

CS322 W'05 Lecture Notes Lecture 5

Boolean expressions

Many languages extend “high-level” expression facility to non-arithmetic values, such as **booleans**.

- Operands: true, false, boolean-valued variables.
- Operators: and, or, not.
- Contexts: wherever a boolean value makes sense (ifs, wheres, etc.)

Remember that booleans are typically a separate type (C/C++ is an exception).

Issue: Does language use **short-circuit** evaluation for boolean expressions?

- a AND b : evaluate b only if a evaluates to true.
- a OR b : evaluate b only if a evaluates to false.

```
if (x < 7 && costly(y) > 6) ...
```

```
if (p != NULL && p->x > 7) ...
```

```
if (x < 7 || (printf("hello\n"), y > 6)) ...
```

Common misuse of booleans:

```
BOOLEAN flag;  
flag := IF (x < 2) THEN true ELSE false;
```

Expressions

- Essential component of “high-level” languages.
- Most familiar for arithmetic operators.
- Abstract away from precise order of evaluation, naming of intermediate results.

```
x1 = (-b + sqrt(b*b - 4*a*c)) / (2 * a)
```

```
t1 = -b  
t2 = b*b  
t3 = 4*a  
t4 = t3*c  
t5 = t2 - t4  
t6 = sqrt(t5)  
t7 = t1 + t6  
t8 = 2 * a  
t9 = t7/t8
```

- Issue: Precedence rules (handled in parsing).
- Issue: Mixed-mode expressions and implicit coercions.

```
real a,b;  
int c = a / b;
```

```
?? c = (int a) int/ (int b)  
?? c = (int) (a real/ b)  
?? illegal
```

Richer expression domains

Some languages support expressions over larger values, e.g., vector, strings, etc.

```
int a[10], b[10], c[10];  
c := a * 5 + b;
```

```
C: for (i = 0; i < 10; i++)  
    c[i] = a[i] * 5 + b[i];
```

```
string a, b, c;  
a := b & substring(c,2,4);
```

```
C: char *a,*b,*c;  
int n = max(strlen(c)-2,4);  
a = malloc(strlen(b) + n + 1);  
strcpy(a,b);  
strncpy(a+strlen(b),c+2,n);  
a[strlen(b)+n] = '\0';
```

More generally, can view **operators** as just special way of denoting **functions**. So, to define **expressions** over an arbitrary type, just define appropriate operator functions.

- Operator syntax, precedence, etc. may be fixed for language or programmer-definable.
- Issues like sharing, storage management are tricky.
- Not all operators act like functions.

Statement-level Control Structures

- Sequencing
- Selection
- Iteration
- (• Concurrency)

Primary mechanisms developed in FORTRAN and ALGOL60; mostly minor changes since then (30+ years).

Talk of control “structures” as opposed to “structureless” code using `goto`'s and indirect jumps (“spaghetti code”).

Concurrent computation may be more “natural” (for brains and hardware) but appears hard to reason about accurately!

Structured Programming

(e.g., Edsger Dijkstra, “Go to statement considered harmful,” *CACM*, 11(3), March 1968, 147-148.)

Branches (conditional and unconditional) suffice to program anything; they are what machines use.

BUT problems are **best** solved in terms of higher-level constructs, such as loops and conditional blocks.

- Program text should make programmer's intent **explicit**.
- Static structure of program text should **resemble** dynamic structure of program execution.

Undisciplined use of `GOTO`'s makes these goals hard to achieve.

(Not just “GOTOs are bad.”)

Machine-level Control Flow

- Sequencing; unless otherwise directed, do the next instruction.
- Labels, i.e., addresses in target code.
- Unconditional `GOTO`s.
- Arithmetic and logical **IF ? THEN GOTO** constructs.

These more than suffice to compute anything that can be computed (as best we know).

Structured Programming—Basic Elements

“Single-entry, single-exit.”

Loops:

```
while <condition> loop
  <statements>
end loop
```

Can also put test at end. Sometimes want it in the middle...

```
loop
  <statements>
  exit if <condition>;
  <statements>
end loop
```

Using `exit` violates single-exit goal. If loops are nested, want ability to `exit` any number of levels.

For loops

```
for i in <lower-bound>..<upper-bound> loop
  <statements>
end loop
```

Common questions:

- When are bounds calculated? Are they recalculated?
- Can <statements> change value of i
- Does i have a defined value after the end loop?
- Can one jump into or out of loop?
- What if upper-bound is less than lower-bound to start with?

C example:

```
for (i = *p; i > 0; i--)
```

can be optimized better than

```
for (i=1; i <= *p; i++)
```

Conditionals and Cases

```
if <condition> then
  <statements>
elsif < condition> then
  <statements>
elsif ...
else
  <statements>
endif
```

(Various parts can be missing.)

```
case <expression> of
<value1>: <statements>
<value2>: <statements>
...
otherwise: <statements>
end case
```

Permits more efficient code (a jump table) if values are “dense.”

