Verificação de Tipo

- Atividade que certifica que os operandos de um operador são de tipos compatíveis.
- São considerados operadores e operandos:
  - operadores usuais (aritméticos, relacionais, etc)
  - subprogramas (operadores) e parâmetros (operandos)
  - atribuição (operador) e variavel / expressão (operandos)
- Os tipos de operandos são compatíveis com um operador se:
  - são aceitos pelo operador ou
  - podem ser convertidos implicitamente pelo compilador (coerção)
- Erro de tipo: aplicação de um operador a um operando de tipo não apropriado.

Verificação de tipo estática

- Feita antes da execução
- Amarração de tipo estática permite quase sempre a verificação de tipo estática
- Vantagem da detecção de erros em tempo de compilação: quanto antes e erro for encontrado, menor o custo.
- Desvantagem: redução da flexibilidade para o programador
- Verificação de tipo deve ser dinâmica quando a mesma posição de memória pode armazenar valores de tipos diferentes em momentos diferentes durante a execução (record variante, union, equivalence).

Verificação de tipo dinâmica

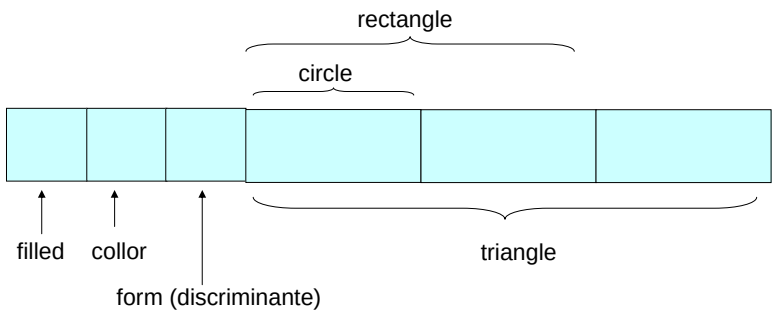
- Feita durante a execução do programa.
- Amarração de tipo dinâmica exige verificação de tipo dinâmica.
- Amarração de tipo dinâmica – o tipo da variável é definido durante a execução, por um comando de atribuição
- Exemplo: APL
  List <- 10.5 5.3 4.1 0.0 (list é um array do tipo float)
  List <- 38 (list é um inteiro escalar)

Unions

- A verificação de tipo dinâmica também deve ser feita quando a mesma posição de memória pode armazenar valores de tipos diferentes em momentos diferentes durante a execução
- Ada e Pascal : record variante
- FORTRAN: EQUIVALENCE
- C, C++ : unions

Exemplo – record variante (Pascal)

```
type shape = (circle, triangle, rectangle);
     colors = (red, green blue);
     figure =
      record
          filled: boolean;
          color: colors;
          case form : shape of
             circle: (diameter : real);
             triangle: (leftside : integer;
                        rightside : integer;
                        angle : real);
             rectangle : (side1 : integer;
                          side2 : integer)
           end;
  var myfigure : figure;
```



Tipagem forte

- Tanto na verificação de tipo estática quanto na dinâmica, alguns erros não podem ser encontrados.
- A capacidade da linguagem de sempre detectar erros está relacionada ao conceito de tipagem forte.
- Uma linguagem é considerada ***fortemente tipada*** se erros de tipo podem sempre ser detectados.
- Isso exige que os tipos de todos os operandos possam ser determinados, em tempo de compilação ou de execução.

- FORTRAN não é fortemente tipada
  - não faz verificação de tipo entre parâmetros formais e reais
  - EQUIVALENCE permite acesso a mesma posição de memória por variáveis de tipos diferentes
- Pascal é quase fortemente tipada
  - record variante permite omissão do tag que armazena o tipo corrente de uma variável

- C, C++ não são fortemente tipadas
  - estruturas do tipo union são checadas
  - permitem uso de funções que não fazem verificação de tipo de parâmetros
  - permitem várias formas de conversão automática de tipo (coerção)
- Java é fortemente tipada
  - permite conversão explicita de tipo, que pode resultar em erro de tipo
  - conversão automática é restrita

Compatibilidade de tipo

- Linguagens podem usar diferentes regras para compatibilidade de tipo.
- Essas regras influenciam a decisão sobre quais tipos de dados e operações serão incluídas na linguagem.
- Métodos de compatibilidade de tipo:
- Compatibilidade de nome
  - Duas variáveis tem o mesmo tipo se estiverem na mesma declaração ou em declaração com o mesmo nome de tipo
- Compatibilidade de estrutura
  - Duas variáveis são compatíveis se seus tipos tiverem a mesma estrutura

Compatibilidade de nome

- Fácil de implementar
- Muito restritiva:
  - Tipo intervalo de inteiros não seria compatível com inteiros

```
type indextype = 1 .. 100;
var     count : integer;
        index : indextype
```

(count e index não seriam compatíveis)

  - Tipos estruturados passados como parâmetros tem que ser declarados globalmente (primeiras versões do Pascal)

Compatibilidade de estrutura

- Mais flexível
- Difícil de implementar
- Outras questões:
  - Toda a estrutura deve ser comparada para verificar a compatibilidade
  - Duas estruturas que tem a mesma estrutura com diferentes nomes para os campos, são compatíveis?

Equivalência por declaração

- As regras de compatibilidade das linguagens são muitoas vezes definidas combinando os dois métodos.
- Compatibilidade por estrutura é mais usada, exceto em algumas situações específicas, como a equivalência por declaração:

```
type
type1 = array [1..10] of integer;
type2 = type1;
```

- Se um tipo é definido com o nome de outro tipo, os dois são compatíveis, mas não são compatíveis por nome.

- C usa equivalência estrutural para quase todos os tipos, exceto tipos estruturados (record, unions)
- Para os tipos estruturados usa equivalência por declaração.
- C++ usa equivalência de nome (typedef não define um novo tipo, apenas define um novo nome para um tipo existente).
- Linguagens OO (C++, JAVA) devem definir regras para implementar compatibilidade entre objetos e sua relação com o mecanismo de herança (discutido no item de POO)

Subprogramas

- Duas formas de abstração são possíveis em linguagens de programação: abstração de processo e abstração de dados.
- Abstração de processos: aparece na forma de subprogramas, que permitem:
  - o reuso de código
  - economia de tempo e de memória.
  - facilitar a leitura do programa (permite ver a estrutura lógica do programa, escondendo detalhes de codificação)

Características dos subprogramas estudados

- Cada subprograma tem um único ponto de entrada
- A unidade chamadora é suspensa durante a execução da unidade chamada – existe um único subprograma em execução a cada momento
- O controle sempre retorna a unidade chamadora quando acaba a execução da unidade chamada
- Outras formas de subprogramas:
  - corotinas
  - unidades concorrentes

- Métodos de POO: semelhantes aos subprogramas

- Principais diferenças:
  - forma de chamada
  - associação com classes e objetos

- Semelhanças:
  - Passagem de parâmetros
  - Variáveis locais