

## Programming Languages I

This is the first of 4 lectures looking at programming language paradigms, issues and concepts.

In this lecture we examine

- variables and scoping

In the next two lectures we will look at

- types and type checking
- abstraction mechanisms

In the last lecture we examine

- main programming language paradigms
- logic programming.

The material is largely based on Watt.

## Variables

Variables are at the core of all(?) programming languages.

A variable contains a value which may be inspected and set.

- The most familiar kind of variable is a program variable, e.g.,  $x = x+1$
- However files and databases are also variables in this sense.

Updatable variables are at the core of imperative and OO languages such as C or C++.

Logic and functional programs provide logical variables. These are not updatable, i.e they may not be set to a new value.

Understanding variables requires understanding:

- storage and lifetime
- bindings
- scoping

## Storage

Variables are understood in terms of **storage**.

- A **store** is a collection of **cells** or locations.  
–I.e. the computer's memory.
- A **cell** has a current status, either it is **allocated** or it is **unallocated**.
- A **cell** has a current content which is either a **value** or **undefined**.

```
int a;  
a = 1;  
a = a+2;
```

## Lifetime

Every variable is **created** (or **allocated**) at some point during program execution and may be **deleted** (or **de-allocated**) at some later time when it is no longer needed.

The period between creation and deletion is called the variable's **lifetime**.

A **local** variable is defined within a program block and is only usable within that block. It is created on entry to the block and deleted on exit to the block. It does not retain its value between different activations of the block.

A **global** variable is a local variable which is defined in the outermost block of the program. Thus it has the same lifetime as the program.

Local variables are usually implemented using a **stack**.

## Lifetime (Cont.)

```
int x;

void R(void)
{ int z;
}

void Q(void)
{ int y;
  R();
}

main ()
{
  Q();
  R();
}

enter main
enter Q
enter R
exit R
exit Q
enter R
exit R
exit main
```

5

## Heap Variables

**Heap** variables can be created or deleted at any time and continue to live when the block in which they are created has been exited.

Usually the programmer uses an explicit command to create them (eg `malloc`) and may need to explicitly delete them (eg `free`).

Usually they are explicitly manipulated using **pointers** but may be implicitly manipulated using **references**.

Nested pointers to heap variables are how most imperative languages support recursive types such as lists or trees.

- It is the programmer's job to do all of the work!

One reason for this is that there is a **mismatch** between assignment and large (possibly recursive) data structures. The problem is that assignment may either be understood as

- **copying** the RHS to the LHS
- assigning a **reference** from the LHS to the RHS.

6

Compare execution of the C code

```
int p, q;  
p = 1;  
q = p;  
q = 2;
```

with that of

```
int p[1], q[1];  
p[0] = 1;  
q = p;  
q[0] = 2;
```

Copying is **expensive**, but assigning a reference is **error-prone**, since changes within the structure change all references to it. Furthermore it is hard for the programmer to know when all references to an object have died.

Thus it is left to the programmer to choose what assignment means for a particular case.

In FP and LP complex objects are always manipulated by reference since because logical variables are used, this is equivalent to copying. Thus you get efficiency + safeness!

## Dangling References

In imperative languages pointers to heap variables are usually first class objects, while pointers to local or global variables are often **not**.

In Pascal for instance, you cannot store a pointer to a local variable.

The reason is that after you exit a block, local variables in that block are deleted, thus pointers to these objects now point at nothing. Such a pointer is said to be a **dangling reference**.

Setting a pointer to refer to a heap variable cannot cause this type of error since, by definition, it is alive while something points at it.

Unfortunately, explicit deallocation can allow the programmer to deallocate the object while it is still alive.

## Dangling Reference Example

```
int *p(void)
{
    int i=1;
    return &i;
}

main()
{
    int *q; *r
    q = p();
    *q = 4;
    r = p();
    printf("%d", *q);
}
```

## Persistent Variables

According to our definition files are composite variables:

- a serial file is a sequence of components and
- a direct file is an array of components.

However, most programming languages treat files very differently to other data structures manipulated by the program.

However conceptually the distinguishing feature of a file is that it is **persistent**, that is their lifetime is longer than that of any particular program.

We note that persistent variables reside in a **file store** which has the same abstract properties as does the ordinary variable store.

In Pascal files may be passed as parameters to the main procedure **program**. However in most languages persistent data types are completely disjoint from the usual transient data types.

Type completeness suggests that all types of the programming language should be allowed for both persistent and transient data types. This would mean that the language would not need special commands for input/output and format conversion when reading or writing from a file would not be needed.

## Persistent Variables (Cont.)

For instance, compare

```
main ()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

with

```
persistent char *stdin;
persistent char *stdout;
```

```
main ()
{
    ?????
}
```

This is a relatively recent idea in programming language design. Eg Napier 88.

## Bindings

A binding assigns a value or variable to an identifier.

Objects which may be bound to an identifier are said to be bindable.

In ML virtually everything is bindable.

An environment is a set of bindings. Each expression and command is interpreted in the context of a particular environment.

