# Concepts in Programming Languages

Anil Madhavapeddy[1]

Department of Computer Science and Technology
University of Cambridge

CST Paper 4: 2023/24 (Easter Term)

www.cl.cam.ac.uk/teaching/2324/ConceptsPL/

[1] Acknowledgement: Many slides are based on Alan Mycroft's 2015-2023 course and previously Marcelo Fiore's 2013-14 course.

# Practicalities

- Course web page:

  `www.cl.cam.ac.uk/teaching/2324/ConceptsPL/`

  with *lecture slides*, *exercise sheet* and *reading material*.
  These slides play two roles – both “lecture notes" and “presentation material”; not every slide will be lectured in detail.
- There are various code examples on the ‘materials’ tab of the course web page.
- One exam question.
- The syllabus and course continues to evolve. I would be grateful for comments on any ‘rough edges’ in the slides, and also for views on material which is either over- or under-represented.

# Main books and papers

- ⋆ Peter van Roy. *Programming Paradigms for Dummies: What Every Programmer Should Know*.
  from Concepts, Techniques and Models of Computer Programming.
- ► J. C. Mitchell. *Concepts in programming languages*.
  Cambridge University Press, 2003.
- ► T. W. Pratt and M. V. Zelkowitz. *Programming Languages: Design and implementation* (3RD EDITION).
  Prentice Hall, 1999.
- ► R. Harper. *Practical Foundations for Programming Languages*.
  Cambridge University Press, 2013.

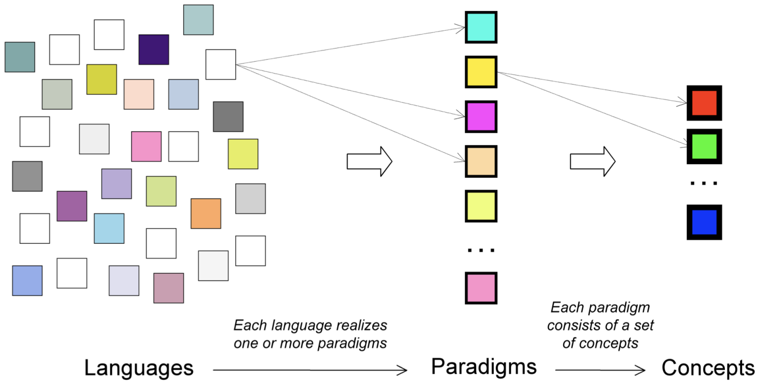
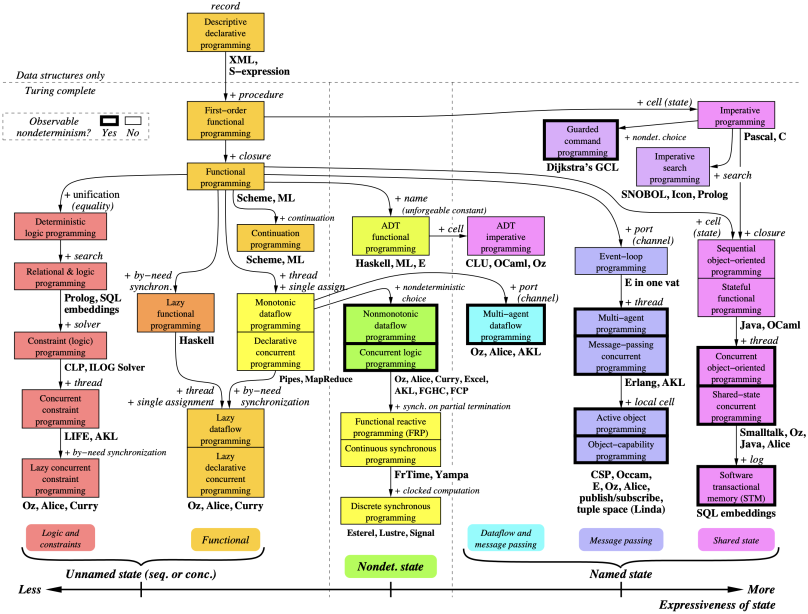


Figure: From “Programming Paradigms for Dummies”



record

Descriptive declarative programming

XML, S–expression

Data structures only

Turing complete

Observable nondeterminism? Yes No

+ procedure

First–order functional programming

+ closure

Functional programming

+ cell (state)

Imperative programming

Pascal, C

+ nondet. choice

Guarded command programming

Dijkstra's GCL

Imperative search programming

+ search

SNOBOL, Icon, Prolog

+ unification (equality)

Deterministic logic programming

+ search

Relational & logic programming

Prolog, SQL embeddings

+ solver

Constraint (logic) programming

CLP, ILOG Solver

+ thread

Concurrent constraint programming

LIFE, AKL

+ by–need synchronization

Lazy concurrent constraint programming

Oz, Alice, Curry

Scheme, ML

+ continuation

Continuation programming

Scheme, ML

+ by–need synchron.

Lazy functional programming

Haskell

+ thread

+ single assign.

Monotonic dataflow programming

Declarative concurrent programming

Pipes, MapReduce

+ thread

+ single assignment

+ by–need synchronization

Lazy dataflow programming

Lazy declarative concurrent programming

Oz, Alice, Curry

+ name (unforgeable constant)

ADT functional programming

Haskell, ML, E

+ cell

ADT imperative programming

CLU, OCaml, Oz

+ nondeterministic choice

Nonmonotonic dataflow programming

Concurrent logic programming

Oz, Alice, Curry, Excel, AKL, FGHC, FCP

+ synch. on partial termination

Functional reactive programming (FRP)

Continuous synchronous programming

FrTime, Yampa

+ clocked computation

Discrete synchronous programming

Esterel, Lustre, Signal

+ port (channel)

Multi–agent dataflow programming

Oz, Alice, AKL

+ port (channel)

Event–loop programming

E in one vat

+ thread

Multi–agent programming

Message–passing concurrent programming

Erlang, AKL

+ local cell

Active object programming

Object–capability programming

CSP, Occam, E, Oz, Alice, publish/subscribe, tuple space (Linda)

+ cell (state)

+ closure

Sequential object–oriented programming

Stateful functional programming

Java, OCaml

+ thread

Concurrent object–oriented programming

Shared–state concurrent programming

Smalltalk, Oz, Java, Alice

+ log

Software transactional memory (STM)

SQL embeddings

Logic and constraints

Functional

Nondet. state

Dataflow and message passing

Message passing

Shared state

Unnamed state (seq. or conc.)

Named state

Less

More

Expressiveness of state

Figure: From “Programming Paradigms for Dummies”

# Context:

## so many programming languages

Peter J. Landin: “The Next 700 Programming Languages”, CACM *(published in 1966!)*.

Some programming-language ‘family trees’ (too big for slide):
`http://www.levenez.com/lang/`
`http://rigaux.org/language-study/diagram.html`

Wikipedia’s historical summary page: `https://en.wikipedia.org/wiki/History_of_programming_languages`

Plan of this course: pick out interesting programming-language concepts and major evolutionary trends.

# Topics

I. Introduction and motivation.

Part A: Meet the ancestors

II. The first *procedural* language: FORTRAN (1954–58).

III. The first *declarative* language: LISP (1958–62).

IV. *Block-structured* languages: Algol (1958–68), Pascal (1970).

V. *Object-oriented* languages: Simula (1964–67), Smalltalk (1972).

Part B: Types and related ideas

VI. *Types* in programming languages: ML, Java.

VII. *Scripting Languages*: JavaScript.

VIII. *Data abstraction* and *modularity*: OCaml Modules.

Part C: Distributed concurrency, Java lambdas, Scala, Monads

IX. Languages for *concurrency and parallelism*.

X. Functional-style programming meets object-orientation.

XI. Miscellaneous concepts: Monads, GADTs.

XII. Up-and-coming ideas: Rust, WebAssembly.

# ∼ *Topic I* ∼

Introduction and motivation

# Goals

- Critical *thinking* about programming languages.
  - What is a programming language!?
  - Who defines it?
  - How can it be changed?
- *Study* programming languages.
  - Be familiar with basic language *concepts*.
  - Appreciate trade-offs in language *design*.
- Trace *history*, appreciate *evolution* and diversity of *ideas*.
- Be prepared for new programming *methods*, *paradigms*.

# Why study programming languages?

- To improve the ability to develop effective algorithms.
- To habitualise the use of familiar languages.
- To grow a vocabulary of useful programming constructs.
- To allow a better choice of programming language.
- To make it easier to learn or design a new language.
- To simulate useful features in languages that lack them.
- To adapt to new language technology as it appears.

# What makes a good language?

- Clarity, simplicity, and unity.
- Orthogonality.
- Naturalness for the application.
- Support of abstraction.
- Ease of program verification.
- Programming environments.
- Portability of programs.
- Cost of use.
  - Cost of execution.
  - Cost of program translation.
  - Cost of program creation, testing, and use.
  - Cost of program maintenance.

# What makes a language successful?

- Expressive power.
- Ease of use for the novice.
- Ease of implementation.
- Standardisation.
- Many useful libraries.
- Excellent compilers (including open-source)
- Economics, patronage, and inertia.

Note the recent trend of big companies to create/control their own languages: C# (Microsoft), Hack (Facebook), Go (Google), Objective-C/Swift (Apple), Rust (Mozilla) and perhaps even Python (Dropbox hired Guido van Rossum).

- ? Why are there so many languages?
  - Evolution.
  - Special purposes.
  - No one language is good at expressing all programming styles.
  - Personal preference.
- ? What makes languages evolve?
  - Changes in hardware or implementation platform
  - Changes in attitudes to safety and risk
  - New ideas from academia or industry

# Motivating purpose and language design

A specific purpose or motivating application provides *focus* for language design—what features to include and (harder!) what to leave out. E.g.

- Lisp: symbolic computation, automated reasoning
- FP: functional programming, algebraic laws
- BCPL: compiler writing
- Simula: simulation
- C: systems programming [Unix]
- ML: theorem proving
- Smalltalk: Dynabook [1970-era tablet computer]
- Clu, SML Modules: modular programming
- C++: object orientation
- Java, JavaScript: Internet applications

# Program execution model

Good language design presents *abstract machine*.

- Fortran: Flat register machine; memory arranged as linear array
- Lisp: cons cells, read-eval-print loop
- Algol family: stack of activation records; heap storage
- BCPL, C: underlying machine + abstractions
- Simula: Object references
- FP, ML: functions are basic control structure
- Smalltalk: objects and methods, communicating by messages
- Java: Java virtual machine

# Classification of programming languages

See `en.wikipedia.org/wiki/Programming_paradigm` for more detail:

- Imperative

| | |
|---|---|
| procedural | C, Ada, **Pascal**, **Algol**, **Fortran**, . . . |
| object-oriented | Scala, C#, Java, **Smalltalk**, **SIMULA**, . . . |
| scripting | Perl, Python, PHP, JavaScript, . . . |

- Declarative

| | |
|---|---|
| functional | Haskell, SML, Lisp, Scheme, . . . |
| logic | Prolog |
| dataflow | Id, Val |
| constraint-based | spreadsheets |
| template-based | XSLT |

# A Standard Model of Languages

A large number of modern languages – from Java and C# to Python and JavaScript to Racket and OCaml – share a common semantic core.

- variables are lexically scoped
- scope can be nested
- evaluation is eager
- evaluation is sequential (per “thread”)
- variables are mutable, but first-order
- structures are mutable and first-class
- functions can be higher-order and close over lexical bindings
- memory is managed automatically

See https://blog.brownplt.org/2024/04/12/behavior-misconceptions.html

# Language standardisation

Consider: `int i; i = (1 && 2) + 3 ;`

**?** Is it valid C code? If so, what's the value of `i`?

**?** How do we answer such questions!?

- ! Read the reference manual (ISO C Standard).
- ! Try it and see!

Other languages may have *informal* standards (defined by a particular implementation but what do we do if the implementation is improved?) or *proprietary* standards. A notable exception is WebAssembly, which was designed from the start with an *executable specification* written in OCaml.

# Language-standards issues

Timeliness. When do we standardise a language?

Conformance. What does it mean for a program to adhere to a standard and for a compiler to compile a standard?

*Ambiguity and freedom to optimise – Machine dependence – Undefined behaviour.*

*A language standard is a* treaty *setting out the rights and obligations of the programmer and the implementer.*

Obsolescence. When does a standard age and how does it get modified?

*Deprecated features.*

# Language standards: unintended mis-specification

- *Function types in Algol 60*, see later.
- In language PL/1 the type `DEC(p,q)` meant a decimal number of `p` digits (at most 15) with `q` digits after the decimal point, so `9`, `8`, `3` all had type `DEC(1,0)`.
  Division was defined to so that `8/3` was `DEC(15,14)` with value `2.66666666666666`.
  But addition was defined so that adding these two was also of type `DEC(15,14)`, which meant that `9 + 8/3` gave `11.66666666666666`, which didn't fit. This gave either *overflow* or the wrong answer of `1.66666666666666`.
- A more recent example is C++11's "out of thin air" behaviour, whereby the ISO specification allows the value 42 to appear as the result of a (concurrent) program only involving assignments of 0 and 1.

Argh! Be careful how you specify a language.

