Objectives

1. To discuss mechanisms for passing parameters to procedures/functions.
2. To discuss approaches to evaluating the parameters of procedures/functions.
3. To discuss positional versus non-positional syntax for parameters, default parameters, and variable-length parameter lists
4. To discuss first class procedures/functions
5. To introduce coroutines

Materials:

1. Projectables of the various demo programs
2. Abillity to demonstrate Prolog

I. Introduction

A. In our discussion of control structures, we saw that most computer machine languages provide four basic operations for controlling the order of statement execution:

   1. Unconditional branch

   2. Conditional branch

   3. Subroutine call

   4. Subroutine return

B. The latter two have found their way into almost every higher-level language in the form of mechanisms for breaking a large program down into smaller procedures (also called subroutines) and/or functions.

   1. Recall that the basic difference between a function and a procedure is that a function returns a value to the caller, and thus is called by being embedded within an expression. A procedure does not return a value, and so is called by a "stand-alone" statement.

      a. From a programming practice standpoint, functions should compute a single result without side effects. This is, of course, a key idea in functional langauges, but some imperative/OO languages also prohibit functions from performing side-effect producing actions - e.g. in Ada, functions are only allowed to take "in" mode parameters, whose value they may not alter.

      b. There is also an issue as to what types of value a function may return.

         i. Motivated in large measure by implementation considerations, some languages (e.g. Pascal) restrict functions to returning values of simple, scalar type.

         ii. Other languages (e.g. Ada) allow a function to return a value of any data type.

iii. Of course, languages in which procedures and/or functions are first-class data objects also allow functions to return a procedure or function as their value.

2. Many languages provide separate "procedure" and "function" entities (e.g. Ada, or FORTRAN's subroutines and functions).

   a. A few langauges provide only procedures.

   b. Others provide only functions, but allow a function's return value to be ignored.

   c. Languages in the C family provide only functions, but define a "void" type that is used as the return type of a function whose value is to be ignored.

   d. In the discussion that follows, we'll use the term "routine" when addressing issues common to both procedures and functions.

3. In addition, most languages (but not all) provide a mechanism whereby the caller of a routine can pass parameters to it. Some languages do not have this, or limit it, though, but we'll not pursue this.

C. The following are the issues we want to consider with regard to routines:

1. Parameter passing mechanisms

2. Evaluation of parameters.

3. Positional and non-positional parameters, default parameters, and variable-length parameter lists

4. First-class procedures/functions.

5. Coroutines

II. Parameter Passing Mechanisms

A. Some terminology

1. When a routine with parameters is defined, its definition includes a list of FORMAL parameters. When it is called, its call specifies a list of ACTUAL parameters.

2. Parameters can be used in one of three different ways, sometimes called MODES:

   a. in - the parameter serves to communicate information from the caller to the routine, but not vice versa.

   b. out - the parameter serves to communicate information from the routine back to the caller - but not vice versa.

   c. in out - the parameter serves to communicate information both ways.

d. Ada is unique among programming languages in incorporating these modes into the syntax of the language - i.e. Ada requires that the formal parameters of a procedure be declared in, out, or in out. (Ada functions can have only in parameters).

e. Other language use various mechanisms for parameter passing that indirectly give the effect of some or all of these modes. (Even the mechanisms actually used by Ada produce some complexities.)

3. Parameters are generally transmitted between a caller and a callee through some sort of PARAMETER LIST. The actual form this takes depends on the underlying hardware.

a. Often, the parameter list is a region in memory, either created on the stack or somewhere else, which ends up looking something like this:

```
Parameter list:     First actual
                    Second actual
                    Third actual
                    ...
```

b. Alternately, the "parameter list" may be a sequence of hardware registers - e.g. MIPS specifies that the first actual parameter to a routine is placed in $4, the second in $5, etc.

c. We'll use the term "parameter list" here to refer to the sequence of actual parameters, without worrying about the hardware implementation - which in any case is usually determined by the architecture of the target machine, rather than by the language (e.g. most languages compile to Pentium code that passes parameters in a parameter list in memory, but to MIPS code that passes them through registers.)

B. One place where languages do differ on a language rather than implementation basis is the mechanism(s) they use for passing parameters. Historically, six different mechanisms have been used, though most languages use but one or two. Mechanisms are distinguished by their answer to the following question: what does the caller of a routine place in the parameter list that it gives to the routine being called?

C. Even before higher-level languages appeared, there were assembly languages - some of which offered a macro facility. This mechanism uses CALL BY TEXT.

1. In call by text, no parameter list per se is used. Rather, the language translator "plugs in" the TEXT of the actual parameter into the code of the procedure at the point it is used. (The answer to the question "what goes in the parameter list" is "nothing" - there is no parameter list.)

a. This means, of course, that a new copy of the procedure must be included in the code for every call.

b. This is, therefore, a procedure mechanism only in the loosest sense of the word, since no "call" and "return" instructions are ever used.

c. Though it often increases the SIZE of the code, it improves its SPEED by eliminating the overhead of procedure calling and parameter passing.

2. One major higher-level language that uses this mechanism is C with its macro facility. Consider the following example:

```
#define cube(x) x * x * x

...

y = cube(m);
z = cube(l -> info)
```

a. The C pre-processor replaces the two occurrences of cube to yield the following:

```
y = m * m * m;
z = l -> info * l -> info * l -> info;
```

b. The C compiler per-se never actually sees the "call" of cube.

c. Note that the TEXTUAL FORM of the actual parameters m and l -> info are actually substituted for the formal parameters. Nothing has to occur at run time to transmit values from the actual parameters to the formals, because the compiler sees only the actuals, never the formals.

d. One problem that can arise with this occurs if the actual parameter is itself an expression with side effects = e.g. if we used   w = cube(m[i++]), this would expand to:

```
w = m[i++] * m[i++] * m[i++]
```

which would multiply three successive elements of the array m instead of cubing one element, and would increment i three times!

