

Programming Language Concepts & Issues II

This is the second of 5 lectures looking at programming languages.

In the first lecture of CSE3322 we discussed the history of programming languages and the reasons for their development.

In this and the next 2 lectures we will examine the main underlying issues and concepts.

In the last lecture we will examine the main programming language paradigms.

In this lecture we will study

- **variables**

The material is largely based on Watt but also draws upon Mitchell and Pratt & Zelkowitz.

Variables

Variables are at the core of all(?) programming languages. They are understood in terms of **storage**.

- All data in a computer is stored as a sequence of bits grouped into bytes or words.
- A **data object** is a collection of bytes which represents one logical data value.
- A data object that is named or defined by the programmer is called a **variable**.
- A **variable** contains a data value which may be **inspected** and **set**.

The most familiar kind of variable is a **program** variable, e.g.,

```
x = x+1
```

Constants are also variables in this sense: they just can't have their value changed.

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.

Overview

Understanding variables requires understanding:

- lifetime
- binding mechanisms
- scoping

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.

Simplicity and orthogonality 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 the Persistent C program

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

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

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

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; /* global variable */

void R(void)
{ int z; /* local variable */
}

void Q(void)
{ int y; /* local variable */
  R();
}

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

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



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.

Often 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!

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);
}
```

Garbage Collection

Because of the difficulty and error proneness of explicitly managing allocation and deallocation of heap storage, starting with Lisp, many languages, such as Prolog, ML and Java, provide automatic memory management in which the language automatically deallocates memory locations when they are no longer required.

More specifically, at a given point in the execution of a program P , a memory location m is **garbage** if no further execution of P from this point can access location m .

Garbage collection is the process of finding memory locations that are garbage and deallocating them.

Mark-and-Sweep Garbage Collection

Many algorithms have been developed for garbage collection. A simple one is **Mark-and-Sweep**.

It assumes that each data-object has a spare tag that can be set to 0 or 1.

It also assumes that we can tell which sequences of bits are pointers.

Mark-and-Sweep Garbage Collection

1. Set all tag bits to 0.
2. Start from each location used directly in the program. Follow all links (pointers), changing the tag bit of each location visited to 1.
3. Deallocate all memory locations with tag still equal to 0.

Advantages and Disadvantages

The main advantage of automatic garbage collection is that memory management errors never occur. It also simplifies code.

The main disadvantage is that it has some space and time overhead (Mitchell suggests 5%). And in general, accurate garbage collection requires all pointers to be identifiable at run-time.

Question: Why do you think C and C++ do not provide garbage collection while Java does?

Bindings

A binding assigns a value or variable to an identifier.

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[];

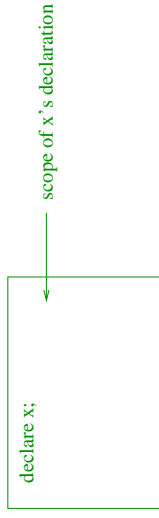
    n = e;
    s = msg;
}
```

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.

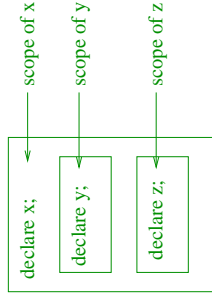
Languages may have a **variety of block structures**.

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



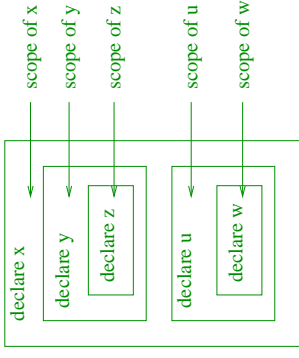
Earlier versions of COBOL were like this.

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.



(Early?) 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`.

Cascal

In this part of the unit we will make use of an **imaginary** language **Cascal** to illustrate different design choices.

As its name suggests Cascal syntax is a cross between C and Pascal.

We are free to decide what the syntax means.

Dynamic vs Static Binding

Consider the Cascal program: what does it write?

```
int s := 2;

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

void A(void)
{
  int s := 3;
  int h := 4;
  writeln( scaled(h));
}

void B(void)
{
  int h := 4;
  writeln( scaled(h));
}

void main(void)
{
  A(); B();
}
```

19

Dynamic vs Static Binding (Cont.)

The result of calling function **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!

Early versions of Lisp and SmallTalk have dynamic binding and have dynamic type checking. TeX also has dynamic binding.

Most other languages, eg C and ML, as well as Scheme and newer versions of Lisp have static binding. We will usually assume static binding in our examples.