# Ultra-brief history

| | |
|---|---|
| 1951–55: | Experimental use of expression compilers. |
| 1956–60: | **Fortran**, COBOL, **Lisp**, **Algol** 60. |
| 1961–65: | APL notation, Algol 60 (revised), SNOBOL, CPL. |
| 1966–70: | APL, SNOBOL 4, Fortran 66, BASIC, **SIMULA**, Algol 68, Algol-W, BCPL. |
| 1971–75: | **Pascal**, PL/1 (Standard), C, Scheme, Prolog. |
| 1976–80: | **Smalltalk**, Ada, Fortran 77, ML. |
| 1981–85: | Smalltalk-80, Prolog, Ada 83, Caml. |
| 1986–90: | C++, SML, Haskell. |
| 1991–95: | Ada 95, TCL, Perl. |
| 1996–2000: | **OCaml**, Java, JavaScript |
| 2000–05: | C#, Python, Ruby, Scala. |
| 1990– : | Open/MP, MPI, Posix threads, Erlang, X10, MapReduce, and many more. |

For more information:

`en.wikipedia.org/wiki/History_of_programming_languages`

# Background Concepts

Modern languages *generally* have four storage areas (segments):

- code – usually read-only or execute-only
- static (top-level) variables
- stack – for return addresses, local variables, and exceptions
- heap

Notes:

- Operating systems also typically provide OS-level *threads* which might execute on a separate CPU core. Each OS thread comes with its own stack.
- Prolog doesn't quite fit this model as its (WAM) virtual machine tends to have: a stack for calls, a heap for allocations, and also a *trail stack* which manages backtracking.

# Digression: Threads and Heaps

There is a choice here (we will talk a bit more about this later):

- a single heap allocator
- one allocator per thread

The former typically requires locks in the allocator (expensive), so the second is generally preferred. So how does the allocator find its next-allocation pointer?

- Another storage area: *thread-local statics*, `thread_local` in C++11; or equivalently:
- A mechanism to find the thread-id for the current thread and then use it to index a vector to give per-thread storage areas.

If we have a shared-memory multiprocessor, then we can have a single heap, but separate efficient per-thread allocators.

On distributed architectures we might have multiple heaps, perhaps not even cheaply addressable from each other . . . .

# Passing arguments to functions

We use the stack (or registers) for this. But there a separate issue of mechanism:

- call-by-value: callee parameters are effectively local variables initialised by values of arguments. [default in C, Java, OCaml],
- call-by-reference: callee parameter is an alias to the location of the caller argument. [C++ form `int &x`, Pascal `var`, default in Fortran]
- call-by-value-result: caller argument location is read to initialise local variable in callee (like call-by-value) but also its final value is copied back.

Exercise: write a program which prints 0, 1 or 2 depending on which of these is used.

- Some authors use ‘pass-by-value’ etc. rather than ‘call-by-value’.

# Passing arguments to functions (2)

Does Java use call-by-vaue or call-by-reference?

- call-by-value (for values of all types).

But aren't objects passed by reference in Java?

- No. An *object reference* (a pointer) is passed by value. This has a similar effect to passing an object by reference in C++.

Stackoverflow attracts questions on this.

# Passing arguments to functions (3)

Less common nowadays are the following mechanisms:

- call-by-name: a closure (also called a ‘thunk’) is passed and evaluating it gives the value or location of the argument [Algol 60].
- call-by-need: a closure is passed and evaluating it gives the value of the argument, but this value is cached and returned on subsequent calls. Also called ‘lazy evaluation’ [Haskell].
- call-by-text: the text (or AST tree) of the argument is passed and the caller uses `eval` when it needs the value [special forms in Lisp]

# Implementing closures

Consider OCaml code like:

```
let f x = fun y -> x + y + 5
let g = f 10 and h = f 20
in g 1 + h 2
```

The result of `f` is a ***closure*** – a structure (often heap-allocated) which contains the values of the free variables (here just `x`) of `fun y -> x+y+5`, along with a pointer to code of the form

```
LDfree 1; LDarg 1; PLUS; LDconst 5; PLUS; RETURN
```

Note that (in the code above) it's vital to ***copy*** `x` into the closure as it will have been deallocated before `g` and `h` are called.

# C++ lambdas: variable capture (value or reference)

Whether free variables are bound by value or by reference depends on the language; C++ provides both:

```
// LLVM use: c++ --std=c++14 to get lambda support
#include <iostream>
int main()
{  int a=0,b=0;
   // C++ use of ‘[]’ (lambda)needs to know how
   //              free variables are bound:
   auto f = [a,&b](int x) ->int { return x+a+b; };
   a++; b+=10;
   std::cout << "f(42)=" << f(42) << std::endl;
   // gives "f(42)=52" -- think why...
   return 0;  }
```

Notes:

- `auto f = [](int x) ->int { return x+a+b;};` gives “error: variable ’a’ cannot be implicitly captured in a lambda with no capture-default specified.”
- C++ calls the type of `f` a *functor*, a class which overloads `operator()`.

# And what about lambdas in Java?

Some neat wording says that local variables used freely by a lambda must be “effectively final”. This forbids code similar to the previous slide by banning both assignments in

```
a++; b+=10;
```

This helps avoid subtle bugs, e.g. where the value of variable is copied into a closure and the programmer is surprised that subsequent updates to the variable have no effect on the closure.

# Programming-language phrases – reminder

[This classification arose in the Algol 60 design.]

- *Expressions*. A syntactic entity that may be evaluated to determine its value.
- *Statement*. A command that alters the state of the machine in some explicit way.
- *Declaration*. A syntactic entity that introduces a new identifier, often specifying one or more attributes.

# ∼ *Part A* ∼

## Meet the ancestors

Santayana 1906: “Those who cannot remember the past are condemned to repeat it.”

# ~ *Topic II* ~

## FORTRAN: A simple procedural language

**Further reading:**

- ***The History of FORTRAN I, II, and III*** by J. Backus. In *History of Programming Languages* by R. L. Wexelblat.

  Academic Press, 1981.

# FORTRAN = FORmula TRANslator (1957)

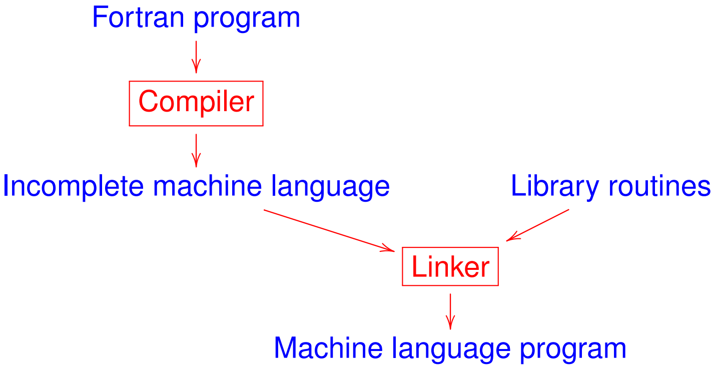
- Developed (1950s) by an IBM team led by John Backus:

  “As far as we were aware, we simply made up the language as we went along. We did not regard *language design* as a difficult problem, merely a simple prelude to the real problem: designing a compiler which could produce efficient programs.”
- The first high-level programming language to become widely used. At the time the utility of any high-level language was open to question(!), and complaints focused on efficiency of generated code. This heavily influenced the design, orienting it towards execution efficiency.
- Standards: 1966, 1977 (FORTRAN 77), 1990 (Fortran 90, spelling change), . . . , 2018 (Fortran 2018).
- Remains main language for scientific computing.
- Easier for a compiler to optimise than C.

# Overview: Compilation

Fortran program = main program + subprograms

- Each is *compiled separately* from all others. (Originally no support for cross-module checking, still really true for C and C++.)
- Translated programs are *linked* into final executable form.



# Overview: Data types and storage allocation

- Numerics: Integer, real, complex, double-precision real.
- Boolean. called logical
- Arrays. of fixed declared length
- Character strings. of fixed declared length
- Files.
- Fortran 90 added ‘derived data types’ (like C structs).

Allocation:

- Originally all *storage was allocated statically* before program execution, even local variables (as early Fortran lacked recursion—machines often lacked the index registers needed for a cheap stack—and we didn’t realise how useful stacks and recursion would be!).
- Modern Fortran has recursion and heap-allocated storage.

# Overview

## Control structures

- FORTRAN 66

  Relied heavily on statement labels and `GOTO` statements, but did have `DO` (for) loops.
- FORTRAN 77

  Added some modern control structures (*e.g.*, if-then-else blocks), but `WHILE` loops and recursion had to wait for Fortran 90.
- Fortran 2008

  Support for concurrency and objects

# Example (Fortran 77)

```
      PROGRAM MAIN
        PARAMETER (MaXsIz=99)
        REAL A(mAxSiZ)
  10    READ (5,100,END=999) K
 100    FORMAT(I5)
          IF (K.LE.0 .OR. K.GT.MAXSIZ) STOP
          READ *,(A(I),I=1,K)
          PRINT *,(A(I),I=1,K)
          PRINT *,’SUM=’,SUM(A,K)
          GO TO 10
 999    PRINT *, ’All Done’
        STOP
        END

C SUMMATION SUBPROGRAM
      FUNCTION SUM(V,N)
        REAL V(N)
        SUM = 0.0
        DO 20 I = 1,N
          SUM = SUM + V(I)
   20     CONTINUE
        RETURN
        END
```

# Example

## Commentary

- Originally columns and lines were relevant, and blanks and upper/lower case are ignored except in strings. Fortran 90 added free-form and forbade blanks in identifiers (use the `.f90` file extension on Linux).
- Variable names are from 1 to 6 characters long (31 since Fortran 90), letters, digits, underscores only.
- Variables need not be declared: implicit naming convention determines their type, hence the old joke “GOD is REAL (unless declared INTEGER)”; good programming style uses `IMPLICIT NONE` to disable this.
- Programmer-defined constants (`PARAMETER`)
- Arrays: subscript ranges can be declared as $(lwb : upb)$ with $(size)$ meaning $(1 : size)$.

- Data formats for I/O.
- Historically functions are compiled separately from the main program with no consistency checks. Failure may arise (either at link time or execution time) when subprograms are linked with main program.
  Fortran 90 provides a module system.
- Function parameters are uniformly transmitted by reference (like C++ '`&`' types).
  But Fortran 90 provided INTENT(IN) and INTENT(OUT) type qualifiers and Fortran 2003 added pass-by-value for C interoperability.
- Traditionally all allocation is done statically.
  But Fortran 90 provides dynamic allocation and recursion.
- A value is returned in a Fortran function by assigning a value to the name of a function.

# Program consistency checks

- *Static type checking* is used in Fortran, but the checking is traditionally incomplete.
- Many language features, including arguments in subprogram calls and the use of `COMMON` blocks, were not statically checked (in part because subprograms are compiled independently).
- Constructs that could not be statically checked were often left unchecked at run time (e.g. array bounds). (An early preference for speed over ease-of-bug-finding still visible in languages like C.)
- Fortran 90 added a `MODULE` system with `INTERFACE`s which enables checking across separately compiled subprograms.

# Fortran lives!

- Fortran is one of the first languages, and the only early language still in mainstream use (LISP dialects also survive, e.g. Scheme).
- Lots of CS people will tell you about all the diseases of Fortran based on hearsay about Fortran 66, or Fortran 77.
- Modern Fortran still admits (most) old code for backwards compatibility, but also has most of the things you expect in a modern language (objects, modules, dynamic allocation, parallel constructs). There's even a proposal for "units of measure" to augment types.
  (Language evolution is preferable to extinction!)
- Don't be put off by the syntax—or what ill-informed people say.

# ∼ *Topic III* ∼

## LISP: functions, recursion, and lists

# LISP = LISt Processing (circa 1960)

- Developed in the late 1950s and early 1960s by a team led by John McCarthy at MIT. McCarthy described LISP as a “a scheme for representing the *partial recursive functions* of a certain class of symbolic expressions”.
- Motivating problems: Symbolic computation (*symbolic differentiation*), logic (*advice taker*), experimental programming.
- Software embedding LISP: Emacs (text editor), GTK (Linux graphical toolkit), Sawfish (window manager), GnuCash (accounting software).
- Current dialects: Common Lisp, Scheme, Clojure, Racket. Common Lisp is ‘most traditional’, Clojure is implemented on JVM.

# Some contributions of LISP

- LISP is an *expression-based* language.
  LISP introduced the idea of *conditional expressions*.
- Lists – dynamic storage allocation, *hd* (CAR) and *tl* (CDR).
- Recursive functions.
- Garbage collection.
- Programs as data.
- Self-definitional interpreter (LISP interpreter explained as a LISP program).

The core of LISP is pure functional, but *impure* (side-effecting) constructs (such as SETQ, RPLACA, RPLACD) were there from the start.

# Overview

- Values in LISP are either *atoms*, e.g. `X`, `FOO`, `NIL`, or *cons cells* which contain two values.
  (Numbers are also atoms, but only literal atoms above can be used as variables below.)
- A LISP program is just a special case of a LISP value known as an *S-expression*. An S-expression is either an *atom* or a NIL-terminated list of S-expressions (syntactically written in parentheses and separated by spaces), e.g. `(FOO ((1 2) (3)) NIL (4 X 5))`.
- So right from the start programs are just data, so we can construct a value and then execute it as a program.
- LISP is a *dynamically typed* programming language, so heterogeneous lists like the above are fine.

- Programs represented as S-expressions are *evaluated* to give values, treating atoms as variables to be looked up in an environment, and lists as a function (the first element of the list) to be called along with its arguments (the rest of the list).

  **Example:**

  ```
  (APPEND (QUOTE (FOO 1 Y)) (CONS 3 (CONS 'Z NIL)))
  ```

  evaluates to `(FOO 1 Y 3 Z)`.