3. Several languages, including Ada and C++, include a facility that allows the programmer to specify that a given procedure be expanded INLINE each time it is used. This gives something of the effect of call by text. For example, consider the following Ada procedure definition and use:

```
function Cube(I: in Integer) return Integer is
    begin
        return I * I * I;
    end Cube;
...
J := Cube(X);
```

a. If this were compiled as an ordinary way, then the code generated for the procedure call would be something like this (on MIPS):

```
lw      $4, X
jal     Cube
sw      $2, J
```

b. But if the procedure were expanded inline, then the following code would be generated instead:

```
lw      $4, X
mul     $2, $4, $4
mul     $2, $2, $4
sw      $2, J
```

(In this case, the inline code is longer but faster than the function call. For some architectures, the inline code might be shorter as well.)

c. Compilers that inline code are written so as to avoid evaluating expressions more than once to avoid the problem we noted above with cube(m[i++]).

D. FORTRAN introduced CALL BY REFERENCE as its only mechanism for passing parameters. The LVALUE of the actual parameter is placed in the parameter list.

1. This mechanism is very general, in that it allows information to flow both ways between caller and procedure.

2. But it does have two weaknesses as a sole mechanism:

   a. Any changes made by the procedure to the parameter are seen by the caller. There is no way to give the procedure a local copy of the parameter for it to work with without "damaging" the caller's copy.

   b. There is a complexity with this mechanism when the actual parameter is a constant or an expression.

      i. As you recall, what goes in the parameter list in this case is the LVALUE of the actual parameter.

      ii. However, constants frequently don't have lvalues, and expressions never do.

      iii. Thus, when the actual parameter is an expression (and sometimes when it is a constant), the compiler must create at run time a temporary variable and pass its address to the procedure.

      iv. On the other hand, in the case of a constant that is stored in a memory location (like a const variable), we can simply pass the address.

      v. We will explore an interesting consequence of this in the homework.

3. In addition, call by reference can result in the ALIASES, where the same entity is accessible by two different names. For example, this shows how aliases could arise in FORTRAN

```
PROGRAM MAIN
COMMON I
CALL SUB(I)
...

SUBROUTINE SUB(J)
COMMON K
```

When called as above, J and K are aliases for what MAIN calls I

4. Nonetheless, call by reference is also used as the sole mechanism for COBOL and is an option for Pascal and C++, and is used in some cases by Ada and Java.

E. Algol introduced two new mechanisms: CALL BY VALUE and CALL BY NAME. We will consider call by value first.

1. Call by value addresses the two major problems of call by reference by giving the procedure a PRIVATE COPY of the RVALUE of the parameter.

   a. This generally means that the parameter list contains the RVALUE of the actual parameter.

   b. However, if the actual parameter is an array or structure, the parameter list may actually contain the LValue, with the code for the routine responsible for making a local copy.

2. The price tag for this, of course, is that the procedure cannot return any information to the caller through the parameter.

   a. Hence, many languages that offer call by value also offer an alternate mechanism (like the reference parameters of C++) for use when two way communication is needed.

   b. C offers only call by value, but has an "address of" operator that allows the effect of call by reference to be achieved using pointer syntax in the called routine.

   c. Both LISP and APL offer only call by value, with no good general way to return values to the caller except through the value of a function - but this is totally consistent with the functional paradigm. (Indeed, offering another parameter passing mechanism would be inconsistent with the paradigm on which these languages are based.

   d. Java uses call by value for primitive types, with no alternative available.

3. Another limitation of call by value is that, when the parameter is a large data structure such as an array, there can be considerable time and space overhead involved in making a local copy every time a procedure is called. (This is true regardless of whether the copy is made by the caller or by the routine.)

   a. Note that it is the desire to avoid this that is one reason why C family languages use call by reference when ARRAYS are passed as parameters, and Java uses call by reference for objects as well.

b. This is not a problem in languages like LISP that have a reference model for variables, because variables are always represented by pointers anyway, so only a pointer need be passed.

   i. Recall that the rvalue of a pointer is, in fact, the lvalue of some other object.

   ii. This means that the routine can change the internals of the object passed to it - but cannot change WHICH object is passed to it.

   Example: (Java)

```
Person p;
Person q;
...
someMethod(p);
...
void someMethod(Person x)
{
     x.salary *= 1.05;   // Changes the Person object p refers to
     x = q;              // Has no effect on p
```

   iii. The text calls this case of call by value call by SHARING (which is actually what it is called in the Python community, though the Java community refers to this as call by value.)

F. Algol's other mechanism was CALL BY NAME. What is placed in the parameter list is the address of a BLOCK OF CODE that can be executed to produce the parameter.

