CSE305 Week 10: Subprograms (Sebesta chs. 9--10)

First, a conceptual distinction apart from the terms various PLs call subprograms:

- A *pure function* returns a value but does not change any program state.
- A *procedure* does not return a value, and so either changes program state or is a no-op.

Functions in the core part of OCaml---in particular not involving **`ref`** or arrays---are pure. The initial versions of Ada also required a `function` to be pure. Void "functions" in C/C++/Java are really procedures, since **`void`** is not a return value (unlike `unit ()` in OCaml). Procedures in Ada, Modula-2, Pascal, PL/1, etc. also do not return values. The other possibilitiy is:

- A subprogram that "walks and chews gum at the same time" by both returning a value and altering program state. Can be called a "function with side effects."

This is the chimera that "pure" languages try most to eliminate. But not all such code is to be panned. Perhaps the simplest impure function is the post-increment operator in code like:

```
item = *itr++;
```

Here `itr` is an **iterator** over a data collection, basically a "pointer object." In C++ the **`*`** and (pre or post) **`++`** operators can be overloaded to make it use the same interface as a native pointer. Or you can code a named function like **`itr.next()`** in Java, which does the same thing: it returns the current object and advances **`itr`** to the next object. Either way, this code is an enduring idiom.

Functions "Versus" Methods

A **method** is formally just a subprogram whose first parameter has privileged access to its own fields and class scope. (But that parameter doesn't know its own name, calling itself "**`this`**" or "**`self`**" or etc.) There are also syntactic differences when (pointers to) methods are passed as arguments. Beyond those surface differences, we can regard methods the same as other subprograms for the concepts in these chapters.

- A purely functional method is somewhere between a "getter" (which strictly speaking just returns the current value of a class field) and a "const method" (which cannot change the invoking object---just the same as if it were a first parameter marked **`const`**---but could change other stuff).
- A purely procedural method can be much more than a "setter": it can read from the invoking object's state before changing it, and/or change other parameters or visible non-local variables, and/or create new objects on the system heap, etc.

Speaking very roughly ("IMPHO"), methods are more apt than other subprograms to do both a change of state and give a value. With these differences digested and put on hold, we can focus on the single idea of a **subprogram**.

Basic Definitions (modified from Sebesta 9.2)

- A **subprogram definition** describes the interface to and the actions of the subprogram abstraction.
- A **subprogram declaration** provides the protocol, but not the body, of the subprogram. Also called a "header" or prototype (in C/C++ especially)
- In C and C++, but not Java or Python or Javascript, a function declaration can be given separately in main code. (They can always be put in interfaces, likewise with signatures in OCaml modules.)
- Besides giving names, the header is basically the *type* of the function, also called the **signature**, so long as you consider the parameter passing modes part of the type. If all parameters are passed by value, as in OCaml, then it is just the type.
- The order of parameters matters to the signature, unless they are *all* **keyword parameters**.
- The notion of protocol or function type is further complicated by the mechanism of allowing variable numbers of arguments and/or default values to some parameters.

In statically compiled languages, by-and-large, all subprogram prototypes must be known at compile time. Whereas in Python, for instance, definitions are ordinary executable code and can be contingent on branching, e.g.:

```
if cond:
   def fun1 (...) :
      ...
else:
   def fun2 (...) :
      ...
```

Well, unless `fun1` and `fun2` have the same name (as the text givens it) and the same protocol, the code after the if-else statement could give an undefined function error.

- A **subprogram call** is an explicit request that the subprogram be executed. It can be a separate statement, or---in the case of a function---can be in the middle of an expression.
- A **formal parameter** is a dummy variable listed in the subprogram header and used in the subprogram.
- An **actual parameter** represents a value or address used in the subprogram call statement. I agree with the "some authors" who prefer the term **argument**. The arguments can be given as expressions, even ones like an array entry **`arr[i]`** that can produce an Lvalue.

Positional and Keyword Parameters, With and Without Defaults (9.2.3)

With **positional parameters**, the binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth.

- **Advantage**: Safe (easily type-checked) and effective.
- **Disadvantage**: `void what_if(int there, float are, vector<int> zillions, char of, set<T> parameters, int whose, string order, int is, double hard, int to, string remember);`

With **keyword parameters**, the name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter---it must be used in the subprogram call.

- **Advantage**: Parameters can appear in any order, thereby avoiding parameter correspondence errors.
- **Disadvantage**: User must know the formal parameter's names. These can be hard to remember exactly too.

When positional and keyword parameters are allowed to be mixed, there is generally a rule saying all the positional parameters come first. A similar rule is used when parameters may have default values.

Command-line parameters are an example of treating **`main`** as an ordinary subprogram. In C, C++, Java, and Javascript there is formally just one argument vector, but it can have arbitrary length. It is possible to store "flags" in the vector and look them up in a way that simulates keyword parameters. Python and Perl and other scripting languages have basically brought this named-argument technique to core language level.

[Show example from chess scripts. Moral is that all three features: keyword arguments (often the Python abbreviation "kwargs" is used for the concept), variable number of arguments (the Java syntax "`...`" for that is often called "varargs"), and default values can combine to great effect.]