  Note: the functions `CONS` and `APPEND` behaves as in ML, and the function `QUOTE` returns its argument unevaluated. Numbers and the atoms `NIL` and `T` (also used as booleans) evaluate to themselves.

- To ease typing LISP programs, `(QUOTE` *e*`)` can be abbreviated `'`*e*, see `'Z` above.
  This is done as part of the `READ` function which reads values (which of course can also be programs).

? How does one recognise a LISP program?

```
( defvar x 1 )
( defun g(z) (+ x z) )
( defun f(y)
    ( + ( g y )
        ( let
            ( (x y) )
            (
              g x )
          ) ) )
( f (+ x 1) )
```

```
val x = 1 ;
fun g(z) = x + z ;
fun f(y)
  = g(y) +
    let
      val x = y
    in
      g(x)
    end ;
f(x+1) ;
```

! It is full of brackets (“Lots of Irritating Silly Parentheses”)!

# Core LISP primitives

The following primitives give enough (Turing) power to construct any LISP function, along with top-level `DEFUN`

- `CONS`, `CAR`, `CDR`: cons, hd, tl.
    - `(car (cons x y)) => x`
    - `(cdr (cons x y)) => y`
- `CONSP`, `ATOM`: boolean tests for being a cons cell or atom.
- `EQ`: boolean equality test for equal atoms (but beware using it on large numbers which may be boxed, cf. Java boxing).
- `QUOTE`: return argument unevaluated.
- `COND`: conditional expression.

Example:

```
(defun subst (x y z)
    (cond  ((atom z)  (cond ((eq z y) x) (T z)))
           (T         (cons (subst x y (car z))
                            (subst x y (cdr z))))
    )
)
```

# Static and dynamic scoping (or binding)

Consider:

```
(defun main () (f 1))
(defun f (x) (g 2 (lambda () x)))
(defun g (x myfn) (apply myfn ()))
```

The question is whether the interpreter looks up the free variable `x` of the lambda in its static scope (getting 1), or in the (dynamic) scope at the time of the call (getting 2).

Historically, LISP was a dynamically scoped language; [Sethi pp.162] writes: when the initial implementation of Lisp was found to use dynamic scope, its designer, McCarthy[1981], “regarded this difficulty as just a bug”.

Newer dialects of LISP, such as Common Lisp and Scheme, use static scoping for this situation. (Common Lisp allows `defvar` to mark a variable as using dynamic binding.)

# Programs as data

- One feature that sets LISP apart from many other languages is that it is possible for a program to build a data structure that represents an expression and then evaluates the expression as if it were written as part of the program. This is done with the function `EVAL`.
- *But* the environment used is that of the caller of `EVAL` so problems can arise if the expression being evaluated contains free variables (see ‘call-by-text’ below).
- McCarthy showed how a *self-definitional* (or *meta-circular*) interpreter for LISP could be written in LISP.

# Parameter passing in LISP

- Function parameters are transmitted either all *by value* or all *by text* (unevaluated expressions); only built-in functions (such as `QUOTE`, `LAMBDA`, `COND`) should really use pass-by-text. *Why: because we need a special variant of `EVAL` to evaluate the arguments to `COND` in the calling environment, and similarly need to capture the free variable of a `LAMBDA` in the environment of the caller*.
- The *actual parameters* in a function call are always expressions, represented as list structures.
- Note that call-by-text (using either a special form, or explicit programmer use of `QUOTE`, and with `EVAL` to get the value of an argument) *resembles* call-by-name (using `LAMBDA` to pass an unevaluated expression, and with `APPLY` to get the value of an argument), but is only equivalent if the `EVAL` can evaluate the argument in the environment of the function call!

# Dangers of `eval`

In LISP, I can write: `(eval '(+ x 1))`

If you've been to IB Semantics then your first question should be "where do I lookup the value of `x`?" This is a real issue – should the following give 11 or 12?

```
(defvar x 1)
(defvar e '(add x 10))
(defun f (x) (eval e))
(f 2)
```

Unless you're very careful any use of `eval` involving user input produces a security hole. Consider: `(eval (read))` and inputs like `(fire-all-missiles)` or `(+ 1 top-security-code)`.

This applies to *any* language with a one-argument `eval`. When using Python (where `eval` takes a string), use the (sadly optional) additional environment-specifying parameters.

# ~ *Topic IV* ~

Block-structured procedural languages
Algol, Pascal and Ada

**Sample Pascal program:** The keyword `var` indicates call-by-reference.

```
program main;
begin
  function f(var x, y: integer): integer;
    begin
      x := 2;
      y := 1;
      if x = 1 then f := 3 else f:= 4
    end;

  var z: integer;
  z := 0;
  writeln( f(z,z) )
end
```

Note the `begin`-`end` brackets, later replaced by `{`-`}` brackets in C and other descendant languages but still used in OCaml.

## Evolution of parameter-passing mechanisms

- Algol 60 supported call-by-value and call-by-name. Call-by-name was found to be inefficient and not popular with programmers – despite its lambda-calculus heritage of beta-reduction.
- A parameter in Pascal is normally passed by value. It is passed by reference, however, if the keyword `var` appears before the declaration of the formal parameter.

```
procedure proc(x: Integer; var y: Real);
```

- Ada supports three kinds of parameters:
  1. `in` parameters, corresponding to value parameters;
  2. `out` parameters, corresponding to just the copy-out phase of call-by-value/result; and
  3. `in out` parameters, corresponding to either reference parameters or value/result parameters, at the discretion of the implementation.
- Beware of passing arrays or large structures by value!

# Algol

HAD A MAJOR EFFECT ON LANGUAGE DESIGN

- The Algol-like programming languages evolved in parallel with the LISP family of languages, beginning with Algol 58 and Algol 60 in the late 1950s.
- The main characteristics of the Algol family are:
  - the familiar semicolon-separated sequence of statements,
  - block structure,
  - functions and procedures, and
  - static typing.

ALGOL IS DEAD BUT ITS MANY DESCENDANTS LIVE ON!

- Ada, C, C++, Java etc.

# Algol innovations

- Use of BNF syntax description.
- Block structure.
- Scope rules for local variables.
- Dynamic lifetimes for variables.
- Nested `if-then-else` expressions and statements.
- Recursive subroutines.
- Call-by-value and call-by-name arguments.
- Explicit type declarations for variables.
- Static typing.
- Arrays with dynamic bounds.
- Functions (closures) as arguments – but entertainingly not “closures as results” as Algol 60 had no heap and so closures were stack-allocated.

# Algol 60

## Features

- Simple statement-oriented syntax.
- Block structure.
  - *blocks* contain declarations and executable statements delimited by begin and end markers.
  - May be nested, declaration visibility: scoping follows lambda calculus (Algol had no objects so no richer O-O visibility from inheritance as well as nesting).
- Recursive functions and stack storage allocation.
- Fewer ad-hoc restrictions than previous languages (*e.g.*, general expressions inside array indices, procedures that could be called with procedure parameters).
- A primitive *static type system*, later improved in Algol 68 and Pascal.

# Algol 60

## Some trouble-spots

- The Algol 60 type discipline had some shortcomings. For instance:
  - The type of a procedure parameter to a procedure does not include the types of parameters.

    ```
    procedure myapply(p, x)
      procedure p; integer x;
      begin p(x);
      end;
    ```

  - An array parameter to a procedure is given type array, without array bounds.
- Algol 60 was designed around two parameter-passing mechanisms, *call-by-name* and *call-by-value*.
  Call-by-name interacts badly with side effects; call-by-value is expensive for arrays.

# Algol 68

- Algol 68 contributed a *regular, systematic type system*. The types (referred to as *modes* in Algol 68) are either *primitive* (`int`, `real`, `complex`, `bool`, `char`, `string`, `bits`, `bytes`, `semaphore`, `format`, `file`) or *compound* (`array`, `struct`, `union`, `procedure`, `set`, `pointer`).
  Type constructors could be combined without restriction – a more systematic type system than previous languages.
- Algol 68 used a *stack* for local variables and *heap* storage. Heap data are explicitly allocated, and are reclaimed by *garbage collection*.
- Algol 68 parameter passing is by value, with pass-by-reference accomplished by pointer types. (This is essentially the same design as that adopted in C.)
- Too complicated (both linguistically and to compile) for its time.

# Pascal (1970)

- Pascal is a *quasi-strong, statically typed* programming language.
  An important contribution of the Pascal *type system* is the rich set of data-structuring concepts: *e.g.* enumerations, subranges, records, variant records, sets, sequential files.
- The Pascal *type system* is more expressive than the Algol 60 one (repairing some of its loopholes), and simpler and more limited than the Algol 68 one (eliminating some of the compilation difficulties).
- Pascal was the first language to propose index checking. The index type (typically a sub-range of integer) of an array is part of its type.
- Pascal lives on (somewhat) as the Delphi language.

# Pascal variant records

*Variant records* have a part common to all records of that type, and a variable part, specific to some subset of the records.

```
type kind = (unary, binary) ;
type                          | type
  UBtree = record             |     ubtree =
    value: integer ;          |          int * ubaux
    case k: kind of           | and ubaux =
      unary: ^UBtree ;        |     Unary of ubtree option
      binary: record          |   | Binary of
        left: ^UBtree ;       |       ubtree option *
        right: ^UBtree        |       ubtree option;
      end
end ;
```

Note the use of `option` to encode `NULL` in the OCaml code on the right.

Pascal variant records introduced *weaknesses* into its type system.

- Compilers do not usually check that the value in the tag field is consistent with the state of the record.
- Tag fields are optional. If omitted, no checking is possible at run time to determine which variant is present when a selection is made of a field in a variant.

C still provides this model with `struct` and `union`. Modern languages provide safe constructs instead (think how a compiler can check for appropriate use):

- OCaml provides `type` and `match` to express similar ideas. In essence the constructor names provide the discriminator `k` but this is limited to being the first component of the record.
- Object-oriented languages provide *subclassing* to capture variants of a class.

See also the 'expression problem' discussion (slide 183).

# ~ *Topic V* ~

## Object-oriented languages: concepts and origins SIMULA and Smalltalk

**Further reading for the interested:**

- Alan Kay's "The Early History Of Smalltalk"
  `http://worrydream.com/EarlyHistoryOfSmalltalk/`

# Basic concepts in object-oriented languages

Four main language concepts for object-oriented languages:

1. Dynamic lookup.
2. Abstraction.
3. Subtyping.
4. Inheritance.

# Dynamic lookup

- *Dynamic lookup*[2] means that when a method of an object is called, the method body to be executed is selected dynamically, at run time, according to the implementation of the object that receives the message (as in Java or C++ *virtual* methods).
- For the idea of *multiple dispatch* (not on the course), rather than the Java-style (or single) dispatch, see

  `http://en.wikipedia.org/wiki/Multiple_dispatch`

# Abstraction

- *Abstraction* means that implementation details are hidden inside a program unit with a specific interface. For objects, the interface usually consists of a set of methods that manipulate hidden data.

[2] Also called ‘dynamic dispatch’ and occasionally ‘dynamic binding’ (but avoid the latter term as ‘dynamic scoping’ is quite a different concept).

# Subtyping

- *Subtyping* is a relation on types that allows values of one type to be used in place of values of another. Specifically, if an object `b` has all the functionality of another object `a`, then we may use `b` in any context expecting `a`.
- The basic principle associated with subtyping is *substitutivity*: If `B` is a subtype of `A`, then any expression of type `B` may be used without type error in any context that requires an expression of type `A`.

Subtlety: just because the type system allows us to use a `B` where an `A` is expected, doesn't generally mean such a `B` behaves in the way that we expect of `A` objects. See Behavioural Subtyping (two slides on).

# Inheritance

- *Inheritance* is the ability to reuse the definition of one kind of object to define another kind of object.
- The importance of inheritance is that it saves the effort of duplicating (or reading duplicated) code and that, when one class is implemented by inheriting from another, changes to one affect the other. This has a significant impact on code maintenance and modification.

NB: although Java treats subtyping and inheritance as synonyms, it is quite possible to have languages which have one but not the other.

- A language might reasonably see `int` as a subtype of `double` but there isn't any easy idea of inheritance here.
- One can simulate inheritance (avoiding code duplication) with C-style `#include`, but this doesn't add subtyping to C.
- Cook et al. write in *Inheritance is not subtyping*: "Subtyping is a relation on interfaces, inheritance is a relation on implementations"

# Behavioural Subtyping – ‘good subclassing’

Consider two classes for storing collections of integers:

```
class MyBag {
  protected ArrayList<Integer> xs;
  public void add(Integer x) { xs.add(x); }
  public int size() { return xs.size(); }
  // other methods here...
}
class MySet extends MyBag
  @override
  public void add(Integer x) { if (!xs.contains(x))
                                   xs.add(x); }
}
```

Questions: Is `MySet` a subclass of `MyBag`? A subtype?

Java says ‘yes’. But should it?

# Behavioural Subtyping – ‘good subclassing’ (2)

It shouldn’t really be a subtype, because it violates *behavioural subtyping* – members of a subtype should have the same behaviour as the members of the supertype. Consider:

```
int foo(MyBag b) { int n = b.size();
                   b.add(42);
                   return b.size() - n; }
```

For every `MyBag` this returns 1. However if I pass it a `MySet` already containing 42, then it returns 0.

