# Introdução

- Uma linguagem de programação apoiada em um paradigma imperativo apresenta algum grau de dificuldade nos aspectos relativos ao contexto;
  - Nas linguagens imperativas o processamento é baseado no acesso freqüente às posições de memória, tanto para recuperar quanto para armazenar valores;
  - Uma variável deve ser declarada antes de ser utilizada, para que se possa efetuar a verificação adequada dos tipos (da variável e da expressão);
  - Um sub-programa que utiliza alguma variável do programa no qual foi declarado não precisa mencionar o escopo de definição desta variável;
- Sob diversos pontos de vista a questão do contexto é crucial.

# 1. Dependência de Contexto

- Em uma linguagem natural o problema da identificação e uso correto do contexto é particularmente complexo.
- Os elementos de um discurso remetem a partes distantes do discurso (escrito ou falado), no qual o interlocutor é responsável por identificar.
  - Em qualquer conversa informal termos, nomes, eventos às vezes não são sequer explicitamente mencionados.
- O conhecimento do interlocutor a respeito de elementos do discurso podem eliminar ambigüidades do discurso.
  - O contexto do discurso, por exemplo. (Uma palestra sobre área específica).
- O tratamento do contexto é complexo.
- A dependência de contexto não é uma característica exclusiva das linguagens naturais.

# O Problema do Contexto

- Há características da estrutura das linguagens de programação que não podem ser descritas usando sintaxe livre de contexto:
  - Declaração de variáveis antes de seu uso.
  - Declarações end <nome> em linguagens (como ADA) no qual <nome> deve coincidir com o nome de algum sub-programa.
  - A compatibilidade de tipos poderia, em tese, ser feita, mas a complexidade é muito grande.
  - São características dependentes de contexto.
- A verificação destas características sintáticas não pode ser feita por qualquer dispositivo livre de contexto.
- Denominou-se de semântica estática este conjunto de tarefas a serem executadas (que nada têm a ver com semântica).
  - O motivo é o processo usado para tratar as questões.

# O Problema do Contexto – cont.

- Uma linguagem cujas construções precisem de informações presentes em partes distintas de suas cadeias é dita dependente de contexto.
  - Ex.: $a^n b^n c^n$, o valor de n deve coincidir nas três sub-cadeias (prova-se usando o bombeamento para linguagens LC que esta linguagem não é LC).
- Portanto as questões anteriores, quando aplicadas a linguagens de programação, ilustram sua dependência de contexto.
  - O uso de gramáticas ou de *parsers* dependentes de contexto para estas linguagens torna-se, freqüentemente impraticável para o tratamento de linguagens de programação.
- Houve duas tentativas com uso de técnicas diferentes:
  - Gramáticas de dois níveis.
  - Gramática de atributos.

# 2. Gramática de Atributos

- Construídas para a definição de sintaxe dependente de contexto, e construção de compiladores.
- Uma gramática de atributos é uma definição sensível ao contexto dirigida pela sintaxe livre de contexto e é explicitada de duas formas:
  - *Definições dirigidas por sintaxe* estendem gramáticas livres de contexto através da inclusão de atributos para cada símbolo da gramática e regras semânticas, sem especificar detalhes de implementação (evidencia detalhes de **definição**);
  - *Esquema de tradução* indicam a ordem na qual as regras que especificam ações semânticas devem ser executadas durante a fase de análise sintática (evidencia detalhes de **implementação**);
  - O uso destas definições pode ser feito dentro de uma ferramenta de geradores de analisadores (yacc, antlr, etc.), porque pode ser diretamente mapeada para atribuições de variáveis e busca em árvore e tabelas.

# Gramática de Atributos

- É uma gramática livre de contexto acrescida de:
  - Para cada símbolo gramatical há um conjunto de valores de atributos;
  - Cada regra tem um conjunto de funções que define certos atributos dos não-terminais da regra;
  - Cada regra tem um conjunto (possivelmente vazio) de predicados para verificar a consistência de atributos;
- Inicialmente há atributos intrínsecos nas folhas da árvore sintática;
- Ex.: Uma expressão como: <id> + <id> deve ser verificada quanto ao tipo, portanto tipo é um atributo de <id>.

