Análise Semântica

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
UFC-Quixadá

Introdução

Atributos e Gramáticas de Atributos

A Tabela de Símbolos

Tipos de Dados e Verificações de Tipos

Análise Semântica

- Estrutura sintática do programa já conhecida;
- computar informações além da capacidade das gramáticas livres de contexto;
- análise semântica **estática**:
  - Ocorre antes da execução;
  - tabela de símbolos para acompanhar o significado de nomes;
  - inferência e verificação de tipos.
- análise pelas regras da linguagem, para verificar correção e garantir execução;
- análise efetuada pelo compilador para melhorar eficiência;
- não há como determinar a correção completa do programa.

Análise Semântica

- Não existe um método padrão como a BNF para especificar a semântica estática;
- **atributos**:
  - equações de atributos ou regras semânticas;
  - gramática de atributos;
  - semântica dirigida pela sintaxe.
- O desenvolvedor de compiladores deve construir uma gramática de atributos manualmente;
- sintaxe abstrata;
- múltiplas passadas:
  - Analisador semântico concorrente ao sintático é complexo;
  - a prática moderna indica o uso de múltiplas passadas.
- não há um **gerador de analisador semântico** consolidado como *lex* e *yacc*.

Atributos e Gramáticas de Atributos

Atributos

Qualquer propriedade de uma construção de linguagem de programação.

- Tipo de dados de uma variável;
- valor de uma expressão;
- localização de uma variável na memória;
- código objeto de um procedimento;
- quantidade de dígitos significativos em um número.

O processo de computar um atributo e associar seu valor computado com a construção da linguagem é chamado de **amarração** (ligação ou *binding*).
O momento durante a compilação ou execução em que ocorre a amarração é o **tempo de amarração** (tempo de ligação ou *binding time*).

Exemplos de Tempo de Ligação

Tipos de dados

Durante a compilação, um **verificador de tipos** é um analisador semântico que computa o atributo de tipo de todas as entidades tipadas e verifica se os valores computados estão de acordo com as regras da linguagem.

Valor de uma Expressão

Os valores de expressão em geral são dinâmicos, definidos em tempo de execução, mas o compilador precisa gerar código para calcular esses valores.

Exemplos de Tempo de Ligação

Localização de uma Variável na Memória

Pode ser definida durante a compilação ou execução, mas em geração é feita na geração de código, pois depende do sistema de memória alvo.

Código Objeto de um Procedimento

É atribuído durante a compilação, quando o procedimento também é compilado.

Quantidade de Dígitos Significativos

A definição é feita fora do processo de compilação, pois considera detalhes da arquitetura. O compilador simplesmente aceita um valor definido pelo desenvolvedor do compilador.

Gramáticas de Atributos

Na **semântica dirigida pela sintaxe**, os atributos são diretamente associados aos símbolos gramaticais da linguagem (os terminais e não terminais). Se X for um símbolo gramatical, e a um atributo, $X.a$ é o valor de a associado a X.

Equação Semântica

Seja uma regra $X_0 \rightarrow X_1 X_2 ... X_n$ e uma coleção de atributos $a_1, ..., a_k$:

$X_i.a_j = f_{ij}(X_0.a_1, ..., X_0.a_k, X_1.a_1, ..., X_1.a_k, ..., X_n.a_1, ..., X_n.a_k)$

É **raro** que um atributo dependa de um número muito grande de outros atributos.

Gramáticas de Atributos

Regra Gramatical	Regras Semânticas
Regra 1	Equações de atributos associadas
.	.
.	.
.	.
Regra *n*	Equações de atributos associadas

Gramáticas de Atributos - Exemplos

número $\rightarrow$ *número dígito*|*dígito*
dígito $\rightarrow$ 0|1|2|3|4|5|6|7|8|9

Considerado o atributo *val* como representando o valor numérico a ser armazenado na memória, como seria a gramática de atributos?

Gramáticas de Atributos - Exemplos

Regra Gramatical	Regras Semânticas
$número \rightarrow número\ dígito$	$número_1.val = número_2.val * 10 + dígito.val$
$número \rightarrow dígito$	$número.val = dígito.val$
$dígito \rightarrow 0$	$dígito.val = 0$
.	.
.	.
.	.
$dígito \rightarrow 9$	$dígito.val = 9$

Como fica a árvore de análise sintática anotada com as computações de atributos para o número 345?

Gramáticas de Atributos - Exemplos

$exp \rightarrow exp + termo | exp - termo | termo$
$termo \rightarrow termo * fator | fator$
$fator \rightarrow (exp) | \textbf{número}$
Considerado o atributo *val* como representando o valor numérico a ser atribuído a expressão, como seria a gramática de atributos?

Gramática de Atributos - Exemplos

Regra Gramatical	Regras Semânticas
$exp_1 \rightarrow exp_2 + termo$	$exp_1.val = exp_2.val + termo.val$
$exp_1 \rightarrow exp_2 - termo$	$exp_1.val = exp_2.val - termo.val$
$exp \rightarrow termo$	$exp.val = termo.val$
$termo_1 \rightarrow termo_2 * fator$	$termo_1.val = termo_2.val - fator.val$
$termo \rightarrow fator$	$termo.val = fator.val$
$fator \rightarrow (exp)$	$fator.val = exp.val$
$fator \rightarrow$ **número**	$fator.val =$ **número.**val

Não existem equações com **número**.*val* à esquerda. Como são calculados?
Como ficaria a árvore de análise sintática para $(34 - 3) * 42$.

Gramáticas de Atributos - Exemplos

decl $\rightarrow$ *tipo var-lista*
tipo $\rightarrow$ **int**|**float**
var-lista $\rightarrow$ **id**, *var-lista*|**id**
Queremos definir um atributo *dtipo* para identificadores e escrever equações para calculá-lo.

Gramáticas de Atributos - Exemplos

Regra Gramatical	Regras Semânticas
$decl \rightarrow tipo\ var\text{-}lista$	$var\text{-}lista.dtipo = tipo.dtipo$
$tipo \rightarrow \mathbf{int}$	$tipo.dtipo = integer$
$tipo \rightarrow \mathbf{float}$	$tipo.dtipo = real$
$var\text{-}lista_1 \rightarrow \mathbf{id}, var\text{-}lista_2$	$\mathbf{id}.dtipo = var\text{-}lista_1.dtipo$
	$var\text{-}lista_2.dtipo = var\text{-}lista_1.dtipo$
$var\text{-}lista \rightarrow \mathbf{id}$	$\mathbf{id}.dtipo = var\text{-}lista.dtipo$

Como ficaria a árvore de análise sintática para a cadeia *float x,y*?

Gramáticas de Atributos - Exemplos

base-num $\rightarrow$ *num basecar*
basecar $\rightarrow$ **o**|**d**
num $\rightarrow$ *num dígito*|*dígito*
dígito $\rightarrow$ 0|1|2|3|4|5|6|7|8|9
Como calcular o valor dos números nas duas bases diferentes? Será um único atributo suficiente?

Gramáticas de Atributos - Exemplos

Regra Gramatical	Regras Semânticas
$\textit{base-num} \rightarrow \textit{num}\ \textit{basecar}$	$\textit{base-num.val} = \textit{num.val}$
	$\textit{num.base} = \textit{basecar.base}$
$\textit{basecar} \rightarrow \textbf{o}$	$\textit{basecar.base} = 8$
$\textit{basecar} \rightarrow \textbf{d}$	$\textit{basecar.base} = 10$
$\textit{num}_1 \rightarrow \textit{num}_2 \textit{dígito}$	$\textit{num}_1\textit{.val} =$
	if $\textit{dígito.val} = \textit{erro}$ **ou** $\textit{num}_2\textit{.val} = \textit{erro}$
	then *erro*
	else $\textit{num}_2\textit{.val} * \textit{num}_1\textit{.base} + \textit{dígito.val}$
	$\textit{num}_2\textit{.base} = \textit{num}_1\textit{.base}$
	$\textit{dígito.base} = \textit{num}_1\textit{.base}$
$\textit{num} \rightarrow \textit{dígito}$	$\textit{num.val} = \textit{dígito.val}$
	$\textit{dígito.base} = \textit{num}_1\textit{.base}$
$\textit{dígito} \rightarrow 0$	$\textit{dígito.val} = 0$
.	.
.	.
.	.
$\textit{dígito} \rightarrow 8$	$\textit{dígito.val} =$
	if $\textit{dígito.base} = 8$ **then** *erro* **else** 8
$\textit{dígito} \rightarrow 9$	$\textit{dígito.val} =$
	if $\textit{dígito.base} = 8$ **then** *erro* **else** 9

Como ficaria a árvore de análise sintática para o número 345o?

Simplificações das Gramáticas de Atributos

- A coleção de expressões permitidas em uma equação de atributos é denominada **metalinguagem** para a gramática de atributos;
- **if-then-else, case, switch** serão utilizados, mas fica entendido que no projeto do compilador, a linguagem em que o mesmo é escrito será utilizada;
- também podemos usar funções que são definidas em outro lugar, por exemplo, uma função $dígito.val = numval(D)$ poderia ser definida como:

```
int numval(char D) {
  return (int)D - (int)'0';
}
```

Simplificações das Gramáticas de Atributos

Uma vez que a sintaxe já foi verificada, podemos simplificar a gramática:
$exp \rightarrow exp + exp | exp - exp | exp * exp | (exp) | \textbf{número}$

Regra Gramatical	Regras Semânticas
$exp_1 \rightarrow exp_2 + exp_3$	$exp_1.val = exp_2.val + exp_3.val$
$exp_1 \rightarrow exp_2 - exp_3$	$exp_1.val = exp_2.val - exp_3.val$
$exp_1 \rightarrow exp_2 * exp_3$	$exp_1.val = exp_2.val * exp_3.val$
$exp_1 \rightarrow (exp_2)$	$exp_1.val = exp_2.val$
$exp \rightarrow \textbf{número}$	$exp.val = \textbf{número}.val$

Como ficaria a **árvore de sintaxe abstrata** para $(34 - 3) * 42$?

Simplificações das Gramáticas de Atributos

Podemos construir a árvore de sintaxe abstrata usando métodos auxiliares:

- *mkOpNode*: recebe três parâmetros (marca do operador e duas árvores) e constrói um novo nó;
- *mkNumNode*: recebe um parâmetro, valor numérico, e constrói um nó folha para esse valor.