So `MySet` shouldn’t be a subtype of `MyBag` as its values behave differently, e.g. results of `foo`. So properties and invariants of `MyBag` which I’ve proved to hold may no longer hold when an `MyBag` object is actually an upcasted `MySet`!!! We say that `MySet` is *not* a *behavioural subtype* of `MyBag`.

[Liskov and Wing’s paper “A Behavioral Notion of Subtyping” (1994) gives more details.]

# Behavioural Subtyping – ‘good subclassing’ (3)

OK, so but perhaps it works the other way? Perhaps `MyBag` is a behavioural subtype of `MySet`?

- Yes? `foo` gives 0 or 1 for supertype and 1 for subtype.
- No: e.g. `add` is idempotent for `MySet` but not for `MyBag`.
- It’s a good idea to document the invariants classes expect their subclasses to maintain.

OK, so how do I code it?

- Good design would have `MyBag` and `MySet` separately inheriting from a common superclass (or separately implementing a common interface).

Java 16 adds *sealed classes* to allow programmers to specify which other classes can extend them (this is more powerful than `final` which is all-or-none). This is useful to forbid a (rogue) distant piece of code from extending `class C` to give behaviours which the coder of `class C` and its planned subclasses did not anticipate.

# History of object-oriented languages

## SIMULA and Smalltalk

- Objects were invented in the design of SIMULA and refined in the evolution of Smalltalk.
- SIMULA: The first object-oriented language.
    - Extremely influential as the first language with classes, objects, dynamic lookup, subtyping, and inheritance. Based on Algol 60.
    - Originally designed for the purpose of *simulation* by Dahl and Nygaard at the Norwegian Computing Center, Oslo,
- Smalltalk: A dynamically typed object-oriented language. Many object-oriented ideas originated or were popularised by the Smalltalk group, which built on Alan Kay's then-futuristic idea of the Dynabook (Wikipedia shows Kay's 1972 sketch of a modern tablet computer).

- A generic event-based simulation program (pseudo-code):

```
Q := make_queue(initial_event);
repeat
  select event e from Q
  simulate event e
  place all events generated by e on Q
until Q is empty
```

naturally requires:

  - A data structure that may contain a variety of kinds of events. ⇝ subtyping
  - The selection of the simulation operation according to the kind of event being processed. ⇝ dynamic lookup
  - Ways in which to structure the implementation of related kinds of events. ⇝ inheritance

# SIMULA: Object-oriented features

- *Objects*: A SIMULA object is a (heap-allocated) activation record produced by call to a class.
- *Classes*: A SIMULA class is a procedure that returns a pointer to its activation record. The body of a class may initialise the objects it creates.
- *Dynamic lookup*: Operations on an object are selected from the activation record of that object.
- *Abstraction*: Hiding was not provided in SIMULA 67; it was added later and inspired the C++ and Java designs.
- *Subtyping*: Objects are typed according to the classes that create them. Subtyping is determined by class hierarchy.
- *Inheritance*: A SIMULA class could be defined, by *class prefixing*, to extend an already-defined class including the ability to override parts of the class in a subclass.

# SIMULA: Sample code

```
CLASS POINT(X,Y); REAL X, Y;
  COMMENT***CARTESIAN REPRESENTATION
BEGIN
  BOOLEAN PROCEDURE EQUALS(P); REF(POINT) P;
    IF P =/= NONE THEN
      EQUALS := ABS(X-P.X) + ABS(Y-P.Y) < 0.00001;
  REAL PROCEDURE DISTANCE(P); REF(POINT) P;
    IF P == NONE THEN ERROR ELSE
      DISTANCE := SQRT( (X-P.X)**2 + (Y-P.Y)**2 );
END***POINT***
```

```
CLASS LINE(A,B,C); REAL A,B,C;
  COMMENT***Ax+By+C=0 REPRESENTATION
BEGIN
  BOOLEAN PROCEDURE PARALLELTO(L); REF(LINE) L;
    IF L =/= NONE THEN
      PARALLELTO := ABS( A*L.B - B*L.A ) < 0.00001;
  REF(POINT) PROCEDURE MEETS(L); REF(LINE) L;
    BEGIN REAL T;
      IF L =/= NONE and ~PARALLELTO(L) THEN
        BEGIN
          ...
          MEETS :- NEW POINT(...,...);
        END;
    END;***MEETS***
  COMMENT*** INITIALISATION CODE (CONSTRUCTOR)
  REAL D;
  D := SQRT( A**2 + B**2 )
  IF D = 0.0 THEN ERROR ELSE
    BEGIN
      A := A/D;  B := B/D;  C := C/D;
    END;
END***LINE***
```

[Squint and it’s almost Java!]

# SIMULA: Subclasses and inheritance

SIMULA syntax: `POINT CLASS COLOUREDPOINT`.
Java syntax: `class COLOUREDPOINT extends POINT`.

Example:

```
POINT CLASS COLOUREDPOINT(C); COLOUR C;    << note arg
BEGIN
  BOOLEAN PROCEDURE EQUALS(Q); REF(COLOUREDPOINT) Q;
    ...;
END***COLOUREDPOINT***

REF(POINT) P;                   // example point
P :- NEW POINT(1.0, 2.5);
REF(COLOUREDPOINT) CP;          // example colouredpoint
CP :- NEW COLOUREDPOINT(2.5, 1.0, RED);    << note args
```

NB: SIMULA 67 did not hide fields. Thus anyone can change the colour of the point referenced by `CP`:

```
   CP.C := BLUE;
```

# SIMULA: Object types and subtypes

- All instances of a class are given the same *type*. The name of this type is the same as the name of the class.
- The class names (types of objects) are arranged in a *subtype* hierarchy corresponding exactly to the subclass hierarchy.
- The Algol-60-based type system included explicit `REF` types to objects.

## **Subtyping Examples** – essentially like Java:

1.
   ```
   CLASS A;    A CLASS B;
   REF(A) a;   REF(B) b;
   a :- b; COMMENT***legal since B is
                  ***a subclass of A
   ...
   b :- a; COMMENT***also legal, but checked at
                  ***run time to make sure that
                  ***a points to a B object, so
                  ***as to avoid a type error
   ```

2.
   ```
   inspect a
     when B do b :- a
     otherwise ...
   ```

# Smalltalk

- Extended and refined the object metaphor.
  - Used some ideas from SIMULA; but it was a completely new language, with new terminology and an original syntax.
  - Abstraction via private *instance variables* (data associated with an object) and public *methods* (code for performing operations).
  - Everything is an object; even a class. All operations are messages to objects. Dynamically typed.
  - Objects and classes were shown useful organising concepts for building an entire programming environment and system. Like Lisp, easy to build a self-definitional interpreter.
- Very influential, one can regard it as an object-oriented analogue of LISP: “Smalltalk is to Simula (or Java) as Lisp is to Algol”.

# Smalltalk Example

Most implementations of Smalltalk are based around an IDE environment (“click here to add a method to a class”). The example below uses GNU Smalltalk which is terminal-based with `st>` as the prompt.

```
st> Integer extend [
      myfact [
       self=0 ifTrue: [^1] ifFalse: [^((self-1) myfact) * self]
    ] ]
st> 5 myfact
120
```

- Send an `extend` message to (class) `Integer` containing `myfact` and its definition.
- The body of `myfact` sends the two-named-parameter message (`[^1]`, `[^((self-1) myfact) * self]`) to the boolean resulting from `self=0`, which has a method to evaluate one or the other.
- ‘`^`’ means return and `(self-1) myfact` sends the message `myfact` to the Integer given by `self-1`.

# Reflection, live coding and IDEs

Above, I focused on Smalltalk's API, e.g. the ability to dynamically add methods to a class.

Note that objects and classes are often shared (via *reflection*) between the interpreter and the executing program, so swapping the 'add' and 'multiply' methods in the Integer class may have rather wider effects than you expect!

While the API is of interest to implementers, often a user interface will be menu-driven using an IDE:

- click on a class, click on a method, adjust its body, etc.
- the reflective structure above gives us a way to control the interpreter and IDE behaviour by adjusting existing classes.
- this is also known as 'live coding', and gives quite a different feel to a system than (say) the concrete syntax for Smalltalk.

Remark: *Sonic Pi* is a live-coding scripting language for musical performance.

# ∼ ***Part B*** ∼

## Types and related ideas

Safety, static and dynamic types, forms of polymorphism, modules

# ∼ *Topic VI* ∼

## Types in programming languages

**Additional Reference:**

- **Sections 4.9 and 8.6** of *Programming languages: Concepts & constructs* by R. Sethi (2ND EDITION).

  Addison-Wesley, 1996.

# Types in programming

- A *type* is a collection of computational entities that share some common property.
- Three main uses of types in programming languages:
  1. naming and organising concepts,
  2. making sure that bit sequences in computer memory are interpreted consistently,
  3. providing information to the compiler about data manipulated by the program.
- Using types to organise a program makes it easier for someone to read, understand, and maintain the program. Types can serve an important purpose in documenting the design and intent of the program.
- Type information in programs can be used for many kinds of optimisations, and warnings about suspicious code.

# Type systems

- A *type system* for a language is a set of rules for associating a type with phrases in the language.
- Terms strong and weak refer to the effectiveness with which a type system prevents errors. A type system is *strong* if it accepts only *safe* phrases. In other words, phrases that are accepted by a strong type system are guaranteed to evaluate without type error. A type system is *weak* if it is not strong.
- Perhaps the biggest language development since the days of Fortran, Algol, Simula and LISP has been how type systems have evolved to become more expressive (and perhaps harder to understand!)

# Type safety

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

| **Safety** | **Example language** | **Explanation** |
|---|---|---|
| Not safe | C, C++ | Type casts,<br>pointer arithmetic |
| Almost safe | Pascal | Explicit deallocation;<br>dangling pointers |
| Safe | LISP, OCaml, Smalltalk, Java | Type checking;<br>garbage collection |

Non-examinable subtlety: Amin and Tate in 2016 reported “Java and Scala’s Type Systems are Unsound: The Existential Crisis of Null Pointers", which perhaps shows that complex interactions of features in a language can be hard for even expert users and designers to follow.

# Type checking

A *type error* occurs when a computational entity is used in a manner that is inconsistent with the concept it represents.

*Type checking* is used to prevent some or all type errors, ensuring that the operations in a program are applied properly.

Some questions to be asked about type checking in a language:

- Is the type system *strong* or *weak*?
- Is the checking done *statically* or *dynamically*?
- How *expressive* is the type system; that is, amongst safe programs, how many does it accept?

# Static and dynamic type checking

Run-time (dynamic) type checking:

- Compiler generates code, typically adding a ‘tag’ field to data representations, so types can be checked at run time.
- Examples: LISP, Smalltalk.
  (We will look at dynamically typed languages more later.)

Compile-time (static) type checking:

- Compiler checks the program text for potential type errors and rejects code which does not conform (perhaps rejecting code which would execute without error).
- Examples: OCaml, Java.
- Pros: faster code, finds errors earlier (safety-critical?).
- Cons: may restrict programming style.

**NB:** It is common for object-oriented languages to use a mixture of compile-time and run-time type checking, see the next slide.

# Java Downcasts

Consider the following Java program:

```
class A { ... };                  A a;
class B extends A { ... };        B b;
```

- Variable `a` has Java type `A` whose valid values are all those of *class* `A` along with those of all classes subtyping class `A` (here just class `B`).
- Subtyping determines when a variable of one type can be used as another (here used by assignment):

  `a = b;` √(upcast)

  `a = (A)b;` √(explicit upcast)

  `b = a;` ×(implicit downcast—illegal Java)

  `b = (B)a;` √(but needs run-time type-check)
- Mixed static and dynamic type checking!

See also the later discussion of subtype polymorphism.

# Type equality

When type checking we often need to know when two types are *equal*. Two variants of this are *structural equality* and *nominal equality*.

Let *t* be a type expression (e.g. `int * bool` in OCaml) and make two type definitions

```
type n1 = t; type n2 = t;
```

- Type names `n1` and `n2` are structurally equal.
- Type names `n1` and `n2` are not nominally equal.
  Under nominal equality a name is only equal to itself.

We extend these definitions to *type expressions* using structural equivalence for all type constructors not involving names.

## Examples:

- **Type equality in C/C++.** In C, type equality is structural for `typedef` names, but nominal for `struct`s and `union`s; note that in

  ```
  struct { int a; } x;   struct { int a; } y;
  ```

  there are two different (anonymously named) `struct`s so `x` and `y` have unequal types (and may not be assigned to one another).
- **Type equality in OCaml.** OCaml has structural equality for most types (including objects) except for datatype names which are only equivalent to themselves.
- **Type equality in Pascal/Modula-2.** Type equality was left ambiguous in Pascal. Its successor, Modula-2, avoided ambiguity by defining two types to be compatible if
  1. they are the same name, or
  2. they are `s` and `t`, and `s = t` is a type declaration, or
  3. one is a subrange of the other, or
  4. both are subranges of the same basic type.

# Type declarations

We can classify type definitions `type n =` *t* similarly:

Transparent. An alternative name is given to a type that can also be expressed without this name.

Opaque. A new type is introduced into the program that is not equal to any other type.

In implementation terms, type equality is just tree equality, except when we get to a type name in one or both types, when we either (transparently) look inside the corresponding definition giving a structural system, or choose insist that both nodes should be identical types names giving a nominal system.

# Type compatibility and subtyping

- Type equality is symmetric, but we might also be interested in the possibly non-symmetric notion of *type compatibility* (e.g. can this argument be passed to this function, or be assigned to this variable).
- This is useful for subtyping, e.g. given Java `A a; B b; a = b;` which is valid only if `B` is a subtype of (or equal to) `A`.

