Procedures as Parameters

It can be handy to pass procedures as parameters to other, higher-order procedures. This feature is supported by many languages, including Pascal, Ada, ML, and C/C++ (but not directly by Java).

Examples:

- Parameterized algorithms (e.g. in C):

```c
typedef int (* leqfn) (int,int);

void isort(int n, int a[], leqfn leq) {
    int i,j,t;
    for (i = n-1; i >= 0; i--) {
        t = a[i];
        for (j = i;
            j < n-1 && leq(a[j+1],t);
            j++)
            a[j] = a[j+1];
        a[j] = t;
    }
}

int up(int p,int q) { return p <= q; }
int down(int p, int q) { return p >= q; }

int a[] = {2,1,3};
isort(3, a, up);   /* a = {1,2,3} */
isort(3, a, down); /* a = {3,2,1} */
```
Procedures as Parameters (cont.)

- Call-backs from surrounding system:

  ```c
typedef void (* click_handler)(int);

void handler(int button) {
    switch(button) {
        case 1: cut();
        case 2: copy();
        case 3: paste();
    }
}

registerClickHandler(handler);
```
Procedures as Parameters (cont.)

- Parameterized data structure traversals (e.g. in ML)

```ml
fun dotoall (g: int -> int,
             u: int list) : int list =
  let fun f (v : int list) : int list =
      case v of
        nil => nil
      | (h::t) => (g h)::(f t)
  in f u
end

fun add3 x = x + 3
fun sub1 x = x - 1
val w = dotoall (add3, [1,2,3])
  (* yields [4,5,6] *)
val z = dotoall (sub1, [1,2,3])
  (* yields [0,1,2] *)
```

ML also supports anonymous function values, i.e., functions that can be defined without being named. Could do above example as:

```ml
val w = dotoall (fn x => x + 3, [1,2,3])
val z = dotoall (fn x => x - 1, [1,2,3])
```
Using Local (Nested) Procedures

- Sometimes want to pass local functions as parameters.

```haskell
fun meandeltas (u : int list) : int list =
    let val mean : int = compute_mean u
        fun compute_delta (x:int) = x - mean
    in dotoall (compute_delta,u)
end
```

- Lexical scope rules apply, so function body can use data associated with outer function.

- Here `compute_delta` uses the value of `mean`, which is local to `deltas`.

- Cannot express this in C/C++/Java, which have no nested functions.

- Possible implementation: pass pair of (code-pointer, env-pointer) as “value” of procedure.

- Must guarantee that env pointer is still valid when procedure is called!
More Nested Procedures

Another example:

```haskell
fun deltas(n:int, u:int list) =
  let fun compute_delta (x:int) = x - n
  in dotoall (compute_delta,u)
  end

...deltas(3,[1,7,5]) (* yields [~2,4,2] *) ..
```

- Here compute_delta depends on the value of variable n, which is a local parameter of deltas.

What if we want to compute deltas on several different lists with a fixed n?

- Can be handy to treat procedure values just like other values, e.g., to return them as function results or store them into variables.
“First-class” Procedures Example

```ml
fun deltas' (n:int) : int list -> int list =
  let fun deltas (u : int list) : int list =
    let fun compute_delta (x:int) : int = x - n
    in dotoall (compute_delta,u)
    end
  in deltas
end

val g : int list -> int list = deltas' 3
...
val x : int list = g [1,7,5] (* yields [~2,4,2] *)
val y : int list = g [2,4,6] (* yields [~1,1,3] *)
val z : int list = deltas' 3 [2,4,6]
  (* yields [~1,1,3] *)
```

ML also provides syntactic sugar to make such “Curried” functions easier to write. Above program is equivalent to:

```ml
fun deltas' (n:int) (u:int list) : int list =
  let fun compute_delta (x:int) = x - n
  in dotoall (compute_delta,u)
end
```
Using Curried functions

- When defining “multi-argument” functions in ML, have a choice using a tuple argument and Currying.
- Can apply Curried version `deltas'` to either one or two arguments.
- Function application associates to the left, so

  \[
  \text{deltas'} \ 3 \ [2,4,6] = \ (\text{deltas'} \ 3) \ [2,4,6]
  \]

- Function type arrows associate to the right, so the type of `deltas'` is

  \[
  \text{int} \rightarrow \text{int list} \rightarrow \text{int list} = \ 
  \text{int} \rightarrow (\text{int list} \rightarrow \text{int list})
  \]

- Currying most often useful when passing partially applied functions to other higher-order functions, e.g.:

  \[
  \text{map} \ (\text{deltas'} \ 3, \ [[1,7,5],[2,4,6]])
  \]
  \[
  (* \text{ yields } [[\sim 2,4,2],[\sim 1,1,3]] \ *)
  \]

(Here \text{map} is a standard library function that works like \text{dotoall}, but can be applied to lists of any type.)
Problems with first-class procedures

Consider activation tree for **deltas’** example:

```
main
  /
  /
  /
  /
  /
  /
  / \
 /   \
/     \
/       \
/         \
deltas'(3)  g([2,4,6]) == deltas([2,4,6])
           /
           /
           /
           /
           / \
           /   \
           /     \
dotoall(compute_delta, [2,4,6])
               /
               /
               / \
               /   \
               /     \
(requires value n = 3) compute_delta(2)
```

Activation of **deltas’** is no longer live when **compute_delta** is called!