1. For example, suppose we had the following - PROJECT

```
procedure p(j); integer j;     -- note: in Algol a parameter
begin                             is passed by name unless
    integer k;                    explicitly declared value
    k := j;                       (opposite of Pascal
    i := i + 1;                    convention)
    j := k + 1
end
...

i := 1;
p(x[i]);
```

a. What would go into the parameter list for the call to p would be the address of a block of code something like the following:

```
lw      $2, I
la      $3, X
sll     $2, $2, 2       # Puts i * 4 (size of integer) in $2
add     $2, $2, $3      # Puts the address of x[i] in $2
```

b. This code would be executed twice by p - once for each time it refers to x. Because i is altered in between, the first reference to j would refer to x[1], but the second to x[2]. Thus, the effect of the call to p would be to set x[2] := x[1] + 1 (and also to increment i.)

c. By way of contrast, if call by reference were used i would also be altered, but the effect would be to set x[1] := x[1] + 1, since the address of x[i] is computed once, at the time the procedure is called.

d. Note that call by name is equivalent to call by reference in the case that the actual parameter is a simple variable, but is quite different otherwise.

2. Call by name makes possible some very interesting coding "tricks". One such trick is known as "Jensen's device".

   a. Consider the following function - PROJECT

```
real procedure sigma(x, j, n); value n; real x; integer j, n;
begin
    real s;
    s := 0;
    for j := 1 step 1 until n do s := s + x;
    sigma := s
end
```

   b. If called by sigma (x[i], i, 10), it will sum x[1] .. x[10]

   c. If called by sigma (a[i]*b[i], i, 10), it will sum a[1]*b[1] .. a[10]*b[10]

   etc.

3. However, call by name also makes some tasks very difficult or at least confusing.

   a. Consider the following simple procedure that swaps its arguments:

```
procedure swap(a, b); integer a, b;
begin integer t;
    t := a;
    a := b;
    b := t
end
```

   b. This works fine when we try something like swap(x,y). But what happens if we try swap(i, a[i])? (ASK CLASS) Answer: the final assignment becomes equivalent to a[a[i]] := t!

4. Further, call by name is very difficult to implement. To see why, consider the following situation, where p is the same procedure as before, and its parameter is to be passed by name:

```
procedure q;
  begin real array j[1:10]; integer k
      k := 1;
      p(j[k]);
  ...
```

a. Notice that the j and k in the actual parameter are local variables of q that happen to have same name as the parameter and local of p. The code that is executed when p evaluates j must refer to the j and k belonging to q, not to the j and k belonging to p. (Observe that this is in contrast to call by text.)

b. Furthermore, since j and k are local, they are allocated storage on the run time stack. Thus, in addition to passing the address of code to evaluate j[k] to p, q must also pass the address of its stack frame.

c. In Algol, the procedures that the compiler generates to handle parameters passed by name are called "thunks". The name comes from the supposed noise the procedure makes as it traverses the static chain of stack frames to access its lexical variables.

5. The complexity of implementing call by name, when coupled with some of the strange results in can produce, has led to its not being used by any language since Algol-60. Pascal and the C family, for example, adopted call by value from Algol, but took call by reference from FORTRAN in place of call by name.

G. Ada uses CALL BY VALUE in some cases, but also uses two new but closely related mechanisms - CALL BY RESULT and CALL BY VALUE RESULT. (Both were first introduced in Algol-W - a precursor to Pascal.)

1. In Ada, formal parameters of a procedure must be declared either in, out, or in out. (The specifier may be omitted, in which case it defaults to in.)

Example:

```
procedure Insert_Into_Tree(Item: in Nametype;
                           Tree: in out Nodeptr;
                           Is_Duplicate: out boolean)
```

2. In parameters are passed by value. However, in contrast with other languages using call by value, Ada specifies that a procedure may not assign new values to its in parameters - i.e. the in parameters are treated like local CONSTANTS within the procedure body.

3. Out parameters are passed by result.

a. The formal parameter is treated like a local variable of the procedure.

b. When the procedure exits, the value of the formal parameter is copied back to the actual.

c. The parameter list needs to actually contain the LVALUE of the actual parameter - but the copying is done from the formal to the actual at the end of the procedure.

4. In out parameters are passed by value result.

a. The formal parameter is treated as a local variable of the procedure.

   b. When the procedure is called, the value of the actual parameter is copied into the formal parameter.

   c. When the procedure exits, the formal parameter is copied back to the actual. (That is, the parameter value is copied twice - once each way.)

   d. Again, the parameter list actually contains the LVALUE of the actual parameter, which is used for making copies both ways.

5. Notice that the Ada mechanism, besides being more straight-forward semantically, avoids the problem of aliasing inherent in reference parameters.

6. Actually, the benefit of this is somewhat reduced by the way Ada handles parameters of structured type.

   a. Call by value, call by result, and call by value result could require much overhead when the parameter being passed is a large data structure.

   b. For this reason, the Ada 83 standard allowed the compiler to use call by reference for parameters of structured types instead of call by value/result/value-result.

      i. This is not a semantic problem for in parameters, because in parameters cannot be modified by the procedure. (The compiler treats in parameters as if they were constants; the procedure can look at them but not assign to them.)

      ii. It does reintroduce the potential problem of aliasing, but only for structured parameters, never for scalars.

      iii. Worse yet, a given program may behave differently under two different compilers if it involves a situation where aliasing could occur and one compiler writer chose to use call by reference while the other stuck with call by result or value-result. For this reason, Ada 83 standard dictates that any program that depends for its correctness on an assumption about which mechanism is used for structured types is erroneous (though a compiler can't catch this!)

      iv. You have a homework problem which asks you to determine which approach gnat uses by looking at the results of running a program whose output differs under call by value-result and call by reference (an erroneous Ada program, of course!)