Similarly we might want type definitions to have one-way transparency. Consider

```
type age = int; type weight = int;
var x : age,  y : weight,  z : int;
```

We might want to allow implicit casts of `age` to `int` but not `int` to `age`, and certainly not `x := y;`.

# Polymorphism

*Polymorphism* [Greek: “having multiple forms”] refers to constructs that can take on different types as needed. There are three main forms in contemporary programming languages:

- **Parametric (or generic) polymorphism.** A function may be applied to any arguments whose types match a type expression involving type variables.
  Subcases: OCaml has parametric polymorphism, but also explicit polymorphism where the user must specify the instantiation (as does C++ with templates, and the type system of “System F”).
- **Subtype polymorphism.** A function expecting a given class may be applied to a subclass instead. E.g. Java, passing a `String` to a function expecting an `Object`.

[Not covered:] *bounded subtype polymorphism* combines aspects of these (e.g. allowing us to restrict the argument to a polymorphic class to being a subclass or superclass of some given class).

- **Ad-hoc polymorphism or overloading.** Two or more implementations with different types are referred to by the same name. E.g. Java.
  (Remark 1: Haskell's *type classes* enable rich overloading specifications. These allow functions be to implicitly applied to a range of types specified by a Haskell *type constraint*.)
  (Remark 2: the C++ rules on how to select the 'right' variant of an overloaded function are arcane.)

Although we've discussed these for function application, it is important to note that Java generics and ML parameterised datatypes (e.g. `Map<Key,Val>` and `'a list`) use the same idea for type constructors.

# Type inference

- *Type inference* is the process of determining the types of phrases based on the constructs that appear in them.
- An important language innovation.
- A cool algorithm.
- Gives some idea of how other static analysis algorithms work.

# Type inference in OCaml – idea

Idea: give every expression a new type variable and then emit constraints $\boxed{\alpha \approx \beta}$ whenever two types have to be equal.
These constraints can then be solved with Prolog-style unification.
For more detail see Part II course: “Types”.

**Typing rule (variable):**

$$\frac{}{\Gamma \vdash x : \tau} \quad \text{if } x : \tau \text{ in } \Gamma$$

**Inference rule:**

$$\frac{}{\Gamma \vdash x : \gamma} \quad \boxed{\gamma \approx \alpha} \text{ if } x : \alpha \text{ in } \Gamma$$

**Typing rule (application):**

$$\frac{\Gamma \vdash f : \sigma \;\texttt{->}\; \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash f(e) : \tau}$$

**Inference rule:**

$$\frac{\Gamma \vdash f : \alpha \qquad \Gamma \vdash e : \beta}{\Gamma \vdash f(e) : \gamma} \quad \boxed{\alpha \approx \beta \;\texttt{->}\; \gamma}$$

**Typing rule (lambda):**

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\texttt{fn}\; x \texttt{=>} e) : \sigma \;\texttt{->}\; \tau}$$

**Inference rule:**

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash (\texttt{fn}\; x \texttt{=>} e) : \gamma} \quad \boxed{\gamma \approx \alpha \;\texttt{->}\; \beta}$$

**Example:**

$$\dfrac{\dfrac{\dfrac{\dfrac{\sqrt{}}{f : \alpha_1, x : \alpha_3 \vdash f : \alpha_5} \qquad \dfrac{\dfrac{\sqrt{}}{f : \alpha_1, x : \alpha_3 \vdash f : \alpha_7} \qquad \dfrac{\sqrt{}}{f : \alpha_1, x : \alpha_3 \vdash x : \alpha_8}}{f : \alpha_1, x : \alpha_3 \vdash f(x) : \alpha_6}}{f : \alpha_1, x : \alpha_3 \vdash f(f(x)) : \alpha_4}}{f : \alpha_1 \vdash \texttt{fn}\ x => f(f(x)) : \alpha_2}}{\vdash \texttt{fn}\ f => \texttt{fn}\ x => f(f(x)) : \alpha_0}$$

$$\boxed{\begin{array}{c} \alpha_0 \approx \alpha_1 \texttt{->} \alpha_2\ , \quad \alpha_2 \approx \alpha_3 \texttt{->} \alpha_4\ , \quad \alpha_5 \approx \alpha_6 \texttt{->} \alpha_4\ , \quad \alpha_5 \approx \alpha_1 \\ \alpha_7 \approx \alpha_8 \texttt{->} \alpha_6\ , \quad \alpha_7 \approx \alpha_1\ , \quad \alpha_8 \approx \alpha_3 \end{array}}$$

**Solution:** $\alpha_0 = (\alpha_3 \texttt{->} \alpha_3) \texttt{->} \alpha_3 \texttt{->} \alpha_3$

# let-polymorphism

- The ‘obvious’ way to type-check `let` $x$ `=` $e$ `in` $e'$ is to treat it as `(fun` $x$ `->` $e'$`)` `(`$e$`)`.
- But Milner invented a more generous way to type let-expressions (involving *type schemes*—types qualified with $\forall$ which are renamed with new type variables at every use).
- For instance

  ```
  let f = fun x -> x in f(f);;
  ```

  type checks, whilst

  ```
  (fun f -> f(f)) (fun x -> x);;
  ```

  does not.
- Exercise: invent OCaml expressions $e$ and $e'$ above so that both forms type-check but have different types.

# Interaction of subtyping and generics—variance

In Java, we have that `String` is a subtype of `Object`.

- But should `String[]` be a subtype of `Object[]`?
- And should `ArrayList<String>` be a subtype of `ArrayList<Object>`?
- What about `Function<Object,String>` being a subtype of `Function<String,Object>`?

Given generic `G` we say it is

- *covariant* if `G<String>` is a subtype of `G<Object>`.
- *contravariant* if `G<Object>` is a subtype of `G<String>`.
- *invariant* or *non-variant* if neither hold.
- variance is a per-argument property for generics taking multiple arguments .

But what are the rules?

# Java arrays are covariant

The Java language decrees so. Hence the following code type-checks.

```
String[] s = new String[10];
Object[] o;
o = s;                 // decreed to be subtype
o[5] = "OK so far";
o[4] = new Integer(42);  // whoops!
```

However, it surely can't run! Indeed it raises exception `ArrayStoreException` at the final line. Why?

- The last line would be unsound, so all *writes* into a Java array need to check that the item stored is a subtype of the array they are stored into. The type checker can't help.
- Note that there is no problem with *reads*.

# Java generics are invariant (by default)

The Java language decrees so. Hence the following code now fails to type-check.

```
ArrayList<String> s = new ArrayList(10);[3]
ArrayList<Object> o;
o = s;                         // fails to type-check
o.set(5,"OK so far");          // type-checks OK
o.set(4, new Integer(42));     // type-checks OK
```

So generics are safer than arrays. But covariance and contravariance can be useful.

- What if I have an immutable array, so that writes to it are banned by the type checker, then surely it's OK for it to be covariant?

[3] Legal note: it doesn't matter here, but to exactly match the previous array-using code I should populate the ArrayList with 10 NULLs. Real code would of course populate both arrays and ArrayLists with non-NULL values.

# Java variance specifications

In Java we can have safe covariant generics using syntax like:

```
ArrayList<String> s = new ArrayList(10);
ArrayList<? extends Object> o;
o = s;                // now type checks again
```

But what about reading and writing to `o`?

```
s.set(2,"Hello");
System.out.println((String)o.get(2)+"World"); //fine
o.set(4,"seems OK");     //faulted at compile-time
```

The trade is that the covariant `ArrayList o` cannot have its elements written to, in exchange for covariance.

# Java variance specifications (2)

Yes, there is a contravariant specification too (which allows writes but not reads):

```
ArrayList<? super String> ss;
```

So `ss` can be assigned values of type `ArrayList<String>` and `ArrayList<Object>` only.

For more information (beyond the current course) see:

- en.wikipedia.org/wiki/Generics_in_Java
- en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)

# Topic VII

## Scripting Languages and Dynamic Typing

# Scripting languages

"*A scripting language is a programming language that supports scripts; programs written for a special run-time environment that automate the execution of tasks that could alternatively be executed one-by-one by a human operator. Scripting languages are often interpreted (rather than compiled). Primitives are usually the elementary tasks or API calls, and the language allows them to be combined into more complex programs.*" [Wikipedia]

From this definition it's clear that many (but not all) scripting languages will have a rather ad-hoc set of features – and therefore tend to cause computer scientists to have strong views about their design.

# Scripting languages (2)

- A *script* is just a program written for a scripting language, indeed the usage seems to be ‘respectability driven’: we write a Python *program* (respectable language) but a shell *script* or a Perl *script*.
- The definition is a bit usage-dependent: ML is a ‘proper’ stand-alone language, but was originally was the scripting language for creating proof trees in the Edinburgh LCF system (history: ML was the *meta-language* for manipulating proofs about the *object language* PP$\lambda$).
- Similarly, we’ve seen Lisp as a stand-alone language, but it’s also a scripting language for the *Emacs* editor.
- But it’s hard to see Java or C as a scripting language. (But note the recent “Java Shell” – Java not to be ‘outdone’??) Let’s turn to why.

# Scripting languages (3)

- *scripting language* means essentially “language with a REPL (read-evaluate-print loop)” as interface, or “language which can run interactively” (otherwise it’s not so good for abbreviating a series of manual tasks, such API calls, into a single manual task).
- interactive convenience means that scripting languages generally are dynamically typed or use type inference, and execution is either interpretation or JIT compilation.

Incidentally, CSS and HTML are generally called *mark-up languages* to reflect their weaker (non-Turing powerful) expressivity. Although, HTML combined with CSS is Turing powerful...

# Dynamically typed languages

We previously said:

- Using types to organise a program makes it easier for someone to read, understand, and maintain the program. Types can serve an important purpose in documenting the design and intent of the program.

So why would anyone want to lose these advantages?

- And why is JavaScript one of the most-popular programming languages on surveys like RedMonk?

Perhaps there is a modern-politics metaphor here, about elites using Java and ordinary programmers using JavaScript to be free of the shackles of types?

# Questions on what we teach vs. real life

- why the recent rise in popularity of dynamically typed languages when they are slower and can contain type errors?
- why C/C++ still used when its type system is unsafe?
- why language support for concurrency is so ‘patchwork’ given x86 multi-core processors have existed since 2005?

Chatley, Donaldson and Mycroft (“The next 7000 programming languages”) explore these questions in more detail: https://doi.org/10.1007/978-3-319-91908-9_15

We start by looking at a recent survey (by RedMonk, considering GitHub projects and StackOverflow questions) on programming language popularity.

# RedMonk language rankings 2022

| **Rank** | **RedMonk** (2022) |
|---:|---:|
| 1 | JavaScript |
| 2 | Python |
| 3 | Java |
| 4 | PHP |
| 5 | C# |
| 6 | CSS |
| 7= | C++ |
| 7= | TypeScript |
| 9 | Ruby |
| 10 | C |
| 11 | Swift |
| 12= | R |
| 12= | Objective-C |
| 14 | Shell |
| 15= | Scala |
| 15= | Go |
| 17= | PowerShell |
| 17= | Kotlin |
| 19= | Rust |
| 19= | Dart |

Note 1. CSS (Cascading Style Sheets) not really a language.
Note 2. Swift is Apple’s language to improve on Objective C.
Note 3. Don’t trust such surveys too much.

# What are all these languages?

- Java, C, C++, C#, Objective-C, Scala, Go, Swift, Haskell
  These are small variants on what we already teach in Tripos; all are statically typed languages.
- JavaScript, Python, PHP, TypeScript, Ruby, R
  These are all dynamically typed languages (but notably TypeScript is an optionally typed version of JavaScript)
- Most of the dynamically typed languages have a principal use for scripting.
- Some static languages (Scala, Haskell, ML) are also used for scripting (helpful to have type inference and lightweight top-level phrase syntax).

Let's look at some of them.

# JavaScript

- Originally called Mocha, shares little heritage with Java (apart from curly braces), but renamed ‘for advertising reasons’.
- Designed and implemented by Brendan Eich in 10 days (see fuller history on the web).
- Dynamically typed, *prototype-based* object system (there was a design requirement not to use Java-like classes).
- Has both object-oriented and functional language features (including higher-order functions)
- Implemented within browsers (Java *applets* did this first, but security and commercial reasons make these almost impossible to use nowadays).
- *callback*-style approach to scheduling within browsers.

# Browsers: Java Applets and JavaScript

There are two ways to execute code in a browser:

- [Java Applets] compile a Java application to JVM code and store it on a web site. A web page references it via the `<applet>` tag; the JVM code is run in a browser sandbox. In principle the best solution, but historically beset with security holes, and unloved by Microsoft and Apple for commercial reasons. Effectively dead by 2017.
- [JavaScript] the browser contains a JavaScript interpreter. JavaScript programs are embedded in source form in a web page with the `<script>` tag, and interpreted as the page is rendered (now sometimes JIT compiled). Interaction between browser and program facilitated with the DOM model. Questionable confidentiality guarantees.

https://en.wikipedia.org/wiki/Java_applet
https://en.wikipedia.org/wiki/JavaScript

# The JavaScript language

Three language inspirations [quoting Douglas Crockford]:

- Java (for its syntax)
- Scheme (a Lisp dialect for its lexical closures)
- Self (a Smalltalk dialect with prototypes instead of classes)

What’s ‘wrong’ with classes? Java has:

- Classes to define the basic qualities and behaviours of objects.
- Object instances as particular manifestations of a class.

