

CS558 Programming Languages

Winter 2008

Lecture 3

Most commonly-used programming languages are **imperative**: they consist of a sequence of actions that alter the **state** of the world.

State includes the values of program variables and also the program's external environment (e.g. files the program reads or writes).

Imperative programming is a good match to underlying **Von Neumann** machine programs, which are sequences of instructions that modify the contents of registers and memory locations.

- User-program variables are mapped to machine locations.
- User-program operations correspond to primitive machine instructions.

Imperative languages are also suitable for writing **reactive** programs that interact with the state of the "real world." Examples:

- Reading mouse clicks and modifying the contents of a display.
- Controlling a set of relays in an external device.

Imperative programming is the dominant paradigm, but there are alternative "declarative" paradigms too...

STATEMENTS AND EXPRESSIONS

Many languages put have a separate syntactic category of **statements** (or **commands**) that includes stateful operations which don't produce a result value.

But in some languages, certain **expressions** can also affect the state (in which case they are said to have **side-effects**) in addition to returning a result.

Also, most languages support user-defined **functions**, which contain statements but return a value and are invoked in an expression context; this is another way expressions can have side-effects.

ASSIGNMENT

The basic primitive stateful operation is typically **assignment**, which alters a value stored in a **location**.

Depending on language, assignments are statements (with no result value), or expressions (maybe with result value).

In the simplest form, the location is associated with a simple **variable**, e.g.,

$$a := a + 2$$

(Will use := for assignment, = for equality relational operator. C/C++/Java use =, == respectively: a bad idea, because **both** form expressions.)

In most languages, the variable name *a* means different things on the left-hand and right-hand sides.

On the LHS, *a* denotes the **location** of the variable *a*, into which the value of the RHS expression is to be stored.

On the RHS, *a* denotes the **value** currently contained in *a*, i.e., it indicates an implicit **dereference** operation.

ML REFERENCES

In ML, ordinary “variables” are **immutable**, i.e., they are really just names for values (computed at runtime), rather than for locations. Updatable variables, called **references**, must be explicitly created as such, and always serve as l-values. The contents of the variable must be **explicitly** dereferenced:

```
let val x = ref 2
in x := !x + 2
end

let val y = ref 0
    fun setto10 (x: int ref) = x := 10
in setto10 y
end
```

This is somewhat more verbose, but removes any confusion between l-value and r-value.

DEFINITE ASSIGNMENT

So the Java language reference manual carefully details a **conservative**, computable, set of conditions, which every program must meet, that guarantee there will be no uses before definition.

This is called the **definite assignment** property; just defining it takes 16 pages of the reference manual.

Some programs that **do** in fact initialize before use will be rejected because they violate the conditions.

Legal example:

```
int a;
if (b) /* b is boolean */
    a = 3;
else
    a = 4;
a = a + 1;
```

Illegal example:

```
int a;
if (b)
    a = 3;
if (!b)
    a = 4;
a = a + 1;
```

INITIALIZATION VALUES

Most languages require variables (and other sources of l-values) to be **declared** before they are used: gives them a type and scope, and **optionally**, an initializing expression.

In fact, it is surely a **bug** to use any variable as an r-value unless it has previously assigned a value. But many languages permit this, resulting in runtime errors.

The simplest fix is to **require** an initial value to be given for every declared variable. ML requires this for mutable `ref` variables (and also of course for ordinary immutable variables).

Java takes a slightly more sophisticated approach:

- variables do not need to be initialized at the point of declaration; but
- they **must** be initialized before they are actually used.

But in any reasonably powerful language, checking initialization before use is an **uncomputable** problem.

ORDER OF EVALUATION

Order of stateful operations affects program semantics (behavior).

Statements are always explicitly ordered, making these differences obvious.

Expressions can also have side-effects, but order of evaluation is often **under-specified** (precedence and associativity don't always fix order).

ANSI C example:

```
a = 0;
b = (a = a + 1) - (a = a + 2);
```

Result (1-3 = -2 or 3-2 = 1 ?) depends on compiler whim.

HIDDEN SIDE EFFECTS

Side-effects are not always obvious:

```
int a = 0;
int h (int x, int y) { return x; }
int f (int z) { a = z; return 0; }
h(a,f(2)); // = 0 or 2 ??
```

Keeping expression evaluation order or argument evaluation order undefined sometimes lets compiler generate more efficient code.