7. Finally, we should notice that the result and value-result mechanisms introduce some new questions, centering on WHEN the address of an out or in out parameter is evaluated.

a. Consider the following code: - PROJECT

```
procedure P(I: in out Integer; J: out Integer) is
    begin
        I := I + 1;
        J := 0;
    end P;

....

X: array(1..10) of Integer;
K: Integer;

begin
    K := 1;
    P(K, X(K));
    ...
```

What element of X is set to 0?

i. If the address of X(K) is computed before the call to P, then X(1) is altered.

ii. If the address of X(K) is computed after the return from P, then depending on whether this is done before or after the new value is stored in K, either X(1) or X(2) could be zeroed.

b. The problem could become even more complex with an in out parameter - conceivably one could evaluate the address of the actual parameter TWICE - once before the call to get its value to send to the procedure, and once after the call to decide where to store the result sent back.

c. Ada handles this problem by explicitly specifying that the identity of an out or in out parameter is to be established once, before the procedure is called (which is the natural way to do things anyway if the parameter list contains the LVALUE of the actual.)

H. To summarize the difference between the various mechanisms, consider the following (very contrived) program. Assume it is written in a Pascal-like language that allows the programmer to specify the parameter passing mechanism from among the various choices we have considered.

PROJECT

```
int a[10];
int b;
int i;

void p(int ___ x, int ___ y, int ___ z)        <-- mechanism to be
{                                                  specified here by
    int i;                                         the programmer
    x = 3;                                         (assume the same
    i = 2;                                         mechanism is used in
    y++;                                           all 3 cases.)
    z = b;
    x = 4;
}

void main(int argc, char ** argv)
{
    i = 1;
    b = 37;
    p(i, b, a[i]);
    ...
}
```

We now ask, regarding the call to p, what element of the array a is altered, and what is it changed to? (The procedure will also affect other variables, but we focus on this one.)

1. If the mechanism for parameters is call by value, the answer is that NO ELEMENT of a is changed.

2. If the mechanism for parameters is call by reference, the answer is a[1] := 38. (b and y are aliases, as are z and a[1])

3. If the mechanism for parameters is call by text, the answer is a[2] := 38. (After textual substitution, the procedure body becomes:

```
i = 3;         <-- note: the i referred to is always
i = 2;             the local variable i, not the
b++;               global
a[i] = b;
i = 4
```

4. If the mechanism for parameters is call by name, the answer is a[3] := 38. (x and global i are aliases, as are z and b. When the assignment z = b is done, global i (the one visible to the caller that created the "thunk" is 3 and b is 38.)

5. If the mechanism for parameters is call by value-result, the answer could be either a[1] := 37 - if the address of a[i] is calculated only once - or a[4] := 37 - if the address of a[i] is calculated twice and results are stored into the actual parameters left to right. (Ada would always produce a[1] := 37). The reason why the value becomes 37 rather than 38 is that call by value-result prevents aliasing, so the assignment y := y + 1 has no effect on b until AFTER the procedure exits.

III. Evaluation of Parameters

A. When an actual parameter is an expression rather than a variable or constant, the question arises as to when the expression is evaluated.

B. Commonly, the answer is what is called strict or applicative-order evaluation: the argument expression is evaluated just before the routine is invoked.

1. A subordinate question that arises when there are multiple expression parameters is what order they are evaluated in.

   e.g. given

   ```
   int test(int a, b) ...

   int evalArg(int arg) {
      cout << arg << ' ';
      return arg;
   }
   ```

   will test(evalArg(1), evalArg(2)) write 1 2 or 2 1?

   a. Left-to-right seems (1 2) seems more intuitive.

   b. However, if the parameter list is pushed on a stack, then given that stacks typically grow from high addresses to low addresses, there is an advantage to pushing parameters on the stack in right to left order. This may make it advantageous to evaluate the parameter expressions in right-to-left order (which yields the output 2 1 in this case).

   c. You have a homework problem concerning this.

2. Of course, this isn't an issue at all if the expressions don't have side effects (they are referentially transparent.)

C. Sometimes, though, it may be desirable to defer the evaluation of a parameter until _after_ the routine is called and actually needs it.

1. There are a couple of cases where this might occur.

   a. This could happen in a case where evaluation of the parameter would result in an error in some cases - but the routine wouldn't need it anyway in those cases.

   b. This could happen if evaluation of the parameter might construct an object such as a list of indeterminate size, where the actual size needed is determined by the something in the called routine.

2. Such cases may be handled by a strategy called "lazy evaluation", in which evaluation of the parameter expression is postponed until the parameter's value is actually used.

   a. This is, of course, what call by name actually does.

b. As we saw in our discussion of call by name, if a given parameter is used several times in a routine, then call by name evaluates it several times.

i. At best, this can be inefficient if the evaluation is complex.

ii. At worst, this could result in getting a different value each time if the expression has side effects or some value it uses has been changed in between evaluations.

c. An alternative approach is to cache the results of a computation - so if the same parameter is used more than once, it is evaluated once and then the cached value is used subsequently.

IV. Positional and Non-Positional Parameters; Default Parameters; Variable-Length Parameter Lists

A. By way of review, recall that when a routine is DECLARED, the parameters appearing in the declaration are called FORMAL PARAMETERS. When it is CALLED, the parameters appearing in the call are called ACTUAL PARAMETERS.

1. In statically-typed languages, it is normally expected that the actual parameters have the same number, type, and order as the formal parameters, and in strongly typed languages this is generally enforced by the compiler.

2. In dynamically-typed languages, of course, there is no expectation regarding the type of the parameters, but passing a parameter of an inappropriate type may either result in a run time error or a bizzare result.

B. Ordinarily, actual parameters are matched with formal parameters by POSITION - i.e. the first actual parameter is matched with the first formal, the second actual with the second formal, etc. Further, it is expected that there will be exactly as many actual parameters as formals.

