Overloading

An operator or function is overloaded if its meaning depends on the number or types of its arguments

- `+` (real, integer)
- `-` (unary, binary)

Also called ad-hoc polymorphism

Some languages permit overloading of user-defined functions.

Good idea??

Coercion

- **Implicit, or automatic, type conversion.**

```
var x:integer;
    y,z:real;
    ...
y := x + z;
```

→ **"mixed-mode" expression**

- **Coercion is different from explicit type conversior**

```
int x,y
...
(double) x / (double) y
```

Coercion Extremes (Algol 68)

```
int    i;
real   r;
[1:1] int rowi;
ref int refi;
union (int,real)  ir;
proc int p;
   ...
r := i/r;          -- widening (of i)
ir := i;           -- uniting
i := refi;         -- dereferencing
i := p;            -- deproceduring
rowi := 5          -- rowing
```

Relational and Boolean Expressions

- **Relational operators compare two operands and return a boolean**

  = < > <= >= <>

- **Lower precedence than arithmetic operators**

  a + b < c + d ≡

  (a + b) < (c + d)

- **Boolean values: true, false**

- **Boolean operators: and, or, xor, not, =**

- **True boolean values are helpful.**

  → **In C, use integers:**

  0 = false

  other = true

  → **a > b > c is legal!**

Short-Circuit Evaluation

- Get a result without evaluating entire expression.

  x and y ≡ if x then y else false

  x or y ≡ if x then true else y

  (x and y are arbitrary boolean expressions)

- Same as regular evaluation in the absence of side-effects.

- May have a choice, e.g. Ada:

  x or else y

  x and then y

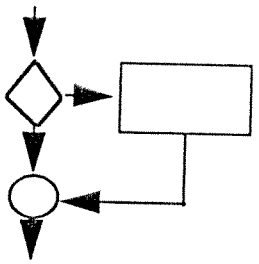
  Otherwise, need to know if it's used.

Statement-Level Control

- **Three types:**
  - → **sequencing (not much to be said about this)**
  - → **selection**
  - → **repetition**

Selection

- **Issues:**
  - → How is selection controlled?
  - → How many choices? Can none of the choices be selected
- **Single-way selectors**

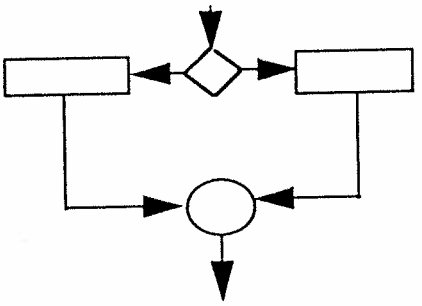


if <bool> then <stmt>

- → Without compound statements, need GOTO

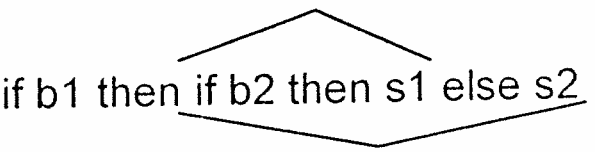
Two-Way Selection

- **Two-way selectors**



if <bool> then <stmt> else <stmt>

- **Recall potential ambiguity with nested 2-way selectors:**



→ **Several possible remedies. . .**

Avoiding Ambiguity

- **Disallow nested conditionals**

<stmt> --> <a_stmt> | <if_stmt> | <c_stmt>

<c_stmt> --> begin <stmt_list> end

<if_stmt> --> if <bool> then < s_stmt>

| if <bool> then <s_stmt> else <stmt>

<s_stmt> --> <a_stmt> | <c_stmt>

```
if b1 then
    begin if b2 then s1 else s2 end

if b1 then
    begin if b2 then s1 end
        else s2
```

Multiple-Way Selection

- **Multiple-way Selection**
  - → **New issues:**

    How to handle unrepresented selector values?

    Should selectable segments be followed by implicit branches of construct?

  - → **Lots of forms of multiple selectors. . .**

Multiple-Way Selection -- Early Approaches

- **Three-way selectors (Fortran)**

  IF (<arith_exp>) L1, L2, L3

  → **Must jump out of segment:**

```
      IF (...) 10, 20, 30
10    ...
      GO TO 40
20    ...
      GO TO 40
30    ...
40    ...
```

- **Computed GO TO (Fortran)**

  GO TO (L1, L2, ..., Ln), exp

  → goes to first label if exp=1, second if exp=2, etc.

