Overloaded Operators

- The multiple use of an operator is called operator overloading
  - E.g., “+” is used to specify integer addition, floating-point addition, and string catenation
- Do not use the same symbol for two completely unrelated operations, because that can decrease readability
  - In C, “&” can represent a bitwise AND operator, and an address-of operator

Type Conversion

- Narrowing conversion
  - To convert a value to a type that cannot store all values of the original type
  - E.g. (Java), double->float, float->int
- Widening conversion
  - To convert a value to a type that can include all values belong to the original type
  - E.g., int->float, float->double

Narrowing Conversion vs. Widening Conversion

- Narrowing conversion are not always safe
  - The magnitude of the converted value can be changed
  - E.g., float->int with 1.3E25, the converted value is distantly related to the original one

- Widening conversion is always safe
  - However, some precision may be lost
  - E.g., int->float, integers have at least 9 decimal digits of precision, while floats have 7 decimal digits of precision (reduced accuracy)

Implicit Type Conversion

- One of the design decisions concerning arithmetic expressions is whether an operator can have operands of different types.

- Languages that allow such expressions, which are called **mixed-mode expressions**, must define conventions for implicit operand type conversions because computers do not have binary operations that take operands of different types.

- A **coercion** is an implicit type conversion that is initiated by the compiler

Implicit Type Conversion

```
var x, y: integer;
      z: real;
      ...
y := x * z;    /* x is automatically converted to "real"  */
```

- Implicit type conversion can be achieved by narrowing or widening one or more operators
- It is better to widen when possible
  - E.g., x = 3, z = 5.9, what is y's value if x is widened? How about z narrowed?

Key Points of Implicit Coercions

- They decrease the type error detection ability of compilers
  - Did you really mean to use “mixed-mode expressions” ?
- In most languages, all numeric types are coerced in expressions, using widening conversions

Explicit Type Conversion

- Also called “casts”
- Ada example

```
FLOAT(INDEX)-- INDEX is an INTEGER
```

- C example:

```
(int) speed /* speed is a float */
```

Short-Circuit Evaluation

- A **short-circuit evaluation** of an expression is one in which the result is determined without evaluating all of the operands and/or operators
  - Consider `(a < b) && (b < c)`:
    - If `a >= b`, there is no point evaluating `b < c` because `(a < b) && (b < c)` is automatically false
- (x && y) ≡ if x then y else false
- (x || y) ≡ if x then true else y

Short-Circuit Evaluation

- Short-circuit evaluation may lead to unexpected side effects and cause error
  - E.g., (a > b) || ((b++) / 3)
- C, C++, and Java:
  - Use short-circuit evaluation for Boolean operations (&& and ||)
  - Also provide bitwise operators that are **not short circuit** (& and |)

Short-Circuit Evaluation

- Ada: programmers can specify either

Non-SC eval	SC eval
(x or y)	(x or else y)
(x and y)	(x and then y)

Control Structures

- Selection
- Iteration
  - Iterators
- Recursion
- Concurrency & non-determinism
  - Guarded commands

Structured and Unstructured Flow

- Assembly language: conditional and unconditional branches.
- Early Fortran: relied heavily on `goto` statements (and labels):

```
      IF (A .LT. B) GOTO 10        ! ".LT." means "<"
      …
10
```

- Late 1960s: Abandoning of `GOTO` statements started.
- Move to structured programming in 1970s:
  - Top-down design (progressive refinement).
  - Modularization of code.
  - Descriptive variable.
- Within a subroutine, a well-designed imperative algorithm can be expressed with only sequencing, selection, and iteration.
- Most of the structured control-flow constructs were introduced by Algol 60.

Structured Alternatives to `goto`

- With the structured constructs available, there was a small number of special cases where `goto` was replaced by special constructs: `return`, `break`, `continue`.
- Multilevel returns: branching outside the current subroutine.
  - Unwinding: the repair operation that restores the run-time stack of subroutine information, including the restoration of register contents.
- Errors and other exceptions within nested subroutines:
  - Auxiliary Boolean variable.
  - Nonlocal `GOTO`s.
  - Multilevel returns.
  - Exception handling.