# Atributos

- Acrescentam informação aos símbolos da gramática que representam alguma construção da linguagem;
- Seus valores são calculados através de regras semânticas;
- Considere uma produção de uma gramática livre de contexto da forma:
  - $Y \rightarrow X_1 X_2 ... X_n$, atributos são associados a cada instância de um símbolo gramatical, ou seja, a cada nó na árvore sintática;
  - Um atributo b é calculado: $b = f(c_1, c_2, ..., c_k)$, $f$ é a função;
  - Exemplos de atributos podem ser valores, tipos, tamanhos de nomes, etc;
- Para cada instância distinta de um símbolo gramatical haverá uma instância separada do atributo;
- Atributos podem herdados ou sintetizados.

# Atributos (cont.)

- Os valores dos atributos sintetizados para tokens são determinados pelo analisador léxico. Define-se que tokens não possuem atributos herdados.
  - Comando: comandoatribuição
  - comandoatribuição: var ‘=’ expr;
- O tipo da expressão é *herdado* do tipo da variável.
- Ex.: Em Java é ilegal
  - amInteiro = 2.3 + 3.4; // tipo de amInteiro é int

# Atributos (cont.)

- Um atributo de um símbolo gramatical é dito *sintetizado* se este é computado através dos atributos de seus descendentes em uma árvore de derivação sintática.
- Exemplo (obtém o somatório dos dígitos):

  $N \rightarrow N_1$ num  {N.val = $N_1$.val + num.lexval}

  $N \rightarrow$ num  {N.val = num.lexval}

- Um atributo de um símbolo gramatical é dito herdado se este é computado através dos atributos de seu ancestral (pai) ou de seus irmãos.
- Exemplo (obtém o somatório dos dígitos):

  $N \rightarrow$ num  {X.h = num.lexval}  X  {N.val = X.s}

  $X \rightarrow$ num  {$X_1$.h = X.h + num.lexval}  $X_1$  {X.s = $X_1$.s}

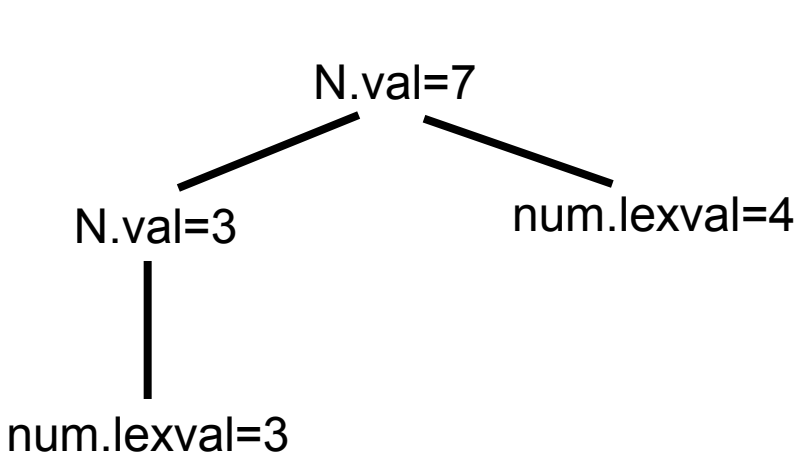
  $X \rightarrow \varepsilon$  {X.s = X.h}

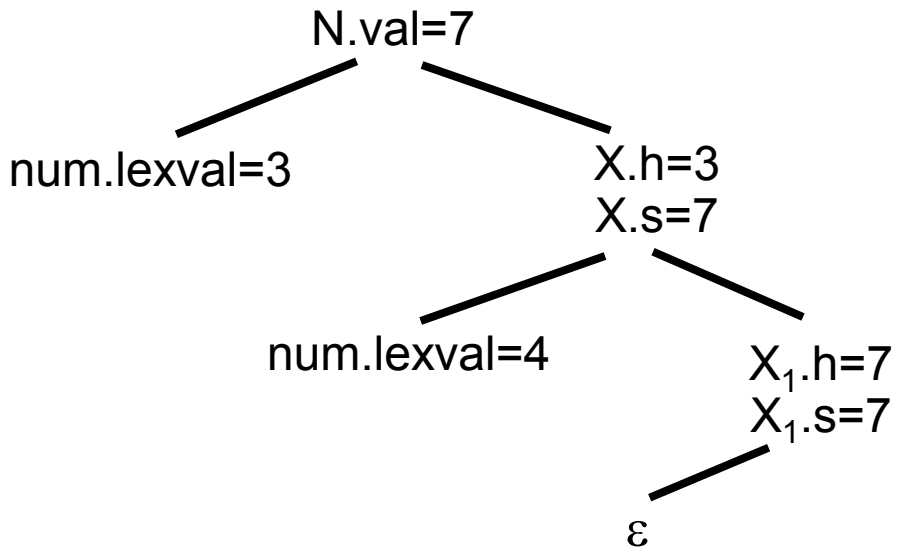
# Atributos (cont.)