But modern languages (e.g., Java, ML) have moved towards precise definition of evaluation order within expressions (e.g., left-to-right).

STRUCTURED CONTROL FLOW

All modern higher-level imperative languages are designed to support **structured programming**.

Loosely, a structured program is one in which the **syntactic structure** of the program text corresponds to the **flow of control** through the dynamically executing program.

Originally proposed (most famously by Dijkstra) as an improvement on the incomprehensible “spaghetti code” that is easy to produce using the labels and jumps supported directly by hardware.

More specifically, structured programs use a very small collection of (recursively defined) **compound statements** to describe their control flow.

KINDS OF COMPOUND STATEMENTS

- Sequential composition: form a statement from a sequence of statements, e.g.

(Java) { x = 2; y = x + 4; }

(Pascal) begin x := 2; y := x + 4; end

- Selection: execute one of several statements, e.g.,

(Java) if (x < 0) y = x + 1; else z = y + 2;

- Iteration: repeatedly execute a statement, e.g.,

(Java) while (x > 10) output(x--);

(Pascal) for x := 1 to 12 do output(x*2);

SELECTION: IF

The basic selection statement is based on boolean values

if e then s_1 else s_2

which translates to

```
evaluate  $e$  into  $t$ 
cmp  $t$ , true
brneq  $l_1$ 
 $s_1$ 
br  $l_2$ 
 $l_1$ :  $s_2$ 
 $l_2$ :
```

SELECTION: CASE

To test types with more than two values, multi-way selections against constants are appropriate:

```
case e of
  c1 : s1
  c2 : s2
  ...
  cn : sn
  default : sd
```

The most efficient translation of case statements depends on **density** of the value c_1, c_2, \dots, c_n within the range of possible values for e .

SPARSE CASES

For **sparse** distributions, it's best to translate the case just as if it were:

```
t := e;
if t = c1 then
  s1
else if t = c2 then
  s2
else
  ...
else if t = cn then
  sn
else
  sd
```

DENSE CASES

For a **dense** set of labels in the range $[c_1, c_n]$, it's better to use a **jump table**:

```
evaluate e into t
cmp t, c1
brlt ld
cmp t, cn
brgt ld
sub t, c1, t
add table, t, t
br *t
table: l1
      l2
      ...
      ln
      l1: s1
      br done
      l2: s2
      br done
      ...
      ln: sn
      br done
      ld: sd
      done:
```

The best approach for a given case may involve a combination of these two techniques. Compilers differ widely in the quality of the code generated for case.

ITERATION

The basic loop construct is

```
while e do s
```

corresponding to:

```
top: evaluate e into t
     cmp t, true
     brneq done
     s
     br top
done:
```

A commonly-supported variant is to move the test to the bottom:

```
repeat s until e
```

which is equivalent to:

```
s;
while not e do s
```

LOOP EXITS

It is sometimes desirable to exit from the middle of a loop:

```
loop
  s1;
  exitif e;
  s2
end
```

is equivalent to:

```
top: s1
  evaluate e into t
  cmp t, true
  breq done
  s2
  br top
done:
```

C/C++/Java have an unconditional form of `exit`, called `break`. They also have a `continue` statement that jumps back to the top of the loop.

MULTI-LEVEL break

But we **can** do as well in Java, using a named, multi-level `break`:

```
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 construct was invented by Knuth in the 1960's, but not adopted into a mainstream language for about 30 years!)

USES FOR goto?

An efficient program with `goto`:

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

In most languages (e.g., Modula, C/C++) there is **no** equivalently efficient solution without `goto`.

COUNTED LOOPS

Since iterating a definite number of times is very common, languages often offer a dedicated statement, with basic form:

```
for  $i := e_1$  to  $e_2$  do  $s$ 
```

Here s is executed repeatedly with i taking on the values $e_1, e_1 + 1, \dots, e_2$ in each successive iteration.

The detailed semantics of this statement vary, and can be tricky. Often, s is prohibited from modifying i , which (under certain other conditions) guarantees that the loop will be executed exactly $e_2 - e_1 + 1$ times.

C/C++/Java have a much more general version of `for`, which guarantees much less about the behavior of the loop:

```
for ( $e_1; e_2; e_3$ )  $s$ ;
```

is exactly equivalent to:

```
 $e_1$ ; while ( $e_2$ ) {  $s$ ;  $e_3$  }
```

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.