

# Análise Semântica: Verificação de Tipos

# Introdução

- Verifica se as construções sintaticamente corretas possuem significado lógico dentro da linguagem
- Verifica a consistência da declaração e uso dos identificadores
- Além disso, deve fazer conversões necessárias, e permitidas, para dar significado a uma sentença

# Introdução

- Durante a análise semântica também podem ser obtidas algumas conclusões que permitem ao compilador alterar a árvore sintática, de modo que redundâncias e ineficiências sejam, sempre que possível, eliminadas, obtendo-se uma árvore que descreve um programa equivalente, porém mais eficiente.

# Introdução

- A verificação de tipos pode ser estática e dinâmica
  - Verificação estática - é feita no código fonte, no processo de compilação
  - Verificação dinâmica – é feita durante o tempo de execução
- Concentraremos mais na primeira

# Verificação Estática

- Exemplos de verificação estática incluem:
  - *Verificação de tipos:*
    - Um compilador deve relatar um erro se um operador for aplicado a um operando incompatível
    - Por exemplo:
      - Se uma variável array é somada a um identificador de um procedimento
      - Uma variável do tipo inteiro é somada com uma variável do tipo string em Pascal

# Verificação Estática

- Considere o seguinte exemplo de código em C:

```
int f1(int a, float b)
{
    return a % b;
}
```

- A tentativa de compilar esse código irá gerar um erro detectado pelo analisador semântico, mais especificamente pelas regras de **verificação de tipos**, indicando que o operador módulo % não pode ter um operador real.

# Verificação Estática

## – *Verificação do fluxo de controle:*

- Os comandos que fazem o fluxo de controle deixar uma construção precisam ter algum local para onde transferir o controle .
- Exemplo: O comando break em C faz com que o fluxo de controle saia do while, for ou switch mais interno. Um erro ocorre se este comando não aparecer dentro de pelo menos um while, for ou switch

# Verificação Estática

## – *Verificações de unicidade:*

- Existem situações nas quais um objeto precisa ser definido exatamente uma vez
- Por exemplo: Em Pascal:
  - Um identificador precisa ser declarado univocamente no mesmo nível
  - os rótulos em enunciados case precisam ser distintos.
  - os elementos em um tipo escalar não podem ser repetidos

# Verificação Estática

- *Verificação relacionada aos nomes:*
  - Algumas vezes, o mesmo nome precisa figurar duas ou mais vezes.
  - Por exemplo:
    - em Ada, um laço ou bloco precisa ter um nome que apareça ao início e ao final da construção.
    - O compilador precisa verificar se o mesmo nome é usado em ambos os locais

# Verificação de Tipos

- Nas linguagens de programação em geral os tipos podem ser:
  - **Básicos** - não possuem estrutura interna. Ex.: inteiro, real, booleano, carácter, *string*, intervalo e enumeração
  - **Construídos** - possuem uma estrutura interna composta por tipos básicos e outros tipos construídos. Ex.: ponteiros, estruturas, registros, vetores e matrizes

# Verificação de Tipos

- A **expressão de tipo** é o tipo de uma construção de linguagem qualquer
- São expressões de tipos:
  - Os tipos básicos  
`char, integer, real, boolean`
  - Nomes de expressões  
`type linha = record  
    idade: interger;  
    nome: array [1..15] of char;  
end;  
var tabela : aray [1..101] of linha;`

# Verificação de Tipos

- Um construtor de tipo aplicado sobre uma expressão de tipo (vetores, apontadores, funções e procedimentos)

```
var A: array[1..10] of integer;
```

```
var p: ^integer;
```

```
função c(a,b: integer): integer;
```

# Verificação de Tipos

- Um **sistema de tipos** é uma coleção de regras para expressões de tipos de um programa
- Podem ser especificados na forma de **definições dirigidas pela sintaxe (ações semânticas)**

# Verificação de Tipos

- Declaração de Tipos

D → id: T {incluir\_ID(id.entrada, T.tipo)}

T → char {T.tipo:= caracter}

T → integer {T.tipo:= inteiro}

T → array [num] of T {T.tipo:= vetor(1..num.val, T.tipo)}

T → ↑ T {T.tipo:= apontador(T.tipo)}

# Verificação de Tipos

- $\text{incluir\_ID}(e, t)$ : insere um identificador  $e$  com o tipo  $t$  na tabela de símbolos
- $\text{caracter}$  e  $\text{inteiro}$ : são tipos básicos
- $\text{vetor}(a1, an, t)$ : tipo construído vetor de elementos do tipo  $t$  e intervalo de variação de índices  $a1$  até  $an$
- $\text{apontador}(t)$ : tipo construído apontador para o tipo básico  $t$