- Exemplo (obtém o somatório dos dígitos): cadeia “34”

  $N \rightarrow N_1$ num  {N.val = $N_1$.val + num.lexval}

  $N \rightarrow$ num  {N.val = num.lexval}




- Exemplo (obtém o somatório dos dígitos): cadeia “34”

  $N \rightarrow$ num  {X.h = num.lexval}  X  {N.val = X.s}

  $X \rightarrow$ num  {$X_1$.h = X.h + num.lexval}  $X_1$  {X.s = $X_1$.s}

  $X \rightarrow \varepsilon$  {X.s = X.h}

# Regras Sintáticas

- Uma regra de produção gramatical $Y \rightarrow X_1X_2...X_n$, pode ter zero ou mais regras semânticas associadas. Cada regra tem a forma $b=f(c_1,c_2,...,c_k)$, tal que:
  1. b é um atributo sintetizado e $c_1,c_2,...,c_k$ são os atributos dos símbolos gramaticais do lado direito da regra.
  2. b é um atributo herdado de algum dos símbolos gramaticais do lado direito da regra e $c_1,c_2,...,c_k$ são quaisquer outros atributos na regra de produção.

# Gramática de Atributos

- Um **esquema de tradução** no qual as funções semânticas não têm efeitos colaterais é gramática de atributos sem dependência de contexto.
- Exemplo de esquema de tradução:

| **Produção** | **Regras Semânticas** |
|---|---|
| $S \rightarrow E$ | **imprime(E.val)** |
| $E1 \rightarrow E2 + T$ | **E1.val=E2.val+T.val** |
| $E \rightarrow T$ | **E.val=T.val** |
| $T1 \rightarrow T2 * num$ | **T1.val=T2.val*num.val** |
| $T \rightarrow num$ | **T.val=num.val** |

- Os atributos são herdados ou sintetizados?
  - **Sintetizados**.

# Exemplo (cont.)

- Comentário: Uma regra semântica como imprime(E.val) pode ser pensada como computação de um atributo sintetizado para S como S.val=imprime(E.val)
- Ordem de avaliação dos atributos:
  - A regras semânticas definem dependências entre instâncias de atributos, as quais definem a ordem de avaliação (ordem na qual um valor de atributo é computado antes de ser usado).
  - Pode ser necessário delinear um grafo de dependência para resolver as dependências e estabelecer a ordem de avaliação.

# Avaliação de Atributos

- Se a ordem de análise é consistente com a ordem de avaliação, então a avaliação dos atributos ser feita durante a fase de análise, sem necessidade de explicitamente construir a árvore de análise.
- Tipos de esquemas de tradução:
  - Esquemas S-atribuídos: opera somente com atributos sintetizados. São implementados através de extensões de analisadores ascendentes (redutivos);
  - Esquemas L-atribuídos: opera somente com atributos herdados, restringindo o seu uso, para permitir que as ações semânticas possam ser executadas durante a análise sintática em um único passo. São implementados através de extensões de analisadores descendentes.
  - Para esquemas com atributos herdados que não são L-atribuídos é necessário obter a relação de dependência entre os atributos.

# Esquemas S-atribuídos

- Um esquema S-atribuído é um esquema de tradução que possui somente atributos sintetizados.
- Apresentam as ações semânticas à direita das regras de produção.
- Os atributos sintetizados podem ser armazenados em símbolos extra para cada símbolo na pilha de análise.
- Os atributos podem ser avaliados à medida que a cadeia de entrada é reconhecida pelo analisador ascendente.
  - Sempre que ocorre uma redução são computados os atributos do símbolo do lado esquerdo da produção, a partir dos atributos que estão na pilha.

