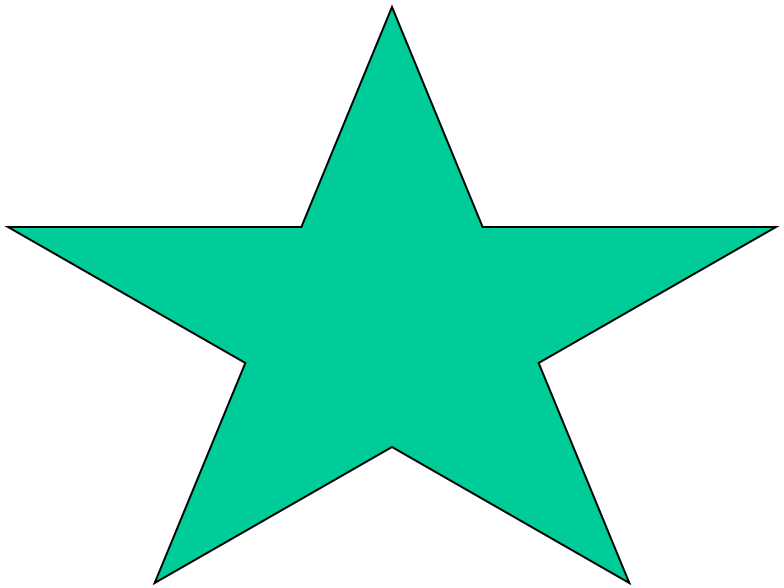
Control Flow

February 9, Lecture 7

Copyright © 2009 Elsevier



Expressions

- A **simple object**
  - Literal constant
  - Named variable
  - Constant

- Or a **function** applied to arguments
  - For built-in functions we use the term **operator** (like +,*)

ALGOL family

```
my_func(A, B, C)
```

```
a + b
- c
```

Copyright © 2009 Elsevier

Expressions

- a+b is syntactic sugar for **actual** internal functions::
  - Ada: "+"(a,b)
  - C++: a.operator+(b)

- Where does the name of the function appear?
  - Prefix
  - Infix
  - Postfix

```
(* (+ 1 3) 2)                ; that would be (1 + 3) * 2 in infix
(append a b c my_list)
```

For example in LISP we have prefix, with function name inside parentheses

Copyright © 2009 Elsevier

Expressions

- Where does the name of the function appear?
  - Prefix
  - Infix
  - Postfix

For example in LISP we have prefix, with function name inside parentheses

```
(* (+ 1 3) 2)              ; that would be (1 + 3) * 2 in infix
(append a b c my_list)
```

Copyright © 2009 Elsevier

Expressions

- Where does the name of the function appear?
  - Prefix
  - Infix
  - Postfix

Some languages (like R) allow the programmer to define infix functions

Some languages use **only** infix notation

Smalltalk

```
myBox displayOn: myScreen at: 100@50
```

Copyright © 2009 Elsevier

Expressions

- Conditional expressions (mixfix)

```
a := if b <> 0 then a/b else 0;
```

Algol

```
a = b != 0 ? a/b : 0;
```

C

Copyright © 2009 Elsevier

Expressions

- Precedence and associativity (only for infix notation)

```
a + b * c**d**e/f
```

```
((((a + b) * c)**d)**e)/f
```