Simplificações das Gramáticas de Atributos

Regra Gramatical	Regras Semânticas
$exp_1 \rightarrow exp_2 + termo$	$exp_1.árvore = mkOpNode(+, exp_2.árvore, termo.árvore)$
$exp_1 \rightarrow exp_2 - termo$	$exp_1.árvore = mkOpNode(-, exp_2.árvore, termo.árvore)$
$exp \rightarrow termo$	$exp.árvore = termo.árvore$
$termo_1 \rightarrow termo_2 * fator$	$termo_1.árvore = mkOpNode(*, termo_2.árvore, fator.árvore)$
$termo \rightarrow fator$	$termo.árvore = fator.árvore$
$fator \rightarrow (exp)$	$fator.árvore = exp.árvore$
$fator \rightarrow$ **número**	$fator.árvore = mkNumNode(\textbf{número}.lexval)$

Algoritmos para Computação de Atributos

O Problema

Encontrar uma ordem para a avaliação e atribuição de atributos que garanta que todos os outros atributos necessários já estejam disponíveis.

Grafo de Dependências

Dado um atributo $X_i.a_j$ com X_i em uma regra gramatical $X_0 \rightarrow X_1X_2...X_n$, **existe um nó** para esse atributo no grafo de dependências. Considerando que:

$$X_i.a_j = f(..., X_m.a_k, ...)$$

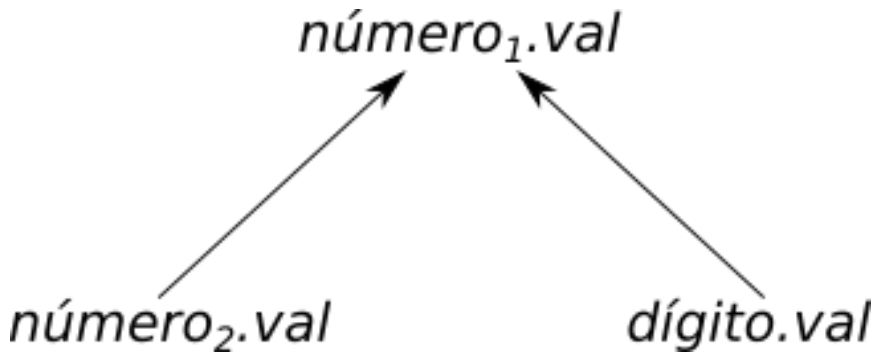
é a equação de atributos associada, **existe uma aresta** de cada $X_m.a_k$ direcionada para o nó $X_i.a_j$.

Construção do Grafo de Dependências

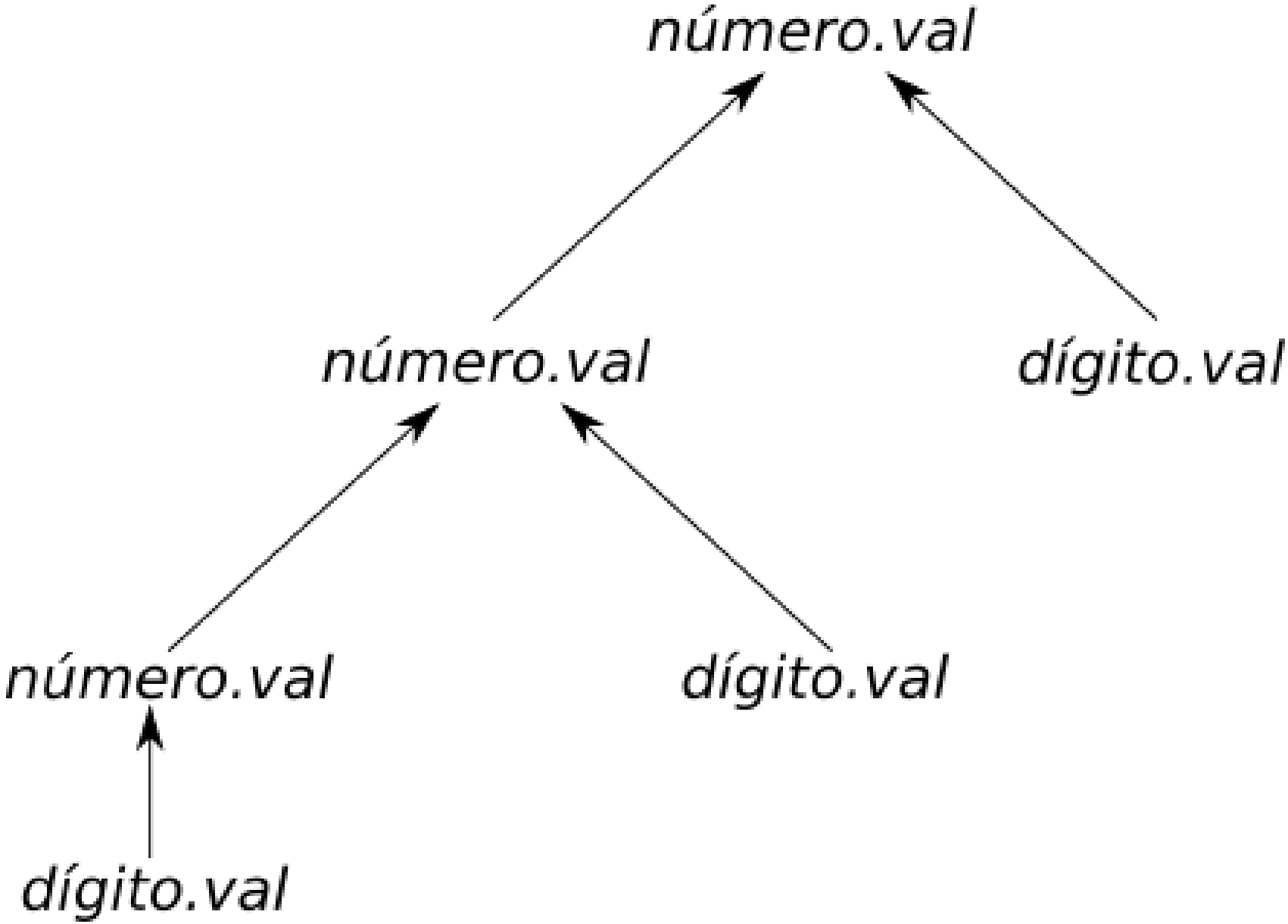
Considerando a gramática de atributos para números decimais, a regra $número_1 \rightarrow número_2\ dígito$ tem a seguinte equação de atributos associada:

$$número_1.val = número_2.val * 10 + dígito.val$$

Considerando a regra de definição de nós e vértices, teremos:



Para a cadeia 345, no próximo *slide*, mostramos o grafo completo.


número.val
número.val
dígito.val
número.val
dígito.val
dígito.val

Construção do Grafo de Dependências

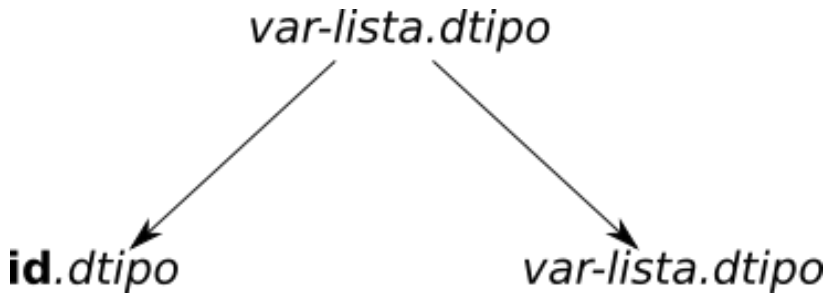
Para a regra gramatical:

$$\textit{var-lista}_1 \rightarrow \textbf{id}, \textit{var-lista}_2$$

temos as equações de atributos:

$$\textbf{id}.\textit{dtipo} = \textit{var-lista}_1.\textit{dtipo}$$
$$\textit{var-lista}_2.\textit{dtipo} = \textit{var-lista}_1.\textit{dtipo}$$

que resultam no seguinte grafo de dependências:



A regra gramatical *var-lista* $\rightarrow$ **id** tem como grafo a ligação direta entre os dois atributos. As regras *tipo* $\rightarrow$ **int** e *tipo* $\rightarrow$ **float** são triviais, com atributos preenchidos pelas fases anteriores da compilação.

Construção do Grafo de Dependências

Para a regra gramatical:

$$\textit{decl} \rightarrow \textit{tipo var-lista}$$

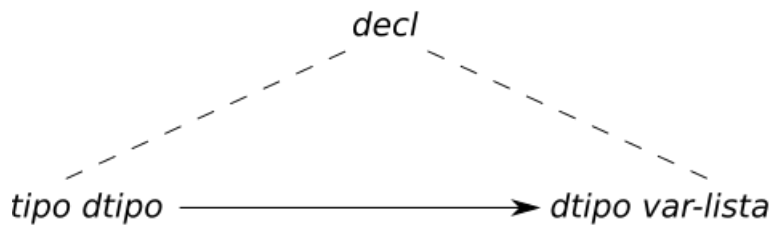
temos a equação de atributos:

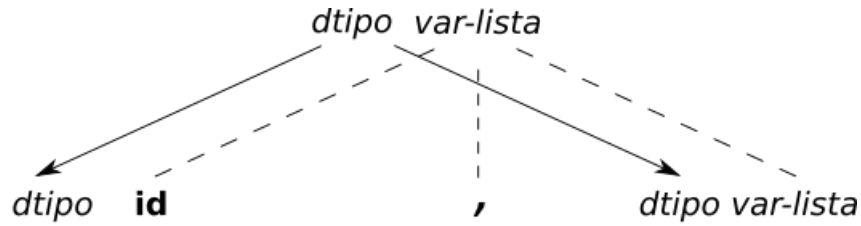
$$\textit{var-lista.dtipo} = \textit{tipo.dtipo}$$

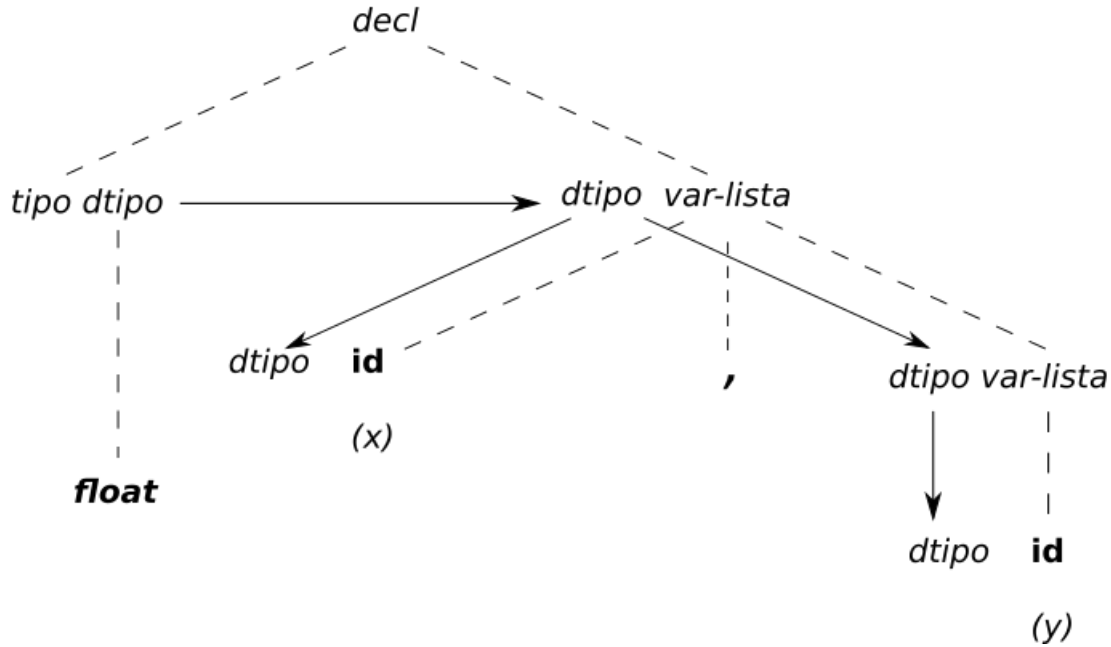
o grafo de dependências tem o formato:

$$\textit{tipo.dtipo} \longrightarrow \textit{var-lista.dtipo}$$

O problema é que, sem *decl* visível, nós não sabemos qual regra origina o grafo. Podemos desenhar o grafo de dependência sobreposto a árvore de análise sintática para deixar claro qual regra define cada relação entre os atributos.

