subprograms
- Used as a tool for decomposition
- Sometimes can be translated separately and combined after translation
  - Independent compilation
  - Separate compilation
- Two flavors: procedures, functions (methods)
- Variables used: local, non-local or global
- L-value: reference to memory which stores routine body
- R-value: execution of the routine’s body
- Declaration and definition of functions.
- At execution time: function instance
  - code segment and activation record
- Binding of function declaration with its body:
  - Static policy (C, FORTRAN, Ada): at definition or link time.
  - Dynamic binding policy: at invocation time, via polymorphism.

functions as first class values
- In most languages functions are “constant data”.
- Functions as first class values:
  - function types
  - Variables of those types
  - function as arguments
  - function as results
  - functions as part of an ADT
- Functions as variables in some languages (C, Haskell, ML)
  • C: pointer to fun int (*)(int); ps = & sum; int (*ps)(int);
  • Haskell: factor :: Int -> (Int -> Int)
  factor n = product where product m = n * m
doubler = factor 2

Parameters
- Formal and actual
- Binding between formal and actual: positional, name
- Matching of formal with actual: type, number, order
- Optional parameters
- Indefinite-length argument list

Parameter passing
- Copy parameter passing: Do not modify actuals
  - Call-by-value: copy-in
  - Call-by-result: copy-out
- Argument modifiers:
  - Call-by-value-result: copy-in, copy-out
  - Call-by-reference
  - Call-by-pointer (simulation of previous)

execution environment of a function
- Local environment
- Plus its non-local environment determined by the time of its call from the state of the Stack of activation records.
- Closure: a function activation record along with its non-local environment.
Pass-by-reference (aliases) vs pass-by-value-result

```c
int i = 3;
void fun( int a, int b){
i = b;
}
void main(){
int list[10];
list[i]=5;
fun(i, list[i]);
printf(" %d and %d", i, list[i]);
}
```

Pass-by-reference (aliases) vs pass-by-value-result

```c
void main(){
int value = 2, list[] = {1,3, 5, 7,9};
swap(value, list[0]);
swap(list[0], list[1]);
swap(value, list[value]);
}
void swap(int a, int b){
int temp;
temp = a;
a = b;
b = temp;
}
```

parameter evaluation

- Order of expression evaluation
- Order of evaluation of list of arguments
- Argument evaluation time:
  - Eager, used by procedural and strict languages.
  - Evaluate all arguments at time of invocation
  - delayed (by need, Church-Rosser order), (non-strict languages)
    - Evaluate an argument every time is needed in the body of function
  - Lazy, efficiency improvement over delayed in F Prog.
    - Evaluate an argument when needed in body and save its value.

Evaluation time example

```c
int a = 1;
int b = 6;
int c = 3;
f(a*b/c, a – (b – c), c/a * b);
int f ( double x, int y, int z ){ if ( x >= 1){a = a + 1; b = b – 1; c = c + 2; return x + y; }else{a = a – 1; b = b + 1; c = c – 2; return x + z; } }
```

Resolving references

- When functions use only common data types as parameters we know how to resolve reference to variables in the execution of the body of the function:
  - i. find name in local environment
  - else follow static/dynamic chain of activation records looking for the name.
- Which chain to follow depends on the scope disciplined used.

Resolving references

- The situation becomes more complex when we allow functions as parameters:
  - this implies that the function may not be invoked in the same environment where it was defined. In other words we need to come up with a method to determine the "non-local environment".
  - Languages must choose between:
    - i. environment of definition of function: static scope.
    - ii. environment where the function is passed as a parameter.
    - iii. environment where function is invoked: dynamic scope.
  - see example and you get 3 different answers.
typedef void (*function) ();
int x = 1;

void fun1(){
  fun3();
}

void fun2(){    // declaration of fun2
  printf("value of x>> %d", x)
}

void fun3(){
  int x = 3;
  fun4(fun2);  //passing fun2 as parameter
}

void fun4(function X){
  int x = 4;
  X();    // invoking fun2
}

Example of use of Lazy evaluation: Infinite List

// definition of an infinite list
// pre: n an integer number
// post: a list of all integer numbers starting from n
// List from (int n) {
// return List (n, (from (+ n 1))
//}

// removes multiples of num in numList
// pre: num an integer; numList an infinite list of numbers
// post: returns a list formed by all entries in numList which are not multiples of num
// List filter (int num, List numList) {
// if (first (numList) % num == 0)
//   return filter(num, rest(numList));  // first should not be in result
// else
//   return List (car (numList) , filter (num, rest (numList)));
// }

// returns the kth element of a list.
// pre: list must have at least k elements
// post: returns kth element. Zeroth element is the first element of list.
// int nth (int k, List aList) {
// if (k == 0)
//   return first (aList);
// else
//   return nth (rest (k-1, aList));
// }