Sequencing

- The principal means of controlling the order in which side effects occur.
- Compound statement: a delimited list of statements.
- Block: a compound statement optionally preceded by a set of declarations.
- The value of a list of statements:
  - The value of its final element (Algol 68).
  - Programmers choice (Common Lisp – not purely functional).
- Can have side effects; very imperative, von Neumann.
- There are situations where side effects in functions are desirable: random number generators.

Selection

- Selection statement: mostly some variant of `if…then…else`.
- Languages differ in the details of the syntax.
- Short-circuited conditions:
  - The Boolean expression is not used to compute a value but to cause control to branch to various locations.
  - Provides a way to generate efficient (jump) code.
  - Parse tree: inherited attributes of the root inform it of the address to which control should branch:

```
if ((A > B) and (C > D)) or (E ≠ F) then
  then_clause
else
  else_clause
```

```
    r1 := A  r2 := B
    if r1 <= r2 goto L4
    r1 := C  r2 := D
    if r1 > r2 goto L1
L4: r1 := E  r2 := F
    if r1 = r2 goto L2
L1: then_clause
    goto L3
L2: else_clause
L3:
```

Case/Switch Statements

- Alternative syntax for a special case of nested `if..then..else`.

```
CASE … (* expression *)
    1:      clause_A
|   2, 7:   clause_B
|   3..5:   clause_C
|   10:     clause_D
    ELSE    clause_E
END
```

- Multiple selectors
- Code fragments (clauses): the arms of the CASE statement.
- The list of constants are CASE statement labels:
  - The constants must be disjoint.
  - The constants must of a type compatible with the tested expression.
- The principal motivation is to *facilitate the generation of efficient target code*: meant to compute the address in which to jump in a single instruction.
  - A jump table: a table of addresses.

Case/Switch Statements

```
switch (index) {
case 1:
case 3: odd += 1;
        sumodd += index;
        break;
case 2:
case 4: even += 1;
        sumeven += index;
        break;
default: printf("Error in switch,
index = %d\n", index);
}
```

Iteration

- Iteration: a mechanism that allows a computer to perform similar operations repeatedly.
- Favored in imperative languages.
- Mostly some form of loops executed for their side effects:
  - Enumeration-controlled loops: executed once of every value in a given finite set.
  - Logically controlled loops: executed until some Boolean condition changes value.
  - Combination loops: combines the properties of enumeration-controlled and logically controlled loops (Algol 60).
  - Iterators: executed over the elements of a well-defined set (often called containers or collections in object-oriented code).

Design Issues

- What are the type and scope of the loop variable?
- Should it be legal for the loop variable or loop parameters to be changed in the loop, and if so, does the change affect loop control?
- Should the loop parameters be evaluated only once, or once for every iteration?

Enumeration-Controlled Loops

- Originated with the `DO` loop in Fortran I.
- Adopted in almost every language but with varying syntax and semantics.
- Many modern languages allow iteration over much more general finite sets.
- Semantic complications:
  1. Can control enter or leave the loop in any way other than through the enumeration mechanism?
  2. What happens if the loop body modifies variables that were used to compute the end-of-loop bound?
  3. What happens if the loop body modifies the index variable itself?
  4. Can the program read the index variable after the loop has completed, and if so, what will its value be?
- Solution: the loop header contains a declaration of the index.

Combination Loops

- Algol 60: can specify an arbitrary number of “enumerators” – a single value, a range of values, or an expression.
- Common Lisp: four separate sets of clauses – initialize index variables, test for loop termination, evaluate body expressions, and cleanup at loop termination.
- C: semantically, `for` loop is logically controlled but makes enumeration easy - it is the programmer’s responsibility to test the terminating condition.
  - The index and any variables in the terminating condition can be modified within the loop.
  - All the code affecting the flow of control is localized within the header.
  - The index can be made local by declaring it within the loop thus it is not visible outside the loop.