Even in languages with static binding, choosing which exception handler to use when an exception is thrown is usually dynamic. **Why?**

20

Dynamic vs Static Binding Exercise

Summary

Consider the Cascal program:

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

int 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?

We have looked at

- variables
- lifetime and garbage collection
- binding mechanisms
- 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 Language Concepts & Issues III

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.

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, subtype and parametric polymorphism
- coercion
- type checking and type inference

The material is loosely based on Watt and Mitchell.

Types (Cont.)

Types in modern programming languages can be divided into

Primitive types These are types 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.

Composite types These are types made by combining other types. Examples include **tuples**, **records**, **disjoint union** and **reference** or pointer types.

Unusually Pascal provides **powersets** of finite types:

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

Recursive types

Mapping types

Mapping Types

A mapping $m : S \rightarrow T$ maps elements in type S to elements in type 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
```

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

Since 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.

```
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 | Comp of foo * string * foo;
```

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 .

Pascal allows the programmer to declare such subtypes.

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

Class subtypes. Type T_1 is a class subtype of T_2 if T_1 supports at least the same functions and operations as does T_2 . This is based on the class hierarchies in object oriented languages and type classes in functional and logic programming languages.

Class Subtype Example

The following C++ code states that `ColourPoint` is a class subtype of `Point`.

```
class Point {
public:
    int getX();
    int getY()
    void move(int,int);
protected: ...
private: ...
}

class ColourPoint: public Point {
public:
    int getX();
    int getY()
    void move(int,int);
    int getColour();
    void setColour(int);
protected: ...
private: ...
}
```

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.

Subtype polymorphism in which a function expecting an argument of type T_1 can take any type T_2 which is a class subtype of T_1 .

Subtype polymorphism is a core part of object oriented languages. However it is also found (under the name **type classes**) in recent functional and logic programming languages.

In some senses it is a natural generalisation of both parametric and ad hoc polymorphism.

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 tuples.)

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.

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.

Subtype Polymorphism

All object oriented languages provide subtype polymorphism.

For instance the function `translatePoint` in the C++ code will also work with `ColourPoints`.

```
class Point {
public:
    int getX();
    int getY();
    void move(int,int);
protected: ...
private: ...
}

class ColourPoint: public Point {
public:
    int getX();
    int getY();
    void move(int,int);
    int getColour();
    void setColour(int);
protected: ...
private: ...
}

/* function to move point p by (x,y) */
void translatePoint(Point *p, int x, int y) {
    p->move(p->getX() + x, p->getY() + y);
}
```

35

Subtype Polymorphism

One subtlety of subtype polymorphism is precisely defining what being a class subtype means.

Recall that type T_1 is a class subtype of T_2 if T_1 supports at least the same functions and operations as does T_2 . This means that T_1 can be safely substituted for T_2 .

So if T_2 has a function $f_2 : S \rightarrow T$ and T_1 has a function $f_1 : S' \rightarrow T'$ what should be the relationship between S and S' and between T and T' ?

This has led to a lot of debate. However we require that f_1 can be safely substituted for f_2 . This means that we require S' should be a class subtype of S and that T should be a class subtype of T' .

36

Type Classes

Recall the homework 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, the most general type should be any type supporting a comparison operation >.

Type classes allow this generic form of subclass polymorphism. They were introduced into the functional programming language Haskell by Phil Wadler and were inspired by classes in object-oriented languages. Now used in logic programming languages.

Type Classes (Cont.)

Unfortunately ML was designed before type classes existed. But if ML had type classes we would be able to write

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

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

instance Orderable string where
  op > = op > : (string * string) -> bool;
  op < = op < : (string * string) -> bool;
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 : Orderable 'a => 'a list -> 'a
```

Type Classes Ordering

Like classes in object oriented languages, the programmer can organise type classes into a hierarchy.

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

```
class Equality 'a where
  op = : ('a * 'a) -> bool;
  op <> : ('a * 'a) -> bool;
end
```

```
class Orderable 'a, Equality 'a => Comparable 'a where
  op >= : ('a * 'a) -> 'a;
  op <= : ('a * 'a) -> 'a;
end;
```

Indicates that Comparable is a subclass of both Orderable and Equality. An instance of Comparable must provide the 6 functions >, <, =, <>, >=, <=.

Another Type Class Example

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 any type supporting the usual arithmetic operations including `*`.

If ML had type classes we would be able to write

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

Another Type Class Example (Cont.)