or

```
a + (((b * c)**d)**(e/f))
```

or

```
a + ((b * (c**(d**e)))/f)
```

C is very good and intuitive with precedence

Pascal not

```
if A < B and C < D then (* ouch *)
```

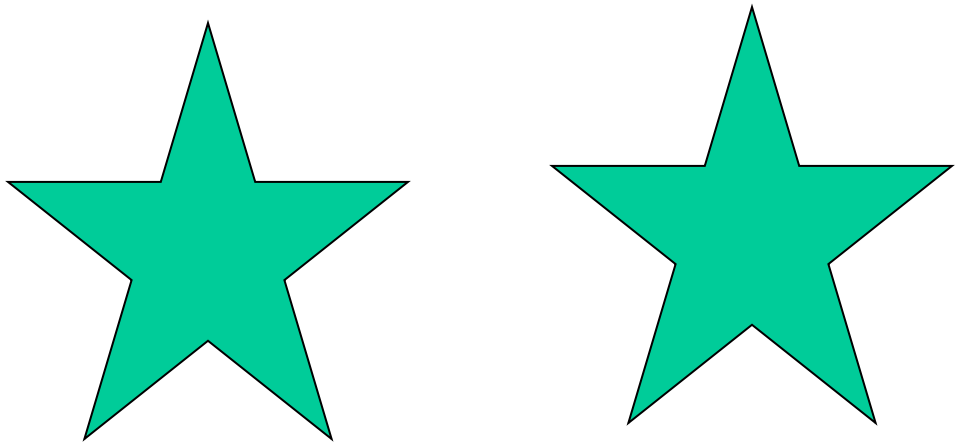
Copyright © 2009 Elsevier

Expressions

- Precedence and associativity (only for infix notation)

- Associativity is more standard
  - Exceptions exist, for example 2**3**4 is not allowed in Fortran

Copyright © 2009 Elsevier



Expressions

- **Expressions** provide values to the surrounding context

```
(defun gcd2 (a b)
 (if (zerop b) a
 (gcd2 b (mod a b))))
```

LISP code

- **Functional** languages contain only expressions
- Complex computations are done via recursion

Copyright © 2009 Elsevier

Statements

- **Imperative** languages compute by means of **side-effects**
  - A future computation is influenced in a way other than returning a value to the surrounding context
  - For example changing the value of a variable affects all future parts of the program where the variable is used.

- In functional languages there are **no** side-effects
  - The value of an expression is independent of the time it is evaluated
  - A future evaluation will be the same, because there are no side-effects

- Some imperative languages distinguish between expressions and statements. Statements **exist only** for their side-effects.

Copyright © 2009 Elsevier

References and values

- Assignments appear to be simple.
- However there are differences in their semantics
  - What they "mean" in each language
- This has an impact on more complex programs with pointers

Reference value

```
d = a;
a = b + c;
```

Location value

Copyright © 2009 Elsevier

References and values

- Not all expressions can be l-values
  - 2+3 = a
  - a = 2+3 (when a is a constant)

- L-values can be very complicated expressions

```
(f(a)+3)->b[c] = 2;
```

What does is do?