# Esquemas L-atribuídos

- Opera somente com atributos herdados, nos quais cada atributo $X_i$ de $A \rightarrow X_1X_2...X_n$ depende somente:
  - Dos atributos de $X_j$, $1 \leq j < i$ à esquerda de $X_i$ na produção, e
  - Dos atributos herdados de A.
- São implementados em analisadores descendentes.
- Quando há atributos herdados e sintetizados:
  - Deve-se computar em ação semântica antes (à esquerda) um atributo herdado associado a símbolo do lado direito.
  - Uma ação semântica somente pode se referir a atributo sintetizado de símbolo que apareça à esquerda dessa ação.
  - Atributo sintetizado associado a não-terminal do lado esquerdo deve ser computado somente depois de computar todos os atributos que ele referencia.

# Exemplo

- Considere a gramática livre de contexto abaixo:

  <atrib> → <var> := <expr>

  <expr> → <var>[1] + <var>[2]

  <var> → id

- Acrescentando as seguintes regras semânticas:

  <expr>.expected_type = <var>.actual_type

  <expr>.actual_type ← <var>[1].actual_type

  <var>.actual_type ← lookup (id, <var>)

- E os predicados:

  <var>[1].actual_type == <var>[2].actual_type

  <expr>.expected_type == <expr>.actual_type

- As regras descritas garantem a verificação e a correção dos tipos na atribuição.

# Tradução Dirigida por Sintaxe

- Para tradução dirigida por sintaxe são necessárias as definições:
  - Definições dirigidas por sintaxe estendem gramáticas livres de contexto através da inclusão de atributos pra cada símbolo da gramática, e regras semânticas que especificam ações semânticas a serem executadas durante a análise. (Também chamado de **esquema de tradução**). Isto pode ser feito dentro de uma ferramenta de análise (antlr, yacc, etc.).

# Tradução Dirigida por Sintaxe

- A tradução de um texto em linguagem fonte para a linguagem objeto pode ser realizada a partir da sintaxe, para isso é necessário:
  - Acompanhar a sintaxe livre de contexto através da árvore sintática;
  - Tratar a sintaxe dependente de contexto e armazenar os valores dos atributos;
  - Usar esta estrutura pronta durante a análise sintática para gerar código (diretamente ou através de algum código intermediário).

# Implementação de Esquemas S-Atribuídos

- Um esquema da forma:
  - $A \rightarrow X\ Y\ Z$ $\quad\quad\quad\quad\quad\quad\quad\quad$ { A.a = *f*(X.x, Y.y, Z.z) }
- Permite a execução da função *f* diretamente dos dados na pilha porque no momento em que *f* for executada (na redução para A) os dados dos atributos estarão na pilha, tendo sido coletados anteriormente. Assim usa-se uma pilha de atributos para implementar o esquema. Desta maneira, cada símbolo do lado direito da regra terá um atributo empilhado (pode ser vazio). Ex.:
  - $L \rightarrow E =$ {imprime(Atributo[topo])}
  - $E \rightarrow E1 + T$ {Atributo[topo] := Atributo[topo-2] + Atributo[topo]}
  - $E \rightarrow T$
  - $T \rightarrow T1 * F$ {Atributo[topo] := Atributo[topo-2] * Atributo[topo]}
  - $T \rightarrow T1 / F$ {Atributo[topo] := Atributo[topo-2] / Atributo[topo]}
  - $T \rightarrow F$
  - $T \rightarrow ( E )$ {Atributo[topo] := Atributo[topo-1]}
  - $F \rightarrow num$ {Atributo[topo] := num.lexval}