(Other) Design Issues For Subprograms (9.3, including short section 9.4)

Sebesta's laundry list, showing ones I'll emphasize:

1. Are local variables static or dynamic?
2. Can subprogram definitions appear in other subprogram definitions?
3. **What parameter passing methods are provided?**
4. **Are parameter types checked?**
5. **If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?**
6. **Are functional side effects allowed?**
7. What types of values can be returned from functions?
8. How many values can be returned from functions?
9. **Can subprograms be overloaded?**
10. Can subprograms be generic?
11. If the language allows nested subprograms, are **closures** supported?

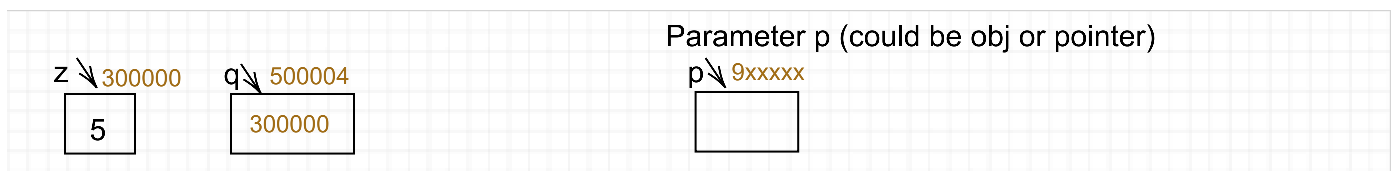
From the "modern" perspective it is IMPHO possible to give short shrift to some of these:

1. Generally stack-dynamic, not static.
2. Yes, even in scripting languages.
3. **Treated at length below.**
4. Often yes, even if the types are **inferred**, but Python is an example of a language that is weaker in this regard.
5. **Treated at length below**, after covering section 10.3.
6. **Themed at the beginning above.**
7. Whether functions can be returned is another running theme. If so, then basically any type; **void** is not a return type.
8. Formally one value, but it can be a collection object. The illusion of returning multiple values can come from having reference parameters and/or write-only (**out** mode in Ada) parameters and/or other side effects.
9. **Yes-and-no**, deferred to when we consider methods in OOP.
10. Often yes, also deferred to OOP.
11. May be picked up at the end in connection with chapter 15.

We have already largely covered questions 1 and 2 in section 9.4 when considering nested scopes in terms of chapter 5. Section 9.6 will raise a much bigger aspect of this. So let's move on to section 9.5.

Parameter Passing (9.5)

Suppose we have a subprogram called `bar`. Let it have one parameter `p` of a general type `T` (or it could be a pointer of type `T*`). Focus for now on one storage object that is involved in the call. Call it **z** ↘. And to cover all bases, let's suppose that there is an explicit pointer `q` that points at `z`, in other words, has **z**↘ as its value. The picture so far looks like:



The exact binding address of the storage object for the parameter will depend on where its stack frame is allocated, but let's say it's a number beginning with 9. Here are three main ways of treating the **mode** of the parameter---in C++ terms:

1. **Call By Value**. C/C++/Java syntax: `bar(T p)`. Just the value held by z is copied into p.
2. **Call By Reference**. C++ syntax: `bar(T& p)`.
3. **Call By Pointer**. C/C++ syntax: `bar(T* p)` or `bar(T *p)`.

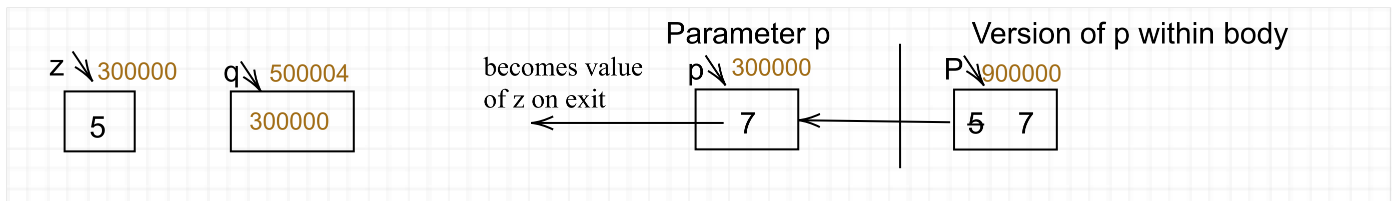
In call-by-reference, just the address **z**↘ is given via `p`. It is not the value "of" p but rather supersedes the 9xxxxx binding that would be given to p, so that p becomes an **alias** of z. Any change made in the subprogram body to `p` is immediately reflected in `z`. In call-by-pointer, the address **z**↘ does become the value of `p`. Looking up the value requires an extra **`fetch`** compared to when p is an alias or when the value is directly given to `p`. Three other modes merit special mention right away:

- **Call By Constant Reference**. C++ syntax: `bar(const T& p)`.
- **Call By Value-Return** (Ada): `bar(p: in out T)`.
- **Call-By-Name** (Algol-60): The subprogram code is run as if it used the name "`z`" rather than "`p`". Dicey when the actual argument is an expression, in particular an array item `arr[i]`.

Ada reflected mathematical purity of the notion of how values go in and come out:

- Just read the value v of z: Ada **in** mode.
- Just write to p without reading what storage object is given to it: Ada **out** mode.
- Both: Ada **in out** mode, which is closest to how reference parameters work in C++.