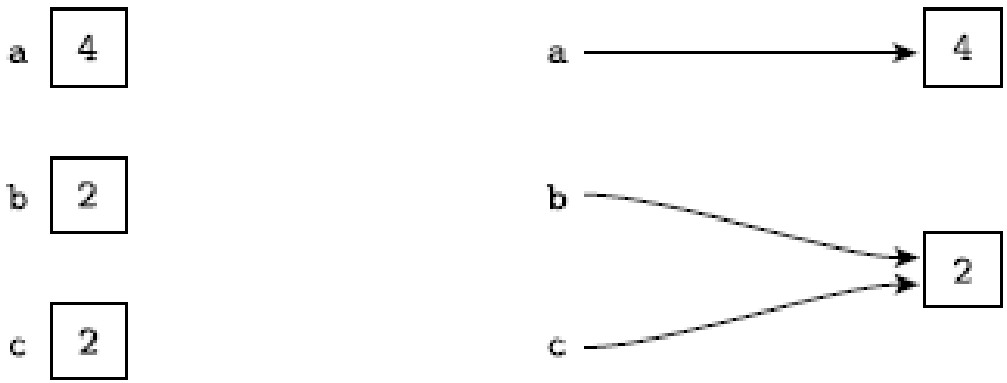
```
g(a).b[c] = 2;
```

Possible in C++ only

Copyright © 2009 Elsevier

Reference vs value model

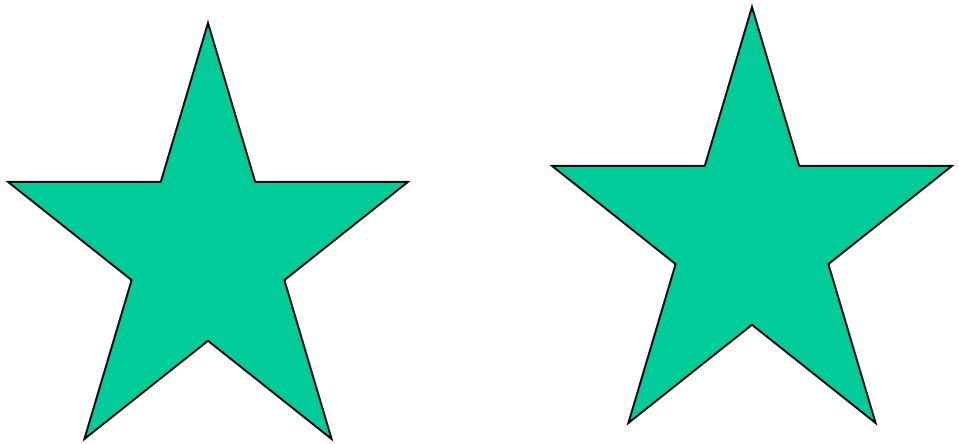
- Two semantics



Difference in thinking

```
b := 2;
c := b;
a := b + c;
```

Copyright © 2009 Elsevier



Boxing

- The need for boxing, because of the value model

Older Java

```
import java.util.Hashtable;
...
Hashtable ht = new Hashtable();
...
Integer N = new Integer(13);        // Integer is a "wrapper" class
ht.put(N, new Integer(31));
Integer M = (Integer) ht.get(N);
int m = M.intValue();
```

Newer Java

```
ht.put(13, 31);
int m = (Integer) ht.get(13);
```

Copyright © 2009 Elsevier

Orthogonality

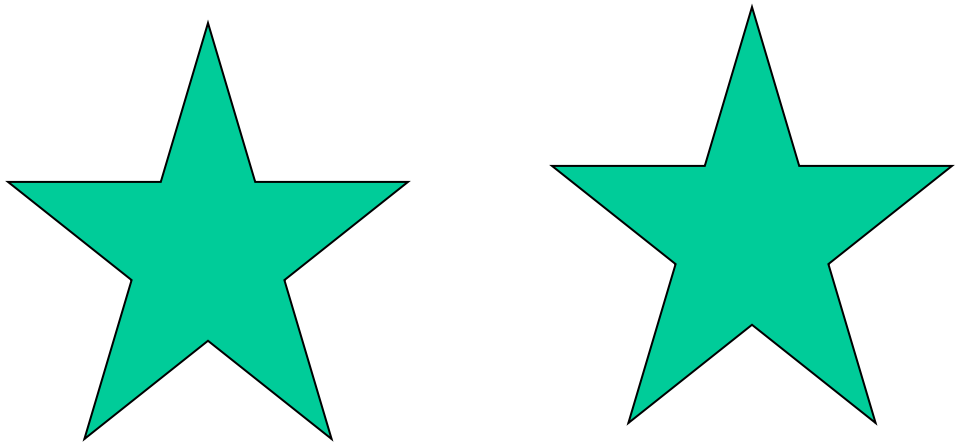
- Features are consistent and can be used in any combination
- Expression-oriented languages
  - Things that look like statements work like expressions

```
begin
    a := if b < c then d else e;
    a := begin f(b); g(c) end;
    g(d);
    2 + 3
end
```

Algol 68

In contrast Pascal makes clear distinction

Copyright © 2009 Elsevier



Orthogonality

- In C the distinction is made
- But it allows expressions to appear in "expression statements"
  - Effectively allowing expressions to appear where statements are expected

```
if (a == b) {
    /* do the following if a equals b */

if (a = b) {
    /* assign b into a and then do
       the following if the result is nonzero */
```

Source of bugs in C

This will generate error in other languages

Copyright © 2009 Elsevier

Combination Assignment Operators

- Common things:
  - a = a+1
  - b.c[3].d = b.c[3].d*e;