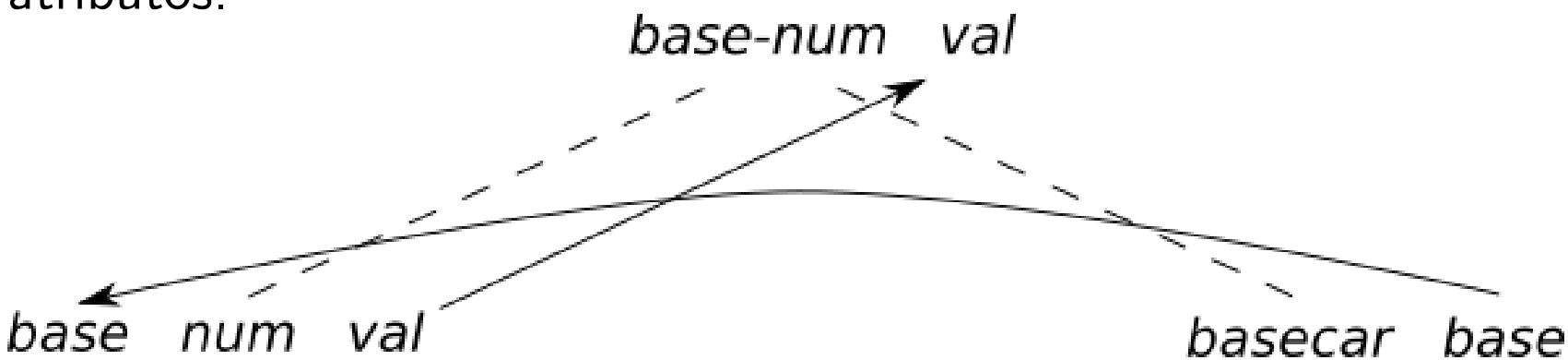

decl
tipo dtipo
dtipo var-lista


dtipo var-lista
dtipo
id
,
dtipo var-lista


decl
tipo dtipo
dtipo var-lista
dtipo
id
(x)
,
dtipo var-lista
float
dtipo
id
(y)

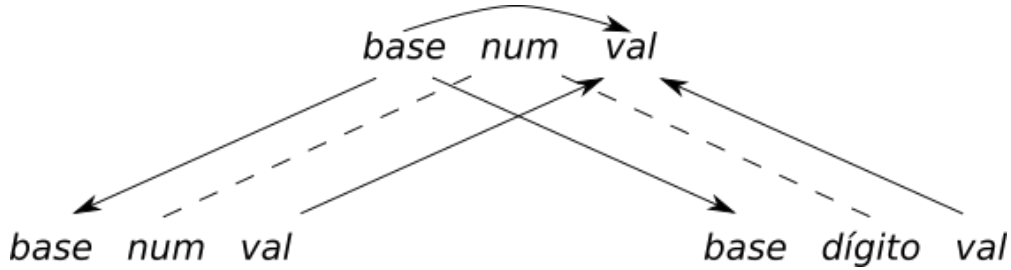
Construção do Grafo de Dependências

Na gramática para números octais e decimais, a regra *base-num* $\rightarrow$ *num basecar* tem as equações *base-num.val* $=$ *num.val* e *num.base* $=$ *basecar.base*. O detalhe importante para a construção do grafo é que agora são dois atributos.

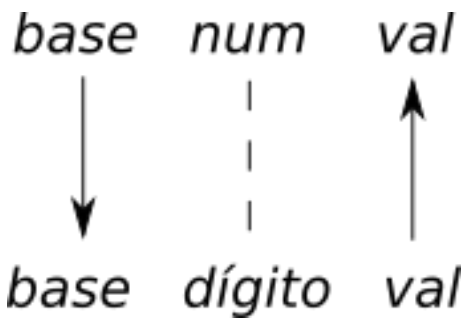


Construção do Grafo de Dependências

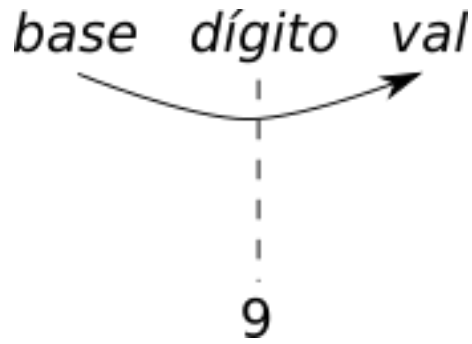
Regra *num* → *num dígito*:



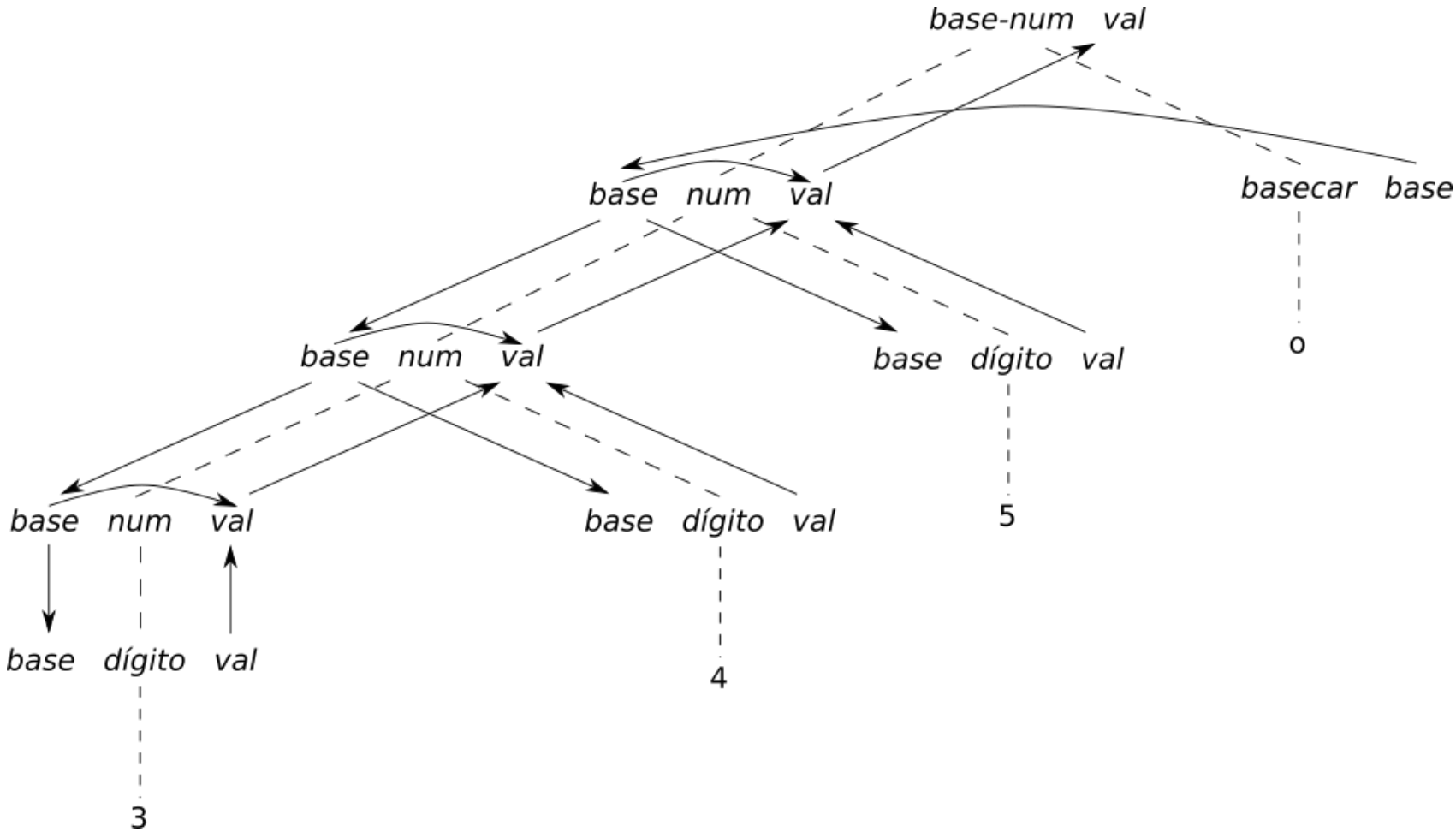
Regra *num* → *dígito*:



Regra *dígito* → 9:

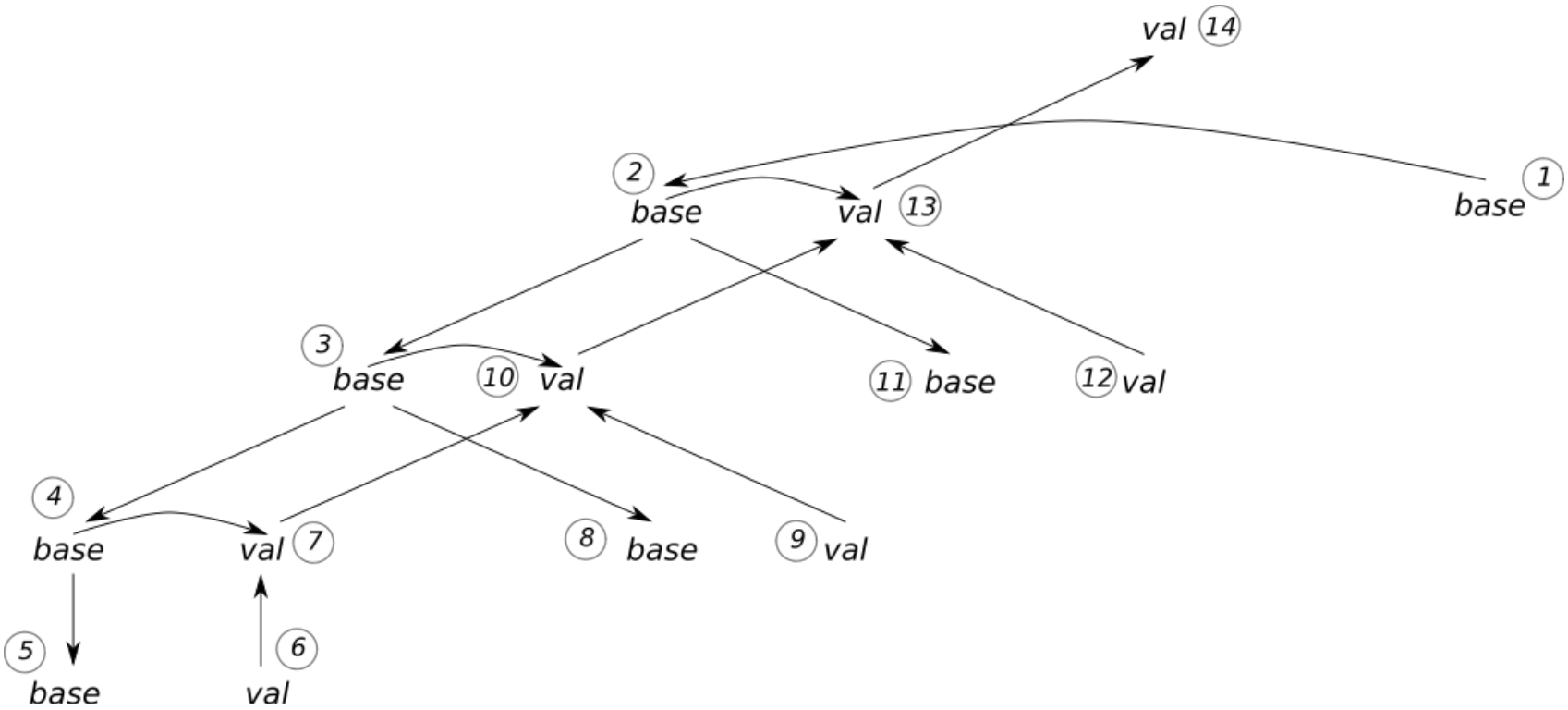


Como seria o grafo de dependências para 345o?


base-num val
base num val
basecar base
base num val
base dígito val
o
base num val
base dígito val
5
base dígito val
4
3

Algoritmos para Computação de Atributos

- Qualquer algoritmo deve computar o atributo em cada nó no grafo antes de **tentar** computar atributos sucessores.
- **ordenação topológica**: percurso no grafo que obedece a restrição acima;
- só existem ordenações topológicas para **grafos direcionados acíclicos**, DAGs;
- uma **raiz** de um grafo é um nó sem predecessores.


val 14
2
base
val 13
base 1
3
base
10 val
11 base
12 val
4
base
val 7
8 base
9 val
5
base
6
val

Método de Árvore de Análise Sintática

- Realizar a análise de atributos construindo um grafo de dependências e uma ordenação topológica;
- eficaz desde que a gramática de atributos não seja **circular**;
- existem algoritmos para testar se uma gramática de atributos é circular ou não, em tempo exponencial;
- usando esses algoritmos, e técnicas para construção de ordenações topológicas, teríamos um gerador de analisadores semânticos;
- a maioria dos compiladores, entretanto, utiliza o **método baseado em regras**, ou seja, determinar através de código a ordem de avaliação dos atributos;
- é suficiente para gramáticas **fortemente não circulares**.

Atributos Sintetizados

Para definir um método baseado em regras, seja percorrendo ou não a árvore sintática, precisamos estabelecer a natureza dos atributos.

Definição

Todas as suas dependências apontarem de filho para pai na árvore de análise sintática. Atributo a é sintetizado se dada uma regra $A \rightarrow X_1 X_2 ... X_n$, a única equação de atributos associada com um a a esquerda é da forma:

$$A.a = f(X_1.a_1, ..., X_1.a_k, ..., X_n.a_1, ..., X_n.a_k)$$

Uma gramática de atributos em que todos os atributos são sintetizados é denominada **gramática S-atribuída**.

Como Programar o Cálculo de Atributos Sintetizados

Se a gramática de atributos for S-atribuída, podemos utilizar um percurso ascendente (pós ordem) na construção da árvore de análise sintática.

```
procedure Pós-Eval(T:nó-árvore);
begin
   for cada filho C de T do
      Pós-Eval(C);
   compute cada atributo sintetizado de T;
end;
```

Um Algoritmo para a Gramática de Atributos das Expressões Aritméticas

```
typedef enum {Plus, Minus, Times} OpKind;
typedef enum {OpKind, ConstKind} ExpKind;
typedef struct streenode {
   ExpKind kind;
   OpKind op;
   struct streenode *lchild, *rchild;
   int val;
} STreeNode;
typedef STreeNode *SyntaxTree;
```

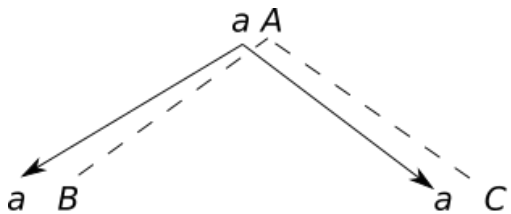
Um Algoritmo para a Gramática de Atributos das Expressões Aritméticas

```
void postEval(SyntaxTree t) {
   int temp;
   if (t->kind == OpKind) {
      postEval(t->lchild);
      postEval(t->rchild);
      switch (t->op) {
         case Plus:
            t->val = t->lchild->val + t->rchild->val;
            break;
         case Minus;
            t->val = t->lchild->val - t->rchild->val;
            break;
         case Times;
            t->val = t->lchild->val * t->rchild->val;
            break;
      } /* end switch */
   } /* end if */
} /* end postEval */
```

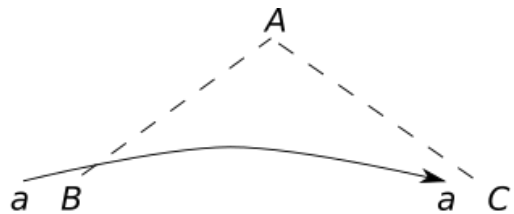
Atributos Herdados

Definição

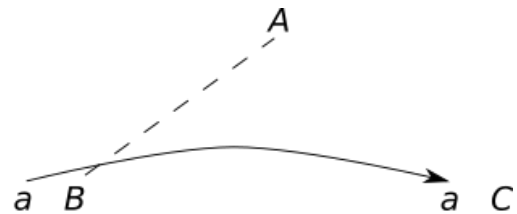
- Um atributo que não é sintetizado é denominado **herdado**;
- os atributos herdados têm dependências que fluem de pai para filhos ou entre irmãos;
- a razão de chamar os dois casos de herança é porque a transmissão de atributos entre irmãos é geralmente implementada de modo que os valores dos atributos passam entre os irmãos através do pai.



Herança de pai para filhos



Herança entre irmãos



Herança entre irmãos através de ponteiros

Métodos Algorítmicos para Calcular Atributos Herdados

Percurso em pré-ordem, ou a combinação de pré-ordem e *in*-ordem, para percorrer a árvore.

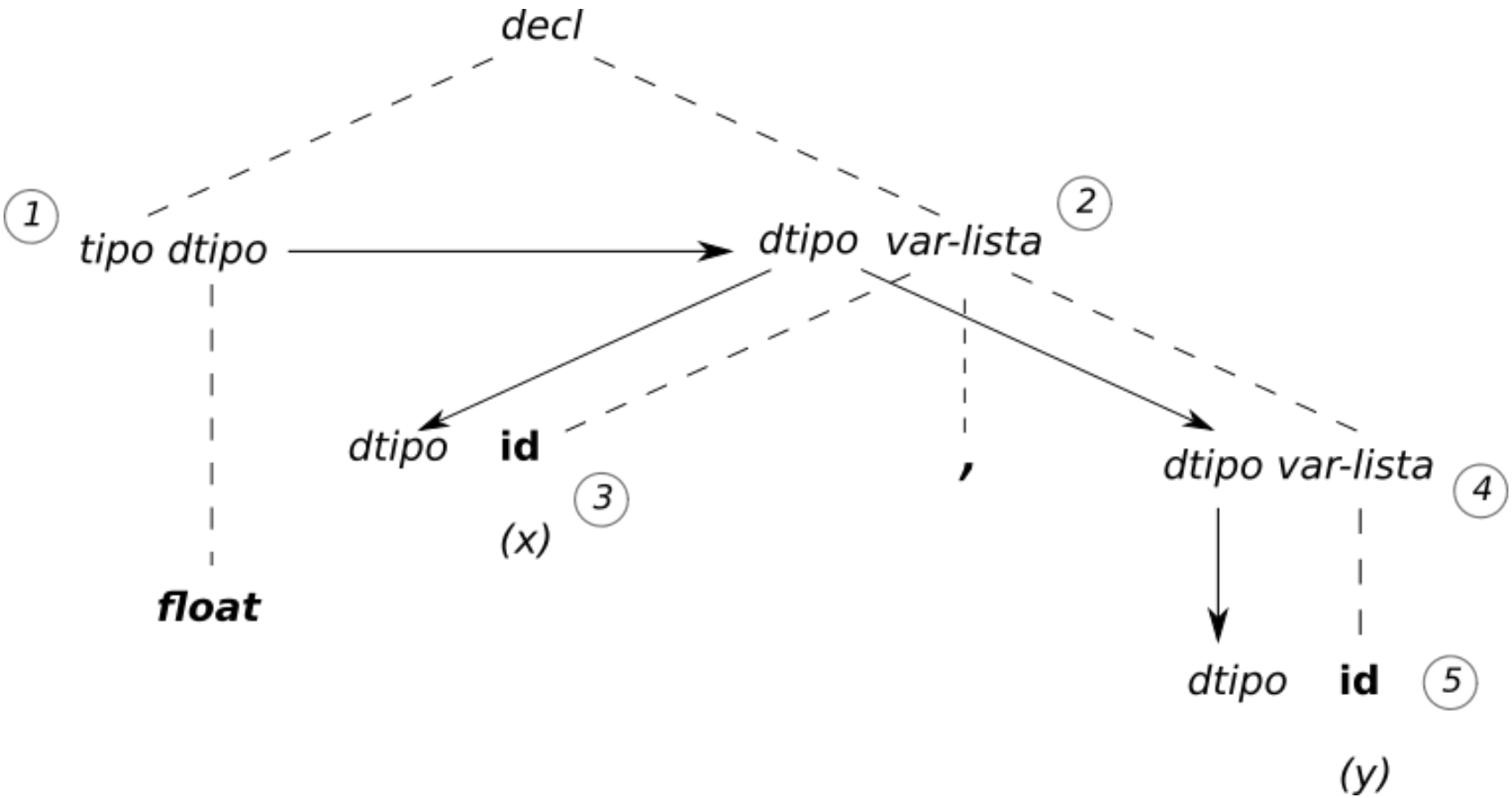
```
procedure Pré-Eval (T:nó-árvore);
begin
   for cada filho C de T do
      compute cada atributo herdado de C;
      Pré-Eval(C);
end;
```

- A ordem agora importa;
- a sequência de visitas na definição do **for**;
- requisitos das dependências.