```
const double e = 2.7182;
const char msg[] = "Warning!";

main()
{
    int e = 1;
    int n;
    char s[];

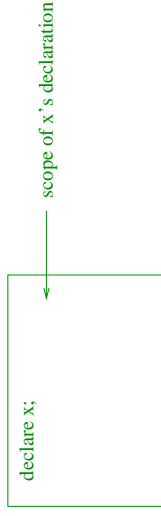
    n = e;
    s = msg;
}
```

## Scope

A **block** is any program phrase that delimits the scope of any declarations it may contain where the **scope** of an identifier is that part of the program text over which the declaration is effective.

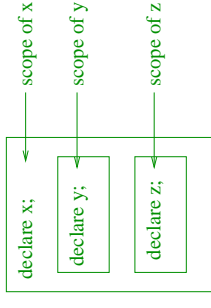
Languages may have a **variety of block structures**.

The simplest is a **monolithic block structure** in which the entire program is a single block.



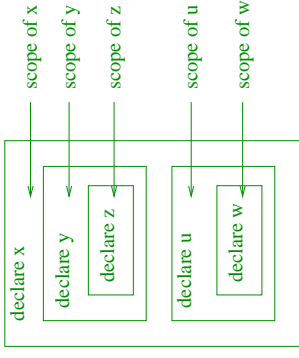
Earlier versions of COBOL were like this.

Some languages have a **flat block structure** in which each subroutine or function in the program forms a single block. Subroutine and function names are global. A variable can be declared within a subroutine in which case it is local to that subroutine or else it may be declared in the outermost block, in which case it is global and all subroutines may refer to it.



**FORTRAN is like this.**

Other languages have a **nested block structure** in which **blocks may be arbitrarily nested inside each other**.



ML is like this with **local ... in ... end** used to nest functions and **let ... in ... end** used to nest expressions.

C is a mixture of the flat and nested block structures since you cannot nest subroutine definitions but you can nest variable declarations:

```
p(int i) {
  while (i-- > 0) {
    real i=0.0;
    ... }
}
```

Notice that where **multiple bindings for the same identifier are within scope, the innermost binding is used**. That is binding an identifier **I** within a block usually **hides** any outer block bindings to **I**.

## Dynamic vs Static Binding

Consider the **Cascal program**

```
const s=2;

function scaled(d:int)
{
  return d*s;
}

procedure A(void)
{
  const s=3;
  int h;

  h := 4;
  writeln( scaled(h));
}

procedure B(void)
{
  int h;

  h := 4;
  writeln( scaled(h));
}
```

## Dynamic vs Static Binding (Cont.)

The result of calling procedure `A` depends on how we interpret the occurrence of `s` in the body of `scaled`.

The most common interpretation is called **static binding**. In this case the function body is evaluated in the environment of the function **definition**.

The alternative interpretation is called **dynamic binding**. In this case the function body is evaluated in the environment of the function **call**.

Dynamic binding does not mix-well with static type checking, since you cannot know the type of an identifier until runtime!

Lisp and SmallTalk have dynamic binding and have dynamic type checking.

Most other languages, eg C and ML, have static binding. We will usually assume static binding in our examples.

## Dynamic vs Static Binding Exercise

Consider the Cascal program:

```
const s = 2;
const d = 3;

int function inc(void) {
    return d+s;
}

void main(void) {
    int s := 5;
    int d := 4;
    s := d;
    writeln( inc() );
}
```

What will be written by the above program if Cascal uses dynamic binding?

What about if it uses static binding?

## Declarations

**A declaration** is a program phrase that will be elaborated to produce a binding.

**Definitions:** The only effect of a definition is to produce a binding.

For instance, `#define PI 3.14` in C and function definitions.

In ML we have the general purpose value definition

```
val I = E
```

that works for both functions and other values.

Eg. `val odd = not o even.`

**Type declarations.** We can distinguish between type definitions which serve to bind an identifier to an existing type and new-type declarations which also define a new type.

`typedef` in C and `type` in ML are type definitions, while ML's `datatype` is a new type declaration.

**Variable declarations.** Again we can distinguish between variable definitions which serve to bind an identifier to a variable and new-variable declarations which also creates a new and distinct variable, ie storage location.

Most imperative languages (including C) provide only new-variable declarations.

FORTRAN and Ada also provide variable definition. The Ada code

```
pop: Integer renames population(state);
```

binds an identifier to a variable.

In ML we use `val` as usual and an explicit allocator `ref` to create a new variable

```
val count = ref 0;  
val count1 = count;
```

## Summary

We have looked at

- variables
- storage and lifetime
- bindings
- scoping

## Homework

- Read Chapter 3 and 4 of Watt.
- Design a persistent variable extension to C. Use this to rewrite your favorite C file manipulation program.
- Consider an extension to C which would allow nesting of functions. Do you see any problems?
- What sort of block structure does C++ have?
- Does C++ use dynamic or static binding?

## Programming Languages II

In the last lecture we examined

- variable storage and lifetime
- identifier bindings and their scope

In this lecture we will look at

- types
- ad hoc and parametric polymorphism
- coercion
- type checking and type inference

The material is loosely based on Watt.

## Types

Values are the “things” that are evaluated, passed as arguments, stored etc in computer programs.

Most programming languages group values into “types.”

But what is a type?

A **type** is a set of values whose elements exhibit uniform behaviour with respect to the operators defined for that type.

For instance, integers exhibit (almost) uniform behavior for the arithmetic operations.

The set of even length lists form a type with respect to list concatenation.

## Primitive Types

A **primitive type** is one that whose values are atomic in the language. The choice of these reflects the use of the language.

Eg, ML has primitive types `real`, `int`, `char` etc.

In some languages you can define new primitive types using

- **Enumeration.** Eg in C