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
```

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 Safety

A programming language is **type safe** if no program is allowed to violate its type distinctions.

Lisp, ML, Smalltalk and Java are all type safe. C and C++ are not. **Why?**

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 more programmer errors and allows more efficient implementation, but sometimes since the type checking must be conservative, the type system is restrictive.

In **dynamically typed** languages arguments are tested for type correctness at run-time. E.g. Smalltalk.

This is more flexible but cannot be used to guarantee that a program is free from type errors and introduces some runtime overhead.

Sometimes a mixture of static and dynamic typing is used in which case as much type checking as possible is done at compile time while some type checking is done at run-time. E.g. Java.

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

Type Checking and Inference

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, subclass and parametric polymorphism
- coercion
- type checking and type inference

Homework

- Read relevant chapters of Watt (2 and 7).
- C++ provides context-independent overloading. In the lecture we stated that context-independent overloading allows the function to be called to be uniquely identified by the types of the actual parameters. Then why does C++ sometimes complain of ambiguity when you use overloaded operators?
- 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;
```

And the compiler would call the appropriate version. This is how templates work in C++.

Another way would be to associate an appropriate dictionary of operations with each variable and pass this along with the variable. This is similar to how most object oriented languages support objects.

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

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

Programming Language Concepts & Issues IV

In the last two lectures we examined

- variables
- types

In this lecture we examine abstraction mechanisms.

The material is loosely based on Watt but also draws upon Mitchell and Pratt & Zelkowitz.

Abstraction

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

In programming languages abstraction mechanisms allow us to hide implementation details of a program component from the **client** using the program component. How the client can use the component is specified by an **interface**.

We will look at 4 types of abstraction mechanisms:

- Procedural abstraction including parameter passing mechanisms
- Abstract data types
- Modules
- Objects

Procedural Abstraction

In computer programming we define an procedural abstraction to be an entity that embodies a computation, i.e. abstracts a sequence of program commands.

A procedural abstraction has a well-defined interface made explicit in the code. This gives the name of the function and the input parameters. It may provide type information. Usually the implementation may use local variables to hide information.

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

Procedural Abstraction (Cont.)

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

A **command** is a program command 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.

Based on this Watt distinguishes between:

- **Function abstractions** These abstract an expression to be evaluated. They return a single value and should have no side effects.
- **Procedure abstractions** These abstract a command that updates variable values.
- **Selector abstractions** These embody a command to access a variable and return a reference to the variable.

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

Pascal (sort of) 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. In essence we **assign** the value of the actual parameter to the formal parameter. Thus these mechanisms are often similar to variable assignment.

Variable Assignment

Variable assignment is usually implemented in one of two ways:

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

Copying is expensive for larger data structures, 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. This is the problem of tracking **aliasing**, i.e. knowing when two variables refer to the same object.

Example: Variable Assignment in Java

In Java what happens depends on the type of the variable: primitive types such as `char` are copied while objects are assigned a reference.

Compare execution of the Java code

```
char c='c';
char d='d';

c = d;
d = 'e';
System.out.print(""+c+d);
```

with that of

```
public class MyChar {
    private char c;
    public MyChar(char c) {this.c = c;}
    public char get() {return c;}
    public void set(char c) {this.c = c;}
}

MyChar c=new MyChar('c');
MyChar d=new MyChar('d');

c = d;
d.set('e');
System.out.print(""+c.get()+d.get());
```

Variable Assignment

In C++ primitive types are copied while the programmer is free to define what assignment means for other types.

In functional programming and logic programming complex objects are always manipulated by reference since because variables cannot have their value changed, this is equivalent to copying. Thus you get efficiency + safeness!

Copy Mechanisms

Just like variable assignment the two most common approaches to parameter passing are copying and reference assignment.

A copy mechanism allows for values to be copied in or out of an procedural 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 **pass-by-value** parameter (passing) is used in C. In this mechanism the actual parameter is copied to the local variable X when calling the procedure. 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 **(pass-by) 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. Used in Ada.

We can combine these to give a **(pass-by) 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 and Ada

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);
}
```

Pass-by-reference

pass-by-reference is the other main approach to parameter passing. In this the formal parameter X becomes bound directly to the actual argument, i.e. it is just another name for the argument.

In the case of a **variable** 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. Thus Watt calls this the **definitional mechanism**.

FORTRAN and **ML** use only pass-by-reference.

What happens in:

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

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

Aliasing

The advantage of pass-by-reference is efficiency for larger data structures. The disadvantage is that aliasing between the parameters 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!

Parameter Passing in Ada

If we look at the example we can see that aliasing between the arguments in pass-by-reference causes problems only if we modify the value of one of these arguments inside the procedure.

For this reason Ada requires the programmer to indicate whether a parameter is:

- **in**: an input parameter whose value is passed into the procedure and can only be read inside the procedure
- **out**: an output parameter whose value is only passed out of the procedure but which can be read and written inside the procedure but must be written first
- **in out**: the value is passed in and out of the procedure and can be read and written inside the procedure.