So has Smalltalk.

# Problems with classes

- When designing a system with the waterfall design method, then classes and objects are natural.
- In more flexible design styles (spiral and agile), then we might want to change the classes during program evolution.
- In Smalltalk this can be achieved by changing the structure of a class, or the class an object belongs to, by assignment.
- But it's not quite so simple, in that users inheriting a class may be surprised by the behaviour change of the superclass (the *fragile base class* problem)
- One partial solution is to use *prototypes* . . .

# Prototypes instead of classes

- In JavaScript (and Self), there are no classes.
- When a constructor function is defined, it acquires a prototype property (the JS for ‘field’ and ‘method’) initially an empty object.
- When a constructor is called, created objects inherit from the prototype (i.e. field and method lookup look first in the object and then in the prototype).
- You can add properties to the prototype object, or indeed replace the it entirely with another object.
- This gives a prototype inheritance chain similar to a class inheritance chain.
- The Self community argue that this is a more flexible style of programming, but many JavaScript systems start by defining prototypes which play the role of classes, and copying from these plays the role of *new*.

# Prototypes example

[Taken from the ‘prototype’ section of https://www.w3schools.com/js/]

```
function Person(first, last, age, eye) {
    // Constructor: caller needs to use ‘new’
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eye;
}
Person.prototype.nationality = "English";
var myFather = new Person("John", "Doe", 50, "blue");
```

After this, then `myFather.nationality` gives `"English"`.

Note the difference between updating a field in the prototype and replacing the prototype with a new object.

# Duck typing

If we have classes, we can use Java-style type-testing:

```
if (x instanceof Duck} { ... }
```

This is a form of *nominal* property. But what do we do without classes? One answer is *Duck Typing*:

> *"In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with." [Alex Martelli, comp.lang.python (2000)]*

More precisely, we take a *structural* rather than *nominal* view of being a duck: if an object has sufficiently many methods and fields (perhaps `walks()` and `quacks()` to be considered a Duck) then it *is* a Duck.
[Beware: JavaScript also has `instanceof` but this is merely shorthand for a test on an object's prototype.]

# The JavaScript DOM model

JavaScript is most often used in browsers, so needs to be tightly connected with web pages. This is done by:

- The DOM (Document Object Model), a W3C standard.
- The browser converts the HTML into a tree of objects representing the HTML page structures. This is made available as the JavaScript variable *document*.
- Executing JavaScript code reads and modifies *document* which the browser renders on-screen.
- Example *button-click2* on 'materials' tab on course web page
  (view the source as well as clicking).

# JavaScript interaction within the browser

- Uses event-based GUI programming, based on callbacks, just like the Swing library for Java.

# WebAssembly – turning full circle?

Because JavaScript is the only language universally supported by browsers (we're not counting plug-ins because of lack of ubiquitous support and security risks), we find that various compilers have developed JavaScript-targeting back-ends.

- E.g. js_of_ocaml, ghcjs, Scala.js.
- Try out OCaml in a browser: `http://ocsigen.org/js_of_ocaml/2.8.3/files/toplevel/index.html`
- But JavaScript is a terrible compiler target language (a Java Applet compiled to JVM would be more effective).
- Upcoming answer: WebAssembly. Machine-agnostic, safety-checked assembly language JIT'ted in the browser.
- Isn't this just a variant of applets?

# Some other scripting Languages

- Python: popular for scripting scientific package API calls.
- Ruby: popularised by Heinemeier Hansson's *Rails* framework (server-side web-application programming and easy database integration).
- Clojure: Lisp meets the JVM.
- Typescript: Microsoft's typed version of JavaScript.
- R: a statistical scripting package.
- Lua: original for scripting for game engines.
- Matlab.

# Gradual type systems

Dynamically typed languages are great (at least for small programs):

- *“You can write your whole prototype application in Python while a Java programmer is still writing his/her class hierarchy”*

Statically typed languages are better for large programs, due to the documentation and cross-module static checks, provided by types.

- Imagine the discussions at Facebook when management started to confront the issue that they had one million lines of PHP in their system.

So what you might want is:

- a language in which small programs can be written without types, but the IDE encourages you to add types faster than software rust overtakes your system – *gradual types*.

# Gradual type systems (2)

*“Gradual typing is a type system in which some variables and expressions may be given types and the correctness of the typing is checked at compile-time (which is static typing) and some expressions may be left untyped and eventual type errors are reported at run-time (which is dynamic typing). Gradual typing allows software developers to choose either type paradigm as appropriate, from within a single language. In many cases gradual typing is added to an existing dynamic language, creating a derived language allowing but not requiring static typing to be used. In some cases a language uses gradual typing from the start.”* [Wikipedia]

Example languages:

- Microsoft’s TypeScript
- Facebook’s Hack (for PHP)
- Cython and mypy (for Python)
- C#’s *dynamic* type.

Sometimes gradual types are called *optional types*.

# Object-orientation: static vs dynamic, compiled vs interpreted

- Java and Scala are probably sweet-spots for ahead-of-time compilation for object-oriented languages, due to their static type systems.
- The dynamic features of Self and JavaScript (and indeed Smalltalk) mean that these languages need to be interpreted or JIT'ed.
- You might (non-examinably) think about what makes a nice type system possible and what is nice about a type system to enable ahead-of-time compilation.

# ∼ *Topic VIII* ∼

## Data abstraction and modularity
## OCaml Modules

**Additional Reference:**

- **Part 1** of *Real World OCaml* (2ND EDITION). CUP, 2022. `https://realworldocaml.org/`

**May also be useful:**

- **Chapter 7** of *ML for the working programmer* (2ND EDITION) by L. C. Paulson. CUP, 1996.
- *Purely Functional Data Structures* by Chris Okasaki, Cambridge University Press. [Clever functional data structures, implemented in Haskell and SML/OCaml Modules.]

# Need for modules

- We want control over name visibility in large systems.
- Name visibility is important because it expresses large-scale architecture (which components can use which other components). Excessive visibility increases the attack surface for malevolent use.
- Traditional Java does quite well on class-level visibility (private, protected, public) – but less well on package-level visibility (no formal way to control who might use a package-level export, or malevolent uses of reflection).
- Java 9 added a module system in 2017.
- We focus on the OCaml module system here, as you have used the core language already.

There are also issues (only briefly mentioned) concerning *versioning*, when modules evolve over time and multiple versions are available, e.g. Linux packages or Windows DLLs.

# The Core and Modules languages

OCaml consists of two sub-languages:

- The *Core* language is for *programming in the small*, by supporting the definition of types and variables denoting values of those types.
  (You used this in 1A Foundations of Computer Science)
- The *Modules* language is for *programming in the large*, by grouping related Core definitions of types and variables into self-contained units, with descriptive interfaces.

The *Core* expresses details of *data structures* and *algorithms*. The *Modules* language expresses *software architecture*. Both languages are largely independent.

# The Modules language

Writing a real program as an unstructured sequence of Core definitions quickly becomes unmanageable.

```
type nat = int
val zero = 0
fun succ x = x + 1
fun iter b f i =
   if i = zero then b
               else  f (iter b f (i-1))
...
(* thousands of lines later *)
fun even (n:nat) = iter true not n
```

The OCaml module language lets one split large programs into separate units with descriptive interfaces.

# OCaml Modules

## Signatures and structures

An *abstract data type* is a type equipped with a set of operations, which are the only operations applicable to that type.
Its representation can be changed without affecting the rest of the program.

- *Structures* let us *package* up declarations of related types, values, and functions.
- *Signatures* let us *specify* what components a structure must contain.
- Structures are like OCaml records `{a = 3, b = [4,5]}` but can contain definitions of types (and exceptions) in addition to definitions of values.

Signatures are to structures what types are to values.

# Structures

In Modules, one can encapsulate a sequence of Core *type* and *value* definitions into a unit called a ***structure***.
We enclose the definitions in between the keywords

`struct ... end`.

**Example:** A structure representing the natural numbers, as positive integers:

```
module IntNat = struct
  type nat = int
  let zero = 0
  let succ x = x + 1
  let rec iter b f i =
    if i = zero then b
    else f (iter b f (i-1))
end
```

# Concrete signatures

*Signatures* specify the 'types' of structures by listing the specifications of their components.

A signature consists of a *sequence* of component specifications, enclosed in between the keywords `sig ... end`.

```
module IntNat : sig
    type nat = int
    val zero : int
    val succ : int -> int
    val iter : 'a -> ('a -> 'a) -> int -> 'a
end
```

This signature fully describes the 'type' of `IntNat` on the previous slide.
(We'll later see signatures which *restrict* the names visible in a structure.)

The specification of type `nat` is *concrete*: it must be `int`.

# Opaque signatures

On the other hand, the following signature

```
module IntNat : sig
  type nat
  val zero : int
  val succ : int -> int
  val iter : 'a -> ('a -> 'a) -> int -> 'a
end
```

specifies structures that are free to use *any* implementation for type `nat` (perhaps `int`, or `word`, or some recursive datatype).

This specification of type `nat` is *opaque*.

# Naming structures and signatures

The `module` keyword binds a structure to an identifier:

```
module IntNat = struct
  type nat = int
  ...
  let rec iter b f i = ...
end
```

The `sig` keyword similarly names signatures:

```
module NAT : sig
  type nat
  ...
  val 'a iter : 'a -> ('a -> 'a) -> nat -> 'a
end
```

# Modules and dot notation

Dot notation accesses components of structures and signatures, e.g.

```
let even (n : IntNat.nat) = IntNat.iter true not n
```

**NB:** Type `IntNat.nat` is statically equal to `int`.
Value `IntNat.iter` dynamically evaluates to a closure.

There is an OCaml *convention* of having each module export a type `t` which names the principal representation type defined by the module.
Hence e.g. `List.t` is the type of lists, and we have:

```
List.cons 1 [2] : int List.t
```

## **Example:** Polymorphic functional stacks.

```
module type STACK = sig
  exception E
  type ’a t                                <--- opaque type
  val make: ’a t
  val push: ’a -> ’a t -> ’a t
  val pop: ’a t -> ’a t
  val top: ’a t -> ’a
end

module MyStack: STACK = struct
  exception E
  type ’a t = ’a list                      <--- varied later
  let make = []
  let push x s = x::s
  let split = function                     <--- not exported
   | h::t -> h,t
   | _ -> raise E
  let pop s = snd (split s)
  let top s = fst (split s)
end
```

```
# let e = MyStack.make ;;
val e : ’a MyStack.t = <abstr>

# let s0 = MyStack.push 0 e ;;
val s0 : int MyStack.t = <abstr>

# let s01 = MyStack.push 1 s0 ;;
val s01 : int MyStack.t = <abstr>

# let s0 = MyStack.pop s01 ;;
val s0 : int MyStack.t = <abstr>

# MyStack.top s0 ;;
- : int = 0
```

# Signature matching

**Q:** When does a structure satisfy a signature?

**A:** The type of a structure *matches* a signature whenever it implements at least the components of the signature.

- The structure must *realise* (i.e. define) all of the opaque type components in the signature.
- The structure must *enrich* this realised signature, component-wise:
  - every concrete type must be implemented equivalently;
  - every value must have a more general type scheme;
  - every structure must be enriched by a substructure.

Signature matching supports a form of *subtyping* not found in the Core language:

- A structure with more type, value, and structure components may be used where fewer components are expected.
- A value component may have a more general type scheme than expected.

# Properties of signature matching

- The components of a structure can be defined in a different order than in the signature; names matter but ordering does not.
- A structure may contain more components, or components of more general types, than are specified in a matching signature.
- Signature matching is *structural*. A structure can match many signatures and there is no need to pre-declare its matching signatures (unlike “interfaces” in Java and C#).
- Although similar to record types, signatures actually play a number of different roles.

# Using signatures to restrict access

We can use a *signature constraint* to provide a restricted view of a structure. E.g. (restricting `IntNat` from slide 134)

```
module SubIntNat : sig
  type nat = IntNat.nat
  val succ : nat -> nat
  val iter : nat -> (nat -> nat) -> nat -> nat
end = IntNat
```

Here the constraint `:   sig = str` prunes the structure `str` according to the signature `sgn`. So:

- `SubIntNat.zero` is faulted (“not a member”)
- `SubIntNat.iter` is *less* polymorphic than `IntNat.iter`.

# Opaque signature constraints: “`_ :> _`”

Signature matching is used to enforce *data abstraction* by hiding the identity of types:

```
module AbsNat : sig
  type nat
  val zero : nat
  val succ : nat -> nat
  val iter : 'a -> ('a -> 'a) -> nat -> 'a
end = IntNat
```

The signature here does not expose a type, and so the exposed type is a new *abstract* type for each opaque type in the signature.
The actual implementation of `AbsNat.nat` by `int` is *hidden*, so that `AbsNat.nat` $\neq$ `int`.

- `AbsNat` is just `IntNat`, but with a hidden type representation.
- `AbsNat` defines an ***abstract datatype*** of natural numbers: the only way to construct and use values of the abstract type `AbsNat.nat` is through the operations, `zero`, `succ`, and `iter`.
  E.g., the application `AbsNat.succ(~3)` is ill-typed: `~3` has type `int`, not `AbsNat.nat`. This is what we want, since `~3` is not a natural number in our representation.
  In general, abstractions can also prune and specialise components.