```
enum boolean { TRUE, FALSE};
```

- **Subrange Type.** Eg in Pascal

```
type monthLength = 28..31;
```

## Composite Types

**Cartesian Product.** That is tuples and records.

$$S \times T = \{(x, y) \mid x \in S, y \in T\}$$

**For example:**

```
type person = string * string * int * real;
```

**Note that**  $\#(S \times T) = \#S \times \#T$ .

**Disjoint Union.** This takes the union of two types, adding a tag to indicate which type they originate from.

$$S + T = \{left(x) \mid x \in S\} \cup \{right(y) \mid y \in T\}$$

**For example data constructors in ML:**

```
datatype Number = Exact of int  
                | Approx of real;
```

**Almost the union of C.**

**Note that**  $\#(S + T) = \#S + \#T$ .

## Composite Types (Cont.)

**Powerset.** The powerset of a set  $S$  is the set of all subsets of  $S$ , that is

$$\wp S = \{s \mid s \subseteq S\}$$

**Pascal supports powersets of finite types:**

```
type Colour = { red, green, blue};  
Hue = set of Colour
```

**Reference.** Given a type  $T$ , a reference to  $T$  is a pointer to  $T$ .

**In ML,**

```
val i = ref 0.0;
```

## Mapping Types

A mapping or function  $m : S \rightarrow T$  maps elements in  $S$  to elements in  $T$ .

Arrays are a type of mapping—they map the index set to the array's component set.

In Pascal and Ada the index set can be any discrete primitive type.

```
type Colour = { red, green, blue};  
Pixel = array [Colour] of 0..1
```

Function abstractions implement a mapping by giving an algorithm which computes the result from the argument. In this case  $S$  need not be finite.

Thus, both arrays and functions can be thought of as mappings. The choice of which to use is often an implementation decision.

## Recursive Types

A list is an example of a recursive type. Usually lists are required to be homogenous, ie all elements have the same type.

```
datatype intList = Null | Cons of int * intList;
```

In general, a recursive type  $T$  is defined in terms of recursive type equations.

$$T = \dots T \dots$$

To understand the meaning, we can think of the recursive type equations as a grammar. The type is the language generated by the grammar.

For instance,

```
<intList> ::= Null | Cons( <int>, <intList>)  
<int> ::= 0 | 1 | -1 | ...
```

Exercise: What is the meaning of

```
datatype foo = Empty | Cons of foo * foo | Atom of int;
```

## Strings

Conceptually a **string** is a sequence of characters.

However, programming language designers have modelled strings using a number of different approaches:

- Strings are a **primitive type**. (ML)
- Strings are a **fixed size array** of characters. (Pascal)
- Strings are a **flexible sized array** of characters. (C)
- Strings are a **list of integers(!)** (Prolog)

## Type Systems

In C every value has a **specific type**. The corresponding type system is said to be **monomorphic**.

C forces us to define the exact type of every formal parameter and function return type meaning that every programmer defined function is monomorphic.

However (like almost all other languages) C is not strictly monomorphic.

For example, many built-in operators such as < and – work for more than one type.

Such operations are said to be **generic** in the sense that the same operation makes sense on more than one type.

## Generic Operations

It is useful to distinguish between:

**Parametric polymorphism** (often just called **polymorphism**) in which the function behaves uniformly regardless of the type of the input.

For example, determining the length of the list does not depend on the type of the elements.

**Ad hoc polymorphism** (often called **overloading**) in which the function has different definitions depending on the type of the operands.

For instance, < does very different things in ML depending on if the elements being compared are integers, floats or strings.

Generally overloaded functions do not work for all types just more than one while parametric polymorphic functions work for all types.

## Parametric Polymorphism

We have seen that ML supports parametric polymorphic functions. Eg.

```
fun len [] = 0
  | len (x::xs) = 1 + len xs;
```

has type 'a list → int.

Types with variables are said to be **polytypes**. Those with no variables are said to be **monotypes**.

## Overloading

An operator is overloaded if it simultaneously denotes two or more functions.

In C++ we might define

```
int max(int i, int j)
{ return (i>j) ? i : j;}
float max(float i, float j)
{ return (i>j) ? i : j;}
char *max(char *s, char *t)
{ return strcmp(s,t) > 0 ? s : t;}
```

More generally consider the overloaded identifier  $F$  which denotes both of the functions:

- $F_1 : S_1 \rightarrow T_1$  and
- $F_2 : S_2 \rightarrow T_2$ .

(Note that  $S_1$  and  $S_2$  may be Cartesian products.)

There are two kinds of overloading:

- **Context-independent overloading** requires that  $S_1$  and  $S_2$  are distinct. This is provided in C++.
- **Context-dependent overloading** requires that either  $S_1$  and  $S_2$  are distinct or that  $T_1$  and  $T_2$  are distinct. This is provided in Ada.

## Overloading (Cont.)

Context-independent overloading allows the function to be called to be uniquely identified by the types of the actual parameters.

**Question: Then why does C++ sometimes complain of ambiguity when you use overloaded operators?**

Context-dependent overloading sometimes leads to compile-time ambiguity.

For instance in Ada “/” is overloaded to denote integer division

$Integer \times Integer \rightarrow Integer$

and real division

$Float \times Float \rightarrow Float$ .

We can overload it further

```
function "/" (m,n : Integer) return Float is
begin
  return Float(m) / Float(n)
end
```

**Exercise: Now give an expression which is ambiguous.**

## Type Classes

Recall the `mem` function in ML

```
fun mem(x, []) = false
  | mem(x, y::ys) = (x = y) orelse mem(x, ys);
val mem = fn : 'a * 'a list -> bool
```

where `'a` is an **equality type**, meaning that the type `'a` must have an equality function.

Now recall the home work in which you were asked to write a polymorphic ML function to find the largest item in a list. This was impossible.