The difference with call-by-value is that both the value of z and the address of z are passed but the latter is kept walled off from the subprogram body. The following diagram may look more involved than ways this was actually implemented, but the extra copying depicted is genuine:

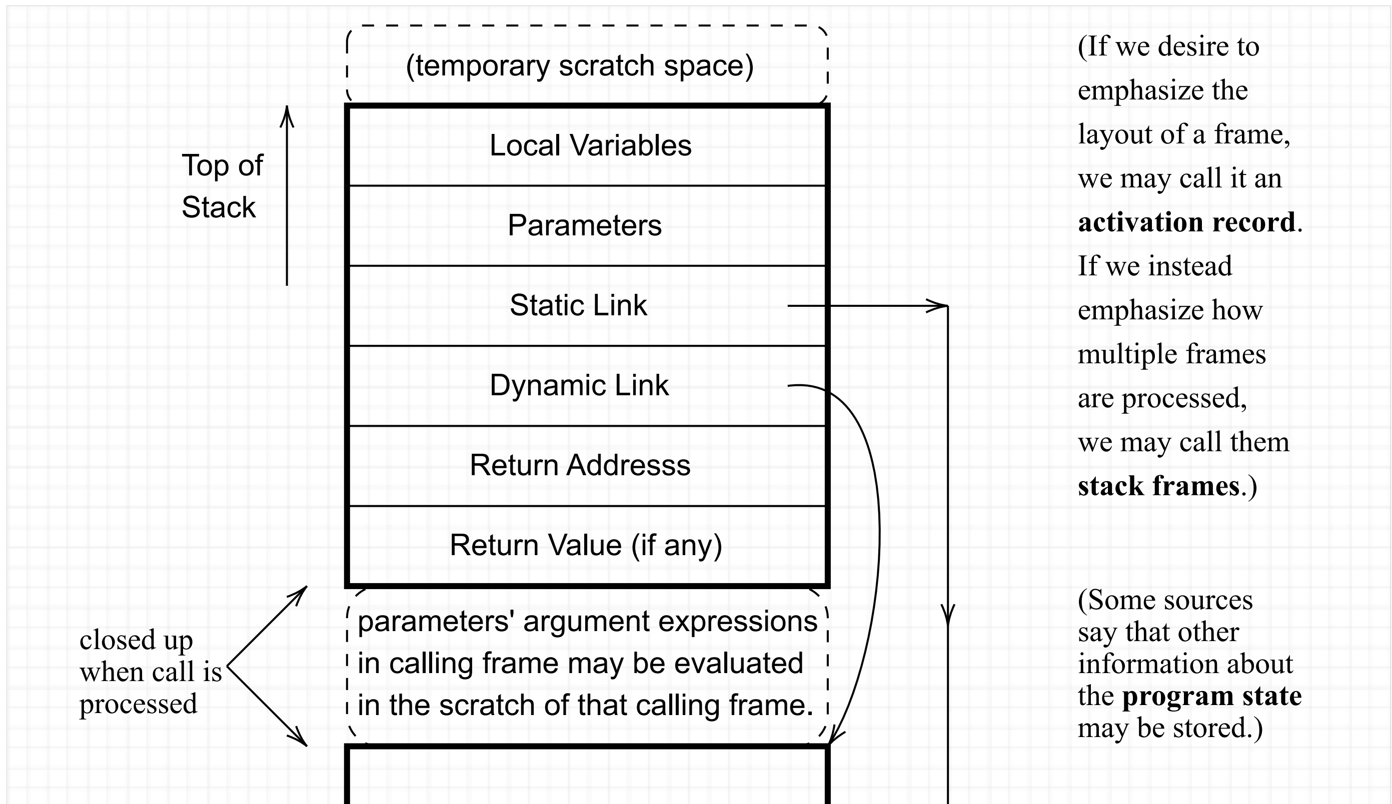


We will say more about differences among these modes after seeing more details about stack frames.

How Subprograms Work (10.1--10.3 after 9.1--9.4)

A subprogram has two footprints in the memory used for execution. Neither affects any part of the system heap.

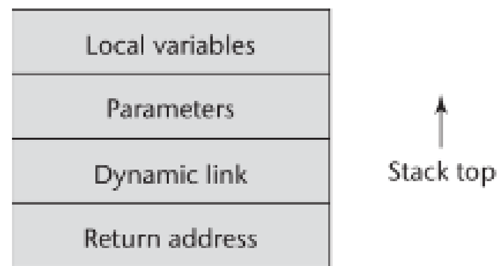
- The executable code belonging to the subprogram.
- Its **activation frame**, which in general contains the following information, not necessarily in the order shown:



1. The **return address** is the IC value of the instruction that will be executed immediately after the call is finished. (It is in the stored compiled obejct-or-machine code; it can be within a source line of code.)
2. The **dynamic link** goes to the calling frame, which is immediately underneath. It is also called the **control link**.
3. The **static link** goes to the activation frame of (*the most recent invocation of*, in case of recursion) the nearest lexically enclosing scope of the subprogram's source code.

Under **static scoping** of references in the subprogram body to a *non*-local variable **x**, the chain of static links is followed until a frame is found where **x** is declared, and its value is read there. This is faithful to the **local referencing environment** of the subprogram's text. Under **dynamic scoping**, the chain of dynamic links is followed instead. Because static scoping is widely presumed, the static link is also what's usually meant by the term **access link**.