  → if exp < 1 or exp > n, no effect

  → must still jump out of segment

Multiple-Way Selection -- Modern Approache

- **Pascal example**

```
case <exp> of
    <const_list> : <stmt>
    ...
    <const_list> : <stmt>
end
```

→ **<exp> of ordinal type**

→ **implicit branch to end after each segment**

→ **"otherwise" option**

→ **selector value unrepresented => error**

More Modern Approaches

- **C example**

```
switch (<exp>)
{
case <const_exp> : <stmt>
...
case <const_exp> : <stmt>
[default : <stmt>]
}
```

  - → **<exp> and <const_exp> yield integers**
  - → **No implicit branches -- use "break" to leave switch**

    Is this a good idea?

- **Elsif (Ada)**
  - → **Like LISP cond**
  - → **Cuts down on indenting**
  - → **More flexible than case statement**

Repetition

- **Iteration (statement-level)**
- **Recursion (unit-level) -- later**
- **Issues:**
  - → **How is iteration controlled?**
  - → **Where in loop is control mechanism?**
    - top
    - bottom
    - anywhere

Counter-Controlled Loops

- Issues:
  - → Type and scope of loop variable
  - → Value of loop variable at loop term
  - → Change loop parameterss in body? Effect?
  - → Branch into loop?
  - → Test at top or bottom?
  - → Loop parameters evaluated when?

FORTRAN loops

- DO <label> <var> = <init>,<term>
  - → Can't change loop variable/parameters inside loop
  - → Loop variable undefined after normal termination; last value if jump out
  - → Can jump out of, back into loop

```
DO 100 I = 1,10
...
100  CONTINUE
```

  - → Note!

```
DO 100 I = 1.10
```

More Loop Examples

- **ALGOL 60**

  → **Very complicated! see book.**

- **ALGOL 68**

```
for i from j by k to m
while b do . . . od
```

- **C**

```
for (<exp>;<exp>;<exp>) <stmt>
```


```
for (i = 0; i <= 10; i++)
    sum = sum + a[i];
```

→ **No special loop variables.**

→ **Can have multiple statements for each <exp>**

Logically Controlled Loops

- **Issues:**
  - → **Pretest or posttest?**
  - → **Jump into loop?**
- **Pascal constructs**

  while <exp> do <stmt>

  repeat <stmt> until <exp>
- **Other exits, e.g. Modula 2, Ada:**

  loop

  ...

  if <exp> then EXIT

  ...

  end
- **CONTINUE, BREAK**

More Statement-Level Control Constructs

- **Explicit loop exits**
  - → **Basic model: (Modula)**

```
loop
. . .
if . . . then EXIT
. . .
end
```

- **Multi-level exits (Ada)**
  - → **named loops**
  - → **conditional EXIT**

exit [<loop name>] [when <condition>]

Ada Example

```
Outer_loop:
        for i in 1. .10 loop
        Inner_loop:
            for j in 1. .20 loop;
                ...
                    EXIT Outer_loop when <cond>;
                ...
        end loop Inner_loop;
            ...
end loop Outer_loop;
```

Iterators (Clu)

- **Lets user specify range of values over which a loop iterates.**

```
for atom:node in list(x) do
    ...

list = iter(z: linked_list) yields (node)
    ...
    yield (n)
    ...
end list
```

→ **list produces elements of type node one at a time**

Guarded Commands

```
if  <bool> -> <stmt>
W  <bool> -> <stmt>
    ...
W  <bool> -> <stmt>
fi
```

- **Semantics**
  - → Evaluate all <bool>s
  - → If ≥ 1 is true, choose one nondeterministically and execute its <stmt>
  - → If none true, error

Guarded Commands (continued)

```
do   <bool> -> <stmt>
W  <bool> -> <stmt>
   ...
W  <bool> -> <stmt>
od
```

- **Semantics**
  - → Evaluate all <bool>s
  - → If ≥ 1 is true, choose one nondeterministically and execute its <stmt>
  - → Repeat until none true, then normal termination