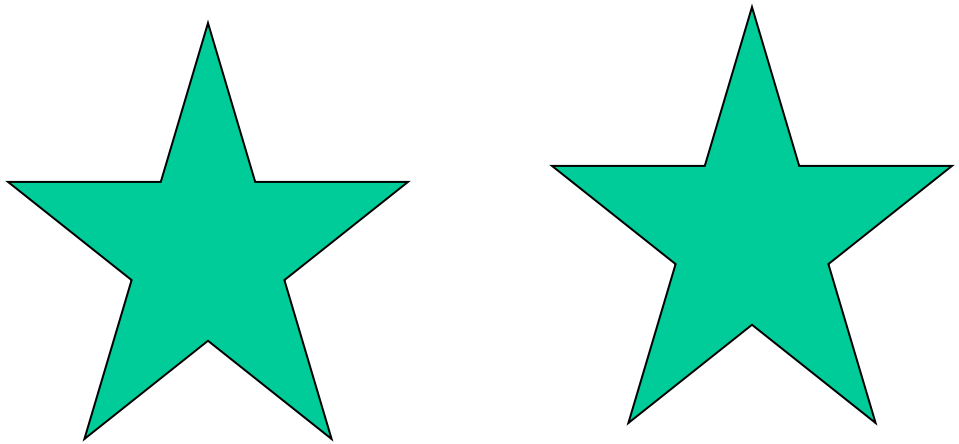
- Cumbersome and requiring extra work (or **compiler** work)

```
void update(int A[], int index_fn(int n)) {
    int i, j;
    /* calculate i */
    ...
    j = index_fn(i);
    A[j] = A[j] + 1;
}
```

```
A[index_fn(i)] = A[index_fn(i)] + 1;
```

Why is this dangerous?

Copyright © 2009 Elsevier



Combination Assignment Operators

- Solution is **assignment operators**

```
A[index_fn(i)]++;
```

or

```
++A[index_fn(i)];
```

```
A[--i] = b;
*p++ = *q++;
```

Traversing arrays

How does this work?

Copyright © 2009 Elsevier

Multiway assignment

- In several languages like Clu,ML,Perl, Python,Ruby

```
a, b := c, d;
```

- It's useful in the following context

```
a, b := b, a;
```

Copyright © 2009 Elsevier

Initialization

- Static variables can be initialized by the compiler
- Heap stored variables must be done at run-time
- Default may be 0 to be consistent with the operating system
- Initialize every variable when declared for **safety**
  - This doesn't help with bugs
  - It rather makes them repeatable

- Some languages allow types that have **constructors**
  - So when a variable of that type is defined it is essentially a call to the constructor