When there are no non-local variables, and when we either talk about a pure procedure or have a regime where the return value is just the value left by the last instruction---which will appear naturally as the item left on the stack when the processing of the frame is done, we get Sebesta's simpler diagram in section 10.3:



The diagram scheme also presumes that the arguments given in a subprogram call---which can be expressions, which can even involve other function calls---are evaluated down to values before the stack frame begins to be processed. Then their values are given when the storage objects for the parameters are created. Those and the initializations of the subprogram's local variables go atop the stack frame---intuitively because their values will be worked on next as the subprogram body is processed. This is called **strict evaluation** of parameters in the text; I prefer the term **eager evaluation** for a better contrast with **lazy evaluation**. (This used to be here but now Sebesta leaves this to the functional languages chapter---"lazy" applies mainly there but "eager" is quote general!)

Assuming strict evaluation, when a subprogram is called, the following happens:

1. The expressions passed as arguments are evaluated down to values. (In case of a reference parameter, such as `var` in Scala or `&` syntax on the parameter in C/C++, the reault must be an Lvalue; in case of a pointer parameter, the "value" is the binding-address value of the pointer.)
2. A lesser issue is, in case of multiple parameters, in which order are their arguments evaluated? [Example `EvalOrder.cpp` versus `EvalOrder.java` shows a difference.]
3. The stack frame is **allocated** on the system stack, directly atop the calling frame. It is possible that the subprogram's code may be "**just-in-time compiled**" at this moment.
4. The IC is loaded with the beginning address of the frame's code, so that execution begins. The address to return to is usually made part of the frame, rather than say the IC has its own stack.
5. Processing begins on that frame---which may allocate further frames, especially in case the subprogram calls itself recursively.
6. On exit of the subprogram, the frame is **deallocated**:
   (a) The storage objects for the parameters and local variables of the frame *cease to exist*.
   (b) Any changes made to non-local variables or via the Lvalues of reference parameters have already occurred.
   (c) Likewise any creation of objects on the system heap or changes to them have already happened.
   (d) The IC is given the return address and the return value (if any) is propagated to the calling frame, whose processing then resumes.
7. But if the subprogram throws an exception `Foo` that is not handled within the subprogram's own code, stacked frames are popped destructively and mercilessly until an activation record with a handler for that exception is found. If none is found, the frame for `main` is popped too and the program exits with an error. (Sebesta puts more detail in chapter 14.)

Some further observations:

- The subprogram has a *single point of entry* and a *single destination of exit*.
- The return value is formally regarded as a single value. It can of course be a compound object, such as an array or list or tuple, but it is a single item.

The grand moral is:

> **Execution follows a stack discipline throughout: from evaluating expressions to executing loops to the macro-scale of processing subprogram calls.**

Two Examples

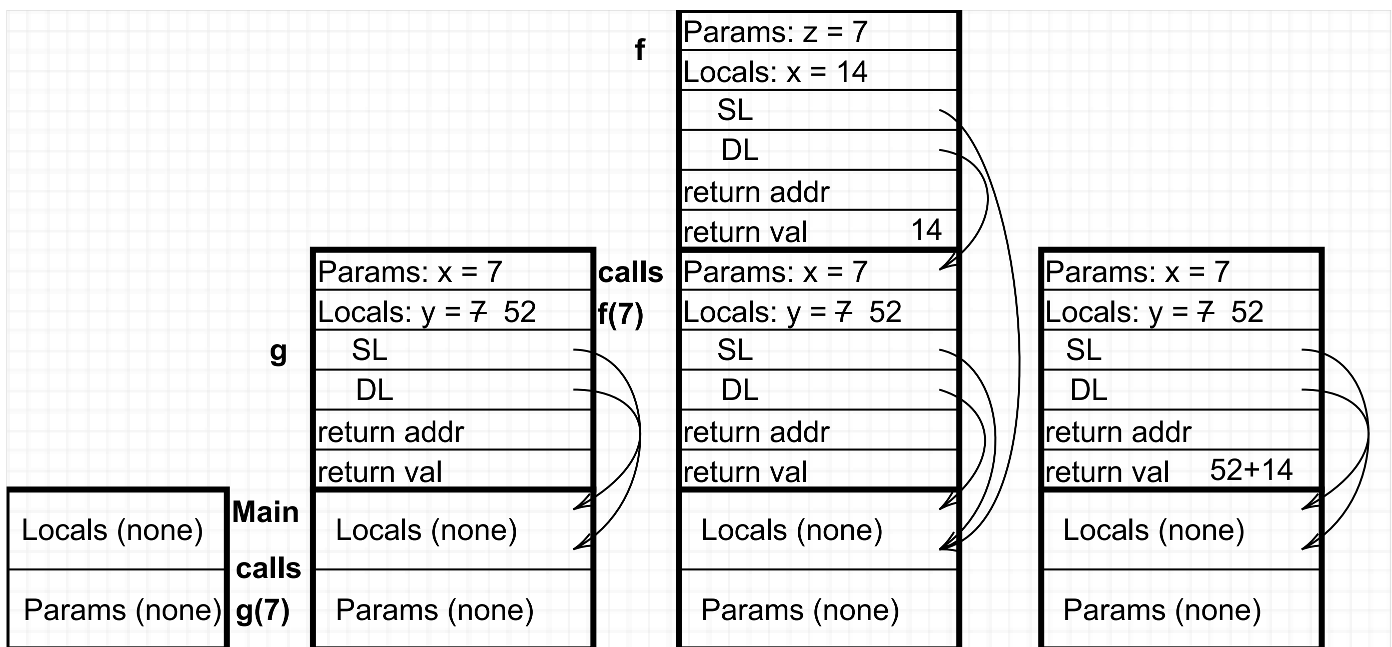
The example in Q18 of the TopHat part of HW3 can be traced more fully as follows:

```
1   #include<stdio.h>
2
3   int f(int z) {
4      int x = 2*z;
5      return x;
6   }
7   int g(int x) {
8      int y = x;
9      for (int x = 0; x < 10; x++) {
10        y += x;
11     }
12     return y + f(x);
13  }
14  int main() {
15      printf("g(7) = %d\n", g(7));
16  }
```

```
object CSE305HW3 extends App {

   def f(z: Int): Int = {
      val x = 2*z
      return x
   }
   def g(x: Int): Int = {
      var y = x
      for (x <- 0 until 10) {
         y += x
      }
      return y + f(x)
   }

   println("g(7) = " + g(7))
}
```