C. However, each of these conventions has problems, which have led to new features in some languages.

1. The matching of actual parameters with formal parameters by position poses two problems.

a. If there are more than one or two parameters to a routine, then it is sometimes burdensome for the programmer to have to remember the exact order of the parameters. Some routines may take a half-dozen or more parameters; someone using one of these will almost certainly have to refer to a reference manual every time he uses it.

b. Further, it is very easy for a programmer to inadvertently switch the order of two actual parameters in routine call. Of course, the compiler may sometimes catch this; but the compiler won't catch it if the two parameters happen to be of the same type.

Example: I always have problems with the C library function strcpy, which copies character strings. It takes two parameters, the first being the destination string and the second being the source. However, because I am used to the Unix cp command, which puts the source first and then the destination, I tend to mix these up. Since both parameters are strings, the compiler can't help me.

2. The requirement that the routine call contain the same number of actual parameters as formals in the routine declaration also poses two problems.

   a. Sometimes, there are certain parameters which are only needed in in certain cases, or which almost always have a certain standard value. It is a nuisance to have to specify them every time. (Creating multiple versions of an overloaded routine with different parameters can address this for the user at the expense of creating extra work for the author.)

   b. In some cases, it would be nice to allow a routine to have any number of parameters of a given kind - as in the C printf routine, which can take any number of parameters.

D. Three features available in some languages address these problems. Though the features are distinct, they are related, so we discuss them together.

1. Some languages allow NON-POSITIONAL syntax for parameters (also called NAMED PARAMETERS).

   a. Example: Ada incorporates this. Suppose we have a procedure that sums a specified range of elements of an array and returns their sum to the caller - declared as follows:

   PROJECT

```
procedure Sum_Array(Arr: Array_Type;
                    Lo_Limit: Integer;
                    Hi_Limit: Integer;
                    Sum: out Float)
```

   If we want to sum up X(3) through X(10) and put the result in the variable X_Sum, we could call this in any of the following ways:

   i. Pure positional syntax:

```
Sum_Array(X, 3, 10, X_Sum);
```

   ii. Pure non-positional syntax - any of the following (and many other possibilities).

```
Sum_Array(Arr => X, Lo_Limit => 3, Hi_Limit => 10, Sum => X_Sum);
Sum_Array(Arr => X, Sum => X_Sum, Lo_Limit => 3, Hi_Limit => 10);
Sum_Array(Sum => X_Sum, Hi_Limit => 10, Lo_Limit => 3, Arr => X);
```

iii. Mixed syntax (many possibilities, just one shown)

```
Sum_Array(X, Sum => X_Sum, Lo_Limit => 3, Hi_Limit => 10);
```

(Note: When mixing positional and non-positional parameters, Ada requires that all positional parameters occur first.)

b. COMMON LISP allows one to declare certain parameters as keyword parameters, in which case non-positional syntax can be used with them (but not with other parameters.)

Example: given the definition

```
(defun f (a &key b c)
    ...
```

one could call f in either of the following ways:

```
(f 1 :b 2 :c 3)
(f 1 :c 3 :b 2)
```

but not

```
(f 1 2 3)
```

(LISP requires that keyworded parameters be called using the explicit keyword names)

2. Some languages allow the specifying of DEFAULT values for certain formal parameters. The caller may choose to omit actual parameters corresponding to these, and the compiler will supply the defaults instead.

   a. For example, Ada allows this. Consider our array sum procedure again. If our type Array_Type had bounds 1 and 10, and we normally expected the user to want to sum an entire array, then we could code our heading with defaults this way:

   PROJECT

```
procedure Sum_Array(Arr: Array_Type;
                    Lo_Limit: Integer := 1;
                    Hi_Limit: Integer := 10;
                    Sum: out Float)
```

   b. Now, the following calls (among others) would be legal, along with the ones we used before where the user explicitly specified the limits:

```
Sum_Array(X, Lo_Limit => 3, Sum => X_Sum); --Hi_Limit defaults to 10
Sum_Array(Arr => X, Sum => X_Sum);         --Lo_Limit defaults to 1,
                                           --Hi_Limit defaults to 10
```

   (Note: Ada requires that any parameter appearing after an omitted default parameter must be specified by name, not positionally)

c. Actually, our procedure would be easier to use if we listed the defaulted parameters last.

i. Our heading would now be:

```
procedure Sum_Array(Arr: Array_Type;
                    Sum: out Float)
                    Lo_Limit: Integer := 1;
                    Hi_Limit: Integer := 10)
```

ii. A possible call would now be:

```
Sum_Array(X, X_Sum);
```

3. More rarely, languages provide a syntax whereby a procedure can have ANY NUMBER of parameters.

a. Certain standard procedures in FORTRAN - such as MAX and MIN - can take any number of arguments. Again, however, this is ability is restricted to intrinsic functions known to the compiler.

b. COMMON LISP has an &rest keyword that allows arbitrarily-long parameter lists.

Example: PROJECT

```
(defun max (best &rest others)
    (mapcar (lambda (x)
                (if (> x best) (setq best x))
            )
            others
    )
    best
)
```

This could handle calls like:

```
(max 2 3)                -- returns 3
(max 2 4 1 3 8 5 6)      -- returns 8
(max 1)                  -- returns 1
```

etc.