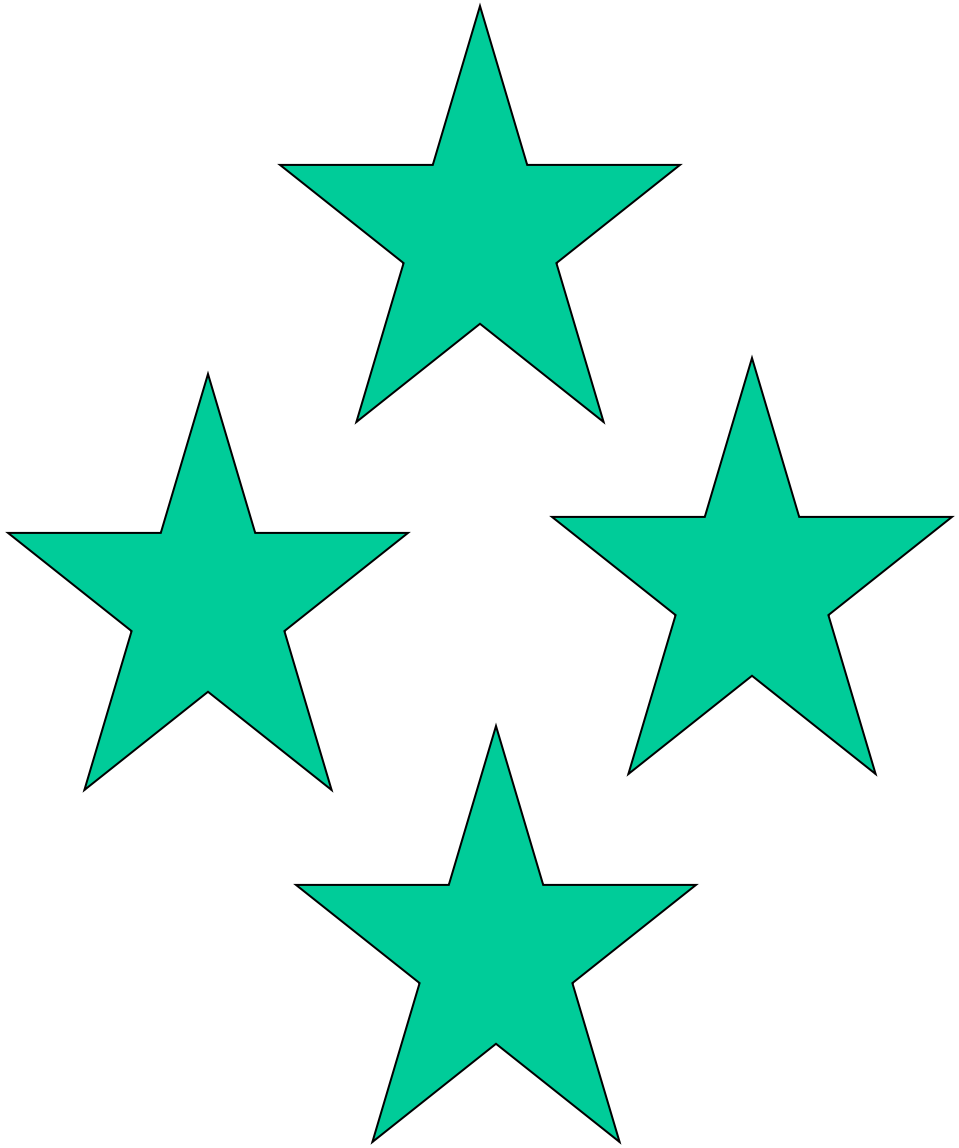
Copyright © 2009 Elsevier

Initialization

- Some languages (Java, C#) require **definite assignment**
- They complain if they see no initialization at compile
- Others do dynamic checks (NaN value), mostly at higher cost
- Focus on safety rather than speed

```
int i;
final static int j = 3;
...
if (j > 0) {
    i = 2;
}
...
if (j > 0) {
    System.out.println(i);
    // error: "i might not have been initialized"
}
```

Copyright © 2009 Elsevier



Ordering

- Associativity doesn't always help predict the value
- This is because of **side-effects**

```
a - f(b) - c * d        f(a, g(b), c)
```

- The order has an effect on efficiency (registers, scheduling)
  - **a*b +f(c)**

```
a := B[i];
c := a * 2 + d * 3;
```

Executing in parallel

Copyright © 2009 Elsevier

Ordering and rearranging

- Most languages leave the order undefined
- Let the compiler decide, for the shake of efficiency
- Some languages fix it, say from left to right

- Rearranging in mathematical expressions
  - Using commutativity

```
a = b + c
d = c + e + b
```

```
a = b + c
d = b + c + e
```

This can be dangerous and unwanted. Why?

Copyright © 2009 Elsevier

Short-circuit evaluation

- The evaluation of some expressions can finish early
  - **(a<b) or (b>c)**
- It may be useful to stop early in practice

```
if (very_unlikely_condition && very_expensive_function())
```

- So saving time is one way it can be used

Copyright © 2009 Elsevier

Short-circuit evaluation

- It changes the semantics
- Code for searching an element in a list

```
p = my_list;
while (p && p->key != val)
    p = p->next;
```

Short-circuiting in C

Non Short-circuiting in Pascal

```
p := my_list;
while (p <> nil) and (p^.key <> val) do     (* ouch! *)
    p := p^.next;
```

Copyright © 2009 Elsevier

Short-circuit evaluation

- Solution in Pascal

```
p := my_list;
still_searching := true;
while still_searching do
    if p = nil then
        still_searching := false
    else if p^.key = val then
        still_searching := false
    else
        p := p^.next;
```

Copyright © 2009 Elsevier

Short-circuit evaluation

- Short-circuiting to prevent from going out of array bounds

```
const MAX = 10;
int A[MAX];                    /* indices from 0 to 9 */
...
if (i >= 0 && i < MAX && A[i] > foo) ...
```

- Division by zero

```
if (d <> 0 && n/d > threshold)
```

Copyright © 2009 Elsevier

Short-circuit evaluation

- Some times short-circuiting is desirable
  - Exploiting side-effects
- Some languages provide both options (cand vs and)

```
function tally(word : string) : integer;
    (* Look up word in hash table.  If found, increment tally; If not
        found, enter with a tally of 1.  In either case, return tally. *)
...
function misspelled(word : string) : Boolean;
    (* Check to see if word is mis-spelled and return appropriate
        indication.  If yes, increment global count of mis-spellings. *)
...
while not eof(doc_file) do begin
    w := get_word(doc_file);
    if (tally(w) = 10) and misspelled(w) then
        writeln(w)
end;
writeln(total_misspellings);
```

Copyright © 2009 Elsevier