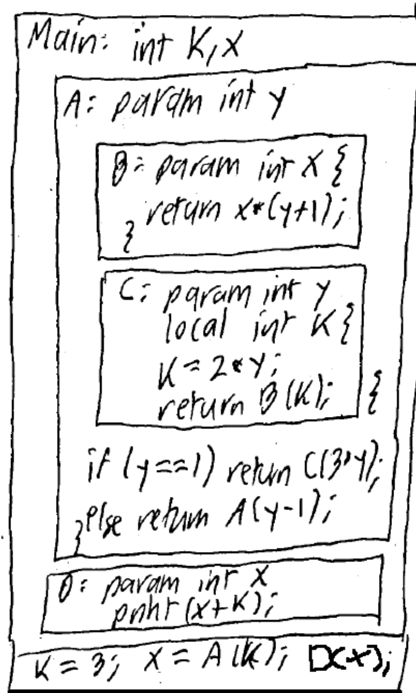


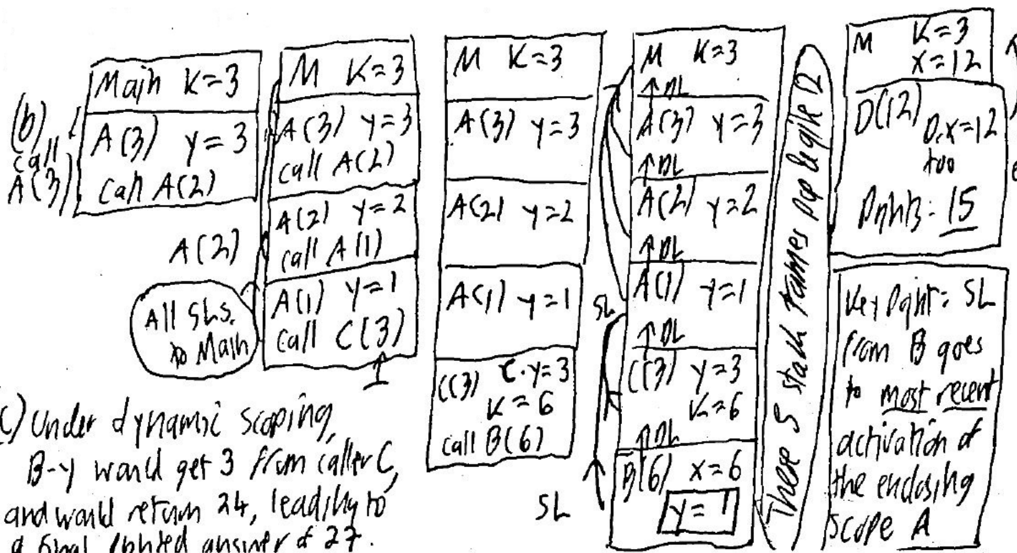
[It was remarked that "`int main()`" should strictly be "`int main(int argc, char** argv)`" and that since those can be referenced inside `main`, they should count as parameters of `main`. Fair point---and something similar is true of the Scala code as well (it was actually used in my Spr'22 CSE250). The fact that the C code as given still compiles on our machines is a laxity of the compilers.]

A more-complicated example:

(3) The following program is written in a mythical language "A" that preceded Ken Thompson's "B," which preceded C.

```
PROC MAIN;
    k,x: int;
    FUNC A(y: int): int;
        FUNC B(x: int): int;
        Begin
            RETURN(x*(y+1));
        End B;
        FUNC C(y:int): int;
            k: int;
        Begin
            k := 2*y;
            RETURN(B(k));
        End C;
    /* Main body of A begins here */
    Begin
        if y = 1 then RETURN C(3*y);
                 else RETURN A(y-1);
        endif;
    End A;
    PROC D(x: int);
    Begin
        Print("Answer is: %d\n",x+k);
    End D;
/* Body of MAIN begins here. */
Begin
    k := 3;
    x := A(k);
    D(x);
End MAIN;
```





[See remarks about Tail Recursion that were inserted into lecture at this time, at bottom.]