Iteration Based on Data Structures

- A data-based iteration statement uses a user-defined data structure and a user-defined function to go through the structure's elements
  - The function is called an **iterator**
  - The iterator is invoked at the beginning of each iteration
  - Each time it is invoked, an element from the data structure is returned
  - Elements are returned in a particular order

Iterators

- True iterators: a container abstraction provides an iterator that enumerates its items (Clu, Python, Ruby, C#).
  - An iterator is a separate thread of control, with its own program counter, whose execution is interleaved with that of the loop.

    ```
    for i in range(first, last, step):
    ```
- Iterator objects: iteration involves both a special from of a for loop and a mechanisms to enumerate the values for the loop:
  - Java: an object that supports `Iterable` interface – includes an `iterator()` method that returns an `Iterator` object.

    ```
    for (iterator<Integer> it = myTree.iterator(); it.hasNext();) {
      Integer i = it.next();
      System.out.println(i);
    }
    ```
  - C++: overloading operators so that iterating over the elements is like using pointer arithmetic.

Logically Controlled Loops

- The only issue: where within the body of the loop the termination condition is tested.

- *Before each iteration*: the familiar `while` loop syntax – using an explicit concluding keyword or bracket the body with delimiters.
- *Post-test loops*: test the terminating condition at the bottom of a loop – the body is always executed at least once. (`do while`)
- *Midtest loops*: often accomplished with a special statement nested inside a conditional – `break` (C), `exit` (Ada), or `last` (Perl).

Recursion

- Recursion requires no special syntax: why?
- Recursion and iteration are equally powerful.
- Most languages provide both iteration (more "imperative") and recursion (more "functional").
- Tail-recursive function: additional computation never follows a recursive call. The compiler can reuse the space, i.e., no need for dynamic allocation of stack space.

```
int gcd(int a, int b) {
  if (a == b) return a;
  else if (a > b) return gcd(a - b,b);
  else return gcd(a, b – a);
}
```

Guarded Commands

- New and quite different forms of selection and loop structures were suggested by Dijkstra (1975)
- We cover guarded commands because they are the basis for two linguistic mechanisms developed later for concurrent programming in two languages: CSP and Ada

Motivations of Guarded Commands

- To support a program design methodology that ensures correctness during development rather than relying on verification or testing of completed programs afterwards
- Also useful for concurrency
- Increased clarity in reasoning

Guarded Commands

- Two guarded forms
  - Selection (guarded if)
  - Iteration (guarded do)

Guarded Selection

```
if <boolean> -> <statement>
[] <boolean> -> <statement>
      ...
[] <boolean> -> <statement>
fi
```

- Sementics
  - When this construct is reached
    - Evaluate all boolean expressions
    - If more than one is true, choose one nondeterministically
    - If none is true, it is a runtime error
- Idea: **Forces** one to consider **all possibilities**

An Example

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + i
fi
```

- If i = 0 and j > i, the construct chooses nondeterministically between the first and the third assignment statements
- If i == j and i ≠ 0, none of the conditions is true and a runtime error occurs

Guarded Selection

- The construction can be an elegant way to state that the order of execution, in some cases, is irrelevant

```
if x >= y -> max := x
[] y >= x -> max := y
fi
```

  - E.g., if x == y, it does not matter which we assign to max
  - This is a form of abstraction provided by the nondeterministic semantics

Guarded Selection

Now, consider this same process coded in a traditional programming language
selector:

```
if (x >= y)
max = x;
else
max = y;
```

This could also be coded as follows:

```
if (x > y)
max = x;
else
max = y;
```

Guarded Iteration

```
do <boolean> -> <statement>
[]  <boolean> -> <statement>
      ...
[]  <boolean> -> <statement>
od
```

- Semantics:
  - For each iteration
    - Evaluate all boolean expressions
    - If more than one is true, choose one nondeterministically, and then start loop again
    - If none is true, exit the loop
- Idea: if the order of evaluation is not important, the program should not specify one

An Example

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

- Given four integer variables: q1, q2, q3, and q4, rearrange the values so that $q1 \le q2 \le q3 \le q4$
- Without guarded iteration, one solution is to put the values into an array, sort the array, and then assigns the value back to the four variables

An Example

- While the solution with guarded iteration is not difficult, it requires a good deal of code
- There is considerably increased complexity in the implementation of the guarded commands over their conventional deterministic counterparts