Métodos Algorítmicos para Calcular Atributos Herdados

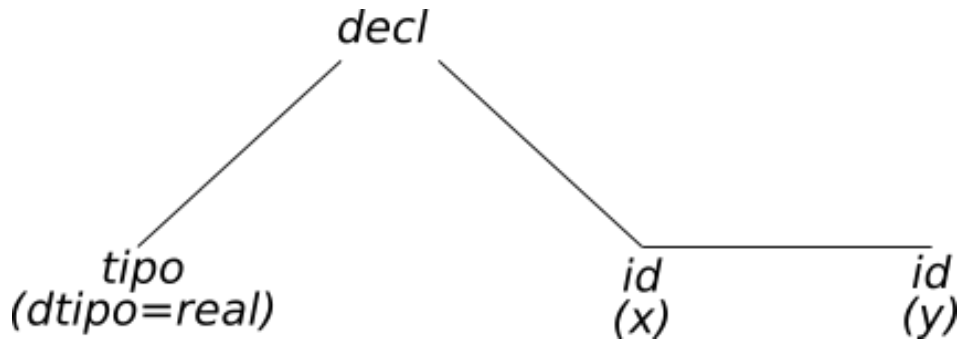
Considerando novamente a gramática:
decl $\rightarrow$ *tipo var-lista*
tipo $\rightarrow$ **int**|**float**
var-lista $\rightarrow$ **id**, *var-lista*|**id**

```
procedure AvalTipo(T:nó-arvore);
begin
   case tipo-nó de T of
   decl:
      AvalTipo(tipo);
      Atribui dtipo de tipo a dtipo de var-lista;
      AvalTipo(var-lista);
   tipo:
      if filho de T = int then T.dtipo := inteiro
      else T.dtipo := real;
   var-lista:
      atribui T.dtipo a primeiro filho de T;
      if terceiro filho de T não é nulo then
         atribui T.dtipo a terceiro filho;
         AvalTipo(terceiro filho de T);
   end case;
end AvalTipo;
```


decl
1
tipo dtipo
2
dtipo var-lista
dtipo
id
3
(x)
,
dtipo var-lista
4
float
dtipo
id
5
(y)

Usando Árvores Sintáticas Simplificadas

Podemos representar *var-lista* como uma lista encadeada de nós **id**.




```
typedef enum {decl, type, id} nodekind;
typedef enum {integer, real} typekind;
typedef struct treeNode {
   struct treeNode *lchild, *rchild, *sibling;
   typekind dtype; /* para nós tipo e id */
   char *name;     /* apenas para nós id */
} * SyntaxTree;
```

Usando Árvores Sintáticas Simplificadas

```
void evalType(SyntaxTree t) {
    switch (t->kind) {
        case decl:
            t->rchild->dtype = t->lchild->dtype;
            evalType(t->rchild);
            break;
        cade id:
            if (t->sibling != NULL) {
                t->sibling->dtype = t->dtype;
                evalType(t->sibling);
            }
    } /* fim do switch */
} /* fim do AvalTipo */
```

Métodos Algorítmicos para Calcular Atributos

base-num $\rightarrow$ *num basecar*
basecar $\rightarrow$ **o**|**d**
num $\rightarrow$ *num dígito*|*dígito*
dígito $\rightarrow$ 0|1|2|3|4|5|6|7|8|9

- Atributo sintetizado *val* e atributo herdado *base*;
- *val* depende de *base*;
- *base* é herdado do filho direito para o filho esquerdo de um *base-num*;
- misturar métodos pré-ordem e pós-ordem.

```
procedure AvalComBase(T:árvore-nó):
begin
   case nó-tipo de T of
   base-num:
      AvalComBase(basecar);
      num.base := basecar.base;
      AvalComBase(num);
      T.val := num.val;
   num:
      filhoEsq := T.arvoreEsquerda;
      filhoEsq.base := T.base;
      AvalComBase(filhoEsq);
      filhoDir := T.arvoreDireita;
      if filhoDir não é nulo then
         filhoDir.base := T.base;
         AvalComBase(filhoDir);
         if (filhoDir.val != erro and
            filhoEsq.val != erro) then
            T.val := T.base *
              (filhoEsq.val + filhoDir.val)
         else T.val := erro;
      else T.val = filhoEsq.val;
   basecar:
      if filho de T = o then
         T.base := 8;
      else T.base := 10;
   dígito:
      if (T.base = 8 and (filho de T = 8
         or filho de T = 9)) then
         T.val := erro;
      else
         T.val := numval(filho de T);
   end case;
end AvalComBase;
```

Métodos Algorítmicos para Calcular Atributos

Nas gramáticas de atributos com sintetizados e herdados, sendo que os sintetizados dependem dos herdados, mas os herdados não dependem de sintetizados, é possível fazer o cálculo em uma única passada.

```
procedure CombinaEval(T:nó-árvore):
begin
   for cada filho C de T do
      compute cada atributo herdado de C;
      CombinaEval(C);
   compute cada atributo sintetizado de T;
end;
```

Situações nas quais atributos herdados dependem de sintetizados exigem mais de uma passada.

Métodos Algorítmicos para Calcular Atributos

Operações interpretadas de maneira diferente, se o número for ponto flutuante ou não.

$$S \rightarrow exp$$
$$exp \rightarrow exp/exp|\mathbf{num}|\mathbf{num.num}$$

- *éFlut*: atributo booleano sintetizado;
- *etipo*: atributo herdado, *int* ou *float*;
- *val*: atributo sintetizado.

Métodos Algorítmicos para Calcular Atributos

Regra Gramatical	Regras Semânticas
$S \rightarrow exp$	exp.etipo = **if** exp.éflut **then** float **else** int S.val = exp.val
$exp_1 \rightarrow exp_2 / exp_3$	exp_1.éFlut = exp_2.éFlut **or** exp_3.éFlut exp_2.etipo = exp_1.etipo exp_3.etipo = exp_1.etipo exp_1.val = **if** exp_1.etipo = int **then** exp_2.val **div** exp_3.val **else** exp_2.val/exp_3.val
$exp \rightarrow$ **num**	exp.éFlut = **false** exp.val = **if** exp.etipo = int **then num**.val **else** Float(**num**.val)
$exp \rightarrow$ **num.num**	exp.éFlut = **true** exp.val = **num.num**.val

Métodos Algorítmicos para Calcular Atributos

Duas passadas:

- A primeira passada computa o atributo sintetizado *éFlut* em pós-ordem;
- A segunda passada computa o atributo herdado *etipo* e o atributo sintetizado *val* com um percurso combinado em pré-ordem e pós-ordem.

Como ficaria o cálculo de atributos para 5/2/2.0?

Atributos como Parâmetros e Valores de Retorno

- No lugar de preencher os valores em uma árvore, podemos transmitir os valores dos atributos como o valor de retorno dos procedimentos recursivos;
- Um único procedimento para percorrer a árvore:
  - Computação dos atributos herdados em pré-ordem;
  - computação dos atributos sintetizados em pós-ordem;
  - transmitir os valores dos atributos herdados como parâmetros para ativações recursivas dos filhos;
  - receber valores dos atributos sintetizados como valores de retorno das ativações.
- essa metodologia já foi demonstrada na gramática do cálculo de expressões;
- para atributos estruturados com maior complexidade, uma tabela de símbolos ou variáveis globais (**estruturas externas**) podem ser empregadas para manter valores entre as ativações.

Atributos como Parâmetros e Valores de Retorno

```
function AvalComBase(T: no-arvore; base:inteiro): inteiro;
var temp, temp2: inteiro:
begin
      case no-tipo de T of
      base-num:
         temp := AvalComBase(filho a direita de T);
         return AvalComBase(filho a esquerda de T, temp);
      num:
         temp := AvalComBase(filho a esquerda de T, base);
         if filho a direita de T nao nil then
            temp2 := AvalComBase(filho a direita de T, base);
            if temp <> erro and temp2 <> erro then
               return base * temp +  temp2;
            else return erro;
         else return temp;
      basecar:
         if filho de T = o then return 9
         else return 10;
      digito:
         if base = 8 and filho de T = 8 ou 9 then return erro
         else return numval(filho de T);
      end case;
end AvalComBase;
```

Atributos como Parâmetros e Valores de Retorno

- Funciona somente porque tanto *base* quanto *val* podem ser representados por inteiros;
- a primeira ativação seria *AvalComBase(nó-raiz, 0);*
- uma possibilidade seria dividir a função em três.

```
function AvalBaseNum(T: no-arvore): inteiro;
(* ativado apenas para a raiz *)
begin
   return AvalNum(filho esquerdo de T,
                  AvalBase(filho direito de T));
end AvalBaseNum
function AvalBase(T: no-arvore): inteiro;
(* ativado apenas para basecar *)
begin
   if filho de T igual ao then return 8
   else return 10;
end AvalBase
```

Atributos como Parâmetros e Valores de Retorno

```
function AvalNum(T: no-arvore): inteiro;
var temp, temp2: inteiro:
begin
      case no-tipo de T of
      num:
         temp := AvalComBase(filho a esquerda de T, base);
         if filho a direita de T nao nil then
            temp2 := AvalComBase(filho a direita de T, base);
            if temp <> erro and temp2 <> erro then
               return base * temp +  temp2;
            else return erro;
         else return temp;
      digito:
         if base = 8 and filho de T = 8 ou 9 then return erro
         else return numval(filho de T);
      end case;
end AvalComBase;
```

O que essa separação quer mostrar é que o desenvolvedor do compilador tem liberdade para refatorar o código.

Estruturas de Dados Externas

- No exemplo anterior, seria realmente necessário a cópia de *base* em vários argumentos?
- podemos supor a existência de estruturas de dados *externas*:
  - Variáveis globais;
  - **tabelas**.

```
base-num:
        AjustaBase(filho direito de T);
        return AvalComBase(filho esquerdo de T);
```

- *AjustaBase* seria um procedimento externo que configura uma variável global;
- todos os outros casos acessam essa variável;
- as regras semânticas podem ser atualizadas para refletir a variável global.