# Exemplo

| Entrada | Análise | Atributo | Produção |
|---|---|---|---|
| 2 * 3 + 4 / 2 = | $ | | |
| * 3 + 4 / 2 = | $2 | | |
| * 3 + 4 / 2 = | $F | 2 | F → num |
| * 3 + 4 / 2 = | $T | 2 | T → F |
| 3 + 4 / 2 = | $T * | 2 _ | |
| + 4 / 2 = | $T * 3 | 2 _ | |
| + 4 / 2 = | $T * F | 2 _ 3 | F → num |
| + 4 / 2 = | $T | 6 | T → T * F |
| + 4 / 2 = | $E | 6 | E → T |
| 4 / 2 = | $E + | 6 _ | |
| / 2 = | $E + 4 | 6 _ | |
| / 2 = | $E + F | 6 _ 4 | F → num |
| / 2 = | $E + T | 6 _ 4 | T → F |
| 2 = | $E + T / | 6 _ 4 _ | **Não há regra E→E/F** |
| = | $E + T / 2 | 6 _ 4 _ | |
| = | $E + T / F | 6 _ 4 _ 2 | F → num |
| = | $E + T | 6 _ 2 | T → T / F |
| = | $E | 8 | E → E + T |
| | $E = | 8 | |
| | $L | 8 | L → E = |

# Implementação de Esquemas L-Atribuídos

- Um esquema L-atribuído terá uma seqüência de percurso da árvore de derivação que corresponde à "descida" e à "subida", computando os atributos herdados na descida e, posteriormente, os atributos sintetizados na subida.
- A pilha de análise é usada para armazenar também as ações semânticas, que serão executadas exatamente na ordem desejada e adequada.
- Transformação de "*bottom-up*" para "*top-down*"
  - **A → A1 Y** **{ A.a = *g* (A1.a, Y.y) }**
  - **A → X** **{ A.a = *f* (X.x)}**
  - Passando para *top-down*:
    - A → X R
    - R → Y R | ε
  - **A → X** **{ R.h = *f* (X.x) }** **R** **{ A.a = R.s }**
  - **R → Y** **{ R1.h = *g* (R.h, Y.y) }** **R1** **{ R.s = R1.s }**
  - **R → ε** **{ R.s = R.h }**

# Exemplo

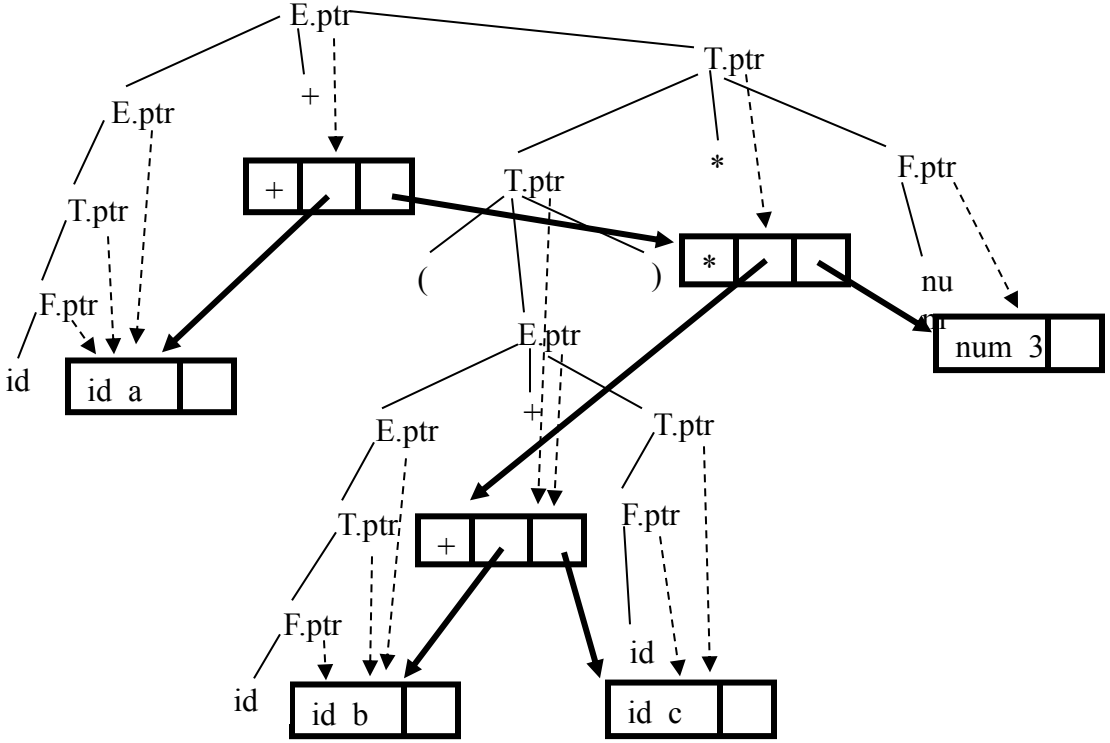
- Bottom-up:
  - $E \rightarrow E1 + T$ { E.ptr := geranodo("+", E1.ptr, T.ptr) }
  - $E \rightarrow E1 - T$ { E.ptr := geranodo("–", E1.ptr, T.ptr) }
  - $E \rightarrow T$ { E.ptr := T.ptr }
  - $T \rightarrow ( E )$ { T.ptr := E.ptr }
  - $T \rightarrow id$ { F.ptr := gerafolha("id", id.nome) }
  - $T \rightarrow num$ { F.ptr := gerafolha("num", num.lexval) }
- Top-down:
  - $E \rightarrow T$ { R.h := T.ptr } R { E.ptr := R.s }
  - $R \rightarrow + T$ { R1.h := geranodo("+", R.h, T.ptr) } R1 { R.s := R1.s }
  - $R \rightarrow - T$ { R1.h := geranodo("–", R.h, T.ptr) } R1 { R.s := R1.s }
  - $R \rightarrow \varepsilon$ { R.s := R.h }
  - $T \rightarrow ( E )$ {T.ptr := E.ptr}
  - $T \rightarrow id$ {F.ptr := gerafolha("id", id.nome)}
  - $T \rightarrow num$ {F.ptr := gerafolha("num", num.lexval)}