# Modules in other languages

Java also provides various structuring and hiding features:

- classes
- interfaces (with default implementations in Java 8)
- packages
- modules (from Java 9)

These offer independently developed, but overlapping, facilities—beyond this current course.

A separate issue is *versioning*. Linux packages (informal modules) come with versions, e.g. `libtest.so.1.0.1` and meta-information says which versions of a package are compatible for use within another versioned package. The similar Windows problem is ‘DLL hell’. Some languages like `https://www.unison-lang.org` are fixing this.

# ∼ *Part C* ∼

## Linguistic ideas beyond Java and ML

Distributed concurrency, Monads, Rust, Resumable Exceptions

# ∼ *Topic IX* ∼

## Languages for Concurrency and Parallelism

# Sources of parallel computing

Five main sources:

1. Theoretical models: PRAM, BSP (complexity theory), CSP, CCS, $\pi$-calculus (semantic theory), Actors (programming model).
2. Multi-core CPUs (possibly heterogeneous—mobile phones).
3. Graphics cards (just unusual SIMD multi-core CPUs).
4. Supercomputers (mainly for scientific computing).
5. Cluster Computing, Cloud Computing.

**NB:** Items 2–5 conceptually only differ in processor-memory communication.

# Language groups

1. Theoretical models (PRAM, $\pi$-calculus, Actors, etc.).
2. C/C++ and roll-your-own using pthreads.
3. Pure functional programming (‘free’ distribution).
4. [Multi-core CPUs] Open/MP, Java (esp. Java 8), Open/MP, Cilk, X10.
5. [Graphics cards] CUDA (Nvidia), OpenCL (open standard).
6. [Supercomputers] MPI.
7. [Cloud Computing] MapReduce, Spark, Dryad, Naiad.
8. [On Chip] Verilog, Bluespec.

**NB:** Language features may fit multiple architectures.

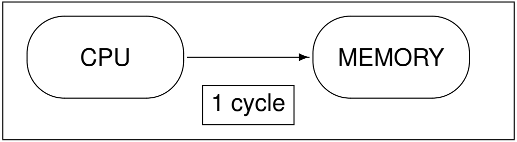
# Painful facts of parallel life

1. Single-core clock speeds have stagnated at around 3GHz for the last twenty years. Moore's law continues to give more transistors (hence multi-core, many-core, giga-core).
2. Inter-processor *communication* is far far far more expensive than *computation* (executing an instruction).
3. Can't the compiler just take my old C/Java/Fortran (or ML/Haskell) program and, *you know*, parallelise it? Just another compiler optimisation? NO! (Compiler researchers' pipe-dream/elephants' graveyard.)

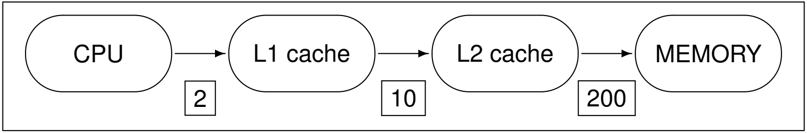
**Takeaway:** optimising performance requires exploiting parallelism, you'll have to program this yourself, and getting it wrong gives slow-downs and bugs due to races.

New programming frameworks for clusters/cloud computing.

# A programmer's view of memory and timings



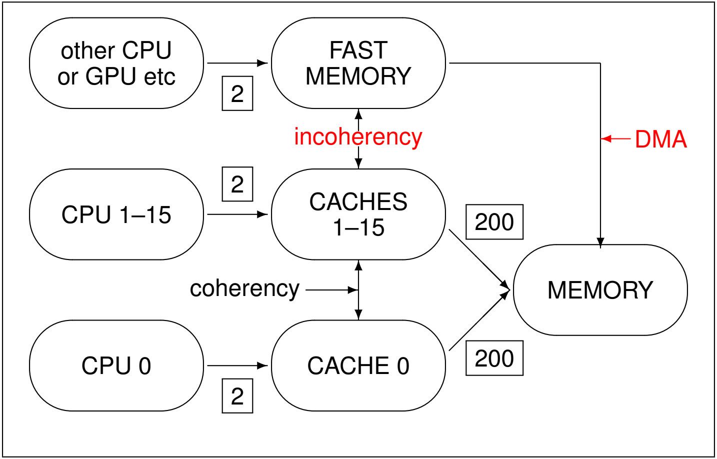
(This model was pretty accurate in 1985.)



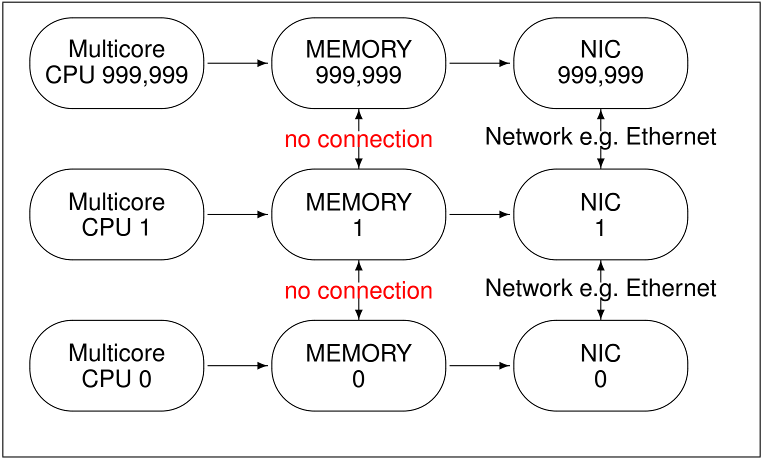
(This model was pretty accurate in 2003.)

# Multi-core-chip memory models

Today’s model (cache simplified to one level):



# A Compute Cluster or Cloud-Computing Server



(The sort of thing which Google uses.)

# Lecture focus: what *programming abstractions*?

- We’ve got a large (and increasing) number of processors available for use within each ‘device’
- This holds at multiple levels of scale (from on-chip to on-cloud). “Fractal” or “self-similar”.
- Memory is local to processor units (at each scale)
- Communication (message passing) between units is much slower than computation.

Question: what are good programming abstractions for a system containing lots of processors?
Answer: no complete answer, rest of this lecture gives fragmentary answers.

# What hardware architecture tells us

- Communication latency is far higher than instruction execution time (2–6 orders of magnitude)
- So, realistically a task needs to have need at least $10^4$ instructions for it to be worth moving to another CPU.
- *Long-running independent computations* fit the hardware best.
- *“Shared memory” is an illusion.* At the lowest level it is emulated by message passing in the cache-coherency protocol.
- Often best to think of multi-core processors as distributed systems.

# Communication abstractions for programming

- “Head in sand”: What communication – I’m just using a multi-core CPU?
- “Principled head in sand”: the restrictions in my programming language means I can leave this to someone else (or even the compiler).
- Just use TCP/IP.
- Shared memory, message passing, RMI/RPC?
- Communication is expensive, expose it to programmer (no lies about ‘shared memory’).

**Reflection:** does a language implicitly create a programmer mental model of communication?

# Concurrent, Parallel, Distributed

These words are often used informally as near synonyms.

- Distributed systems have separate processors connected by a network, perhaps on-chip (multi-core)?
- ‘Parallel’ suggests multiple CPUs or even SIMD, but “parallel computation" isn’t clearly different from “concurrency”.
- Concurrent behaviour *can* happen on a single-core CPU (e.g. Operating System and threads), Theorists often separate ‘true concurrency’ (meaning parallel behaviour) from ‘interleaving concurrency’.

# SIMD, MIMD

- Most parallel systems nowadays are MIMD.
- GPUs (graphical processor units) are a bit of an exception; several cores execute the same instructions, perhaps conditionally based on a previous test which sets per-processor condition codes.
- Programming Languages for GPUs (OpenCL, CUDA) emphasise the idea of a single program which is executed by many tasks. A program can enquire to find out the numerical value of its task identifier, originally its $(x, y)$ co-ordinate, to behave differently at different places (in addition to having separate per-task pixel data).

# Theoretical model – process algebra

- CCS, CSP, Pi-Calculus (calculus = “simple programming language"). E.g.
- Atomic actions $\alpha$, $\overline{\alpha}$, can communicate with each other or the world (non-deterministically if multiple partners offered). Internal communication gives special internal action $\tau$.
- Behaviour $p ::= 0 \mid \alpha.p \mid p + p \mid p|p \mid X \mid rec\ X.p$ (Deadlock, prefixing, non-determinism, parallelism, recursive definitions, also (not shown) parameterisation/hiding and value-passing.)
- Typical questions: “is $\alpha.0|\beta.0$ the same as $\alpha.\beta.0 + \beta.\alpha.0$" and
  “what does it mean for two behaviours to be equal"

# Theoretical model – PRAM model

- PRAM: parallel random-access machine.
- $N$ shared memory locations and $P$ processors (both unbounded); each processor can access any location in one cycle.
- Execute instructions in lock-step (often SIMD, but MIMD within the model): fetch data, do operation, write result.
- Typical question: “given $n$ items can we sort them in $O(n)$ time, or find the maximum in $O(1)$ time"
- BSP (bulk-synchronous parallel) model refines PRAM by adding costs for communication and synchronisation.

# Oldest idea: Threads

Java threads – either extend Thread or implement Runnable:

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long m) { minPrime = m; }
    public void run() {
        // compute primes larger than minPrime
    }
}
...
p = new PrimeRun(143);  // create a thread
new Thread(p).start();  // run it
```

Posix's pthreads are similar.

# Threads, and what's wrong with them

- Need explicit synchronisation. Error prone.
- Because they're implemented as library calls, the compiler (and often users) cannot work out where they start and end.
- pthreads as OS-level threads. Need context switch. Heavyweight.
- Various lightweight-thread systems. Often non-preemptive. Blocking operations can block all lightweight operations sharing the same OS thread.
- Number of threads pretty hard-coded into your program.
- Nice IBM article: `https://www.ibm.com/developerworks/library/j-nothreads/`

# Language support: Cilk

Cilk [example from Wikipedia]

```
cilk int fib (int n)
{   int x,y;
    if (n < 2) return n;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;  // wait for both spawns to finish
    return x+y;
}
```

Compiler/run-time library can manage threads. Neat implementation by “work stealing”. Can adapt to hardware.
X10 (IBM) adds support for partitioned memory.

# Language support: OpenMP

OpenMP [example from Wikipedia]

```
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];
    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;
    return 0;
}
```

The directive “omp parallel for" tells the compiler “it is safe to do the iterations in parallel".
Fortran “FORALL INDEPENDENT".

# Threads and the like don’t work on clusters and clouds

- Memory support for threads, Cilk, OpenMP centres around a *shared address space*. (Even if secretly multi-core machines behave like distributed machines.)
- What about clusters? Cloud Computing?
- Need message-passing (either in the language or behind the scenes).

# Software support for message passing: MPI

- MPI = Message Passing Interface [nothing to do with OpenMP]
- “de-facto standard for communication among processes that model a parallel program running on a distributed memory system.” [no shared memory].
- Standardised API calls for transferring data and synchronising iterations. Message passing is generally synchronous, suitable for repeated sweeps over scientific data.
- Emphasis on message passing (visible and expensive-looking to user) means that MPI programs can work surprisingly well on multi-core, because they encourage within-core locality.

# Software support for message passing: Erlang

- Shared-nothing language based on the actor model (asynchronous message passing).
- Dynamically typed, functional-style (no assignment).
- Means tasks can just commit suicide if they feel there's a problem and someone else fixes things, including restarting them.
- Relatively easy to support hot-swapping of code.

# Cloud Computing

Can mean either any of:

- Doing one computer's worth of work on a server instead of locally. Google Docs.
- A glorified backed-up virtual disk or file server. Google Drive, OneDrive, iCloud.
- Distributing computation across many remote computers, and managing scheduling and dependency management from within a programming framework:
  Example frameworks: MapReduce (seminal), Spark, Dryad, Naiad.
  These have different levels of expressivity for different algorithms (static vs dynamic data dependency graphs).

The first two are really only commercial special cases!

- Note the need for error resiliancy (errors happen often in big computation), so need checkpoints or idempotent (functional-like) computation.

# Embarrassingly Parallel

Program having many separate sub-units of work (typically more than the number of processors) which

- do not interact (no communication between them, not even via shared memory)
- are large

Example: the map part of MapReduce.

# Functional Programming

In pure functional programming every tuple (perhaps an argument list to an application) can be evaluated in parallel.
So functional programming is embarrassingly parallel?
Not in general (i.e. not enough for compilers to be able to choose the parallelism for you). Need to find sub-executions with:

- little data to transfer at spawn time (because it needs copying, even if memory claims to be shared);
- a large enough unit of work to be done before return

Probably only certain stylised code.

# Garbage Collection

While we’re talking about functional programming, and as garbage collection has previously been mentioned . . .
Just how do we do garbage collection across multiple cores?

- Manage data so that data structures do not move from one processor to another?
- “Stop the world" GC with one big lock doesn’t look like it will work.
- *Parallel GC:* use multiple cores for GC.
  *Concurrent GC:* do GC while the mutator (user’s program) is running. Hard?
- Incremental? Track imported/exported pointers?

# How have languages adapted to parallel hardware?

- Hardware change (multi-core parallelism in 2005) is one of the major changes ever, comparable with adding *index registers* and *interrupts* from the 1960s.
- It took many years between the early languages we discussed to get to the globals-stack-heap model from the 1970s to today (for sequential processors).
- Programming language support for parallelism today seems rather fragmentary in comparison. Several incomparable solutions.

  Is 2005 too recent for languages to have got the full answer? Or . . . ?

# Partial language evolution: functional programming ideas

- Concurrency and mutable data structures seem too hard for programmers to correctly use together in large programs.
- Pure functional languages can feel too ‘hair shirt’ for commercial use.
- Observation: languages have evolved to use ideas from functional programming such as higher-order functions and reduced exposure of mutable data structures in APIs used concurrently. Let’s look at Streams in Java 8.

# Java 8: Internal vs. External iteration

Can’t trust users to iterate over data. Consider traditional *external iteration*. It’s easy to start by writing

```
for (i : collection)
{   // whatever
}
```

and then get lazy. Do we really want to write this?

