Type Checking

CIS*4650 (Winter 2020)

Symbol Table

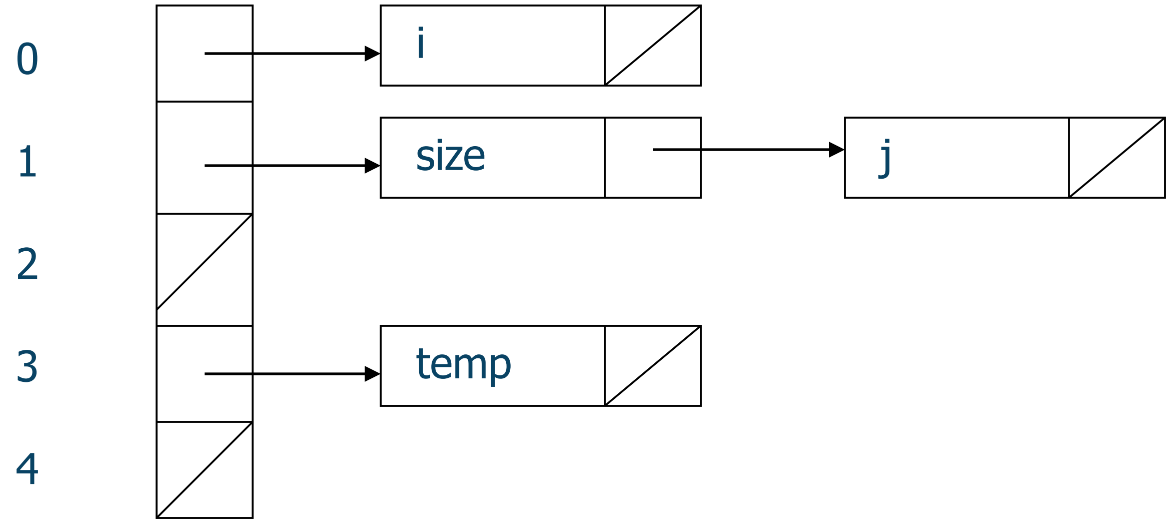
- Keep information associated with identifiers: constants, data types, variables, and functions

- Major operations:
  - Insert: store information from name declarations
  - Lookup: retrieve information associated with names
  - Delete: remove information from the view when the corresponding declaration is out of scope

- Usually implemented as hash tables. Why?

Hash Table

- Hash function: map a key to an address among the available buckets
  - Collisions: multiple keys mapped to the same address



Declarations

Constant: `const int SIZE = 199;`

Type:

```
struct Entry {
   char * name;
   int count;
   struct Entry * next;
};
typedef struct Entry * EntryPtr;
```

Variable:

```
int a, b[100];
struct Entry c;
EntryPtr d;
```

Function/procedure: see the example in slide 6 for an illustration

Name Bindings

- Constant declaration: associate values to names
- Type declaration: bind names to newly constructed types
- Variable declaration: bind names to scopes as well as data types
- Function declaration: bind names to modular language constructs

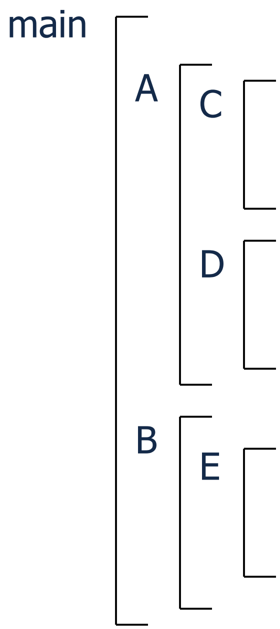
Scope Rules

- Declaration before use: facilitate one-pass processing
- Most closely nested rule for block structures: prefer the declaration in the most closely nested block to the reference.