# Árvores de Sintaxe

- Forma simplificada da árvore de derivação, somente os operandos da linguagem aparecem como folhas, os operadores passam a ser nós internos. Também desaparecem produções $A \rightarrow B$, $B \rightarrow C$.
- Funções:
  - gerafolha("id", nome) – cria um nó identificador com dois campos, rótulo "id" e nome do identificador.
  - gerafolha("num", valor) – cria um nó de número com dois campos, rótulo "num" e valor do número.
  - geranodo(op, esq, dir) – cria um nó de operador com rótulo "op" e dois campos contendo ponteiros para as subárvores esquerda e direita.

# Exemplo

- E → E1 + T {E.ptr := geranodo("+", E1.ptr, T.ptr)}
- E → T {E.ptr := T.ptr}
- T → T1 * F {T.ptr := geranodo("*", T1.ptr, F.ptr)}
- T → F {T.ptr := F.ptr}
- T → ( E ) {T.ptr := E.ptr}
- F → id {F.ptr := gerafolha("id", id.nome)}
- F → num {F.ptr := gerafolha("num", num.lexval)}



# Projeto de um Tradutor Preditivo

- Baseado em um esquema de tradução faça:
  - Para cada não-terminal construa uma função que tenha um parâmetro para cada atributo herdado e que retorne os valores dos atributos sintetizados. Cada atributo que aparecer nas regras desse não-terminal deverá ser mapeado em uma variável local.
  - O código para o não-terminal deve decidir qual regra usar a partir do token de entrada:
    - Para cada token X, verificar se X é o token corrente e avance a leitura;
    - Para cada não-terminal B, gerar c := B(b1, b2, ..., bk) onde cada bi representa uma variável para os atributos herdados de B e c a variável para o atributo sintetizado de B;
    - Para cada ação semântica, copiar o código correspondente substituindo cada referência a atributo pela variável desse atributo

# Exemplo

- Seja a regra:
  - R → op_a T { R1.h := geranodo(op_a.lexval, R.h, T.ptr) } R1 { R.s := R1.s } |
    | ε { R.s := R.h }
- Tem-se:

```
function R(h: ^nodo) : ^nodo;
var h1, ptr, s, s1: ^nodo;
begin
    if token = op_a then
      begin
        token := LETOKEN;
        ptr := T( );
        h1 := geranodo (op_a.lexval, h, ptr);
        s1 := R(h1);
        s := s1;
      end
    else
      s:= h;
    return s;
end;
```

# Tabela de Símbolos

- Repositório de toda informação dentro do compilador. Todas as partes do compilador comunicam-se através desta tabela e acessam os símbolos.
  - São também usadas para armazenar rótulos, constantes, e tipos.
- A tabela de símbolos mapeia um nome (constante, identificador, rótulo numérico) em conjuntos de símbolos.
- Nomes diferentes têm atributos diferentes na tabela (para um identificador pode ser seu tipo, sua localização em uma pilha, sua classe de armazenamento, etc.