Estruturas de Dados Externas

Considerando a gramática de declaração de tipos, podemos ter o procedimento:

```
procedure inserir (nome: cadeia de caracteres; dtipo: tipo);
```

Regra Gramatical	Regras Semânticas
decl $\rightarrow$ *tipo var-lista*	
tipo $\rightarrow$ **int**	*dtipo* = *inteiro*
tipo $\rightarrow$ **float**	*dtipo* = *real*
var-lista$_1$ $\rightarrow$ **id,**var-lista$_2$	*inserir(***id***.nome, dtipo)*
var-lista $\rightarrow$ **id**	*inserir(***id***.nome, dtipo)*

Há uma tabela que armazena o mapeamento do nome ao tipo declarado.

Estruturas de Dados Externas

```
procedure AvalTipo (T: no-arvore);
begin
   case tipo-no de T of
   decl:
      AvalTipo(tipo filho de T);
      AvalTipo(var-lista filha de T);
   tipo:
      if filho de T = int then dtipo := inteiro
      else dtipo := real;
   var-lista:
      inserir(nome do primeiro filho de T, dtipo)
      if terceiro filho de T nao nil then
         AvalTipo(terceiro filho de T);
   end case;
end AvalTipo;
```

Computação de Atributos Durante a Análise Semântica

+

- Uma questão fundamental é quanto dos atributos já podem ser calculados durante a análise sintática;
- como a maioria dos algoritmos de análise sintática processa a entrada da esquerda para a direita, o cálculo de atributos também segue essa direção;
- uma gramática de atributos na qual cada atributo, sintetizado ou herdado, pode ser calculado através de outros atributos que só ocorrem à esquerda na regra gramatical é dita **L-atribuída**.
- podemos acrescentar uma **pilha de valores** e explicitar as **ações semânticas** durante a análise LR;
- a maioria dos compiladores da atualidade não utilizam essa técnica, por sua vez usam mais de uma passada.

A Tabela de Símbolos

- É um atributo herdado;
- também tem relevância na análise sintática e no sistema de varredura;
- estrutura de dados com três operações:
  - *inserir*: armazenar informações fornecidas pelas declarações de nomes;
  - *verificar*: recuperar informações de um nome previamente inserido;
  - *remover*: invalidar uma informação.
- tipos de dados, escopo e localização na memória.

A Estrutura da Tabela de Símbolos

- Na prática, a estrutura mais utilizada é um *dicionário*, no qual você informa uma *chave* e recebe um *valor*;
- Possíveis implementações:
  - Listas lineares;
  - árvores de buscas;
  - tabelas *hash* (mais utilizada. **Por quê?**).
- o estudo avançado dessas estruturas é assunto das disciplinas de Estruturas de Dados.

Tabelas *Hash*

Uma matriz, cujas células são denominadas **repositórios**, indexada por um intervalo de inteiros, em geral de 0 até o tamanho da tabela menos 1.

- Um **função de hashing** transforma a chave de busca em um valor inteiro dentro do intervalo de índices;
- **colisões** de *hashing*;
- **resolução de colisões**:
  - endereçamento aberto;
  - encadeamento separado (repositório como lista linear).
- o tamanho da tabela é fixado durante a construção do compilador;
- deve ser um número **primo**, pois melhora o comportamento da função *hashing*.

Declarações

As declarações de variáveis, tipos, funções, etc (tudo que tem um nome) nas linguagens de programação afetam o comportamento e a implementação da tabela de símbolos.

- Declarações de constantes

```
const int size = 199;
```

- declarações de tipos

```
typed struct Entry * EntryPtr;
```

- declarações de variáveis

```
int a,b[100];
```

- declarações de procedimentos e funções.

```
int hash(char *key) { return char[0] - '0'; }
```

As declarações podem ser **explícitas** ou **implícitas**.

Declarações

- É mais fácil usar uma única tabela de símbolos para todos os tipos de declarações;
- para linguagens modernas, é preferível associar tabelas de símbolos separadas por regiões do programa (procedimento ou função) e ligá-las segundo as regras semânticas;
- os atributos vinculados a cada nome variam de acordo com o tipo de declaração;
- por exemplo, os atributos de uma constante são diferentes dos atributos de uma variável.

Atributos Vinculados a Nomes

Declarações de Constantes

- Associam valores a nomes;
- podem ser valores estáticos substituídos no texto durante a compilação;
- outra alternativa é vinculação dinâmica, como uma variável;
- **atribuição única** em ambos os casos.

Declarações de Tipos

- Vinculam nomes a tipos pré-existentes;
- vamos falar mais sobre tipos adiante.

Atributos Vinculados a Nomes

Declarações de Variáveis

- Vincula nomes a tipos;
- outro atributo é o **escopo**;
- alocação de memória;
- **duração** ou **tempo de vida** da alocação;

Declarações de Procedimentos

- Vincula um nome a uma posição de memória na qual está o código do procedimento;
- parâmetros;
- tipo de retorno.

Regras de Escopo e Estruturas de Blocos

Declarações Antes do Uso

Um nome deve ser declarado no texto de um programa antes de qualquer referência a ele.

Estrutura de Blocos

- **Linguagem Estruturada em Blocos**: permite o aninhamento de blocos dentro de blocos e o escopo das declarações em um bloco é limitado à ele e aos blocos nele contidos.
- **Regra do Aninhamento mais Próximo**: dadas diversas declarações de um mesmo nome, a declaração que se aplica a uma referência é aquela no bloco de aninhamento mais próximo da referência.

Escopos Aninhados

```
int i, j;

int f (int tamanho)
{
   char i, temp;
   ...
   {
      double j;
      ...
   }
   ...
   {
      char *j;
      ...
   }
}
```

- A operação na tabela de símbolos *inserir* não pode escrever por cima declarações anteriores;
- deve ocorrer uma ocultação temporária, para que a operação *verificar* encontre apenas a declaração inserida mais recentemente para um nome;
- a solução é que a tabela de símbolo deve ter comportamento similar a uma *pilha*;
- como ficaria a tabela para o código ao lado?

Alternativas para Construção de Tabela

- Construir uma nova tabela para cada escopo;
- fazer a ligação das tabelas em uma lista, do escopo mais interno para o mais externo;
- a operação *verificar* percorre a lista até encontrar o nome buscado;
- as regras de resolução de escopo podem ser ignoradas utilizando um **operador de resolução**.

Escopo Estático *versus* Escopo Dinâmico

```c
#include <stdio.h>

int i = 1;

void f(void) {
    printf("%d\n", i);
}

void main(void) {
    int i = 2;
    f();
    return 0;
}
```

- Estamos concentrando nossas discussões no escopo estático, pois é o mais utilizado;
- nesse caso, a tabela de símbolos é construída em sua maior parte durante a compilação;
- o escopo dinâmico exige que tabela seja construída durante a execução.

Interação em Declarações de Mesmo Nível

- Para a maioria das linguagens, nomes não podem ser repetidos no mesmo escopo:

```
typedef int i;
int i;
```

- declaração sequêncial:

```
int i = 2, j = i + 1;
```

- algumas linguagens funcionais adotam a declaração **colateral**, o oposto da sequêncial;
- declaração recursiva:

```
int gcd(int n, int m) {
  if (m == 0) return n;
  else return gcd(m, n % m);
}
```

- o uso de protótipos de funções ameniza os problemas da recursão.

Exemplo de Gramática de Atributos para Construção da Tabela de Símbolos

$S \rightarrow exp$
$exp \rightarrow (exp) | exp + exp | \mathbf{id} | \mathbf{num} | \mathbf{let}\ dec\text{-}lista\ \mathbf{in}\ exp$
$dec\text{-}lista \rightarrow dec\text{-}lista, decl | decl$
$decl \rightarrow \mathbf{id} = exp$

As declarações após a marca **let** estabelecem os nomes para expressões, os quais, quando aparecem na *exp* após a marca **in**, substituem os valores. Em outras palavras, as expressões *let* representam os blocos dessa linguagem.

Exemplo de Gramática de Atributos para Construção da Tabela de Símbolos

```
    -- Expressões Corretas
    let x = 2 + 1, y = 3 + 4 in x + y
    let x = 2, y = 3 in
        (let x = x + 1, y = (let z = 3 in x + y + z)
             in (x + y))
    -- Não pode haver redeclaração - Errado!
    let x = 2, x = 3 in x + 1
    -- Um nome precisa ser declarado - Errado!
    let x = 2 in x + y
    -- Aninhamento mais próximo
    let x = 2 in (let x = 3 in x)
    -- Declaração seqüêncial
let x = 2, y = x + 1 in (let x = x + y, y = x + y in y)
```

Objetivos da Tabela de Símbolos para o Exemplo

- Vamos usar a tabela para determinar se uma expressão é errônea;
- atributo booleano sintetizado *err* (**true** expressão errada, **false** caso contrário);
- atributo herdado *simtab* para representar a tabela;
- atributo herdado *nivelaninh*, para identificar o nível do bloco;
- $inserir(s, n, l)$ retorna uma nova tabela de símbolos que contém todas as informações de s, mas com o nome n associado ao nível l;
- $estaem(s, n)$ verdadeiro ou falso se n está em s ou não;
- $verificar(s, n)$ retorna o aninhamento de n em s.

Gramática de Atributos para Tabela de Símbolos

Regra Gramatical	Regras Semânticas
$S \rightarrow exp$	$exp.simtab = tabelavazia$ $exp.nivelaninh = 0$ $S.err = exp.err$
$exp_1 \rightarrow exp_2 + exp_3$	$exp_2.simtab = exp_1.simtab$ $exp_3.simtab = exp_1.simtab$ $exp_2.nivelalinh = exp_1.nivelalinh$ $exp_3.nivelalinh = exp_1.nivelalinh$ $exp_1.err = exp_2.err$ **or** $exp_3.err$
$exp_1 \rightarrow (exp_2)$	$exp_2.simtab = exp_1.simtab$ $exp_2.nivelalinh = exp_1.nivelalinh$ $exp_1.err = exp_2.err$
$exp \rightarrow$ **id**	$exp.err =$ **not** $estaem(exp.simtab.$**id**$.nome)$
$exp \rightarrow$ **num**	$exp.err =$ **false**

Gramática de Atributos para Tabela de Símbolos

Regra Gramatical	Regras Semânticas
$exp_1 \rightarrow$ **let** *dec-lista* **in** exp_2	$\textit{dec-lista.enttab} = exp_1.\textit{simtab}$ $\textit{dec-lista.nivelalinh} = exp_1.\textit{nivelalinh} + 1$ $exp_2.\textit{simtab} = \textit{dec-lista.saitab}$ $exp_2.\textit{nivelalinh} = \textit{dec-lista.nivelalinh}$ $exp_1.\textit{err} = (\textit{decl-lista.saitab} = \textit{errtab})$ **or** $exp_2.\textit{err}$
$\textit{decl-lista}_1 \rightarrow \textit{dec-lista}_2, \textit{decl}$	$\textit{dec-lista}_2.\textit{enttab} = \textit{dec-lista}_1.\textit{enttab}$ $\textit{dec-lista}_2.\textit{nivelalinh} = \textit{dec-lista}_1.\textit{nivelalinh}$ $\textit{decl.enttab} = \textit{dec-lista}_2.\textit{saitab}$ $\textit{decl.nivelalinh} = \textit{dec-lista}_2.\textit{nivelalinh}$ $\textit{decl-lista}_1.\textit{saitab} = \textit{decl.saitab}$
$\textit{decl-lista} \rightarrow \textit{decl}$	$\textit{decl.enttab} = \textit{dec-lista.enttab}$ $\textit{decl.nivelalinh} = \textit{dec-lista.nivelalinh}$ $\textit{dec-lista.saitab} = \textit{decl.saitab}$

Gramática de Atributos para Tabela de Símbolos

Regra Gramatical	Regras Semânticas
$decl \rightarrow \textbf{id} = exp$	$exp.simtab = decl.enttab$ $exp.nivelalinh = decl.nivelalinh$ $decl.saitab =$ $\textbf{if } (decl.enttab = errtab) \textbf{ or } exp.err$ $\textbf{then } errtab$ $\textbf{else if } (verificar(decl.enttab, \textbf{id}.nome) = decl.nivelaninh$ $\textbf{then } errtab$ $\textbf{else } insert(decl.enttab, \textbf{id}.nome, decl.nivelaninh)$

Tipos de Dados e Verificações de Tipos

Inferência de Tipos

Computar e manter informações sobre tipos de dados de acordo com as declarações.

