Abstraction

What is an Abstraction?

- **A mode of thought:** concentrate on general ***ideas*** rather than on the specific ***manifestation*** of these ideas
  - Philosophy
  - Mathematics
  - Computer Science
- **Abstraction in systems analysis**
  - **Essential aspects of a problem**
  - **Ignore extraneous aspects**
  - **Example: air traffic control**
    - Essential: destination, location in space, velocity, …
    - Irrelevant: color, names of passengers
- **Abstraction in programming**: make a distinction between
  - Essential: <u>What</u> does a piece of program do?
  - Inessential**:** <u>How</u> is it implemented?

Kinds of Abstractions

- Every programming language is an abstraction of machine code
- Higher levels of abstraction:
  - **Data:** abstract data types – what the objects stand for and what operations can be applied to them, not how they are represented
    - Sets can be implemented in many ways with vastly different properties
  - **Control:** captures the order in which commands are performed or values are being accessed
  - **Procedural:** what does a procedure (or function) do?
    - ***How*** is essential only when we implement and analyze it
- We shall focus on procedural abstractions, but first...

Motivation

- **Deja-vu as a warning signal:** If while reading your code you experience deja-vu – beware! This is a sign of a potential disaster.
  - **Danger:** since programs are subject to constant change, it is very likely that changes would be made to one occurrence of a "conceptual" abstraction without being made to the other.
  - **Cure:** Capture the abstraction using an *abstraction mechanism.*
- **Abstraction Mechanism:** a programming language *construct* that allows the programmer to *capture* an abstraction and represent it as part of the program.
- There are even simpler linguistic constructs, such as symbolic constants and macros.

Example: Identifying Abstractions in Arrays

Recurring Entity

```
double a[500];
…
for (j = 0; j < 500; j++){
     …
}
```

Capturing the abstraction using the pre-processor #define mechanism

```
#define N 500
double a[N];
…
for (j = 0; j < N; j++){
     …
}
```

Deeper Abstraction found for Arrays

```
#define N 500
#define M 400
double a[N], b[M];
…
for (j = 0; j < N; j++){
     a[j] = …
}
…
for (j = 0; j < M; j++) {
     … = … b[j] …
}
```

N *is the size of* a, M *is the size of* b, *but nowhere in the code this linkage is being enforced.*

Capturing the abstraction using the pre-processor #define mechanism

```
#define SIZEOF(a) (sizeof(a)/sizeof(a[0]))
double a[500], b[400];
…
for (j = 0; j < SIZEOF(a); j++){
     a[j] = …
}
…
for (j = 0; j < SIZEOF(b); j++) {
     … = … b[j] …
}
```

A “Looping” Abstraction: Iteration

```
#define SIZEOF(a) (sizeof(a)/sizeof(a[0]))
double a[500], b[400];
…
for (j = 0; j < SIZEOF(a); j++){
     a[j] = …
}
…
for (j = 0; j < SIZEOF(b); j++) {
     … = … b[j] …
}
```

Recurring entity: looping over an array

Capturing the abstraction using the pre-processor #define mechanism, but in a very cumbersome way

```
#define SIZEOF(a) (sizeof(a)/sizeof(a[0]))
#define ITERATE(a, commands) { \
                    int j;
                    for (jj = SIZEOF(a); j++){ \
                         commands;\
                    }\
               }
double a[500], b[400];
…
ITERATE(a, { ...    a[j] = … })
ITERATE(b, { … = … b[j] … })
```

Alternatives to the Iteration Macro

- Macros are a very poor substitute for programming language constructs
- **Change your programming language**: Replace macros with powerful lingual mechanisms in the programming language
  - **CLU**: iterators as first class language constructs
  - **ADA:** mechanisms for determining the size of an array
  - **C++:** a Standard Template Library (STL) package providing iteration concepts
  - **GNU-C**: generic functions, i.e., functions without a name
  - **ML:** lists instead of arrays, anonymous functions, …
- Sometimes, when the language is not powerful enough, it is better not to capture an abstraction, rather than capturing it in a clumsy way