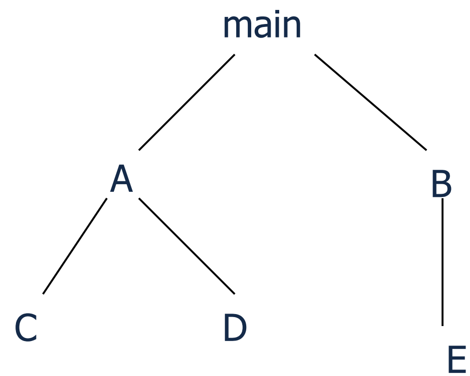
```
int i, j;
int f( int size ) {
    char i, temp;
    ...
    {  double j;
       ...i...j...
    }
    ...i...j...
    {  char * j;
       ...i...j...
    }
}
```


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

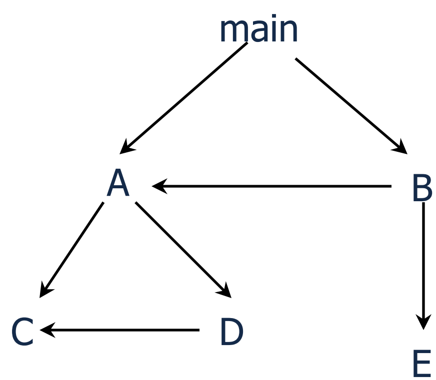
Block Structure in Pascal



Program Structure



Tree Structure

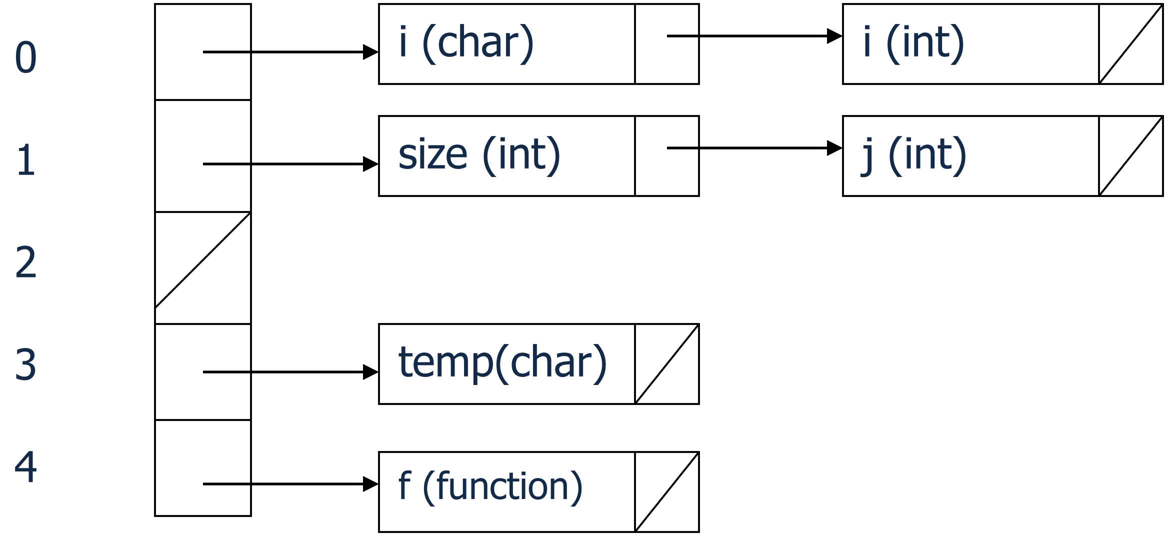