Verificação de Tipos

Garantir que cada parte de um programa faça sentido para as regras de tipo da linguagem.

- Em geral, chamamos as duas tarefas apenas de **verificação de tipos**;
- informações estáticas sobre tipos:
  - quantidade de memória requerida;
  - forma de acesso da memória.

Tipos de Dados e Verificações de Tipos

Tipos de Dados

É um conjunto de valores ou, mais precisamente, um conjunto de valores com certas operações sobre esses valores.

- O tipo **int**, representa o conjunto de valores inteiros, nos quais podemos executar operações aritméticas;
- expressões de tipos:

```
// Apenas um nome de tipo
int a, b, c;
// Expressão estruturada
int vetor[100];
// Definição de novo tipo
typedef int notas[100];
```

Tipos de Dados e Verificações de Tipos

- As informações de tipo podem ser explícitas, como nos exemplo em C;
- ou implícitas nas linguagens dinâmicas como Python:

```
universidade = "UFC Quixadá"
```

- considerando a linguagem C, a referência:

```
a[i]
```

- os tipos de dados de *a* e *i* são recuperados da tabela de símbolos;
- se *a* for do tipo *double*[] e *i* for *int*, a subexpressão *a*[*i*] será do tipo *double*;

Expressões de Tipos e Construtores de Tipos

- Os tipos **predefinidos** ou **primitivos** correspondem a tipos de dados numéricos fornecidos internamente por diversas arquiteturas de máquinas, como *int*, *double*, etc;
- os **construtores de tipos** como *typedef* ou *struct* podem ser vistos como funções que tomam tipos existentes como parâmetro e retornam novos tipos;
- são os chamados **tipos estruturados**;
- um construtor de tipos corresponde a uma operação de conjuntos sobre os conjuntos subjacentes de valores dos seus parâmetros.

Expressões de Tipos e Construtores de Tipos

Matrizes ou Vetores

Dois parâmetros de tipos: **tipo de índice** e **tipo de componente**.
tipo componente vetor[*tipo índice*];

- Na maioria das linguagens, o tipo de índice deve pertencer ao conjunto de **tipos ordinais**;
- uma matriz representa valores que são sequências de valores do tipo componente, indexados pelos valores do tipo do índice;
- tipo índice com conjunto de valores I e o tipo componente com valores C: um vetor é o conjunto de funções $I \rightarrow C$.
- matrizes multidimensionais: deteminadas pela **coluna** ou pela **linha**;
- a tabela pode ter apenas o endereço inicial e o tipo componente, ou também incluir o tamanho.

Expressões de Tipos e Construtores de Tipos

Registros

Recebe uma lista de nomes e tipos associados e constrói um novo tipo.

```
struct {
   double r;
   int i;
}
```

- Para o exemplo acima, o registro corresponde a $(\mathbf{r}xR)x(\mathbf{i}xI)$ onde r e i representam o espaço de nomes possíveis, enquando R e I representam todos valores possíveis reais e inteiros;
- o registro é em geral alocado em sequência na memória;
- na tabela, entram o endereço inicial e o tipo de cada membro.

Expressões de Tipos e Construtores de Tipos

União

Corresponde à operação de união dos conjuntos de valores possíveis.

```
union {
    double r;
    int i;
}
```

- aloca-se o espaço necessário para o maior membro, nele se armazena o valor, seja acessado como inteiro ou real;
- a representação em *bytes* do inteiro é diferente da representação real;
- $(\mathbf{r}xR) \cup (\mathbf{i}xI)$;
- a tabela tem que guardar o endereço da memória e uma lista de tipos possíveis.

Expressões de Tipos e Construtores de Tipos

Ponteiro

O valor de um tipo ponteiro é um endereço de memória.

- o espaço ocupado depende da arquitetura;
- se a linguagem permitir aritmética de ponteiros, a tabela tem que guardar, além do endereço, o tamanho do tipo do ponteiro, para saber quantos *bytes* devem ser avançados por vez;
- a operação básica do tipo ponteiro é **derreferenciação**.

Expressões de Tipos e Construtores de Tipos

Função

A função é um bloco de código invocável.

- a tabela precisa guardar o endereço onde começa o código da função;
- o tipo do valor de retorno;
- a lista de argumentos e seus tipos.

Classe

Similar a registros, porém com funções membro ou **métodos**.

- em linguagens orientadas à objetos, uma classe pode definir um novo tipo;
- a tabela armazena o endereço de um objeto, que deve ter sua memória organizada de acordo com o tipo definido pela classe;

Nomes de Tipos, Declarações e Tipos Recursivos

```c
// Declaração de tipo em C
typedef struct {
    double r;
    int i;
} RealIntRec;
// Declaração detalhada
struct RealIntRec {
    double r;
    int i;
};
typedef struct RealIntRec RealIntRec;
```

As declarações de tipos levam os nomes de tipos declarados a serem fornecidos na tabela de símbolos.

Nomes de Tipos, Declarações e Tipos Recursivos

```c
// Declaração de tipo ilegal em C
struct intBST {
    int isNull;
    int val;
    struct intBST left, right;
};
// Declaração válida
struct intBST {
    int val;
    struct intBST *left, *right;
};
typedef struct intBST * intBST;
```

É importante lembrar que ponteiros sempre ocupam o mesmo espaço na memória.

Equivalência de Tipos

Motivação

Dadas as expressões possíveis de tipos de uma linguagem, um *verificador de tipos* deve responder se duas expressões de tipos representam o mesmo tipo.

Imagine uma função:

```
function tipoIgual(t1, t2: TipoExp): Booleano;
```

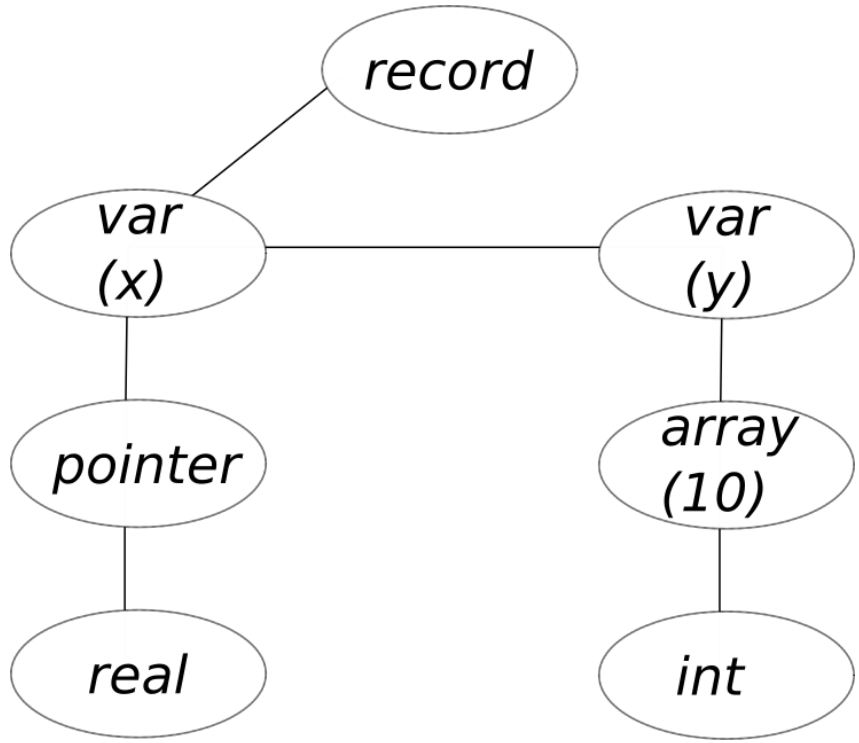
Recebe duas árvores, que representam derivações de expressões de uma gramática de linguagem, e retorna se ambas expressões são do mesmo tipo.

Equivalência de Tipos

var-decls → *var-decls*;*var-decl*|*var-decl*
var-decl → **id:***tipo-exp*
tipo-exp → *tipo-simples*|*tipo-estruturado*
tipo-simples → **int**|**bool**|**real**|**char**|**void**
tipo-estruturado → **array [num] of** *tipo-exp*
|**record** *var-decls* **end**
|**union** *var-decls* **end**
|**pointer to** *tipo-exp*
|**proc** (*tipo-exps*) *tipo-exp*
tipo-exps → *tipo-exps*,*tipo-exp*|*tipo-exp*