The Simplest Abstraction Mechanism

- **Procedures and functions:** entities which embody computation
  - **Functional Abstraction:** embody an *expression* to be evaluated
  - **Procedural Abstraction:** embody a *command* to be executed
- The embodied computation is performed whenever the abstraction is called
- Separation of concerns:
  - **Implementer is concerned with how the computation is to be performed**
  - **Caller is concerned with what the computation does**
- Effectiveness is enhanced by *parameterization*

Function Abstraction

- An expression to be evaluated; when called will yield a value.
- Pascal: `function` I(`$FP_1$`; ... ; FP_n): T; B

```
function power(x: Real; n: Integer): Real;
begin (* assume that n > 0 *)
    if n = 1 then
        power := x
    else
        power := x * power (x, n-1)
end
```

This binds power to a function abstraction. The function call power(b, 10) will yield the 10*th* power of b.

Pascal Functions Are Untidy

- Function body always includes commands...
  - **Invites programmers to introduce side-effects**
- Result is defined by assignment to pseudo-variable:
  - Execution may end without such an assignment
  - The function identifier denotes two different things in the same scope:

    ```
    power := x * power (x, n-1)
    ```
  - The variable name is linked to the function name. If the programmer changes the function name, then he or she must also change the variable name. If they fail to do so:
    - Compilation error (*good*)
    - Bug if the same name is defined in an external context
- In general:
  - **The function body is a command by syntax, but**
  - **a very strange kind of expression by semantics**
- It is more natural to allow a function body to be only an expression
  - **This is how it is done in ML**

ML Functions

```
fun power(x: real; n: int) =
 (* assume that n > 0 *)
    if n = 1 then
        x
    else
        x * power (x, n-1)
end
```

- There is no loss of generality. If commands and variables are necessary, then *command expressions* can be used.
- However, one should observe that command expressions can always create side-effects.

Command Expressions

- Suppose that we want to write an *expression* to evaluate the following polynomial at a point *x*:

  $a_n x^n + ... + a_2 x^2 + a_1 x + a_0$

- A recursive solution is possible but it is very unnatural to the problem
- An iterative solution uses commands, but what we are really after is an expression
- **Answer:** *command expression!* Here is how to do it in a hypothetical programming language which includes such a construct:

```
var p: Real; i: integer;
begin
  p := a[n];
  for i:= n-1 downto 0 do p := p * x + a[i];
  yield p;
end
```

What is a Function Definition?

`function` I`(`FP_1`; ... ;` FP_n`)` **`is`** E

- Binds an identifier I to a certain function abstraction.
- The function abstraction yields a result when called with appropriate parameters.
- User’s view of a function call is a *map* between the arguments and the result.
- Implementer's view: evaluation of the function body. Change of algorithm is the implementer's concern only.

```
fun power(x: real; n: int) = (* assume that n > 0 *)
    if n = 1 then
        x
    else if even(n)
          then power(sqr(x), n div 2)
             else power(sqr(x), n div 2) * x
    end
```

Definition and Creation

- In most languages, the only way to construct a function abstraction is by defining it
- In ML, these two operations are distinct:

Construct a function abstraction which implements the cube function:

```
fn(x: real) => x * x* x
```

Binding a name to the above:

```
val  cube = fn(x: real) => x * x* x
```

Or in abbreviated form (syntactic sugar):

```
fun cube(x: real) = x * x* x
```

- A function abstraction can be used in ML as is, without binding a name to it:

```
fun integral(a: real, b: real, f: real->real) = ...
```

And then,

```
... integral(0.0, 1.0, cube) ..
... integral(0.0, 1.0, fn(x: real) => x * x * x)
```

Try doing this in Pascal!

Procedure Abstraction

- Embodies a command to be executed
  - **The user observes only changes to variables**
- In Pascal: you cannot create a procedure abstraction without binding a name to it. General format is:

procedure $I(FP_1; \ldots ; FP_n)\ \ B$

- User's view:

```
type Dictionary = array [...] of Word;
procedure sort(var words: Dictionary);
...
...
sort(a); (* observe change to variable a *)
```

- Implementer's view: the algorithm encoding the details

The Abstraction Principle

- Summary:
  - **Function Abstraction:** an abstraction over an expression
    - A function call is an expression which will yield a value by evaluating the function body
  - **Procedure Abstraction:** an abstraction over a command
    - A procedure call is a command which updates variables by executing the procedure's body
- Generalization:

> It is possible to construct abstractions over any syntactic class, provided only that the phrases of the class specify some kind of computation.