# Verificação de Tipos

- Tipos de Expressões

$E \rightarrow \text{literal}$                        $\{E.\text{tipo} := \text{caracter}\}$

$E \rightarrow \text{num}$                                $\{E.\text{tipo} := \text{inteiro}\}$

$E \rightarrow \text{id}$                                  $\{E.\text{tipo} := \text{procurar}(\text{id.entrada})\}$



# Checagem compilação vs execução

- Checagem dinâmica x estática
- Estática (em tempo de compilação)
  - Antes da execução
  - Compilador maior, restringe flexibilidade, menos expressiva
- Dinâmica (tempo de execução)
  - Pode ser muito tarde
  - Cara, mas mais flexível

# Exemplo de checagem de Tipos

– Uma linguagem de programação simples:

- Variáveis devem ser definidas antes de serem usadas

N: integer;

N mod 1999

# Exemplo de checagem de Tipos

- P -> D ; E
- D -> D ; D
- D -> id: T {addTS(id.lexema, T.tipo);}
- T -> char {T.tipo=char;}
- T -> integer {T.tipo=integer;}
- T -> array [num] of T<sub>1</sub> {T.tipo=array(num,T<sub>1</sub>.tipo);}
- T -> ^T<sub>1</sub> {T.tipo=ponteiro(T<sub>1</sub>.tipo);}
- E -> literal {E.tipo=char;}
- E -> num {E.tipo=integer;}
- E -> id {E.tipo=consulta\_TS(id.lexema);}
- E -> E<sub>1</sub> mod E<sub>2</sub> {if E1.tipo==integer && E2.tipo==integer then  
    E.tipo=integer;  
    else  
        E.tipo=tipo\_erro;}
- E -> E<sub>1</sub> [E<sub>2</sub>] {if E<sub>2</sub>.tipo==integer && E<sub>1</sub>.tipo==array then  
    E.tipo=array;  
    else  
        E.tipo=tipo\_erro;}

# Exemplo de checagem de Comandos

- P -> D ; S
- D -> D ; D
- D -> id: T {addTS(id.lexema, T.tipo);}
- T -> char {T.tipo=char;}
- T -> integer {T.tipo=integer;}
- T -> array [num] of T<sub>1</sub> {T.tipo=array(num,T<sub>1</sub>.tipo);}
- T -> ^T<sub>1</sub> {T.tipo=ponteiro(T<sub>1</sub>.tipo);}
- S -> id=E {if id.tipo==E.tipo then  
    S.tipo=void;  
    else  
        S.tipo=tipo\_erro;}
- S -> if E then S<sub>1</sub> {if E.tipo==boolean then  
    S.tipo=S<sub>1</sub>.tipo;  
    else  
        S.tipo= tipo\_erro;}
- S -> while E do S<sub>1</sub> {if E.tipo==boolean then  
    S.tipo=S<sub>1</sub>.tipo;  
    else  
        S.tipo= tipo\_erro;}
- S -> S<sub>1</sub> ; S<sub>2</sub> {if S<sub>1</sub>.tipo==void && S<sub>2</sub>.tipo==void then  
    S.tipo=void;  
    else  
        S.tipo=tipo\_erro;}

**Crie as regras de produção e as regras semânticas que faltam para os símbolos E**

# Tabela de Símbolos

# Tabela de Símbolos

- Um compilador necessita coletar e usar informações sobre os nomes (identificadores) que aparecem no programa-fonte
- Estas informações são colocadas numa estrutura de dados chamada Tabela de Símbolos (TS)

# Tabela de Símbolos

- As informações coletadas sobre o nome incluem
  - a cadeia de caracteres pelo qual é denotado, isto é, o *lexeme* do identificador
  - seu *tipo* - inteiro, real, booleano etc
  - sua *forma* - variável simples, identificador de tipo, identificador de função, identificador de procedimento...
  - seu *tamanho*
  - sua *localização na memória* e outros atributos, dependendo da linguagem

# Tabela de Símbolos

- Algumas mudanças serão realizadas em nosso analisador lexico/sintático
- A primeira mudança é a retirada da inserção dos identificadores na Tabela de Símbolos na análise léxica
- As ações semânticas inseridas no analisador sintático serão responsáveis por inserir os identificadores na Tabela de Símbolos.