```
fun max [x] = x
  | max (x::xs) =
    let val xmax = max xs in
      if x > xmax then x else xmax
    end;
```

```
val max = fn : int list -> int
```

However, similarly to `mem` the most general type should be `'a>` where `'a>` is any type supporting the comparison operation `>`.

## Type Classes (Cont.)

Type classes allow this generic form of ad hoc polymorphism.

If ML had type classes we would be able to write

```
class Comparable 'a where
  op > : ('a * 'a) -> bool;
end;

instance Comparable int where
  op > = op > : (int * int) -> bool;
end;

instance Comparable string where
  op > = op > : (string * string) -> bool;
end;
end;
```

## Type Classes (Cont.)

Type inference would now deduce that:

```
fun max [x] = x
  | max (x::xs) =
    let val xmax = max xs in
      if x > xmax then x else xmax
    end;
```

has type

```
val max = fn : Comparable 'a => 'a list -> 'a
```

Type classes were introduced into the functional programming language Haskell by Phil Wadler and are similar to classes in OO languages.

Now used in logic programming languages.

## Type Classes (Cont.)

As another example consider

```
fun square x = x*x;
```

ML will give `x` either the type `integer` (or `real`) but not both.

However the most general type would be `'aNum` where `'aNum` is any type supporting the usual arithmetic operations including `*`.

If ML had type classes we would be able to write

```
class Num 'a where
  op +, op * : ('a * 'a) -> 'a;
  op ~ : 'a -> 'a;
end;

instance Num int where
  op + = op + : (int * int) -> int;
  op * = op * : (int * int) -> int;
  op ~ = op ~ : int -> int;
end;

instance Num real where
  op + = op + : (real * real) -> real;
  op * = op * : (real * real) -> real;
  op ~ = op ~ : real -> real;
end;
```

Type inference will now deduce that:

```
- fun square x = x**x;  
val square = fn : Num 'a => 'a -> 'a
```

and similarly

```
- fun squares (x,y,z) = (square x, square y, square z);  
val squares = fn : Num 'a, Num 'b, Num 'c =>  
  ('a * 'b * 'c) -> 'a * 'b * 'c
```

## Subtypes

There are two sorts of subtypes in typing.

**Domain subtypes.** Type  $T_1$  is a domain subtype of  $T_2$  if the elements in  $T_1$  are a subset of the elements in  $T_2$ .

As we have seen Pascal allows for such subtypes.

```
type monthlength = 28..31;
```

**Class subtypes.** Type  $T_1$  is a class subtype of  $T_2$  if  $T_1$  supports all of the same functions and operations as does  $T_2$ . This is based on the class hierarchies in OO languages and type classes in FP and LP languages.

```
class Equality 'a where  
  op = : ('a * 'a) -> bool;  
end  
  
class Equality 'a => Num 'a where  
  op +, op * : ('a * 'a) -> 'a;  
  op ~ : 'a -> 'a;  
end;
```

This states that Num is a subtype of Equality.

## Coercion

Many programming languages have operators that explicitly **coerce** a value from one type to another type. For example, type casts in C.

Also many languages have implicit coercion. In C, for example, an integer may be automatically coerced to a float and a float to a double.

In ML no implicit coercion takes place. This is because coercion doesn't work well with type inference.

In C++ you may define your own coercion rules. This is quite complex and may lead to subtle errors or ambiguities.

## Type Checking

In **statically typed** languages each variable has a well-defined type either provided by the programmer or inferred at compile-time. Eg ML.

This catches programmer errors and allows more efficient implementation, but sometimes the type system is restrictive.

In **dynamically typed** languages arguments are tested for type correctness at run-time. E.g. SmallTalk and Prolog (sort of).

This is more flexible, but does not catch programmer errors and is inefficient.

A big area of research is to develop more flexible type systems which can still be checked at compile-time.

For statically typed languages we distinguish between:

- **Type checking** where all function and variable type declarations are provided by the programmer, e.g. C and C++.
- **Type inference** where some type declarations are inferred, e.g. ML.

Generally speaking, type inference is harder than type checking.

## Summary

In this lecture we have looked at

- types
- ad hoc and parametric polymorphism
- coercion
- type checking and type inference

## Homework

- Read relevant chapters of Watt (2 and 7).
- Imagine a language that allows an array to be indexed by a **record**, assuming that that all of the record's component types are suitable. Give an example of a program using this. .
- Discuss a modification to C which would allow recursive types to be directly supported. Consider carefully the meaning of assignment.
- How would you implement type classes? One way would be to have different version of the function for each possible class. Thus square would give rise to

```
fun square_int x = x*x : int;  
fun square_real x = x*x : real;
```

Another way would be to pass in a dictionary of operations for each different type.

