereço de X)
while S ¬= 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)
```

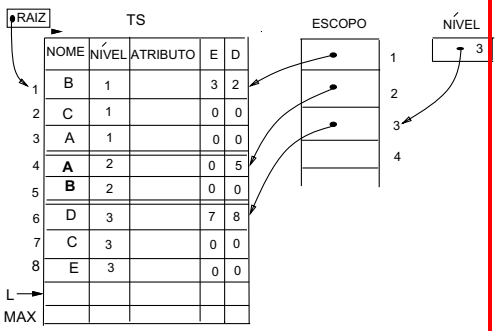

··· ÁRVORE BINÁRIA

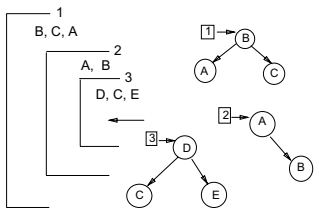


4. INSTALA(X, ATRIBUTO):

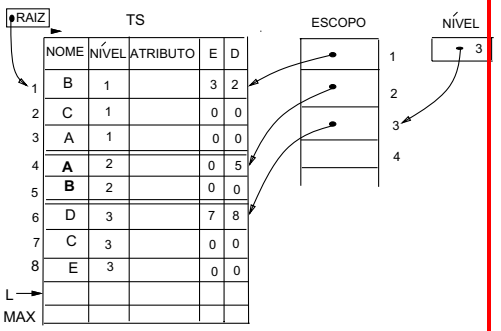
```
S := RAIZ; K := 0; i := RAIZ
while S ¬= 0 do i := S
  if X = TS.NOME[S] then K:=S; S:=TS.D[S]
  else if X < TS.NOME[S]
       then S:=TS.E[S] else S:= TS.D[S]
end
if K >= ESCOPO[NíVEL] then erro( )
if L >= MAX + 1 then erro( )
TS.NOME[L]:= X;  TS.NíVEL[L]:= NíVEL
TS.ATRIB[L]:=ATRIB; TS.E[L]:=TS.D[L]:=0
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
```

3. Organização: FLORESTAS de ÁRVORES BINÁRIAS





··· FLORESTAS de ÁRVORES BINÁRIAS



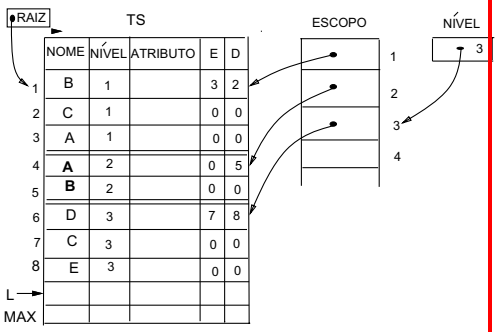
- **1. Entrada de Bloco:**

```
NíVEL := NíVEL + 1
if NíVEL > NMAX then erro( )
ESCOPO[NíVEL] := 0
```

- **2. Saída de Bloco:**

```
if ESCOPO[NíVEL] ¬= 0
then L := ESCOPO[NíVEL]
NíVEL := NíVEL - 1
```

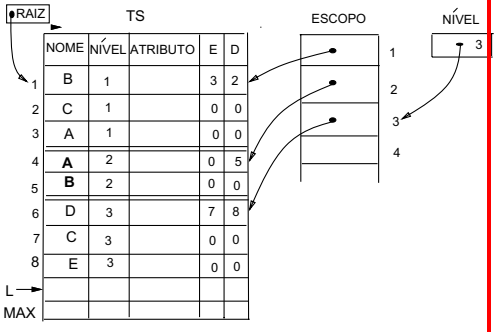

··· FLORESTAS de ÁRVORES BINÁRIAS



3. 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)
```

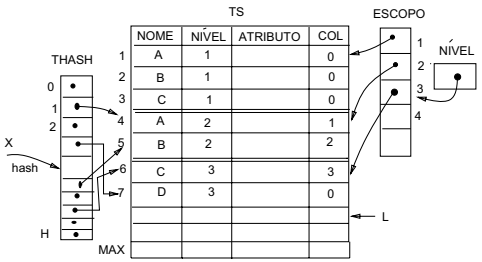

··· FLORESTAS de ÁRVORES BINÁRIAS



4. 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
```

4. Organização: HASH



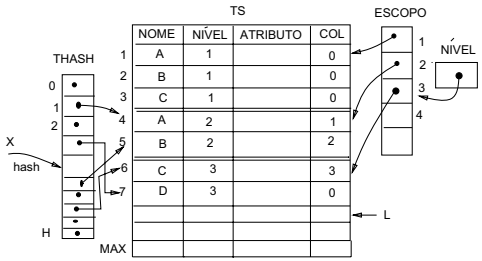
- **1. Entrada de Bloco:**