Abstraction over Declaration

- **Generic declaration:** an abstraction over a *declaration* (given parameters and elaborates a declaration)
- **Body:** a declaration
- **Generic Instantiation (call):** a declaration that will *produce* a binding by elaborating the *generic* abstraction body
- Not in Pascal, but in C++ templates, Ada generics, and ML

Abstraction over Variable References

- **Selector:** abstraction over variable reference (given parameters and elaborates a reference)
- **In Pascal** – only built-in selectors:
  - `F^`: reference given a pointer F
  - `V[E]`: reference given an array V and index E
  - `R.A`: reference given record R and element A
- **But there is no way for the programmer to define a new selector**
  - For example: no way to write a function that given a list L of integers, returns a reference to the first integer element in L

Parameters

- **Parameterization:** generalizing the abstraction

```
val pi = 3.14159;
val r = 1.0;
fun circum() = 2 * pi * r;

fun circum (r: real) = 2 * pi * r;
```

formal parameters

```
circum (1.0);
circum (a+b);
```

Arguments

actual parameters

Arguments

- **Pascal**
  - **primitive values**
  - **composite values (except files)**
  - **pointers**
  - **references to variables**
  - **procedure and function abstractions**

- **ML**
  - **primitive values**
  - **composite values**
  - **references to variables**
  - **function abstractions**

Parameter Passing

- The associations between formal and actual parameters
- A variety of mechanisms
  - *by value*
  - *by result*
  - *by value-result*
  - *by name*
  - *by reference*
  - *procedural parameters*
  - *…*
- Can all be described in terms of 2 concepts:
  - **Copy,** *e.g.* by-value
  - **Definitional,** *e.g.* by-reference

Copy Mechanisms

- **Formal parameter:** a local variable
- **Actual parameter:**
  - **value parameter**
    - **legality:** any first-class value
    - **entry effect:** evaluated, value assigned to formal parameter
    - **exit effect:** none
  - **result parameter**
    - **legality:** a variable
    - **entry effect:** none
    - **exit effect:** returned value assigned to actual parameter
  - **value-result parameter**
    - **legality:** a variable
    - **entry effect:** value assigned to formal parameter
    - **exit effect:** returned value assigned to actual parameter

Pseudo Pascal Example

```
TYPE Vector = array[1..n] of Real;

Procedure add(value v, w: Vector, result sum:
Vector);
VAR i: 1..n;
Begin
    for i := 1 to n do
        sum[i] := v[i] + w[i];
End;
```

```
TYPE Vector = array[1..n] of Real;

Procedure normalize(value result u:
Vector)
VAR
        i: 1..n;
        s: Real;
Begin
    s := 0.0;
    for i:=1 to n do
        s := s+sqr(u[i]]);
    s := sqrt(s);
    for i:=1 to n do
        u[i] := u[i]/s;
End;
```

What's the effect of
`add(a,b,c)`
`normalize(c)`
?

Definitional Mechanisms

- **A formal parameter (X) is bound directly to an actual parameter**
  - **constant parameter**
    - argument is (first-class) value
    - X is bound to the *value* during activation
  - **variable (reference) parameter**
    - argument is a reference to a variable (Y)
    - X is bound to the *reference* of Y during activation
    - any inspection/update of X actually inspects/updates Y
  - **procedural/functional parameter**
    - argument is a procedure/function abstraction (Y)
    - X is bound to Y during activation
    - any call to X is an indirect call to Y

Pseudo-Pascal Example

```
TYPE Vector = array[1..n] of Real;

Procedure add(const v, w: Vector, var sum: Vector);
VAR i: 1..n;
Begin
     for i := 1 to n do
          sum[i] := v[i] + w[i];
End;
```

```
TYPE Vector = array[1..n] of Real;

Procedure normalize(var u: Vector);
VAR
          i: 1..n;
          s: Real;
Begin
     s := 0.0;
     for i:=1 to n do
          s := s+sqr(u[i]);
     s := sqrt(s);
     for i:=1 to n do
          u[i] := u[i]/s;
End;
```

What's the effect of
`add(a,b,c)`
`normalize(c)`
now?