Call Graph

Implementing Nested Scopes

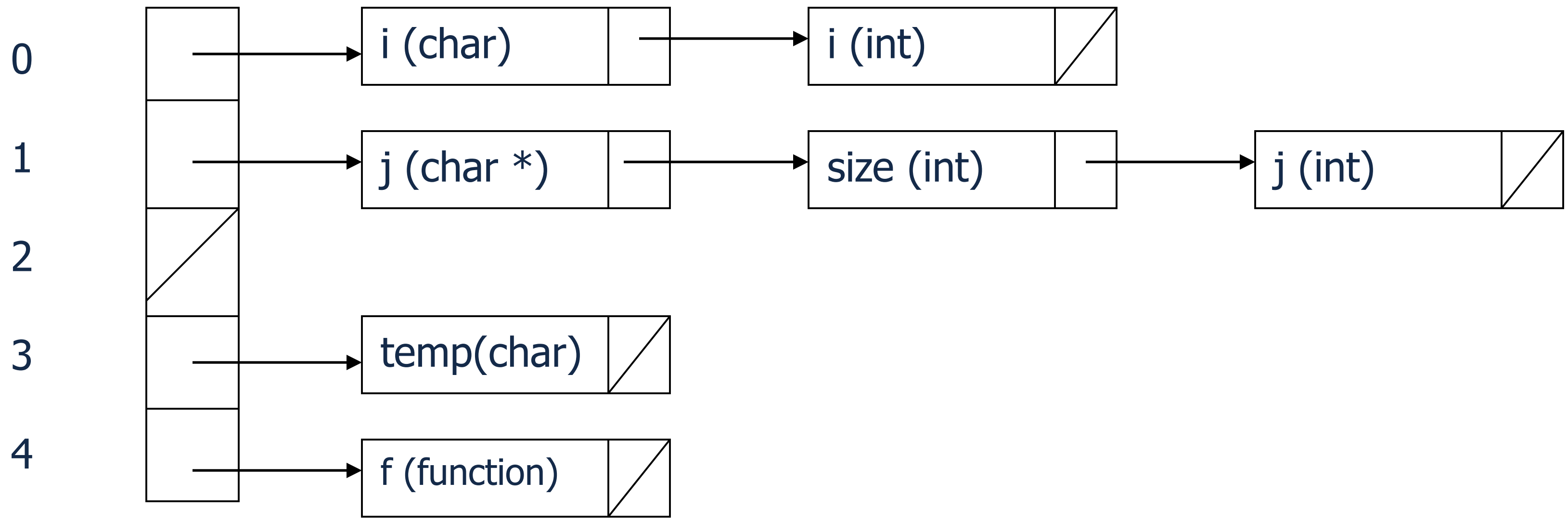
- Insert shouldn't overwrite existing names; just temporally hide irrelevant information

- After processing the declarations of function f



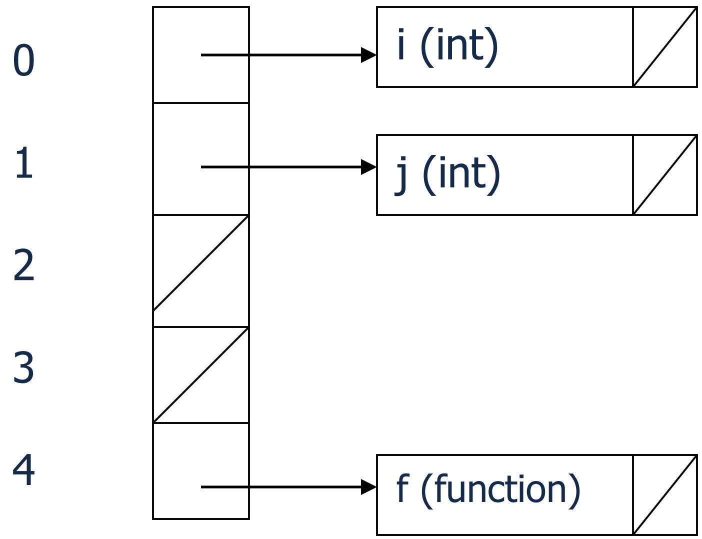
Implementing Nested Scopes

- After processing the declarations of the second nested block within the body of function f