```
fun square NumDict x = NumDict.* (x,x);
```

Consider the implementation of squares. Which method do you prefer and why?

- What type will ML infer for

```
fun twice f x = f (f x);
```

- What type will ML infer for

```
fun K x y = x
```

- What type will ML infer for

```
fun S x y z = x z (y z)
```

## Programming Languages III

In the last two lectures we examined

- variable storage and lifetime
- identifier bindings and their scope
- types

In this lecture we examine

- abstraction
- parameter passing
- encapsulation
- objects

The material is loosely based on Watt.

## Abstraction

Abstraction is a mode of thought in which we concentrate on general ideas rather than specific details.

An **expression** is a program phrase that is evaluated to give a value, i.e.  $(x+y, 3.0)$ .

A **command** is a program phrase which when executed updates the values of variables. (Sometimes they are called statements). E.g.  $a[i] = j$

A **selector** provides variable access. E.g. In C,  $V[E]$  for arrays and  $V.I$  for records.

In computer programming we define an **abstraction** to be an entity that embodies a computation.

- A **function abstraction** embodies an expression to be evaluated.
- A **procedure abstraction** embodies a command to be executed.
- A **selector abstraction** embodies a command to access a variable. It returns a reference to the variable.

Only the programmer who implements the abstraction is concerned with details of **how** it is done. Other programmers who use it are only interested in **what** it does.

## Abstraction (Cont.)

In C and ML there is no real syntactic distinction between function and procedure abstractions — they are both regarded as “functions.”

Pascal distinguishes between the two using `function` for the first and `procedure` for the second.

Few languages provide selector abstractions, although C++ does.

## Parameters

An identifier used within an abstraction to denote an argument is called a **formal parameter**.

An expression that yields an argument is called an **actual parameter**.

```
val pi = 3.14;  
fun circum (r) = 2.0 * pi * r;  
- circum(4.0+pi);
```

There have been many mechanisms suggested for “passing” parameters.

## Copy Mechanisms

A **copy** mechanism allows for values to be copied in or out of an abstraction. The formal parameter **X** is a local variable in the abstraction. The actual parameter's value is copied into **X** on entry and/or copied out of **X** (to a nonlocal variable) on exit.

So called **value** parameter (passing) is used in C. In this mechanism the actual parameter is copied to the local variable **X**. Since **X** is a local variable any updating of **X** has no effect on any non-local variables.

The mirror-image to this is the **result** parameter. In this case the argument is a (reference) to a variable. Again the local variable is created but it is uninitialized. On exit from the abstraction, the final value of **X** is assigned to the argument variable.

We can combine these to give a **value-result** parameter. In this case the argument is a reference to a variable and the formal argument is copied to **X** on call and on exit the final value of **X** is copied to the argument variable. Used in Algol W.

**Exercise** Consider the Cascal program. Write down the result when each of the above parameter passing mechanisms is used

```
void dummy(int x)
{
    x := x+1;
    print(x);
}

main()
{
    int y;
    y := 1;
    dummy(y);
    print(y);
}
```

## Definitional Mechanisms

A **definitional mechanism** allows for the **formal parameter X** to be bound **directly to the argument**.

In the case of a **constant parameter** the argument is a first class value which **X** is bound to.

In the case of a **variable** or **reference parameter** the argument is a reference to a variable. Thus the procedure can also **update** the argument variable.

In the case of **procedural** or **functional parameters** the argument is a procedure or function abstraction.

Notice that these are not really distinct, all bind the formal parameter to the actual parameter. ML uses only the definitional mechanism.

What happens in:

```
void dummy(ref int x) {  
  x := x+1;  
  print(x);  
}
```

```
main() {  
  int y;  
  y := 1;  
  dummy(y);  
  print(y);  
}
```

51

## Aliasing

The disadvantage of reference parameters is that **aliasing** may make the program hard to understand.

```
void dummy(ref int x, ref int y)  
{  
  x := 1;  
  x := x + y;  
}  
  
main()  
{  
  int i;  
  i := 4;  
  dummy(i,i);  
  print(i);  
}
```

Here aliasing is easy to detect but in general it is **impossible** to determine!

52

## Call-by-Name

In **call-by-name** parameter passing the formal parameters are textually replaced by the actual parameters in the abstraction body.

Also called **normal-order** evaluation.

Call-by-name was the default parameter passing mechanism used in Algol 60.

It is also how macros work in C.

Despite its intuitive simplicity, call-by-name was largely given up after Algol 60 because the presence of side effects made it too difficult to understand programs.

```
void swap(name int x, y)
{
    int t;
    t := x; x := y; y := t;
}
```

**Exercise:** What does `swap(n, A[n])` do?

## Evaluation Order

An important issue in parameter passing is **when** is the actual parameter evaluated?

There are **basically three methods**:

- **Eager evaluation** – i.e. at the time of procedure call. Also called **applicative-order** evaluation.
- **Lazy evaluation** – i.e. when needed. Evaluation is only performed once when first needed and the result is “memoed” to be used when needed subsequently.
- **Call-by-name** parameter passing.

For pure functional languages we have that if they terminate, all three methods give the same result. This is called the **Church-Rosser property**.

However for languages with side-effects (ie updatable variables) this is not true.

## Packages

Since the 1970s programming language design has focused more on supporting programming in the **large**. This is about construction of **modules**.

A **procedure** exemplifies **encapsulation** of commands.

Here we look at **data abstractions** and related notions.

The simplest way to group a collection of related functions, procedures and declarations together is a **package**. This is simply a **named** collection of such components.

Usually this is combined with some way of specifying which components are **exported** by the package and which components are **hidden** in the package.

In ML packages are provided by **structures** and hiding is provided by declaring the structure's **signature** or by using **local** declarations.

In C a package is identified with a **file** and the programmer labels functions to indicate if they are **exported** or not.

This is similar to SICStus Prolog's **module** system.

## Abstract Types

Another way of organising functions and procedures is around a type whose representation we can hide. Functions and procedures which use this type are placed in a single module called an **abstract type**.

ML provides the **abstype** for this purpose, while in Ada you can use a **package declaration**:

```
package directory_type is
  type Directory is limited private;
  procedure init(dir: out Directory);
  procedure insert(dir: in out Directory;
                 newname: in Name;
                 newnumber: in Number);
  procedure lookup(dir: in Directory;
                 name: in Name;
                 number: out Number;
                 found: out Boolean);
private
  type DirNode is record
    entryname: Name;
    entrynumber: Number;
  end record;
  type Directory is record
    dict: array(0..100) of DirNode;
    num: 0..101;
  end record;
end directory_object
```

This defines the “interface” the actual implementation is defined in the “body”

```
package body directory_type is
  procedure init(dir: out Directory) is
    dir.num = 0;
  procedure insert(dir: in out Directory; newname: in Name;
                  newnumber: in Number) is
    ... put element in dictionary ...;
  procedure lookup(dir: in Directory; name: in Name;
                  number: out Number; found: out Boolean) is
    ... iterate through elements in array...;
end directory_type;
```

Values of type Directory can only be accessed and displayed using the functions declared in the package interface.

```
use directory_type;
homedir: Directory;
workdir: Directory;
init(homedir);
init(workdir);
insert(homedir, kim, 96088913);
insert(workdir, kim, 55525);
lookup(workdir, kim, kimNo, ok);
```

Abstract types look similar to objects in C++. What is the essential difference?

## Object-Based Programming

According to Watt an **object** is a **hidden** variable in a package or module recording its current state.

The following Ada package defines a dictionary “object”:

```
package directory_object is
  procedure insert( newname: in Name;
                  newnumber: in Number)
  procedure lookup( name: in Name;
                  number: out Number;
                  found: out Boolean)
end directory_object
```

## Object-Based Programming (Cont.)

### The package definition is

```
package body directory_object is
  type DirNode is record
    entryname : Name;
    entrynumber: Number;
  end record
  num: 0..101;
  dict: array(0..100) of DirNode;

  procedure insert( newname: in Name;
                  newnumber: in Number) is
    ... put element in dictionary ...;
  procedure lookup( name: in Name;
                  number: out Number; found: out Boolean) is
    ... iterate through elements in array...;
  begin
    num = 0;
  end directory_object;
```

### Elaboration of this package creates a single object

```
directory_object.insert(kim, 55525);
directory_object.insert(bernd, 52240);
directory_object.lookup(kim, kimNo, ok);
```

59

## Object-Based Programming (Cont.)

A generic package declaration does not create an object but rather defines a whole class of objects. To create an object we create a new instance of this class and its hidden variable.

```
generic package directory_class is
  procedure insert( newname: in Name;
                  newnumber: in Number)
  procedure lookup( name: in Name;
                  number: out Number;
                  found: out Boolean)
end directory_class
```

### The definition is as before.

### We can now create multiple dictionary objects