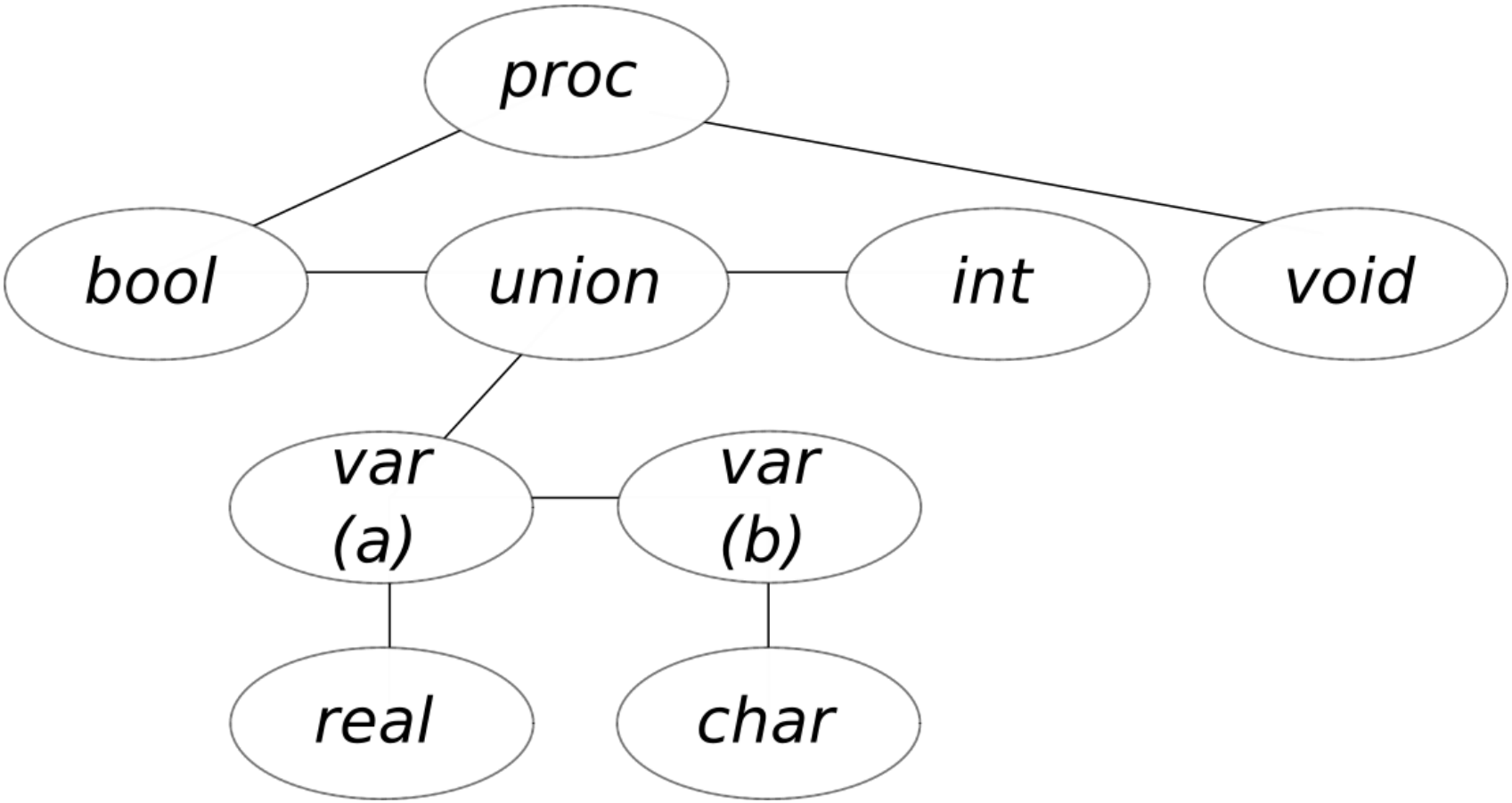
Equivalência de Tipos

```
record
  x: pointer to real;
  y: array [10] of int
end
```



Equivalência de Tipos

```
proc(bool, union a:real; b:char end, int):void
```



Equivalência de Tipos

Equivalência Estrutural

Dois tipos são iguais se e somente se tiverem a mesma estrutura.

- Em termos de árvores, dois tipos são equivalente se suas árvores tiverem a mesma **estrutura**;
- os atributos em cada nó das árvores podem ser diferentes.

Equivalência de Tipos

```
function tipoIgual (t1, t2:TipoExp):Booleano;
var temp:Booleano;
    p1, p2:TipoExp;
begin
   if t1 e t2 são de tipos simples then
      return t1 = t2
   else if (t1.tipo = matriz
            and t2.tipo = matriz) then
      return (t1.tamanho = t2.tamanho
            and tipoIgual(t1.filho1, t2.filho1))
   else if ((t1.tipo = registro
            and t2.tipo = registro)
            or (t1.tipo = uniao
            and t2.tipo = uniao)) then
   begin
      p1 := t1.tipo1;
      p2 := t2.tipo1;
      temp := true;
      while temp and p1 != nil and p2 != nil do
         if p1.nome != p2.nome then
            temp := false
         else if not tipoIgual(p1.filho1, p2.filho1)
         then temp := false
         else begin
           p1 := p1.irmão;
           p2 := p2.irmão;
         end;
      return temp and p1 = nil and p2 = nil;
   end
   else if (t1.tipo = ponteiro
            and t2.tipo = ponteiro) then
      return tipoIgual(t1.filho1, t2.filho1)
   else if (t1.tipo = proc and t2.tipo = proc) then
   begin
      p1 := t1.filho1;
      p2 := t2.filho1;
      temp := true;
      while temp and p1 != nil and p2 != nil do
         if not tipoIgual(p1.filho1, p2.filho1)
         then temp := false
         else begin
           p1 := p1.irmão;
           p2 := p2.irmão;
         end;
      return temp and p1 = nil and p2 = nil
             and tipoIgual(t1.filho2, t2.filho2)
   end
   else return false;
end; (* tipoIgual *)
```

Equivalência de Tipos

Uma alteração na gramática para permitir novos nomes para tipos:

var-decls → *var-decls* **;** *var-decl*|*var-decl*

var-decl → **id :** *exp-tipo-simples*

tipo-decls → *tipo-decls* **:** *tipo-decl*|*tipo-decl*

tipo-decl → **id =** *tipo-exp*

tipo-exp → *exp-tipo-simples*|*tipo-estruturado*

exp-tipo-simples → *tipo-simples*|**id**

tipo-simples → **int**|**bool**|**real**|**char**|**void**

tipo-estruturado → **array [num] of** *tipo-exp*

|**record** *var-decls* **end**

|**union** *var-decls* **end**

|**pointer to** *tipo-exp*

|**proc** (*tipo-exps*) *tipo-exp*

tipo-exps → *tipo-exps*,*exp-tipo-simples*|*exp-tipo-simples*

Equivalência de Tipos

No lugar de:

```
record
    x: pointer to real;
    y: array [10] of int
end
```

precisamos ter:

```
    t1 = pointer to real;
    t2 = array [10] of int;
    t3 = record
             x: t1;
             y: t2
    end
```

Equivalência de Nomes: duas expressões de tipos são equivalentes se e somente se forem o mesmo tipo simples ou o mesmo nome de tipo.

Equivalência de Tipos

```
function tipoIgual(t1,t2: TipoExp): Booleano;
var temp: Booleano;
    p1, p2: TipoExp;
begin
   if t1 e t1 são de tipos simples then
      return t1 = t2
   else if t1 e t2 são nomes de tipos then
      return t1 = t2
   else return falso;
end;
```

Equivalência de Tipos

- A **equivalência estrutural** tem alta complexidade;
- a **equivalência de nomes** é muito restrita;
- **equivalência de declarações**:
  - cada nome é equivalente a um nome básico ou a uma expressão de tipos;
  - toda expressão de tipos é cadastrada na tabela de símbolos;
  - novos nomes que utilizam a mesma expressão apontam para a mesma entrada na tabela.
- a tabela de símbolos deve oferecer uma nova operação *capturaNomeBaseTipo*.

Equivalência de Tipos

Considere, para equivalência de declarações:

```
t1 = array [10] of int;
t2 = array [10] of int;
t3 = t1;
```

Tanto $t1$ quanto $t2$ e $t3$ apontam para a entrada na tabela de *array [10] of int*. O compilador precisa, para cada expressão de tipos, verificar se a mesma já existe na tabela.

Inferência e Verificação de Tipos

Considerando que temos:

- Uma operação *tipolgual* que retorna verdadeiro ou falso;
- uma operação de *inserir* na tabela de símbolos;
- uma operação de *verificar* na tabela de símbolos.

Vamos simplificar a gramática para:
programa $\rightarrow$ *var-decls* **;** *decls*
var-decls $\rightarrow$ *var-decls* **;** *var-decl* | *var-decl*
var-decl $\rightarrow$ **id :** *tipo-exp*
tipo-exp $\rightarrow$ **int** | **bool** | **array [num] of** *tipo-exp*
decls $\rightarrow$ *decls* **;** *decl* | *decl*
decl $\rightarrow$ **if** *exp* **then** *decl* | **id :** *exp*
Vamos definir uma gramática de atributos que verifique se as expressões estão corretas de acordo com o sistema de tipos.

Inferência e Verificações de Tipos

Regra Gramatical	Regras Semânticas
var-decl $\rightarrow$ **id :** *tipo-exp*	*inserir*(**id**.*nome*, *tipo-exp.tipo*)
tipo-exp $\rightarrow$ **int**	*tipo-exp.tipo* := *inteiro*
tipo-exp $\rightarrow$ **bool**	*tipo-exp.tipo* := *booleano*
tipo-exp$_1$ $\rightarrow$ **array[num] of** *tipo-exp*$_2$	*tipo-exp*$_1$.*tipo* := criaTipoNó(matriz, **num**.tamanho,*tipo-exp*$_2$.tipo)
decl $\rightarrow$ **if** *exp* **then** *decl*	**if not** *tipolgual*(*exp.tipo, booleano*) **then** *tipo-erro(decl)*
decl $\rightarrow$ **id :=** *exp*	**if not** *tipolgual*(*verificar*(**id**.*nome*) , *exp.tipo*) **then** *tipo-erro(decl)*

Inferência e Verificações de Tipos

Regra Gramatical	**Regras Semânticas**
$exp_1 \rightarrow exp_2 + exp_3$	**if not** (*tipoIgual*(exp_2.*tipo*,*inteiro*) **and** *tipoIgual*(exp_3.*tipo*, *inteiro*)) **then** *tipo-erro*(exp_1); exp_1.*tipo* := *inteiro*;
$exp_1 \rightarrow exp_2$ **or** exp_3	**if not** (*tipoIgual*(exp_2.*tipo*,*booleano*) **and** *tipoIgual*(exp_3.*tipo*, *booleano*)) **then** *tipo-erro*(exp_1); exp_1.*tipo* := *booleano*;
$exp_1 \rightarrow exp_2[exp_3]$	**if not** *éTipoMatriz*(exp_2.*tipo*) **and** *tipoIgual*(exp_3.*tipo*, *inteiro*) **then** exp_1.*tipo* := exp_2.*tipo*.*filho*1 **else** *tipo-erro*(exp_1)

Inferência e Verificações de Tipos

Regra Gramatical	**Regras Semânticas**
$exp \rightarrow$ **num**	$exp.tipo := inteiro$
$exp \rightarrow$ **true**	$exp.tipo := booleano$
$exp \rightarrow$ **false**	$exp.tipo := booleano$
$exp \rightarrow$ **id**	$exp.tipo := verificar(\textbf{id}.nome)$

Tópicos Adicionais em Verificações de Tipos

- Sobrecarga de operadores;
- conversão e coação entre tipos;
- tipos polimórficos.

FIM

Dúvidas?