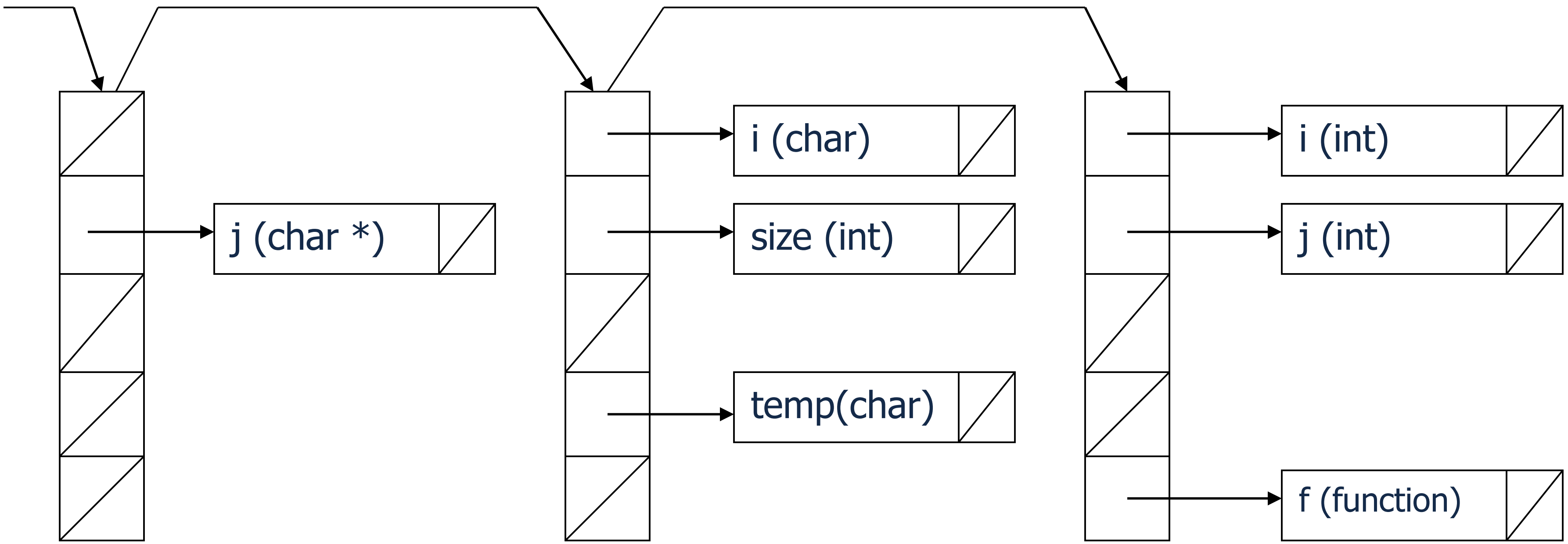
Implementing Nested Scopes

- After exiting the body of function f and deleting its declarations



Implementing Nested Scopes

- Use multiply linked hash tables so that deletion can be done easily



Displaying Symbol Tables

- Table content changes as we enter a scope, and as a result, it's better to show the nested scoping structures and display the content just before we leave each scope.

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

```
Entering the global scope:
    Entering the scope for function f:
        Entering a new block:
            j: double
        Leaving the block
        Entering a new block:
            j: char *
        Leaving the block
        size: int
        i: char
        temp: char
    Leaving the function scope
    i: int
    j: int
    f: (int) -> int
Leaving the global scope
```

Interacting Declarations

- The same name can't be re-declared in the same scope
  - Solution: perform a lookup before each insert

e.g.,
```
typedef int i;
int i;
```

- Indirect recursions: use function prototype

```
void f( void ) {
   ... g()...
}

void g( void ) {
   ... f() ...
}
```

⟹

```
void g( void );      /* function prototype */

void f( void ) {
   ... g()...
}

void g( void ) {
   ... f() ...
}
```

Type Checking

- Declarations provide the initial type definitions
- Type inference: compute and maintain type information
  - Given the data types of operands, determine the data types of expressions
- Type checking: use type information to ensure that all constructs are valid under the type rules
  - For example, boolean variables can't be added and integer variables can be or'ed

Type Checking

- Static type checking: performed at compilation time
  - Variables are declared
  - Rules for type compatibility

- Dynamic type checking: performed at execution time
  - Array: range is often known at run time
  - Reference

- All type checks can be done dynamically, but static checking is preferred where possible.