```
package homedir is new directory_class;
package workdir is new directory_class;
homedir.insert(kim, 96088913);
workdir.insert(kim, 55525);
workdir.lookup(kim, kimNo, ok);
```

**Exercise: What are the key differences between object classes and abstract types?**

Programming languages which allow objects in the preceding sense are said to be **object-based**.

60

## Summary

We have looked at

- Abstractions
- Parameter Passing
- Encapsulation
- Object-based programming

## Homework

- Read Chapters 5 and 6 of Watt.
- Do you think verbatim replacement of the formal parameter by the actual parameter is the way to formalize call-by-name parameter passing? Consider what happens if a local variable has the same name as a variable in the actual parameter.
- What do you think is the difference between object-based and object-oriented programming languages?

## Programming Languages IV

This is the last of 4 lectures looking at programming language paradigms, issues and concepts.

In previous lectures we have looked at unifying concepts for all programming languages

- variables and scoping
- types and type checking
- abstraction mechanisms

In this lecture we examine

- logic programming
- main programming language paradigms

## Programming Language Paradigms

Recall that programming languages provide

- An underlying computation model
- Data types and operations,
- Abstraction facilities
- Checking and enforcement

Different computation models lead to different programming language paradigms.

The traditional four main paradigms are

- **Imperative (or procedural)**: based on the von Neumann architecture with updatable memory locations.
- **Object-oriented**: based on “objects” which encapsulate state and which communicate by message passing.
- **Functional**: based on mathematical functions (more precisely the lambda calculus).
- **Logic**: based on mathematical relations (more precisely predicate calculus).

Are there other paradigms?

## Imperative Programming Paradigm

**Imperative programming** is so called because it is based on commands that update variables held in storage. (imperare is Latin for to command)

The first high-level programming languages (**FORTAN, COBOL**) were imperative because of the close match with the underlying computer architecture with updatable memory locations meant that

- they were easily compiled
- could be implemented efficiently.

What are some more examples of imperative programming languages? What are they good for?

The imperative programming paradigm used to be the dominant programming paradigm, is this still true?

## Object-Oriented Programming Paradigm

According to Watt an **object** requires a **hidden** variable in a package recording its current state.

Since the programmer can change internal state of the object, object-oriented programming is really an extension of the imperative programming paradigm.

In **object-oriented programming** languages

- **objects** and **classes** (generic packages) are fundamental concepts
- objects are **first-class** values
- objects are characterised by the **methods** or **actions** which can be applied to it
- classes are organised into a **hierarchy** and sub-classes can **inherit** method definitions from their super-classes.

Other (once) common features are dynamic typing and dynamic binding.

Object-oriented languages originated with Simula, but with C++, Java and C# are rapidly becoming the dominant programming paradigm.

Arguably this is because they accord with the inbuilt human cognitive model for understanding the world in terms of entities, state and actions.

What are some more examples of OO programming languages?  
What are they good for?

## Functional Programming Paradigm

Functional languages have the following characteristics:

- Everything is a function or a value.
- First-class higher-order functions.
- The underlying conceptual model is the lambda calculus.

They are high-level languages with implicit memory management and often provide lazy evaluation and nowadays type classes.

What are some more examples of functional programming languages? What are they good for?

## The Logic Programming Paradigm: History

- **early 1970's Kowalski:** computational interpretation of logic.

The logic statement:

$A$  if  $B_1$  and  $B_2$  and  $\dots$  and  $B_n$

is read computationally as:

to solve (execute)  $A$ ,

solve (execute)  $B_1$  and  $B_2$  and  $\dots$  and  $B_n$

- **early 1970s Colmerauer:** develops a specialized theorem prover (written in Fortran) embedding Kowalski's procedural interpretation: **Prolog** (stands for Programmation et Logique).
- **1981 Japanese Fifth Generation Project:** logic programming becomes the hot new programming paradigm.
- **late 1980s** numerous commercial Prolog implementations, programming books, and a de facto standard, the **Edinburgh Prolog family**.
- **late 1980s Constraint Logic Programming** (initially developed at Melbourne, Monash, Marseille, etc) and **Deductive Databases** developed.
- **Now Typed logic and constraint logic programming languages (Mercury and HAL)** are under development at Melbourne and Monash (almagamate logic and functional programming).

## Simple Prolog Example

Consider the Prolog program `solar.pl`

```
orbits(mercury, sun).
orbits(venus, sun).
orbits(earth, sun).
orbits(mars, sun).
orbits(moon, earth).
orbits(phobos, mars).
orbits(deimos, mars).
```

Using **SICStus Prolog** we can read in this Prolog program and interactively ask questions about the facts in it:

```
% sicstus
booting, please wait...
SICStus 3 #6: Mon Nov 3 19:53:41 MET 1997
| ?- consult('solar.pl').
yes
| ?- orbits(Planet, sun).

Planet = mercury ? ;
Planet = venus ? ;
Planet = earth ? ;
Planet = mars ?
yes
```

## Simple Prolog Example (Cont.)

```
| ?- orbits(earth, X).  
X = sun ? ;  
no  
  
| ?- orbits(Satellite, Planet), orbits(Planet, sun).  
  
Planet = earth,  
Satellite = moon ? ;  
Planet = mars,  
Satellite = phobos ? ;  
Planet = mars,  
Satellite = deimos ?  
  
yes
```

## Simple Prolog Example (Cont.)

The Prolog facts define a **relational database** which the programmer can query.

A query can have zero or more answers.

A query can be a **conjunction**.

Variables start with an uppercase letter, constants with a lower case letter.

## Rules

Not all information is expressed as facts, we also have **rules** to deduce information:

**A planet is a body that circles the sun.**

The Prolog rule is:

```
planet(B) :- orbits(B, sun).
```

The symbol **`:-`** is read as “if.”

Another example of a rule is:

**A satellite is a body that circles a planet:**

The corresponding Prolog rule is:

```
satellite(B) :- orbits(B, P), planet(P).
```

We can add these rules to our Prolog program and then read it in and query our solar system database

```
| ?- satellite(B).  
B = moon ? ;  
B = phobos ? ;  
B = deimos ? ;  
no
```

## Execution Mechanism

Prolog executes a query by repeatedly

- Selecting the leftmost atom and replacing it by its definition
- Trying the definitions in turn and backtracking when these lead to failure.

## Example of Execution

```
orbits(mercury, sun).
orbits(venus, sun).
orbits(earth, sun).
orbits(mars, sun).
orbits(moon, earth).
orbits(phobos, mars).
orbits(deimos, mars).

planet(B) :- orbits(B, sun).
satellite(B) :- orbits(B, P), planet(P).
```

How does Prolog execute the query `satellite(B)?`

## Data Structures

Prolog provides numbers (either floats or integers) and terms.

Terms are just like ML's data constructors.

Also like ML, pattern matching is used.

However, you do not have to define or declare the types since Prolog is typeless.

## Simple Terms

The simplest type of data structures are **records**.

For example consider the **C** structs

```
struct lecturer {
    char[] first;
    char[] last; }

struct time {
    Days day;
    int start;
    int finish;
}
```

We represent these in Prolog by terms:

```
lecturer(kim,marriott)
time(monday,10,11)
```

They are a lot like ML's data constructors:

```
datatype lecturer = Lecturer of string * string;
datatype weekday = Monday | Tuesday | Wednesday | Thursday
datatype time = Time of weekday * int * int;
```

75

## Simple Terms (Cont).

Example database:

```
course(prolog_programming,
        time(monday,10,11),
        lecturer(kim,marriott),
        s12).
course(lisp_programming,
        time(tuesday,10,11),
        lecturer(ann,nicholson),
        c1).
```

We can query the database to find out about courses:

- **Where is Prolog programming?**  
?- course(prolog\_programming, -, -, Loc)  
Loc = s12.
- **When is Prolog programming?**  
?- course(prolog\_programming, Time, -, -)  
Time = time(monday,10,11).
- **When does Lisp programming start?**  
?- course(prolog\_programming, time(\_, Start, \_), -, -)  
Start = 10.

In Prolog data structures are matched, much like ML. More powerful, though. **Unification** is used for matching so data structures can have variables in them and variables can appear in the match more than once.

76

## Lists

Like data constructors terms can be recursive. This provides recursive data structures like lists and trees.

Because lists are so important, special notation is used to represent both lists and partially constructed lists.

- `[]` is the empty list
- `[X1, X2, ..., Xn]` is a fixed size list containing the elements X<sub>1</sub>, ..., X<sub>n</sub>. (Like ML).
- `[X1, X2, ..., Xn | Y]` is a list whose first *n* elements are X<sub>1</sub>, ..., X<sub>n</sub> and whose remaining elements are the list Y.  
(In ML this is `X1 :: X2 :: ... :: Xn :: Y`.)

For example:

- `[a, b]` and `[a|X]` match with `X = [b]`.
- `[a]` and `[a|X]` match with `X = []`.
- `[a]` and `[a, b|X]` do not match.

## Programming with Lists

As in ML the key to programming with lists is to reason recursively about the list.

Write a predicate `member(X, Y)` which holds if X is a member of the list Y. E.g. `member(X, [1,2,3])` should return the answers `X = 1`, `X = 2` and `X = 3`.

What if the list Y is empty? Do not return anything.

What if the list Y is `[U | V]`? Two cases:

- X is the first element U;
- X is an element of the remainder of the list V.

The program is:

```
/* member(X,Y) :- X is a member of list Y */
member(X, []) :- fail.
member(X, [U|V]) :- X=U.
member(X, [U|V]) :- member(X, V).
```

A better program is:

```
/* member(X,Y) :- X is a member of list Y */
member(X, [X|_V]) .
member(X, [_|V]) :- member(X, V) .
```

Compare this to the ML function

```
fun mem(x,[]) = false
  | mem(x,y::ys) = (x = y) orelse mem(x,ys)
```

## Member (Cont.)

We can use `member(X, Y)` in a variety of ways:

- **Checking whether an element is in a list –**

```
| ?- member(2, [3,2,4]).  
yes
```

- **Finding the elements in a list –**

```
| ?- member(X, [3,2,4]).  
X = 3 ? ;  
X = 2 ? ;  
X = 4 ? ;  
no
```

- **Finding a list containing an element –**

```
| ?- member(2, X).  
X = [2|_A] ? ;  
X = [_A,2|_B] ? ;  
X = [_A,_B,2|_C] ?
```

- **Or even:**

```
| ?- member(X, Y).  
Y = [X|_A] ? ;  
Y = [_A,X|_B] ? ;  
Y = [_A,_B,X|_C] ? ;
```

This shows the difference between Prolog and most other programming paradigms: with Prolog you are programming relations not functions.

## Predicate Logic View

Prolog is very close to first-order predicate logic.

Prolog variables are like **logic variables** i.e can take only one value.

Different rules defining the same predicate correspond to **disjunction**:

```
parent(P,Par) :- father(P,Par).  
parent(P,Par) :- mother(P,Par).
```

is essentially

**For all  $X$ , for all  $Y$ ,  $parent(X, Y)$  is true if and only if  $father(X, Y)$  is true or  $mother(X, Y)$  is true.**

Different atoms in a body are **conjoined together and local variables are existentially quantified**:

```
grandparent(P, GdPar) :- parent(P, Par), parent(Par, GdPar).
```

is

**For all  $X$ , for all  $Y$ ,  $grandparent(X, Y)$  is true if and only if for some  $Z$ ,  $parent(X, Z)$  and  $parent(Z, Y)$  is true.**

## Logic Programming Paradigm

Logic programming languages are based on **relations**. They can be formalised in terms of predicate calculus.

**PROLOG** is the best known example.

Traditional application areas:

- natural language understanding (DCGs)
- expert systems
- other AI applications
- databases
- compiler writing
- rapid prototyping

**Constraint Logic Programming (CLP)** application areas:

- engineering analysis and design
- financial modelling
- combinatorial optimization

## Summary

We have reviewed the 4 main programming paradigms – imperative, object-oriented, functional and logic.

We have used **PROLOG** to introduce the logic programming paradigm and looked at

- How Prolog defines **relations**.
- How Prolog can be used to model information expressed as **facts** and **rules**.
- How to define arithmetic relations in Prolog.
- How Prolog computes the answers to a **query**.
- Prolog data structures and how to interpret them.
- A first-order logic interpretation of Prolog programs.

## Homework

- Read Chapters 10, 12, 13 and 14 of Watt.
- Write a query to find out which objects orbit mars.
- Write a logic program to return the last element in a list.
- Give the logical interpretation of your definition.