Here, incidentally, is the table of references to variables given by tracing the static links:

Var	Main	A	B	C	D
x	M.x	M.x	B.x	M.x	D.x
y	undef	A.y	A.y	C.y	undef!
K	M.K	M.K	M.K	C.K	M.K

The fact of the static link going to the most recent invocation is consonant with why the big numbers "777" and "888" were not involved in the recursive computation on Prelim I:

```
let rec f ell (x,y) = match ell with
   | [] - > x + y
   | [x] - > x + y
   | x::y::rest - > x + f(y:;rest) (1,2)

f[5;4;3] (777,888)
         --> 5 + f[4;3] (1,2)
         --> 5 + 4 + f[3] (1,2)
         --> 5 + 4 + (x+y) where x = C.x = 3 but y = A.y = 2
```

Here, however, there is no need to trace links: the "`y = 2`" is already in the stack frame for the call `f[3] (1,2)`.

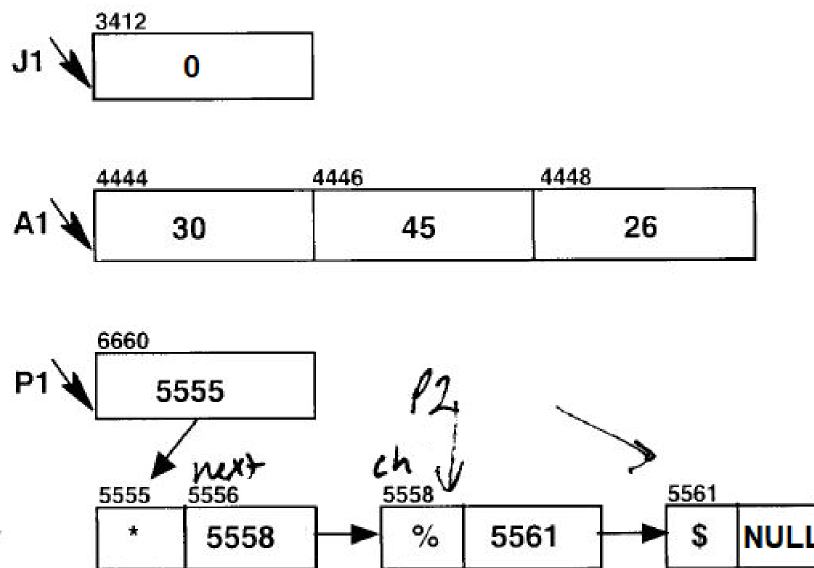
Parameter Passing Example

Here is an example using call by value and call by reference in C++ notation, followed by imaging how it would work under call-by-name. It has an array and a linked list made via a `Node` class with fields `char ch` and `Node* next`.

```
void testVal(int J2, int A2, Node* P2) {
   cout ≪ "Incoming values: " ≪ J2 ≪ " " ≪ A2 ≪ " " ≪ P2->ch ≪ endl;
   J2++;
   P2 = P2->next;
   cout ≪ "Outgoing values: " ≪ J2 ≪ " " ≪ A2 ≪ " " ≪ P2->ch ≪ endl;
}
```

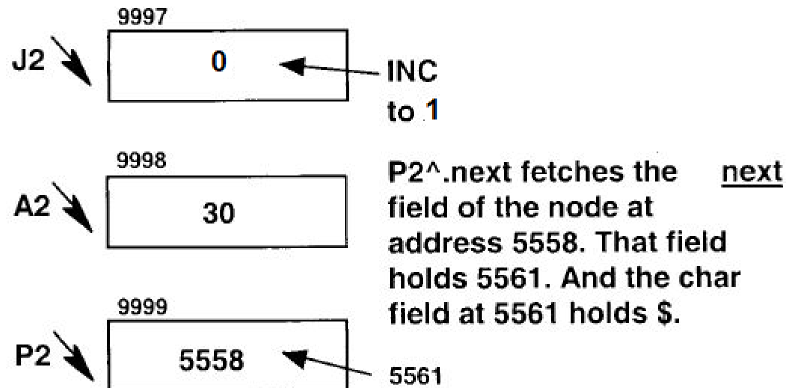
```
int main() {
   int J1 = 0;
   int A[3] = { 30, 45, 26 };
   int A1 = A[J1];
   Node* P1 = new Node('*', new Node('%', new Node('$', NULL)));
   cout << "Before Call: " << J1 << " " << A[J1] << " " << P1->next->ch << endl;
   test(J1,A1[J1],P1->next);
   cout << "After Call: " << J1 << " " << A[J1] << " " << P1->next->ch << endl;
   return 0;
}
```

The status before any call:



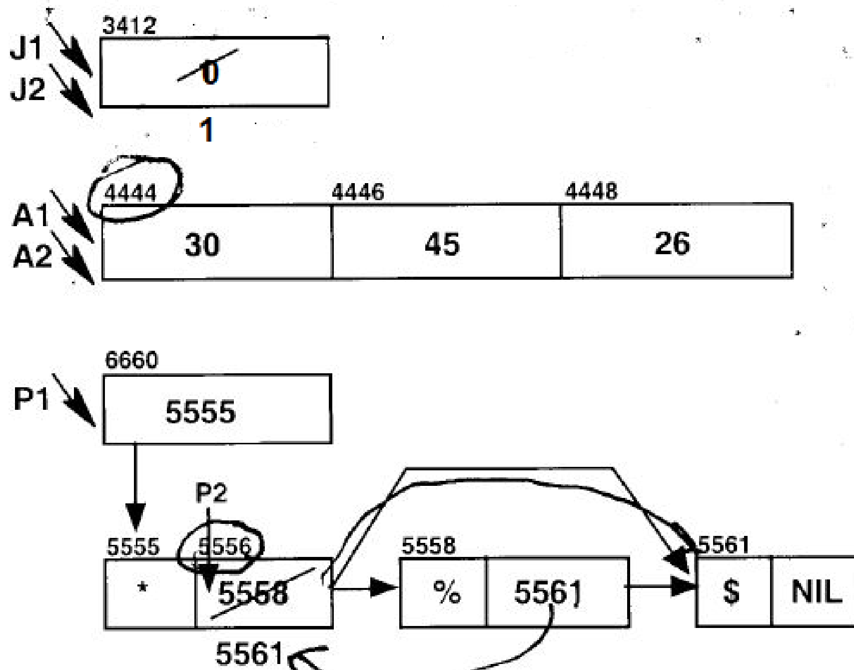
[Remark: During lecture I confused myself by thinking "`next`" was the arrow coming down from the separate storage object for the P1 reference. It is the second field of the first list `Node` with value 5558, meaning it is the arrow from that node to the node with the '%' character. So there is no off-by-one error and the subsequent diagrams are correct after all. (It was a brain blip, since I did after all change the array part from 1-based to 0-based.) Here is `testVal` again:]

```
void testVal(int J2, int A2, Node* P2) {
   cout << "Incoming values: " << J2 << " " << A2 << " " << P2->ch << endl;
   J2++;
   P2 = P2->next;
   cout << "Outgoing values: " << J2 << " " << A2 << " " << P2->ch << endl;
}
```



The Code With Call-By-Reference

```
void testRef(int& J2, int& A2, Node*& P2) {
   cout << "Incoming values: " << J2 << " " << A2 << " " << P2->ch << endl;
   J2++;
   P2 = P2->next;
   cout << "Outgoing values: " << J2 << " " << A2 << " " << P2->ch << endl;
}
```



Incoming Values:	0	30	%
Outgoing Values:	1	30	$
Objects After Call:	1	45	$

And the list got un-linked!

In this case, the **fetch** and **store** commands in the stack frame fetched from and stored to the original objects, not local copies.

The Unspeakable Call-By-Name

For the call `test(J1,A1[J1],P1->next);`

```
void testName(int J2, int A2, Node* P2) {
   cout << "Incoming values: " << J2 << " " << A2 << " " << P2->ch << endl;
   J2++;
   P2 = P2->next;
   cout << "Outgoing values: " << J2 << " " << A2 << " " << P2->ch << endl;
}
```

would compiled as if it were literally

```
void testName(int J1, int A1[J1], Node* P2) {
   cout << "Incoming values: " << J1 << " " << A1[J1] << " " << P1->next->ch <<
endl;
   J1++;
   P1->next = P1->next->next;
   cout << "Outgoing values: " << J1 << " " << A1[J1] << " " << P1->next->ch <<
endl;
}
```

What does Java do?

A common Java question is whether Java uses call-by-value or call-by-reference. The answer is: only the former, but for non-primitive types this becomes call-by-pointer, which can simulate call-by-reference! Using this, however, requires a sharp understanding of the difference between re-assigning a pointer and modifying the object it points to—an issue that Java's erasure of C++ *'s serves to blur. A full example:

```
public class CallByString {
  static void put(String s) { System.out.println(x); }
  public static void modify(StringBuffer p, String q,
                                    StringBuffer r) {
    p.setCharAt(0,' '); p.setCharAt(1,' ');
    q = "changed";
    r = new StringBuffer("changed");
  }
  public static void main(String args[]) {
    StringBuffer a = new StringBuffer("unchanged");
    String b = "unchanged";
    StringBuffer c = new StringBuffer("unchanged");
    put("Pre: a = " + a + ", b = " + b + ", and c = " + c);
    modify(a,b,c);
    put("now a = " + a + ", b = " + b + ", and c = " + c);
   }
}
Pre: a = unchanged, b = unchanged, and c = unchanged
now a =   changed, b = unchanged, and c = unchanged
```

Passing Subprograms As Parameters (9.6)

Pascal allows passing whole procedures or functions as parameters to other routines. C and C++ achieve a similar effect by passing pointers to functions. A typical example where one wants to do this is where you have an array A of Items with various fields, and want to (re)sort A with different sorting functions on various fields. In ANSI C/C++ one can write:

```
void arraySort (Item A[],
                bool (*lessThan)(Item, Item))
{  ...
   if ( (*lessThan) (A[i]. A[j]) ) ...
   ...
}
```

This is fine for a disciplined programmer, but as you might imagine, if the "`lessThan`" function does weird assignments to global variables etc., it could get even hairier to figure out what is going on! Section 9.6 covers the issue of whether to determine the **referencing environment** of `lessThan` as:

- where it was defined (buzzword: "deep binding");
- wherever `arraySort` calls it ("shallow binding");
- where `arraySort` itself is called ("ad hoc binding").

In any statically-scoped language, clearly only option 1 is consistent, and that could be all we need to say about it. But the mechanics are still important.

[The lecture ended here. I had intended to go to Sebesta's slides to touch on a little more about section 9.6 and then fly over the text's remaining points in chapters 9--10. I'd actually forgotten when preparing the lecture that I had missed 10-15 intended minutes on tail recursion the previous week. I actually spent about 5 minutes on tail recursion during the lecture itself when expanding on the "more-complicated example" of tracing activation frames, since it had two tail-calls without being full tail recursion. Since the following also harks back to an example that discussed tail recursion in week 5, I've expanded that into the following "officially covered" notes.]