Type Expressions

<var-decls> -> <var-decls> ; <var-decl> | <var-decl>

<var-decl> -> id : <type-exp>

<type-exp> -> <simple-type> | <structured-type>

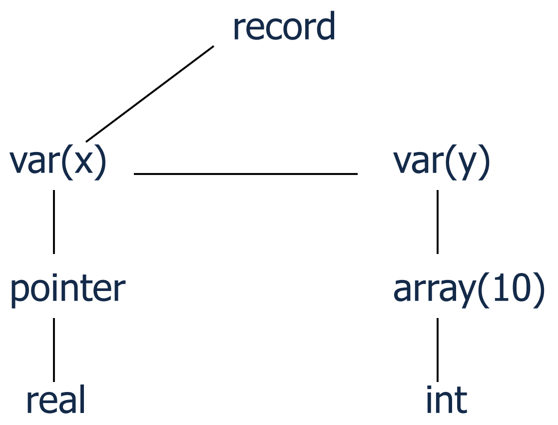
<simple-type> -> int | bool | real | char | void

<structured-type> -> array [num] of <type-exp> |
record <var-decls> end |
union <var-decls> end |
pointer to <type-exp> |
proc(<type-exps>) type-exp

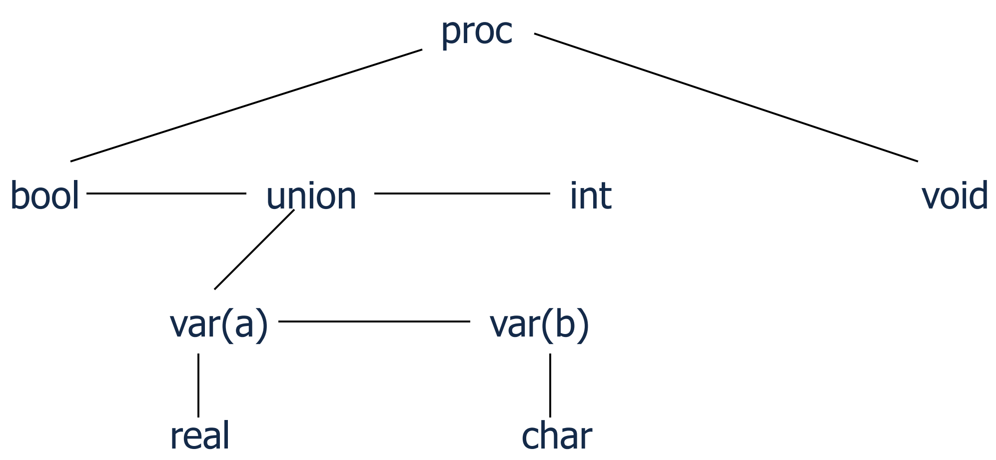
<type-exps> -> <type-exps> , <type-exp> | <type-exp>

Type as Syntax Tree

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



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



Type Equivalence

- Structural equivalence: iff two types have the same structure (syntax tree)

- Name equivalence: iff two types are the same simple type or are the same type name

```
e.g.,   t1 = int;              // t1 and t2 are not equivalent since they are
        t2 = int;              // different names
```

- Declaration equivalence: also called aliases
  - Implementation: introduce base type in addition to associated type

```
e.g.,   t1 = array [10] of int;      // t1 and t2 are not equivalent
        t2 = array [10] of int;      // t1 and t3 are equivalent
        t3 = t1                      // t2 and t3 are not equivalent
```

Type Declarations

```
<var-decls> -> <var-decls> ; <var-decl> | <var-decl>
<var-decl> -> id : <simple-type-exp>

<type-decls> -> <type-decls> ; <type-decl> | <type-decl>
<type-decl> -> id = <type-exp>

<type-exp> -> <simple-type-exp> | <structured-type>
<simple-type-exp> -> <simple-type> | id
<simple-type> -> int | bool | real | char | void
<structured-type> -> array [num] of <simple-type-exp> |
                     record <var-decls> end |
                     union <var-decls> end |
                     pointer to <simple-type-exp> |
                     proc( <type-exps> ) simple-type-exp

<type-exps> -> <type-exps> , <simple-type-exp> | <simple-type-exp>
```

