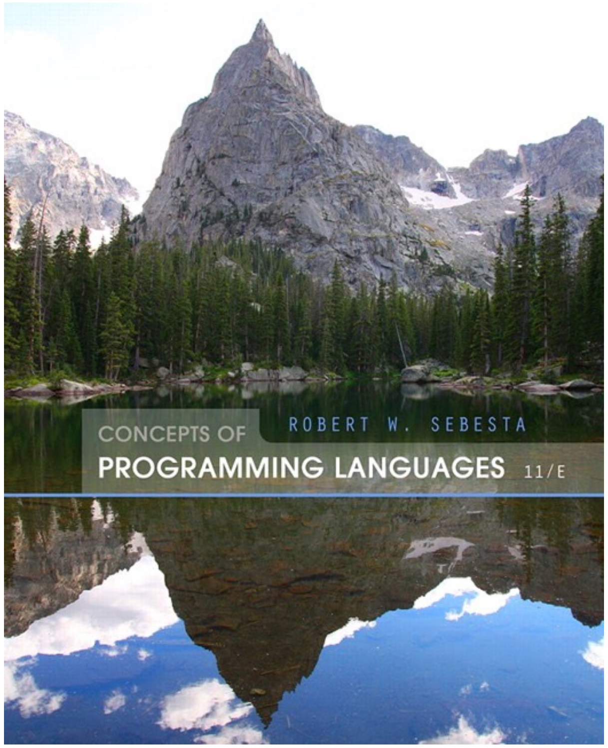
Chapter 6

Type Checking



ISBN 0-321-49362-1

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- Type checking is the activity of ensuring that the operands of an operator are of compatible types
- A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler- generated code, to a legal type
  - This automatic conversion is called a coercion.
- A type error is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is strongly typed if type errors are always detected
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing

Language examples:

- FORTRAN 95 is not
- C and C++ are not: parameter type checking can be avoided; unions are not type checked
- Ada, Java, C#: is, almost strongly typed (e.g., types can be explicitly cast which could result in error)
- ML is strongly typed
- Ruby, Python are strongly typed (determined at run time)

Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Name Type Equivalence

- Name type equivalence means the two variables have equivalent types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not equivalent with integer types
  - Formal parameters must be the same type as their corresponding actual parameters

Structure Type Equivalence

- Structure type equivalence means that two variables have equivalent types if their types have identical structures
- More flexible, but harder to implement

Type Equivalence (continued)

- Consider the problem of two structured types:
  - Are two record types equivalent if they are structurally the same but use different field names?
  - Are two array types equivalent if they are the same except that the subscripts are different?

    (e.g. [1..10] and [0..9])
  - Are two enumeration types equivalent if their components are spelled differently?
  - With structural type equivalence, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)