BUT NOT:

```
(max)
```

i. The first parameter of the actual call is bound to best. The remaining parameters (if any) are put into a list and bound to the &rest parameter others.

ii. The mapcar construct iterates down the list others, comparing each element to best. If it is bigger than best, then it replaces best.

iii. Ultimately, the function returns the value of best.

c. Both ANSI C and C++ allow an ellipsis (...) to occur in a function prototype (possibly after one or more fixed arguments) to indicate that the function may be called with an arbitrary number of arguments (of arbitrary type.) This is used in conjunction with a stdarg facility to allow access to the individual arguments in the function implementation. In contrast to the facility found in LISP, there is no way to determine the NUMBER of arguments actually passed - the argument values must specify this in some way.

   Example - you cannot write the following function in C/C++:

```
int max(...)
/* Returns the maximum of an arbitrary number of ints */
```

   What you would have to write is something like this:

   PROJECT

```
#include <stdarg.h>

/* Returns the maximum of n integers (n >= 1) */
int max(int n, ...)
{
    int best, i;
    va_list ap;

    if (n == 0)
    {
        perror("Cannot find the maximum of 0 items");
        return 0;
    }
    va_start(ap, n);
    best = va_arg(ap, int);
    for (i = 1; i < n; i ++)
    {
        int next = va_arg(ap, int);
        if (next > best) best = next;
    }
    va_end(ap);
    return best;
}
```

   Which you could call by calls like:

```
max(2, x, y)
max(7, a, b, c, d, e, f, g)
```

   (max(0) would also be syntactically legal, but would print an error message)

d. Since version 1.5, Java has included provisions for what the language specification calls variable arity methods. A variable arity method has an ellipsis (...) after its final formal parameter.

i. This parameter is considered by the compiler to be an ARRAY, of its declared type, though it is not declared as an array. (If the formal is an array, then the compiler regards it as an array of arrays.)

ii. The method can be called with 0 or more actual parameters corresponding to this formal The compiler generates code to construct an array of the appropriate length.

iii. It is _not_ necessary for the caller to specify the number of arguments; the length of the constructed array that the formal parameter refers to is equal to the number of actual arguments placed in it.

The following is the Java equivalent of the above:

PROJECT

```
int max(int args ...)
{
    int best, i;

    if (args.length == 0)
        throw new IllegalArgumentException(
            "Can't take max of 0 items");

    best = args[0];
    for (i = 1; i < args.length; i ++)
        if (args[i] > best)
            best = args[i];
    return best;
}
```

Which you could call by calls like:

```
max(x, y)
max(a, b, c, d, e, f, g)
```

(max() would also be syntactically legal, but would print an error message)

e. Ada, however, has no such facility.

V. Routines as First Class Data Types In Imperative Languages

A. We have already introduced, in conjunction with the study of functional languages, the notion of functions as a first class data type - i.e. a function can be stored in a variable, used as an actual parameter to a routine, or returned by another function. Actually, some imperative and OO languages also provide some of these capabilities.

1. Example: the following Pascal function will approximate the derivative of any real-valued function at a specified point:

   PROJECT

```
function derivative(function f(a: real): real;
                    x: real): real;
const
    epsilon = 1.0E-4;
begin
    derivative := (f(x + epsilon) - f(x))/epsilon
end;
```

   The following would be legal calls to this function:

```
writeln(derivative(sin, 0.0));
writeln(derivative(sin, 1.57));
writeln(derivative(cos, 0.5));
...
```

2. Similar facilities exist in FORTRAN, C, and MODULA-2 - but (curiously) not in Ada - though Ada can achieve the same effect (in a much more cumbersome way) by the use of generics.

3. OO Languages like Java achieve the same effect by the use of interfaces and objects (cf the notion of an ActionListener), or by the use of delegates as in C#.

4. There is an implementation complexity that arises with routine parameters in block structured languages.

a. Suppose we have the following in Pascal: PROJECT

```
procedure p(procedure q);
    var
        i: integer;
    begin
        i := 2;
        q
    end;

procedure r;

    var
        i: integer;

    procedure s;
        begin
            writeln(i)
        end;

    begin
        i := 1;
        p(s)
    end;

begin (* main *)

    r

end.
```

b. What should s write when it is called by p? (ASK)

i. The answer is that s must write the value of i which is lexically visible at the point it is declared - i.e. the value of i that belongs to r (since s is contained in r) - i.e. 1.

ii. This implies that it is not sufficient, when passing a procedure as a parameter to another procedure - to simply pass the address of the procedures code. The call must also pass the address of a LEXICAL ENVIRONMENT for the procedure - i.e. the location (on the stack) of the stack frame for the procedure that contains it.

iii. This is, of course, not an issue in non-block-structured languages like FORTRAN or even C (since functions cannot be defined inside other functions.)

iv. The complexity of doing this may be one reason why Ada has not included this facility (though it has included a lot of things that are much more complex to implement!)

B. Some languages (e.g. C, Modula-2) carry this idea further and allow one to declare VARIABLES whose value is a routine, and also to RETURN routines as the result of functions - by allowing the declaration of a data type that is actually a routine.