```
NíVEL  :=  NíVEL + 1
if NíVEL > NMAX  then  erro( )
ESCOPO[NíVEL] := L
```

- **2. Saída de Bloco:**

```
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
```

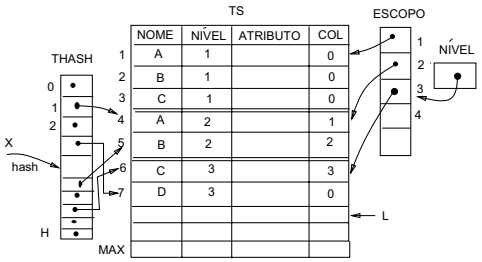

··· HASH



3. GET-ENTRY(X):

```
n:= hash(X)(0:não achou;
                K:endereço de X)
K := THASH[n]
while K ¬= 0 do
     if X = TS.NOME[K]
     then return(K)
     K := TS.COL[K]
end
return(0)
```


··· HASH



4. INSTALA(X, ATRIBUTO):

```
n := hash(X); K := THASH[n]
while K >= 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
```


Comparação das Organizações

- **Parâmetros:**
  - **TEMPO**
    - **Entrada de bloco.**
    - **Saída de bloco.**
    - **Instalação de símbolos.**
    - **Procura de símbolos.**
    - **Salvar a tabela de símbolos.**
  - **ESPAÇO**
  - **FACILIDADE DE PROGRAMAÇÃO**

Comparação das Organizações - TEMPO

Entrada de Bloco:

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

Comparação das Organizações - TEMPO

Saída de Bloco: Linear: necessário atualizar ponteiros e liberar área.

No método de árvores binárias é um pouco mais caro. Os filhos das subárvores que estão na área a ser liberada devem ser eliminados. Portanto, é necessário percorrer a árvore toda a partir da raiz zerando todos os apontadores que endereçam a área liberada.

Se for hash é necessário extrair da tabela hash os elementos que estão sendo eliminados. Esta operação, como no método de árvores binárias é trabalhosa e cara.

Se for floresta de árvores binárias a liberação é bastante eficiente. Basta jogar a árvore fora, ou seja, atualizar os apontadores do nível de escopo.

··· Comparação das Organizações - TEMPO

Instalação de Símbolos:

O método de árvore binária é o mais caro e o mais ineficiente. A tabela na forma de árvore binária deve ser totalmente percorrida a partir da raiz, ainda que em pesquisa binária, para se encontrar uma possível ocorrência de um símbolo antes de instalá-lo.

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.

··· Comparação das Organizações - TEMPO

··· Instalação de Símbolos:

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 uma função hash 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.

··· Comparação das Organizações - TEMPO

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, pois é necessário percorrer todo o vetor do nível mais interno para o nível mais externo.

··· Comparação das Organizações - TEMPO

Procura de Símbolos: Do ponto de vista dos métodos de pesquisa binária, a pesquisa têm um tempo médio que é compatível com a pesquisa em árvore que é da ordem de $O(log_2 n)$.

Na verdade é um valor pior que isto porque tem que percorrer a árvore toda a partir da raiz, ou seja, 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, $O(logn)$.

··· Comparação das Organizações - TEMPO

··· Procura de Símbolos:

As tabelas usando tanto 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 pode-se dizer que o esquema usando hash é mais eficiente neste item pois procura-se 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.

··· Comparação das Organizações - TEMPO

Salvar a Tabela de Símbolos

As vezes é necessário manter a tabela de símbolos porque o código interno a referencia.

Usando o método linear é impossível preservá-la, porque tudo tem que ser contíguo na tabela.

Nos demais métodos isto é perfeitamente possível porque usa-se uma alocação

· · · Comparação das Organizações - ESPAÇO

Em termos de espaço, o método linear aparece em primeiro lugar, seguindo da organização hash. Em terceiro lugar vem 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 propriamente ditos.

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

Tabela hash também consome espaço adicional para armazenar um apontador para o próximo elemento da lista na qual o símbolo está alocado além do espaço gasto para vetor hash que dá acesso à tabela.

Facilidade de Programação

Conclui-se que 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 vários métodos já estão descritos, pode-se dizer que, quanto a facilidade de programação, todos os métodos apresentados são equivalentes.

· · · Tabela de Símbolos

Esta seção mostra um módulo de tabela de símbolos adequado para ser usado em sua íntegra com o tradutor do Apêndice A do livro texto.

Linguagem reduzida: apenas possui as principais construções que usam as tabelas de símbolos; blocos, declarações e fatores.

Um programa nesta linguagem consiste em blocos com declarações opcionais e "comandos" consistindo unicamente em identificadores.

Cada comando desse tipo representa um uso do identificador.

Exemplo de um programa nessa linguagem:

```
{ int x; char y; { bool y; x; y; } x; y; }        (1)
```


Implementação Java das Tabelas de Símbolos

```
1)  package symbols;                    // File Env.java
2)  import java.util.*;
3)  public class Env {
4)     private Hashtable table;
5)     protected Env prev;
6)     public Env(Env p) {
7)        table = new Hashtable(); prev = p;
8)     }
9)     public void put(String s, Symbol sym) {
10)       table.put(s, sym);
11)    }
```

··· Implementação Java das Tabelas de Símbolos

```
                // ··· File Env.java
12)   public Symbol get(String s) {
13)      for( Env e = this; e != null; e = e.prev ) {
14)         Symbol found = (Symbol)(e.table.get(s));
15)         if( found != null ) return found;
16)      }
17)      return null;
18)   }
19)}
```


··· Implementação Java das Tabelas de Símbolos

A classe ENV admite 3 operações:

- Criar uma nova tabela de símbolos.

  O construtor $Env(p)$ nas linhas 6 a 8 do algoritmo cria um objeto Env com uma tabela hash chamada $table$.

  O objeto é encadeado ao parâmetro com valor de ambiente p definindo-se o campo $prev$ como p.

  Embora sejam os objetos Env que formam uma cadeia, é conveniente falar das tabelas sendo encadeadas.

··· Implementação Java das Tabelas de Símbolos

- Colocar uma nova entrada na tabela corrente. A tabela $hash$ contém pares chave e valor, onde:
  - A $chave$ é uma seqüência de caracteres, ou então uma referência a uma seqüência de caracteres.

    Outra alternativa seria usar referências a objetos de token para identificadores como chaves.
  - $Valor$ é uma entrada da classe $Symbol$.

    O código nas linhas 9 a 11 não precisa conhecer a estrutura de uma entrada; ou seja, o código é independente dos campos e métodos na classe $Symbol$.

··· Implementação Java das Tabelas de Símbolos

- Recuperar uma entrada para um identificador pesquisando a cadeia de tabelas, começando com a tabela para o bloco corrente.

  O código para essa operação nas linhas 12 a 18 retorna uma entrada da tabela de símbolos ou $null$.

O encadeamento de tabelas de símbolos resulta em uma estrutura de árvore, pois mais de um bloco pode estar aninhado dentro de um bloco que o envolve.

O Uso de Tabelas de Símbolos

O papel de uma tabela de símbolos é passar informações de declarações para usos.

Uma ação semântica "entra" com informações sobre o identificador x na tabela de símbolos, quando a declaração de x é analisada.

Subseqüentemente, uma ação semântica associada a uma produção como factor $\rightarrow$ id "recupera" a informação sobre o identificador da tabela de símbolos.

··· O Uso de Tabelas de Símbolos

Como a tradução de uma expressão

E_1 op E_2,

para um operador op típico, depende apenas das traduções de E_1 e E_2, e não depende diretamente da tabela de símbolos,

é possível acrescentar qualquer quantidade de operadores sem alterar o fluxo básico de informações de declarações para usos, por meio da tabela de símbolos.

··· O Uso de Tabelas de Símbolos

O esquema de tradução ilustra como a classe Env pode ser usada. O esquema de tradução se concentra em escopos, declarações e usos.

Ele implementa a tradução descrita a seguir. Na entrada

$$\{\ int\ x;\ char\ y; \{\ bool\ y;\ x;\ y;\ \}\ x;\ y;\ \}$$

o esquema de tradução retira as declarações e produz

$$\{\ \{\ \text{x:int; y:bool;}\ \}\ \text{x:int; y:char;}\ \}$$

··· O Uso de Tabelas de Símbolos

$program \rightarrow$ $block$ { top = **null**; }

$block \rightarrow$ '{' { $saved$ = top; top = **new** Env(**top**); **print**("{ "); }

$decls\ stmts$ '}' { top = $saved$; **print**("} "); }

$decls \rightarrow$ $decls\ decl$

| ϵ

$decl \rightarrow$ **type id** ; { s = **new** $Symbol$; s.$type$ = **type**.$lexeme$; top.put(**id**.$lexeme$, s); }

··· O Uso de Tabelas de Símbolos

$$
\begin{array}{lll}
stmts \rightarrow & stmts\ stmt & \\
\quad | & \epsilon & \\
 & & \\
stmt \rightarrow & block & \\
\quad | & factor\ ; & \{\ \textbf{print}(\texttt{"; "});\ \} \\
 & & \\
factor \rightarrow & \textbf{id} & \{\ s = top.get(\textbf{id}.lexeme); \\
 & & \quad \textbf{print}(\textbf{id}.lexeme); \\
 & & \quad \textbf{print}(\texttt{":"}); \\
 & & \quad \textbf{print}(s.type);\}
\end{array}
$$

FIM