Type Checking Example

<program> -> <var-decls> ; <stmts>

<var-decls> -> <var-decls> ; <var-decl> | <var-decl>

<var-decl> -> id : <type-exp>

<type-exp> -> int | bool | array [num] of <type-exp>

<stmts> -> <stmts> ; <stmt> | <stmt>

<stmt> -> if <exp> then <stmt> | id := <exp>

<exp> -> <exp> + <exp> | <exp> or <exp> | <exp> [<exp>] |
num | id | true | false

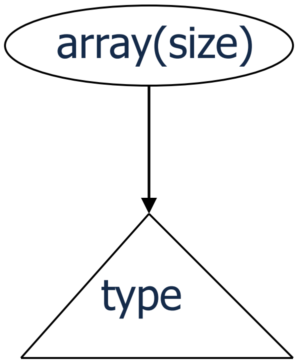
Type Checking Example

Grammar Rule	Semantic Rule
<var-decl> -> id : <type-exp>	insert(id.name, type-exp.type)
<type-exp> -> int	type-exp.type = integer
<type-exp> -> bool	type-exp.type = boolean
<type-exp$_1$> -> array [num] of <type-exp$_2$>	type-exp$_1$.type = makeTypeNode(array, num.size, type-exp$_2$.type)
<stmt$_1$> -> if <exp> then <stmt$_2$>	**if not** typeEqual(exp.type, boolean) **then** type-error(stmt$_1$)
<stmt> -> id := <exp>	**if not** (typeEqual(lookup(id.name), exp-type) **then** type-error(stmt)
<exp$_1$> -> <exp$_2$> + <exp$_3$>	**if not** (typeEqual(exp$_2$.type, integer) **and** typeEqual(exp$_3$.type, integer)) **then** type-error(exp$_1$) **else** exp$_1$.type = integer
<exp$_1$> -> <exp$_2$> **or** <exp$_3$>	**if not** (typeEqual(exp$_2$.type, boolean) **and** typeEqual(exp$_3$.type, boolean)) **then** type-error(exp$_1$) **else** exp$_1$.type = boolean

Type Checking Example

Grammar Rule	Semantic Rule
<exp$_1$> -> <exp$_2$> [<exp$_3$>]	**if** isArrayType(exp$_2$.type) **and** typeEqual(exp$_3$.type, integer) **then** exp$_1$.type = exp$_2$.type.child1 **else** type-error(exp$_1$)
<exp> -> num	exp.type = integer
<exp> -> id	exp.type = lookup(id.name)
<exp> -> true	exp.type = boolean
<exp> -> false	exp.type = boolean

makeTypeNode(array, size, type)



Type Checking for Function Calls/Returns

- For a function definition, the return expression has to match the return type:

```
void foo(void) {
    int x;
    return x;
}
```

- For a function call, the number and types of its arguments have to match those of a function definition:

```
void foo(void) {
}
```

```
void main(void) {
    int x;
    if (x) output(foo())
    else output(x, foo());
}
```

Special Cases for Checkpoint Two

- For a function, variables declared in the parameter list and immediately inside the body should belong to the same scope
  - Suggestion: use (level + 1) for “size” and after entering the block, increment ”level” and then use “level” for “i” and “temp”

```
/* size, i, and temp at the same level */
int f(int size) {
   char i, temp;
   …
}
```

- For the two predefined functions “int input(void)” and “void output(int)”, you may report “undefined” errors when they are used in a program
  - Suggestion: add their definitions to the symbol table at the very start

Special Cases for Checkpoint Two

- Since C- doesn't allow type definitions, we may have to support structural equivalence for arrays. To simplify the task, we only allow an array variable to match a parameter in a function call such as "sort(x, 0, 9)" in the program:

```
int x[10];

void sort(int a[], int low, int high) {
    …
}

void main(void) {
   …
   sort(x, 0, 9);
   …
}
```

Type Conversion and Coercion

- Allow arithmetic expressions of mixed data types
  - e.g., 2.1 + 3

- Compatibility: integer $\subseteq$ float $\subseteq$ double

- Conversion: converting a data type explicitly
  - e.g., 2.1 + float(3)

- Coercion: converting a data type implicitly
  - Similar solution applies to subclasses in object-oriented languages (so-called subtype principle)