# Tabela de Símbolos

- O motivo disto é a checagem da semântica da linguagem PGL
- Vamos inicialmente definir a estrutura da Tabela de Símbolos

# Tabela de Símbolos

- A tabela de símbolos então será responsável pela organização da informação dos identificadores que o compilador “conhece”
- O compilador associa com cada identificador presente na tabela de símbolos um certo número de atributos

# Tabela de símbolos

- Os atributos associados com cada identificador dependerão de sua categoria
- Alguns exemplos são:
  - **Variáveis Simples:** categoria, tipo e endereço
  - **Parâmetro Simples:** categoria, tipo, endereço, mecanismo de passagem (por valor ou referência)
  - **Procedimentos:** categoria, rótulo interno (no programa objeto), nível em que foi declarado, número de parâmetros, tipo e mecanismo de passagem de cada parâmetro

# Tabela de Símbolos

- Nos slides a seguir, é apresentada a estrutura de dados de uma Tabela de Símbolos para a linguagem PGL
- Assume-se que a tabela de símbolos seja representada por uma árvore binária
- A mesma definição poderia ser feita usando-se uma tabela hash

# Categoria dos identificadores

```
typedef enum{  
    PARAMETRO, FUNCAO, VARS,  
    PROCEDIMENTO, IDTIPO,  
    CONSTANTE, _LABEL,  
    CAMPOREG, LEITURA, ESCRITA  
  
} CATEGORIAS;
```

# Tabela de Símbolos

```
typedef struct tab_simbolo {  
    char *identificador; // ponteiro para lexeme  
    CATEGORIAS categoria;  
    int nivel; // nível em que o id foi definido  
    union{  
        // identificador de tipo  
        struct  
        {  
            D_tipos *tipo; // ptr para tipo no DT  
        }id_tipo;  
    };  
};
```

# Tabela de Símbolos

// variavel simples

struct

{

    D\_tipos \*tipo;       // tipo da variável simples

    int deslocamento; // deslocamento na pilha

} vars;

// parametro

struct

{

    D\_tipos \*tipo;       // tipo do parâmetro

    int deslocamento; // deslocamento na pilha

    PP passagem;    // PP pode ser valor ou referência

} param;

# Tabela de Símbolos

```
//procedimento
```

```
struct{
```

```
    int rotulo; // rotulo do procedim.
```

```
    int n;      // número de parâmetros
```

```
    // lista com os ponteiros dos
```

```
    // parâmetros na tabela de Simb.
```

```
    ListaParam *listaParametros;
```

```
}proced;
```

# Tabela de Símbolos

// função

```
struct{
```

```
    D_tipos *tipo; // tipo do retorno
```

```
    int rotulo; // rotulo da funcao
```

```
    int n; // numero de parametros
```

```
    // lista de ptr dos parâmetros na TS
```

```
    ListaParam *listaParametros;
```

```
}func;
```

# Tabela de Símbolos

```
// constante
```

```
struct
```

```
{
```

```
    D_tipos *tipo;
```

```
    int valor;
```

```
} constante;
```

```
// label
```

```
struct {
```

```
    int label;
```

```
} label;
```

```
}u; // fim da union
```

```
struct tab_simbolo *pesq, *pdir; // se for usar arvore binária
```

```
} TAB_SIMBOLO;
```

# Tabela de Símbolos

- Neste caso TAB\_SIMBOLO representa o nó de uma árvore binária
- Se fôssemos implementar um hashing, a primeira mudança seria deixar somente um ponteiro dentro de TAB\_SIMBOLO (\*prox ao invés de \*pesq e \*pdir)

Além disso teríamos que definir:

```
TAB_SIMBOLO *tab_hash[211];
```

# Descritor de tipos

- Antes de definir o descritor de tipos vejamos as estruturas que este utiliza:
- A primeira estrutura é uma lista encadeada de ponteiros para constantes na Tabela de Símbolos

```
typedef struct List_Const
{
    TAB_SIMBOLO *ptr;    // apontador para a
                        // constante na TS
    struct List_Const *prox; // proxima constante
}Lista_Constantes;
```