# Tabela de Símbolos (cont.)

- Os símbolos são coletados em uma tabela de símbolos. O módulo da tabela de símbolos gerencia símbolos e tabelas.
- O gerenciamento de símbolos deve lidar também com o escopo e a visibilidade que são impostos pela linguagem.
- As tabelas de símbolos são normalmente listas de tabelas ***hash***, uma para cada escopo.

# Tabela de Símbolos (cont.)

- Uma tabela típica (Fraser & Hanson, pág. 40)

```
typedef struct table *Table;
struct table {
int level ; /* scope value */
Table previous;
struct entry { struvt symbol sym;
struct entry *link; } *buckets[ 256];
Symbol all; } ;
```

# Interface – Tabela de Símbolos

- geraTab(ptr) – Gera uma tabela de símbolos (filha da tabela apontada por ptr) e retorna um ponteiro para a tabela gerada.
- defTam(ptr, tam) – Armazena na tabela de símbolos apontada por ptr o tamanho da área de dados do procedimento correspondente (local).
- adProc(ptr, nome, pt) – Insere na tabela de símbolos apontada por ptr o nome do procedimento e o ponteiro para a tabela de símbolos desse procedimento.
- adSimb(ptr, nome, tipo, end) – Insere na tabela de símbolos apontada por ptr um novo símbolo, seu tipo, e seu endereço na área de dados local.
- São usadas ainda duas pilhas tabPtr (ponteiro para a tabela de símbolos de cada procedimento) e desloc (próximo endereço disponível na área de dados da memória local do procedimento)

# Geração de Código Intermediário

- A geração de código intermediário é realizada a partir da árvore de derivação produzindo um segmento de código.
- Tradução de código fonte para código objeto apresenta vantagens:
  - Otimização do código intermediário, obtendo-se código objeto mais eficiente
  - Simplificação da implementação do compilador, resolução gradativa das dificuldades
  - Tradução do código intermediário para várias máquinas (ou manter uma máquina virtual).

# Linguagens Intermediárias

- Representações Gráficas
  - Árvore ou grafo de sintaxe (já visto o caso de árvore sintática)
- Notação pós-fixada ou pré-fixada
  - Expressões pré- ou pós fixadas permitem generalização da quantidade de operandos e operadores, ao mesmo tempo facilita a avaliação
- Código de três-endereços
  - Cada instrução faz referência a, no máximo, três variáveis (endereços de memória)

# Código de Três-Endereços

- Instruções:
  - A:=B op C
  - A:= op B
  - A := B
  - goto L
  - if A oprel B goto L
- A, B e C representam endereços de variáveis, op representa operador (binário ou unário), oprel representa operador relacional, e L representa o rótulo de uma instrução intermediária.

# Implementação e Esquema de Tradução

- A implementação pode ser efetuada através de quádruplas ou triplas
  - Quádruplas: 1 operador, 2 operandos, resultado
  - Triplas: 1 operador, 2 operandos (ponteiros para a estrutura)
- Esquema de Tradução: Uso de duas outras funções
  - geratemp: gera o nome de uma variável temporária (o atributo *nome* dos não-terminais armazena o nome de uma variável ou o nome de um temporário)
  - geracod: gera uma cadeia de texto correspondente a uma instrução de código intermediário (atributo *cod* armazena)

# Reutilização de Temporários

- Após uma variável ser referenciada ela pode ser descartada, usando-se um contador de variáveis pode-se controlar a reutilização.
- Algoritmo:
  - Sempre que um novo temporário é gerado usa-se contador e incrementa-se o valor deste de 1.
  - Sempre que um temporário é usado como operando decrementa-se o valor do contador de 1.

# Geração de Código para Comandos de Atribuição

- Esquema de Tradução:
  - Gera código de três-endereços.
  - Usa a função *lookup*(*id.nome*), que procura o identificador armazenado em id.nome na tabela de símbolos (retorna nil se não encontrar ou o índice do nome).
  - Usa a ação semântica de geração de código através de *geracod* (já visto).