That’s All, Folks!

This small set of statements suffices for nearly all programs.

Iteration is Recursion

We can give recursive definitions to the meaning of iterative statements.

Example:

```
while <condition> do <statements>
```

is equivalent to

```
if <condition> then
  begin
    <statements>;
    while <condition> do <statements>
  end
```

Any iteration can be converted to a recursion.

The converse is not true in general. But any **tail-recursion** (such as the one above) can be converted into an iteration. Any decent compiler should take advantage of this (though many don’t).

Taming goto

Completely unrestricted jumps are seldom allowed.

It makes little sense to allow jumps into the middle of a block, since none of the block-local storage will have been properly initialized.

Many languages permit jumps out to enclosing blocks; in a stack allocation scheme, such jumps require quietly popping one or more frames.

Most languages provide special forms of **escapes** from structured program components, such as loop `exit`.

These discourage uses of `goto`, but some good uses remain.

Uses for goto

Problem: Given a key value *k*, search an array *a* for a matching entry and increment the corresponding element of an array *b*. If not found, add the new key to the end of *a*.

A solution with `goto` (in C):

```
int i;
for (i = 0; i < n; i++)
    if (a[i] == k)
        goto found;
n++;
a[i] = k;
b[i] = 0;
found:
b[i]++;
```

A solution with one-level exit (in Java):

```
boolean found = false;
int i;
for (i = 0; i < n; i++) {
    if (a[i] == k) {
        found = true;
        break;
    }
}
if (!found) {
    i = n;
    n++;
    a[i] = k;
    b[i] = 0;
}
b[i]++;
```

This is better, but still requires testing `found` below the loop.

A solution with booleans (in Java):

```
boolean found = false;
int i = 0;
while (i < n && !found) {
    if (a[i] == k)
        found = true;
    else
        i++;
}
if (!found) {
    n = i;
    a[i] = k;
    b[i] = 0;
}
b[i]++;
```

This is clumsier and slower.

A solution with multi-level exit (in Java):

In Java (unlike C/C++), we can break from any named enclosing block.

```
int i;
search:
{ for (i = 0; i < n; i++)
    if (a[i] == k)
        break search;
    n++;
    a[i] = k;
    b[i] = 0;
}
b[i]++;
```

This does the trick. But is it any better than the original `goto` version?

The COME FROM statement

```

10 J = 1
11 COME FROM 20
12 PRINT J
   STOP
13 COME FROM 10
20 J = J + 2

```

(R. Lawrence Clark, "A linguistic contribution to GOTO-less programming," *Datamation*, 19(12), 1973, 62-63.)

But is this really a joke?

Even with a GO TO, we must examine both the branch **and** the target label to understand the programmer's intent.

What to do in an exceptional case?

- In most languages, uncaught exceptions **propagate** to next **dynamically** enclosing handler. E.g, caller can handle uncaught exceptions raised in callee.

```

foo () {
    ... throw Blah(yucc); ...
}

```

```

bar () {
    int icky;
    try {
        icky = foo ()
    } catch (Blah yucc) {
        icky = yucc++;
    }
}

```

- A few languages support **resumption** of the program at the point where the exception was raised.
- Java provides a try...finally construct:

```

f := open_file(n);
try
    ...
catch (Badinput)
    clean_up();
finally
    close_file(f);

```

Exceptions

Programs often need to handle **exceptional** conditions, i.e., deviations from "normal" control flow.

Exceptions may arise from

- failure of built-in or library operations (e.g., division by zero, end of file)
- user-defined events (e.g., key not found in dictionary)

Awkward or impossible to deal with these conditions explicitly without distorting normal code.

Most recent languages (Ada, C++, Java, etc.) provide a means to **define**, **raise**, and **handle** exceptions.

Ada example:

```

Help: exception;

begin
... if (gone wrong) raise Help; ...
... x := a / b; ...
exception
    when Help => ...report problem...
    when Numeric_Error => ...x := -99;...
end

```

Fun with C

Problem: Sending characters to an output device as quickly as possible.

Given:

```

char p[] = "hello world...";
char *m = p;
int n = ... /* length of p */
#define output(c) ... /* do output */

```

Solution 1:

```

for (i = 0; i < n; i++)
    output(*m++);

```

Faster (maybe):

```

if (n) do
    output(*m++)
    while (--n);

```

(Avoids compare with n each time.)

Faster to **unroll** loop, say 4 times:

```
while (n & 3) {
    output(*m++);
    --n;
};
n /= 4;
if (n) do { output (*m++);
           output (*m++);
           output (*m++);
           output (*m++);
        } while (--n);
```

Or (the Duff Loop):

```
i = (n+3)/4;
if (n) switch (n & 3) {
    case 0: do {output(*m++);
               case 3:    output(*m++);
               case 2:    output(*m++);
               case 1:    output(*m++)}
            while (--i);
}
```

“This is the most amazing piece of C I’ve ever seen.” – Ken Thompson