# Descritor de tipos

- A segunda estrutura de dados necessária é uma lista encadeada para armazenar os campos de um registro

```
typedef struct Registro
```

```
{
```

```
    char *ident;           // identificador associado ao registro  
    CATEGORIAS categoria; // categoria do campo de registro  
    D_Tipos *tipo;        // tipo do campo  
    int deslocamento;     // deslocamento dentro do registro  
    struct Registro *prox; // apontador para o proximo registro
```

```
}Tab_Campos;
```

# Descritor de tipos

```
typedef struct Desc_tipos{  
    CONSTRUTOR construtor;  
    int tam;    // tamanho em bytes ocupado pelo tipo  
    union  
    {  
        // intervalo  
        struct{  
            int num_elementos;  
            int lim_inferior;  
            int lim_superior;  
            struct Desc_tipos *tipo; // tipo elem intervalo  
        }  
    }  
};
```

# Descritores de tipos

// enumeração

```
struct{  
    int num_elementos;  
    ListaConstantes *l_constantes;  
} Enum;
```

// array

```
struct{  
    int num_elementos;  
    struct Desc_tipos *tipo_ind;  
    struct Desc_tipos *tipo_elem;  
} Array;
```

# Descritor de tipos

```
// pointer
struct{
    struct Desc_tipos *tipo_elem;
} Pointer;
// registro
struct{
    Tab_Campos *tab_campos;
} record;
} ivariante; // fim da union
}D_tipos;
```

# Exemplo

```
tipo cores = ( branco, vermelho, verde, azul, preto);
```

```
  rgb = vermelho..azul;
```

```
  T = vetor[1..2000] of rgb;
```

```
declare v : T
```

```
  P : vetor[boolean, 1..10] of T;
```

- A definição de P acima é equivalente à:

```
P : vetor[boolean] de vetor [1..10] de T;
```

# Tabela de Símbolos

ID	CAT	NIVEL	TIPO	VAL	DESL
0 - "boolean"	IDTIPO	-1	0	-	-
1 - "false"	CONST	-1	0	0	
2 - "true"	CONST	-1	0	1	
3 - "integer"	IDTIPO	-1	1		
4 - "cores"	IDTIPO	0	2		
5 - "branco"	CONST	0	2	0	
6 - "vermelho"	CONST	0	2	1	
7 - "verde"	CONST	0	2	2	
8 - "azul"	CONST	0	2	3	
9 - "preto"	CONST	0	2	4	
10- "rgb"	IDTIPO	0	3	-	
11- "T"	IDTIPO	0	4	-	
12- "v"	VARIS	0	4	-	0
13- "P"	VARIS	0	6		2000

# Descritor de tipos

	CONST	TAM	NELEM	L.CONST	Mr. V	Ma. V	Tipo	TI	TE
0-	ENUM	1	2	[1,2]*					
1-	INTERVALO	2	-	-	-32768	32767	1	-	-
2-	ENUM	1	5	[5,6,7,8,9]*					
3-	INTERVALO	1	3		1	3	2		
4-	ARRAY	2000	2000					5	3
5-	INTERVALO	1	2000		1	2000	5		
6-	ARRAY	40000	20					0	7
7-	ARRAY	20000	10					8	4
8-	INTERVALO	1	10		1	10	8		

\*São ponteiros  
da tabela de  
símbolos

# Legenda:

- No slide anterior temos as legendas:

CONST = construtor

TAM = tamanho em bytes

NELEM = número de elementos

L.CONST = lista de constantes da enumeração

Mr. V = menor valor

Ma. V = maior valor

Tipo = tipo dos elementos

TI = tipo do índice do array

TE = tipo do elemento do array

# Exercício

- Mostre a tabela de símbolos e o descritor de tipos para o trecho de código abaixo:

```
tipo meses = (janeiro, fevereiro, marco, abril, maio, junho,  
             julho, agosto, setembro, outubro, novembro, dezembro);
```

```
ptrMes = ^meses;
```

```
declare m : meses;
```

```
  p : ptrMes;
```

```
  v1: vetor [meses] de ptrMes;
```

```
  v2: vetor [ 1..10] de meses;
```

```
  v3: vetor [junho..novembro] de array[janeiro..maio] de  
integer;
```

- Os tipos pré-definidos, integer, boolean, true e false devem ser inicialmente inseridos no nível -1

# Exercício

- Os tipos pré-definidos, integer, boolean, true e false devem ser inicialmente inseridos no nível -1