1. For example, the following is possible in Modula-2: PROJECT

```
MODULE DEMO;
FROM InOut IMPORT WriteReal, WriteLn;
FROM MathematicsProcedures IMPORT MTH$SIN, MTH$COS, MTH$TAN;

TYPE
    RealProc = PROCEDURE(REAL): REAL;
VAR
    Funcs: ARRAY[1..3] OF RealProc;
    I: CARDINAL;

PROCEDURE Derivative(F: RealProc; X: REAL): REAL;
    CONST
        epsilon = 1.0E-4;
    BEGIN
        RETURN (F(X + epsilon) - F(X))/epsilon
    END Derivative;

PROCEDURE FunctionReturn(F: RealProc): RealProc;
    (* Just to illustrate the possibility *)
    BEGIN
        RETURN F;
    END FunctionReturn;

BEGIN
    Funcs[1] := MTH$SIN;
    Funcs[2] := MTH$COS;
    Funcs[3] := MTH$TAN;
    FOR I := 1 TO 3 DO
        WriteReal(Derivative(Funcs[I], 0.7535), 10);
        WriteLn
    END
END DEMO.
```

2. C provides a similar facility through the data type "pointer to function" - e.g. PROJECT

```
typedef int (*fp)(int, int);       /* fp is a type-pointer to function
                                      of two ints, returning an int */

int add(int i, int j)              /* add is a function of this type */
{
    return i + j;
}

fp f = add;                        /* f is a variable of type fp, whose
                                      value is add() */

fp foo()                           /* foo is a function returning fp */
{
   return add;                     /* It returns add() */
}

int main(int argc, char ** argv)
{
   printf("%d\n", f(3, 4));        /* Will print 7 */
   printf("%d\n", foo() (1, 2));   /* Will print 3 */
}
```

3. However, this facility is relatively rare. One reason is that it involves a major subtlety when used in a block-structured language.

   a. Suppose the following were legal in Modula-2 (it isn't). (Note: the type PROC is a built-in type for "procedure with no parameters".)

```
PROJECT

MODULE DEMO;
FROM InOut IMPORT WriteCard, WriteLn;

VAR
    P: PROC;

PROCEDURE Outer;

    VAR
        I: CARDINAL;

    PROCEDURE Inner;
      BEGIN
        WriteCard(I, 10); WriteLn
      END Inner;

    BEGIN
        I := 1;
        Inner;
        P := Inner
    END Outer;

BEGIN

    Outer;
    P

END DEMO.
```

   b. It is clear what the call to Inner from Outer should print - the value of I, found in Outer, which is 1.

   c. But what should happen when the main program calls P, given that P was assigned the value "Inner" by Outer?

      i. Obviously, Inner should still print the value of I defined in Outer.

      ii. Unfortunately, though, Outer has already terminated, and its local variables have been destroyed. So there is no variable I still alive for Inner to print.

   d. It is for this reason that Modula requires that only GLOBAL procedure names can be stored in procedure variables. Hence, the assignment P := Inner is illegal.

   e. Note that C does not have this kind of problem, because C does not allow function definitions nested in blocks.

f. Something to think about: can a similar problem arise with PARAMETERS of procedure type in a block structured language (as allowed in both Pascal and Modula)? If so, what is it; if not, why not?

(Answer: No - when a procedure passes a procedure to another procedure, the called procedure must complete before the calling procedure completes, so the lexical environment must exist as long as there is any possibility of using the procedure that was passed.)

VI. Coroutines

A. Ordinarily, when we think of segmenting a program into procedures, we think HIERARCHICALLY.

1. Example: If A calls B, we have

```
 ___
| A |
|___|
  |
__|__
| B |
|___|
```

a. A's execution pauses to let B begin.

b. B's local data is created from scratch.

c. B runs to completion.

d. B's local data is destroyed.

e. A resumes where it left off.

2. If A calls B again, the whole process is repeated.

a. The first execution of B has no effect on the second (except possibly through global data).

b. B always starts execution from the very beginning, and runs through to the end. There is no way for B to "pause in mid-stream" and return control to A, picking up later where it left off.

B. This hierarchical pattern is just what we want for most applications. However, there are some problems for which it doesn't work well, because it is just not possible to establish a clear-cut hierarchy. A good example of this sort of problem is the classic problem of the producer and the consumer.

1. In one version of this problem, a data copying program is to be written that reads data from punched cards and writes it to a disk file, subject to the following formatting requirements. (Of course, the use of punched cards for input is from the early days of computing; but the problem issues remain valid and this example has become traditional.)

a. Each punched card contains exactly 80 characters.

b. A CR/LF sequence is to be appended to the data from each card before writing it to the disk.

c. Data is to be written to disk in blocks of 512 characters. The last block may need to be padded with null characters if it is not completely filled by data from the cards.

2. We can formulate this problem in terms of two subtasks, called the producer and the consumer, where the consumer "produces" characters by reading them from cards or manufacturing them, and the consumer "consumes" characters by writing them to the disk.

   a. The producer executes the following algorithm: PROJECT

```
do
    read a card into card_buffer[80];
    for (int = 0; i < 80; i ++)
        send card_buffer[i] to the consumer;
    send CR to the consumer;
    send LF to the consumer
while (! out of cards);
```

   b. The consumer executes the following algorithm:

```
do
    for (int j = 0; j< 512; j ++)
        if there is another character available then
            accept a character from producer into disk_buffer[j]
        else
            disk_buffer[j] = NULL;
    write disk_buffer[] to disk
while (more characters are available)
```

3. While each of the above is nice and modular, we run into serious problems when we try to put these two algorithms together in a standard hierarchical way.

   a. We could make producer call consumer each time it produces a character.

      i. In this case, producer would call consumer from 3 different places (the three lines that say send --- to consumer.)

      ii. But consumer would need to be entered in the middle of a loop!

      iii. Further, consumer would have to get control one last time after producer finishes to pad out and write the last block.

      iv. Consumer's local variables j and disk_buffer would have to be preserved between calls - either by using globals or (in a language like C) by declaring them static.

   b. Alternately, we could make consumer call producer each time it needs a character.

      i. Now consumer calls producer from just one point, in mid-loop.

ii. But producer needs to be capable of being entered at any of three places, depending on where it is in the processing of a card.

iii. In this case, producer's local variables i and card_buffer must be preserved between calls.