Notes

- Constant and variable parameters (both definitional mechanisms) together give almost what we want from copy mechanisms, at a cheaper price
- If a constant parameter is assigned new values inside the procedure, it is a compilation error
- If you want to initialize a local variable, and do want a new copy, just copy the parameter to it (using copy assignment, or clone)
- Reference is cheaper for composite arguments, but has a key problem: *aliasing*, i.e. multiple names for the same variable

Aliasing

```
Procedure confuse1(var a,b: Integer)
VAR
Begin
    a := 1;
    a := b+a;
End;
```

```
Procedure confuse2(var a,b: Integer)
Begin
    a := b+1;
End;
```

```
VAR
        x,y: Integer;
Begin
    …
    confuse1(x,y);    // like confuse2(x,y)
    confuse1(x,x);    // unlike confuse2(x,x)
End;
```

Aliasing (cont.'d)

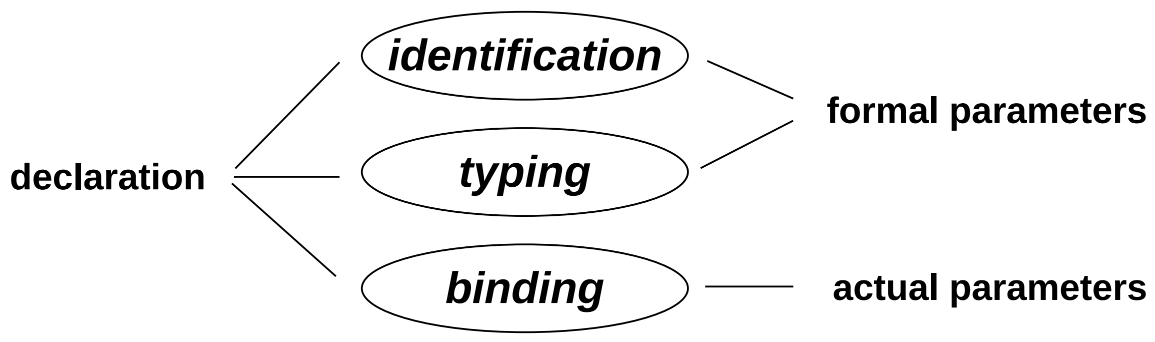
```
Procedure swap1(var a,b: Integer)
VAR
        tmp: Integer;
Begin
    tmp := a;
    a := b;
    b := tmp;

End;
```

```
Procedure swap2(var a,b: Integer)
Begin
    a := a + b;
    b := a - b;
    a := a - b;
End;
```

```
VAR
        x,y: Integer;
Begin
    …
    swap1(x,y);
    swap2(x,y);
    swap1(x,x);
    swap2(y,y);
End;
```

The Correspondence Principle

- **What is the difference between** *declarations* **and** *parameter passing mechanisms***?**



The Correspondence Principle

For each form of declaration there exists a corresponding parameter mechanism, and *vice versa*.

Applying the Correspondence Principle

Declaration	Formal Parameter	Actual Parameter
`Const` `I = E` (constant definition)	**`Const`** `I:T`	An expression *E* of type *T*
`Var` `I := E` (new variable declaration)	`I: T` (value parameter)	An expression *E* of type T
`Var` `I = V` (variable definition)	**`Var`** `I:T`	A variable access V of type T

Evaluation Order

- **When is an actual parameter evaluated?**
- **Eager evaluation (also called applicative order):**
  - **At procedure call**
  - **Present in most languages**
- **Normal-order evaluation (like a macro: substituting actual expression in place of formal parameter within the body, but with the environment of the call)**:
  - **At every use**
  - **In Algol-60 the programmer can define:**
    - *Value parameters*: evaluated eagerly
    - *Name parameters*: evaluated in normal order
- **Lazy evaluation**:
  - **at first use**
  - **In Miranda, Lazy-ML and Haskell**

Evaluation Order: Example

```
fun sqr(n: int) = n*n
p = 2
q = 5
```

How is `sqr(p+q)` evaluated?