If **n** is stored in activation record for **deltas’** and activation-record is stack-allocated, it will be gone at the point where **compute_delta** needs it!

To avoid this problem:

- Pascal prohibits “upward funargs;” procedure values can only be passed downward, and can’t be stored.

- Some other languages only permit “top-level” procedures to be manipulated as procedure values (in C, this means **all** procedures!).
Heap Storage for Procedure Values

- Languages supporting first-class nested procedures (e.g., Lisp, Scheme, ML, Haskell, etc.) solve problem by using heap to store variables like n.

- Simple solution: Just put all activation records in the heap to begin with! (Garbage collection is a must!)

- More refined solution: Represent procedure values by a heap-allocated “closure” record, containing the procedure’s code pointer and values of the non-local variables referenced by the procedure.

- Involves taking copies of the values of non-local variables, so only works when values are immutable.

- Can always introduce extra level of indirection to achieve this.
Using first-class functions

The ability to manipulate functions as first-class values is one of the hallmarks of a functional language.

Functional languages encourage sophisticated abstraction mechanisms. Consider the following problems:

Sum a list of integers

```haskell
fun sum l = 
  case l of 
    nil => 0 
  | h::t => h + (sum t)
```

Multiply a list of integers:

```haskell
fun prod l = 
  case l of 
    nil => 1 
  | h::t => h * (prod t)
```
Folds

Copy a list (of anything):

\[
\text{fun copy } l = \\
\text{    case } l \text{ of} \\
\text{    nil => nil} \\
\text{    | h::t => h::(copy t)}
\]

Query: How does \text{copy} differ from the identity function \text{fn x => x}?

Calculate the length of a list (of anything):

\[
\text{fun length } l = \\
\text{    case } l \text{ of} \\
\text{    nil => 0} \\
\text{    | h::t => 1 + (length t)}
\]
Folds (continued)

We can **abstract** over the common inductive pattern displayed by these examples:

```ml
fun foldr (f, n) = 
    let fun r l =
        case l of
        | nil => n
        | h::t => f(h, r t)
    in r
    end

fun sum l = foldr(fn (x, y) => x+y, 0) l
fun prod l = foldr(op*, 1) l
fun copy l = foldr(op::, nil) l
fun length l = foldr(fn (_, y) => 1+y, 0) l
```

Function `foldr` computes a value working from the tail of the list to the head (from right to left). Argument `n` is the value to return for the nil list. Argument `f` is the function to apply to each element and the previously computed result.
Applicative Programming

Basic advantage of expressions: can describe complex, tree-like computations without having to map them to explicit linear sequences of instructions and temporaries.

Our HW2 expression language had only immutable (unchanging) variables, had no I/O facilities, and computed a single result value.

Num (Int) : Exp → ε
Var (Id) : Exp → ε
Add, Sub, Mul, Div : Exp → Exp Exp
Let (Id) : Exp → Exp Exp
Letfun (Id) (Id) : Exp → Exp Exp
App (Id) : Exp → Exp Exp
Ifzero : Exp → Exp Exp Exp

• Variables are just abbreviations for complex expressions, as in mathematics.

• Expressions can be evaluated with the aid of an environment mapping variables to values. Once set, an environment entry is never changed.

• Evaluation doesn’t require any notion of state.

• Order of evaluation doesn’t matter, except for data dependencies.

Style of programming called functional or applicative.
Imperative Languages

Most commonly-used programming languages are imperative: they consist of a sequence of operations that alter the state of the program.

- The state includes the values of variables, which can change by means of assignment operations.

- The state also includes the input/output history of the program, e.g., the contents of files (or virtual files) read or written by the program’s I/O operations.

- Order of evaluation often matters!

Many languages put have a separate syntactic category of statements (or commands) that includes stateful operations which don’t produce a result value. But expressions can also affect the state (in which case they are said to have side-effects) in addition to returning a result.
Stateful Programming

Stateful 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 operators include 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.
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.

This makes sense when variables are thought of as locations containing values.