Tail Recursion Notes to Insert Above:

A **tail call** (apart from recursion) is a line in a subprogram `A` of the form

```
return B(...)
```

where there is nothing else in the line outside the call to `B` but any stuff may be going on inside the `(...)`. (Or in case **`A`** is a pure procedure, its last line of code is a call to `B(...)`, nothing else.) The "more complicated example" of stack frames above has two such statements in **`A(y)`**, one returning **`C(3*y)`** and the other calling itself recursively as **`A(y-1)`**. The first effect of a tail call to a function is that the return value from `B` immediately becomes the return value from **`A`**, so that no further use of the stack frame for **`A`** is needed.

Note that "**`return A(y)-1`**" or "**`return sum + A(y)`**" is not a tail call, because there is other processing in the expression outside the call and its arguments. Put another way, `B(...)` is a tail call only if **`B`** is the root of the expression tree of what is being returned. In these other two cases, **`-`** and **`+`** are the root operations.

Among various levels of precision in definitions of tail recursion, I think this one works well enough in relevant cases:

Definition: Code **`A`** for a function is **tail recursive** if every branch of **`A`** either returns a value without calling any other subprogram, or is a tail call to **`A`**.

The impact when the tail call is to **`A`** itself is that *we don't have to allocate a new stack frome for* **`A`**. Because the calling frame will have no further action, and because it was a frame for **`A`** to begin with, ***we can re-use it***. Furthermore, because there will be no further calls to other functions, we can dispense with the idea of redoing frames entirely. A compiler can treat the whole body of **`A`** as a ***loop*** that exits when a non-recursive branch is taken. This saves the whole still-relatively-expensive overhead of allocating frames.

Now let us revisit the `countPeaks` example from toward the end of the week 5 notes with these considerations in mind.

```
let rec countPeaks ell = match ell with
     [] -> 0
   | [_] -> 0
   | [_; _] -> 0
   | x::y::z::rest -> if   y > x && y > z
                      then 1 + countPeaks(z::rest)
                      else countPeaks (y::z::rest);;
```

This has 3 nonrecursive branches from the first three match-cases and 2 recursive branches from the `if-then-else`. The `else` branch is already a tail call, but the `then` branch is not because of the "1 +" part. There is a standard code transformation that **hoists** the recursive call to `countPeaks` to be highest in that branch as well. It involves making the output value a second parameter that is **accumulated**, in what is called **accumulator passing style**:

```
let rec countPeaks (ell,n) = match ell with
     [] -> n   (* output not 0 but the running count n *)
   | [_] -> n
   | [_; _] -> n
   | x::y::z::rest -> if   y > x && y > z
                      then countPeaks(z::rest, n+1)
                      else countPeaks(y::z::rest, n);;
```

The base cases now change to output n. The new version has one more parameter, but works the same overall if you call it initially with `n=0`.

Now both branches of the `if-then-else` are tail calls. Thus, this version of `countPeaks` satisfies my definition of tail recursion above. There is, however, a "little fib" going on that is exposed when you follow how Assignment 4 said to process an `if-then-else` (or **? :**) expression under the Postfix conversion. This said to evaluate the condition and both the then and else expressions first, before finally processing the ternary (**? :**) node in the postorder traversal. The enormous downside of following these instructions literally is that we would execute a **double-branch recursion** in a case that is not **divide-and-conquer** and that here would cause an exponential explosion!

In reality, OCaml is able to give efficiency "as though" the `if-then-else` is really control where only one alternative is executed. That is not how it actually behaves, though. Instead, the compiler exploits a second "**hoist**" trick (one that, as far as I gather, is usually easier than determining the accumulator form) exemplified by the following version:

```
let rec countPeaks (ell,n) = match ell with
     [] -> n
   | [_] -> n
   | [_; _] -> n
   | x::y::z::rest -> countPeaks(if   y > x && y > z
                                 then (z::rest, n+1)
                                 else (y::z::rest, n));;
```

Now this complies with stricter definitions of tail recursion, even ones requiring there to be just one recursive branch. The `if-then-else` expression inside the (...) does get evaluated entirely, but because there are no recursive calls inside it, there is no explosion.

Many compilers actually allow further liberalizations of the definition. They can allow preliminary parts of the body of A to call other functions, provided those functions are either non-recursive or themselves tail-recursive in ways the compiler can recognize. Then the compiler can still generate a loop---one whose body calls those functions. There are also cases where two or more *mutually recursive* functions are regimented enough that they would repeat the same two stack frames, so the compiler can convert both together into an iteration. One case that cannot be helped, however, is if you code up "if-then-else" as your own ternary function:

```
let ite (a,b,c) = if a then b else c;;
```

and then rework the first version above as

```
let rec countPeaks ell = match ell with
     [] -> 0
   | [_] -> 0
   | [_; _] -> 0
   | x::y::z::rest -> ite(y > x && y > z,
                          1 + countPeaks(z::rest),
                          countPeaks (y::z::rest));;
```

Now the rules of OCaml require eager evaluation of all three expressions given to your `ite` function, whose body the compiler cannot presume to be able to analyze. This actually **mandates** the naive way Assignment 4 said to evaluate the native if-then-else. Try it---inserting a print statement before the match---and on a list of a dozen-plus elements, you will see the explosion.