```
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;

procedure init(dir: out Directory);
procedure insert(dir: in out Directory;
                newname: in Name;
                newnumber: in Number);
```

Parameter Passing in Ada (Cont.)

In Ada 83 the compiler was free to choose to use pass-by-reference or pass-by-value for input parameters, pass-by-reference or pass-by-result for output parameters and for inout parameters pass-by-reference for compound types and pass-by-value/result for primitive types.

This was not a good idea: Why?

Now Ada is uses pass-by reference for all compound types, and for primitive types pass-by-value, pass-by-result and pass-by-value/result respectively for in, out and in out parameters.

Call-by-Name

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

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.

Data Abstraction

A **procedural abstraction** allows the programmer to abstract program commands. However commands are only part of programming; there is also **data**.

Since the 1970s a focus of programming language design has been supporting programming in the large. In particular, it has focussed on providing support for **data abstraction**, i.e. ways to hide the representation of data from the rest of the program.

There are two common mechanisms for data abstraction:

- **Abstract data types**
- **Modules**

An **abstract data type** consists of a type together with a specified set of operations on the type. Access to the data type is limited to using these operations, so the representation can be hidden from the rest of the program.

ML provides the **abstype** for this purpose. According to Mitchell, CLU was the first language which provided abstract data types.

Modules

Modules generalise abstract data types. They allow the programmer to group a collection of related functions, procedures and type declarations together.

Modules also allow the programmer to specify the module **interface**: i.e., which components are **exported** by the module and which components are **hidden** in the module.

In some languages modules can be hierarchical.

Watt calls modules **packages**.

The language **Modula** developed in the late 1970s provided the basis for modern module systems such as those provided in **ML** and **Ada**.

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

C provides only a very primitive module system. A module is identified with a **file** and the programmer labels functions to indicate if they are **exported** or not.

Ada Modules

In **Ada** you use a package declaration to define a module. It has two parts: the interface or package specification and the implementation or package body.

As an example here is an **Ada** package defining an abstract type **Directory**. The interface is defined by

```
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
```

Ada Modules (Cont.)

Object-Based Programming

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 and modules look similar to objects in C++. What is the essential difference?

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);
```

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);
```

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

Summary

We have looked at

- Procedural abstractions
- Parameter passing
- Abstract data types
- Modules
- Object-based programming

Homework

- Read Chapters 5 and 6 of Watt.
- Give an example of a procedure which behaves differently if pass-by-reference and pass-by-value-result are used.
- 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 are the key differences between object classes and abstract types?
- What do you think are the differences between object-based and object-oriented programming languages?

Programming Language Concepts & Issues V

This is the last of 5 lectures looking at programming language history, issues and concepts.

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

- variables
- types and type checking
- abstraction mechanisms

In this lecture we examine

- main programming language paradigms
- possible new programming paradigms
- DNA computing

This lecture is based on Watt but also draws upon Mitchell.

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).

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 (**FORTRAN, 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 a sub-class can be passed anywhere that a super-class is expected, i.e. **subtype polymorphism** is supported
- Typically sub-classes can **inherit method definitions** from their super-classes

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 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(Satelite, Planet), orbits(Planet,sun).  
  
Planet = earth,  
Satelite = moon ? ;  
Planet = mars,  
Satelite = phobos ? ;  
Planet = mars,  
Satelite = 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.

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
- rapid prototyping

Constraint Logic Programming (CLP) application areas:

- engineering analysis and design
- financial modelling
- combinatorial optimization

New Paradigms

But these may not be the only programming language paradigms. As technology advances computer's underlying architecture may change enormously and we will need programming language that reflect this change.

Some promising new technologies include

- internet computing
- quantum computing
- DNA computing
- cellular computing

We will look at DNA computing in more detail as a representative example.

Summary

We have

- reviewed the 4 main programming paradigms
 - imperative, object-oriented, functional and logic.
- used Prolog to introduce the logic programming paradigm
- looked at DNA computing.

Homework

- Read Chapters 10, 12, 13 and 14 of Watt.
- Write a Prolog query to find out which objects orbit mars.