But note that the value of a variable may be a pointer (or reference) to some other location, e.g., in C:

\[ \text{int } *y; /* y contains a pointer to an int } */ \]
L-values and R-values

Many languages allow more general expressions on the LHS of assignments, as long as they denote locations, e.g. array cells or record fields:

\[ a.x := b.y + 2 \]

The **l-value** of an expression is the location it denotes (if any) when it appears on a LHS; the **r-value** of an expression is its (ordinary) value when it appears on a RHS.

When it is desirable to use the l-value of a variable somewhere other than the LHS of an assignment, special syntax is needed to indicate that the l-value is wanted. 

C example:

```c
int y = 0;
setto10(int *x) {
    *x = 10;
}
setto10 (&y);
```
ML References

In ML, ordinary variables are immutable, so they do not have l-values. 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.
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.
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:

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

Illegal example:

```java
int a;
if (b)
    a = 3;
if (!b)
    a = 4;
a = a + 1;
```
Order of Evaluation

Order of stateful operations affects program semantics.

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:

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

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

Side-effects are not always obvious:

```c
int a = 0;
int h (int x, int y) { return x; }
int f (int z) { return a = z; }
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.

Compounds are of three kinds:

- **Sequential composition**: form a statement from a sequence of statements, e.g.
  
  \begin{align*}
  \text{(Java)} & \{ \ x = 2; \ y = x + 4; \} \\
  \text{(Pascal)} & \text{begin } x := 2; \ y := x + 4; \text{ end}
  \end{align*}

- **Selection**: execute one of several statements, e.g.,
  
  \begin{align*}
  \text{(Java)} & \text{if } (x < 0) \ y = x + 1; \text{ else } z = y + 2; \\
  \end{align*}

- **Iteration**: repeatedly execute a statement, e.g.,
  
  \begin{align*}
  \text{(Java)} & \text{while } (x > 10) \text{ output } (x--); \\
  \text{(Pascal)} & \text{for } x := 1 \text{ to } 12 \text{ do output } (x*2); \\
  \end{align*}
Selection

The basic selection statement is based on boolean values

\[
\text{if } e \text{ then } s_1 \text{ else } s_2
\]

which translates to

\[
\text{evaluate } e \text{ into } t \\
\text{cmp } t, \text{true} \\
\text{brneq } l_1 \\
s_1 \\
\text{br } l_2 \\
l_1: \quad s_2 \\
l_2:
\]

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

\[
\text{case } e \text{ of} \\
c_1 : s_1 \\
c_2 : s_2 \\
\ldots \\
c_n : s_n \\
\text{default} : s_d
\]

The most efficient translation of case statements depends on \textbf{density} of the value \(c_1, c_2, \ldots, c_n\) 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:

\[
\begin{align*}
    t & := e; \\
    \text{if } t = c_1 \text{ then } & s_1 \\
    \text{else if } t = c_2 \text{ then } & s_2 \\
    \text{else } & \\
    \ldots \\
    \text{else if } t = c_n \text{ then } & s_n \\
    \text{else } & s_d
\end{align*}
\]
Dense Cases

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

```plaintext
evaluate e into t
  cmp $t, c_1$
  brlt $l_d$
  cmp $t, c_n$
  brgt $l_d$
  sub $t, c_1, t$
  add table, $t, t$
  br *$t$

table: $l_1$
  $l_2$
  ...
  $l_n$

$l_1$: $s_1$
  br done

$l_2$: $s_2$
  br done
  ...

$l_n$: $s_n$
  br done

$l_d$: $s_d$

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

\[ \text{while } e \text{ do } s \]

corresponding to:

\begin{verbatim}
top:   evaluate e into t
        cmp t, true
        brneq done
        s
        br top

done:
\end{verbatim}

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

\[ \text{repeat } s \text{ until } e \]

which is equivalent to:

\begin{verbatim}
s;
        while not e do s
\end{verbatim}
Loop exits

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

```plaintext
loop
  s_1;
  exitif e;
  s_2
end
```

is equivalent to:

```plaintext
top:  s_1
  evaluate e into t
  cmp t, true
  breq done
  s_2
  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.
Uses for goto?

An efficient program with goto:

```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]++;
```

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

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

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

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


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.
Counted loops

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

\[
\text{for } i := e_1 \text{ to } e_2 \text{ do } s
\]

Here \( s \) is executed repeatedly with \( i \) taking on the values \( e_1, e_1 + 1, \ldots, e_2 \) 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:

\[
\text{for } (e_1; e_2; e_3) \ s;
\]

is exactly equivalent to:

\[
e_1;
\text{while } (e_2) \{
  s;
  e_3
\}
\]
Fun with C

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

Given:

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

Solution 1:

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

Faster (maybe):

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

(Avoids compare with n each time.)
More fun

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

```c
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):

```c
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

Does this work in Java?