4. The following is one way to solve the problem hierarchically, with consumer "on top" calling producer. It uses an additional "state" variable to keep track of the three possible ways producer can generate characters:

```
PROJECT

bool no_more_characters;

char producer()
{
    static enum { READ_NEXT, CARD_NEXT, CR_NEXT, LF_NEXT }
        pState = READ_NEXT;
    static int i;
    static char card_buffer[80];

    switch(pState)
    {
        case READ_NEXT:
        {
            read a card into card_buffer[];
            char c = card_buffer[0];
            i = 1;
            pState = CARD_NEXT;
            return c;
        }
        case CARD_NEXT:
        {
            char c = card_buffer[i];
            i ++;
            if (i >= 80)
                pState = CR_NEXT;
        }
       case CR_NEXT:

            pState = LF_NEXT;
            return CR;

       case LF_NEXT:

           if there are no more cards available then
                no_more_characters = true;
           pState = READ_NEXT;
           return LF;
    }
}
```

```
void consumer()
{
    char disk_buffer[512];

    do
    {
        for (int j = 0; j< 512; j ++)
            if (no_more_characters)
                disk_buffer[j] = NULL;
            else
                disk_buffer[j] = producer();
            write disk_buffer[] to disk
    } while (! no_more_characters);
}

void main(int argc, char ** argv)
{
    no_more_characters = false;
    consumer();
}
```

5. This solution is far from elegant.

   a. We have transformed control flow information into data in the form of the variable pState plus a switch statement.

   b. The producer makes use of several static variables to hold state information. In non-C languages, these would typically have to be global.

C. To deal with cases like this (which do occur quite often in practice), some languages offer a special construct known as a COROUTINE.

1. In contrast to a procedure which is always executed from top to bottom, a coroutine can:

   a. Execute a resume statement at any point. This returns control to another coroutine, WITHOUT DESTROYING THE COROUTINE'S LOCAL STATE OR VARIABLE INFORMATION.

   b. Be resumed by another coroutine, picking up exactly where it left off.

2. The new feature being introduced is some form of RESUME statement, which yields control without destroying local state information the way return does.

3. Using coroutines, our problem could be coded as follows, using two coroutines and two global variables for communication between them. (Note: the code here doesn't correspond to any existing language. However, it could be translated into Modula-2 with little difficulty.)

   PROJECT

```
var
    C: char;
    no_more_characters: boolean;

coroutine producer;

    var
        i: integer;
        card_buffer: array[1..80] of char;

    begin
        no_more_characters := false;
        resume(main);
        repeat
            read a card into card_buffer[1..80];
            for i := 1 to 80 do
              begin
                c := card_buffer[i];
                resume(consumer)
              end;
            c := CR;
            resume(consumer);
            c := LF;
            resume(consumer);
        until there are no more cards available;
        no_more_characters := true
    end;

coroutine consumer;

    var
        j: integer;
        disk_buffer: array[1..512] of char;

    begin

        repeat
            for j := 1 to 512 do
                if no_more_characters then
                    disk_buffer[j] := NULL
                else
                  begin
                    resume(producer);
                    disk_buffer[j] := c
                  end;
            write disk_buffer[1..512] to disk
        until no_more_characters

    end;

begin
    start(producer);
    start(consumer)
end.
```

D. Coroutine facilities are relatively rare, though they have been present in an extended version of Algol, Simula, and Modula-2.

1. One reason why coroutine facilities are relatively rare is that they are complicated to implement.

2. The resume instruction itself is fairly easy to implement, as follows:

   a. First, the current coroutine pushes its own next instruction address onto its own stack.

   b. Then, a switch is made to the stack of the new coroutine.

   c. At this point, an address is popped from the stack of the new coroutine, and control is transferred to it.

   d. That is, the resume combines features of BOTH routine call AND routine return.

3. The real difficulty lies in storage management. Ordinary procedures obey a LIFO discipline; thus, local storage for them can be managed by a single run time stack. But this won't work for coroutines; instead, each coroutine needs ITS OWN stack.

   a. This poses a problem with regard to stack growth, since each stack has neighbors that limit how big it can get.

   b. Some coroutine implementations require that each coroutine pre-declare the maximum stack space it will need, and this amount of space is set aside for its use. This can be dangerous: an underestimate can lead to one coroutine corrupting another's stack.

   c. Other implementations let the current coroutine use the regular system stack, but copy its stack to a holding area when it resumes another coroutine. This can make the resume operation very time-consuming.

E. An alternative way of solving this sort of problem is the use of threads or full concurrency.

1. A fundamental difference between threads and coroutines is that with coroutines

   a. Exactly one coroutine runs at any time. (This is also true of threads on a uniprocessor, but on a multi-core processor it is conceivable that two threads might literally run at the same time.)

   b. Coroutines explicitly yield control to each other, whereas threads may switch at any time. As a result, the kinds of synchronization issues that happen with threads don't arise as much with coroutines, because control transfer is predictable.

2. Actually, a uniprocessor implementation of threads may actually implement them as coroutines, but with some mechanism for forcing transfer of control rather than doing this explicitly.

3. In a few days, we will study another language feature known as TASKING. On a computer system with multiple CPU's, this is even more powerful than coroutines, though on a one CPU system, it may, in fact, be implemented like coroutines.

F. A variant of the coroutine known as a GENERATOR is also a key feature of Icon, and something like it can be done in Prolog.

Example: (Demo)

```
natural(1).
natural(N) :- Natural(X), N is X + 1.

natural(N) - keep entering ; to generate successive values
```