Eager:

1. Evaluate `p+q = 7`
2. Bind n to 7
3. Evaluate `n*n = 49`

Normal-order:

1. Bind n to `p+q`
2. Evaluate `n*n = (p+q)*(p+q) = 49`

On Normal Order

- The original way parameters were conceived: like macro substitution
- Problem: if the environment of the use of the formal parameter differs from the environment of the call
- Could have: there are redefinitions of variables in the actual parameter expression within the scope of the procedure
- Need the environment of the call
- This problem complicated Algol 60 “call by name” so much that no modern language tries to use that kind of parameter passing

Strict *vs.* Non-Strict Functions

- **Strict function**: a function `f` is said to be *strict* in its nth argument if it cannot be evaluated unless this argument is evaluated
- **Non-Strict function**: a function `f` is said to be *non-strict* in its nth argument if there are cases in which this argument cannot be evaluated but `f` can.

```
fun cand (b1: bool, b2: bool) =
    if b1 then b2 else false

cand(n > 0, t/n > 0.5)
```

	n	t	eager	normal-order
◆	2	0.8	`false`	`false`
◆	0	0.8	***FAIL!***	`false`

- The function `cand` is
  - **strict** in its first argument
  - **non-strict** in its second argument

The Church-Rosser Property

- A programming language has the **Church-Rosser (CR)** property if all expressions in it have the CR property
- An expression *E* has the CR property if:
  - **If *E* can be evaluated at all, it can be evaluated by consistently using normal-order evaluation**
    - **Note**: sometimes an expression cannot be evaluated at all since it involves an illegal operation
  - **If *E* can be evaluated in different orders (mixing normal-order and applicative-order evaluation), then all of these evaluations orders yield the same result**

Notes on the CR Property

- An expression that has only unary operators can be evaluated in only one order
- There are multiple orders of evaluating all expressions involving binary operators
- The CR property does not hold in languages that allow side-effects
  - **Reason:** when expression *E* has side effects, the number of times *E* is evaluated in *F(E)* certainly makes a difference
  - **Example:** `sqr(getint(x))` gives different results in eager evaluation and normal-order evaluation

Lazy Evaluation

- **Lazy evaluation:** A Good compromise, although hard to implement efficiently.
  - Saves the overhead of eager evaluation, avoiding side-effect nuisances

```
fun sqr (n: int) = n * n
sqr (getint (f)) – the same result as eager evaluation
```

- **ML simple example:**
  - `E1 andalso E2    =  if E1 then E2 else false`
  - `E1 orelse E2     =  if E1 then true else E2`
- **Extremely-Lazy Evaluation:** allow unevaluated expressions as part of evaluated expressions.
  - Lazy-ML example: `fun from(n) = n::from(n+1)`

    for all n returns the “lazy-list” of all integers greater than n.
  - Given

    ```
    fun fp(n::ns) = if prime(n)then n else fp(ns)
    ```

    fp(from(n)) returns the first prime greater than n [for Lazy-ML]

Pros and Cons of Lazy Evaluation

- **Separation of Control from Calculation**: Specify an algorithm that computes everything that might be needed. Only the needed values are computed.

```
fun NR_sqrt_approximation_list(x) = let
 fun seq(approx)= approx :: seq((approx+x/approx)/2)
in seq(1.0) end

fun appropriate_approximation(e, r1:r2::rest) =
      if abs(r1-r2) <= e then r2
            else appropriate_approximation(e, r2::rest)

val sqrt =
                  appropriate_approximation(0.0001)
            o
                  NR_sqrt_approximation_list
```

- No simple way to produce output, read input, or use variables…
  - **Lazy evaluation requires CR property**

Inefficiencies of Lazy Evaluation

- **Inherent overhead:**
  - **Each expression, each value, and each component of a value carries a tag:**
    - Evaluated
    - Unevaluated, with all the information required for evaluating it
  - **Every time an argument or a component is used, the tag must be checked**
  - **If an argument is used more than once, then its value must be updated once it is computed anywhere**
- **Accidental overhead**: Some expressions compute less efficiently with lazy evaluation:

```
let
  fun fact(n) = if n > 0 then mult(fact(n-1),n) else 1
  and fun mult(n,m)= if m =0 then 0 else mult(n,m-1)+n
  in mult(fact(5),10) end;
```

Evaluates 10 times the expression `fact(5)`, instead of just once