```
for (k = 0; k<NUMPROCESSORS; k++}
{   spawn for (i : subpart(collection,k))
    {   // whatever
    }
}
sync;
// combine results from sub-parts here
```

And what about the maintenance load?

# Internal vs External iteration (2)

- Previous slide was *external iteration*. It's hard to parallelise (especially in Java where iterators have shared mutable state).
- The Java 8 Streams library encourages *internal iteration* – keep the iterator in the library, and use ML-like stream operation to encode the body of the loop

```
maxeven = collection.stream().parallel()
                    .filter(x -> x%2 == 0)
                    .max();
```

- The library can optimise the iteration based on the number of threads available (and do a better job than users make!).

# Java Collections vs Streams

Collections

- ‘Trusty old Java looping’; `for`-loops, internally using mutable state (Iterators).
- Note that Java 8 did not add `map` and `filter` to the Collections classes. Why?

Streams

- Almost ML-style understanding. Items need not be in memory all at the same time (lazy).
- But additional laziness: a stream pipeline, e.g. `.filter().max()` above, traverses the stream only once.
- `.parallel()` authorises parallel/arbitrary order of calls to the stream pipeline. Good for parallelism, beware if you use side-effecting functions on stream elements!

It’s quite practical to use `.stream` on a collection and then process it as a stream and then convert back.

# ~ *Topic X* ~

Functional-style programming meets object-orientation

# Big picture

Historically, object-oriented and functional languages were seen as disjoint programming paradigms.

`en.wikipedia.org/wiki/Programming_paradigm`

- Recently, mainstream OO languages have acquired functional features (Java, C++, C#).
- Why?
- Influences of JavaScript and, more directly, Scala.
- Multi-core parallelism: we just saw how internal iteration can move iteration code into a library, which receives the ‘body of the loop’ as a lambda expression. The run-time system can then schedule separate iterations of the loop body according to the number of available tasks.
- Other programming paradigms like *futures*, *reactive programming* can be added to a language as library features once lambdas are present.

# The ‘billion-dollar mistake’

Java (and the JVM) allow any reference type to be NULL.

- In 2009, Tony Hoare described his (1965) invention of the NULL value as his “billion-dollar mistake”.
- many modern languages, e.g. Scala, C#, Kotlin, forbid reference types from including NULL
- How? Why?
- Most uses of reference types are only NULL due to programmer error (e.g. failed initialiation, creaky old APIs using NULL instead of exceptions).
- If you really want NULL values then use `Optional<t>` in Java or `t option` in OCaml. E.g. `tree option` which has values `None` and `Some x` where `x` is of type `tree`.
- In Kotlin (JVM language used by Android) type `String` only refers to non-NULL JVM strings, but type `String?` includes NULL. This is a good default.

# Java lambdas and closures

Single-argument lambdas in Java implement the `Function` interface:

```
public interface Function<T,R> {
  public <R> apply(T argument);
}
```

A lambda expression `(Int x) -> x+1` allocates an object of a compiler-generated class (only) containing method

```
public Int apply(Int x) { return x+1; }
```

This implements `Function<Int,Int>` as required.

But it's too simple to show the idea of closure. So instead consider:

```
Function<Int,Int> foo(int fv) { return (Int x) -> x+fv; }
```

Now we need *another* compiler-generated class whose instances can hold a copy of the value of `fv` after `foo` has returned, and which also implements `Function<Int,Int>`.

# Function types in Java

Above we saw a unary function `Function<Int,Int>`. But what about *n*-ary functions?

In Scala it's easy: ML `t1*...*tk -> t0` encodes as

```
Functionk[t1,...,tk,t0]
```

for example: `function2[Int, Array[Int], Bool]`.

In Java it's similar but ad-hoc: e.g. `Function1` is spelt `Function` and `Function2` is spelt `BiFunction`.

Worse still in Java, there's the boxed/unboxed issue for generics:you can write `Function<Integer,Boolean>`, but not `Function<Integer,boolean>`. So for efficiency the latter has a special 'function' type

```
Predicate<Integer>
```

It's even more ad hoc as you don't `.apply` a `Predicate`, you `.test` it!!

# Data structures in functional and object-oriented style

Question: how do we write to code print and evaluate expressions such as

```
Times( Plus(Number(1),Number(2)), Number(3) )
```

Logically there are six ($3 \times 2$) cases:

```
eval(Num(n)) = n
eval(Plus(t1,t2)) = eval(t1) + eval(t2)
eval(Times(t1,t2)) = eval(t1) * eval(t2)
prin(Num(n)) = string_of_int(n)
prin(Plus(t1,t2)) = "(" + prin(t1) + "+" + prin(t2) + ")"
prin(Times(t1,t2)) = "(" + prin(t1) + "*" + prin(t2) + ")"
```

How do we program this in OCaml or Java?

# Expr in OCaml

The above six cases are practically OCaml code already – we just have to bundle the cases into *two* function definitions, each of which considers *three* cases. (In OCaml we have to use `function` or `match` for this.)

Additionally we need to define type `Expr` and *three* constructors for expressions:

```
type Expr =
  | Num of int
  | Plus of Expr * Expr
  | Times of Expr * Expr;;
```

# Expr in Java

We define abstract class `Expr` subclassing it *three* times, each subclass having *two* methods `eval` and `prin`. In principle just (omitting some access qualifiers):

```
abstract class Expr { int eval(); String prin(); }
class Num extends Expr {
  private int n; public Num(x) { n = x; }
  int eval() { return n; }
  String prin() { return Integer.toString(n); }
}
class Plus extends Expr {
  private Expr t1, t2; public Plus(x1,x2) {t1=x1; t2=x2;}
  int eval() { t1.eval() + t2.eval(); }
  String prin() { return "(" + t1.prin() + "+" +
                               t2.prin() + ")"; }
}
class Times extends Expr {
  private Expr t1, t2; public Times(x1,x2) {t1=x1; t2=x2;}
  int eval() { return t1.eval() * t2.eval(); }
  String prin() { return "(" + t1.prin() + "*" +
                               t2.prin() + ")"; }
}
```

# Expr in Java (2)

Non-examinable digression: Java 15 added a `record` keyword to Java which behaves like `class` but automatically generates constructors for `Num`, `Plus` etc., and this makes the code tidier.

However, Java records cannot extend other classes; this requires `Expr` to be an interface not a class. While this seems harmless, it allows `Num`, `Plus` etc. (which are now classes) to implement other interfaces beyond `Expr`.

# Expr in OCaml vs. Java

Are there any differences, apart from the Java code feeling more clumsy?

For a small fixed program, these appear to be simple alternatives. But consider a larger program, possibly split over multiple files and subject to ongoing maintenance

Adding a new form of data, e.g. `Divide` to `Expr` is:

- easy in Java, we just add a subclass to `Expr`
- hard in OCaml, we need to find every `match` or `function` involving `Expr`.

Adding a new operation, e.g. `compile` to `Expr` is:

- hard in Java, every subclass of `Expr` needs a method added
- easy in OCaml, just add a new top-level function.

# Expr in OCaml vs. Java – the expression problem

The language-design problem of allowing a data-type definition where one can add new cases to the datatype and new functions over the datatype (without requiring ubiquitous changes) is known as the “expression problem”:

en.wikipedia.org/wiki/Expression_problem

In passing, note that

```
Times( Plus(Number(8),Number(9)), Number(10) )
```

is represented by essentially the same data structure at machine-code level in both OCaml and Java – even the discriminator tags (recall the Pascal lecture) are present. OCaml implements these as numeric case tags, and Java as pointers to the run-time-type information (virtual method table). Using C types, in both, constructor `Plus` returns a pointer to a

```
struct { LanguageDependentTag t0; Expr *t1, *t2; }
```

# Value types

A *value type* is a type representing an a pure value – such values can be copied, compared and subfields extracted but subfields cannot be mutated. Complex numbers and immutable arrays are examples, but mutable arrays are not. Strings of course should be values, but C and C++ historically have had issues as to whether they are `char[]` or `const char[]`.

- In Java, every value is either of primitive type (spelt with lower case) or is a reference to an object. Assignment '`=`' copies bit patterns for primitives, but only copies references for object types. Equality '`==`' differs similarly.
- Worse, every reference-type object has *identity*, so that `new Complex(1.0,2.0)` allocates a new *distinguishable* object.
- This also explains why we are encouraged (given `int i; Integer x;`) to write `x = Integer.valueOf(i);` instead of `x = new Integer(i);`.

# Value types and languages

When talking about value types, we want a type which only admits functional update (copy but changing a field), not mutability and no identity.

Why are value types useful? Increasing use of functional-style APIs (partly due to the need to exploit parallelisms).

Think about defining a class `Complex` and note that every arithmetic operation is likely to do an allocation. What we want is the equivalent of `struct { double re,im; }` in C. C# (Microsoft's Java-style language) provides such values:

> *"Variables [of value type] directly contain values. Assigning one value-type variable to another copies the contained value. This differs from the assignment of reference-type variables, which copies a reference to the object but not the object itself." [C# reference manual]*

# Modelling value types in Java

We can model a value type in Java in two ways

- use `final` to stop modification; or
- use *deep copy* to snapshot values on operations.

Both of these are fragile during program enhancement.

For example a routine may copy an object to another object, but program enhancement may change a value type into a reference type, resulting in code elsewhere getting a value which is *part copied from, and part aliased with,* a source object.

- An object-copy routine which simply copies the fields of an object is known as *shallow copy*
- An object-copy routine which recursively copies the fields of an object is known as *deep copy*

# Value types and inefficient representation

Think how we might represent an array of size 100 of complex values in Java.

- We'll have an array of 100 pointers (around 800 bytes) and 100 complex values (perhaps 24 bytes each).
- But at machine level we really just want to store 200 `double` values (only 1600 bytes).

This costs Java programs in memory use, cache effects and instructions executed.

- Java would benefit from C#-like value types.

Value types form part of the 'Valhalla project' originally planned for Java 10:

http://en.wikipedia.org/wiki/Project_Valhalla_(Java_language)

Incidentally, the Wikipedia entry on value types needs improving (April 2018):

https://en.wikipedia.org/wiki/Value_type

# Aliasing and mutability

Mixing aliasing and mutability – as Java does – can enable subtle bugs. Adding concurrency enables more subtle bugs (and more-subtle bugs!).

There appear to be two ‘easy-to-use’ sweet spots to use current languages:

- use mutability freely, but use deep copy so that mutations don’t accidentally update some other data structure.
- use aliasing freely, but make data structures immutable.

*Ideally,* future languages will allow these two techniques to be mixed and perhaps also allow aliasing and mutability to be used together locally without impacting whole-program maintainability. Rust (see later) incorporates a notion of ‘ownership’ into its type system to address some of these issues.

# ~ *Topic XI* ~

## Miscellaneous (entertaining) concepts

- Monadic I/O
- Generalised Algebraic Data Types (GADTs)
- Continuation-passing style (CPS) and call/cc
  `Wikipedia:Call-with-current-continuation`

# I/O in functional languages

The ML approach: “evaluation is left-right, just let side-effecting I/O happen as in C or Java”.

- Breaks *referential transparency* (‘purity’), e.g. that $e + e$ and $\textbf{let } x = e \textbf{ in } x + x$ should be equal.
- Not an appropriate solution for lazy languages (order of side effects in arguments in a function call would depend on the detail of the called function).
- Haskell is a lazy function language (“laziness keeps us pure”).

Everything I say about I/O applies to other side-effecting operations, e.g. mutable variables, exceptions, backtracking . . .

# Giving different types to pure and impure functions

Suppose we have two types $A \to B$ for functions with no side-effects and $A \leadsto B$ for impure functions.

- Then everything with a side effect would be visible in its type (contagious).
- (Later we might have ways of hiding “locally impure” functions within a pure function, but this doesn’t apply to I/O.)
- Instead of writing $A \leadsto B$ we write $A \to B\ M$ where $M$ is a unary type constructor called a *monad*.
  Factors the idea of ‘call a function’ and ‘do the resulting *computation*’.

Syntax: ML type constructors are postfix so we write $t\ list$ or $B\ M$ whereas Haskell writes $M\ B$ and perhaps $List\ t$.

# So, how does I/O work?

In ML we might read from and write to stdin/stdout with (writing ⇝ for emphasis):

```
MLrdint: unit ⇝ int
MLwrint: int ⇝ unit
```

Using monads (either in Haskell or ML) these instead have type

```
rdint: int IO
wrint: int -> unit IO
```

- The unary type constructor `IO` is predefined, just like the binary type constructor ‘`->`’.
- Shouldn’t we have `rdint: unit -> int IO`?
  We could, but this would be a bit pointless—writing takes an argument and gives a computation, but reading from stdin is just a computation.

# Composing I/O functions

## or Sequencing I/O effects

- In ML we could just write *e*; *e*′ to sequence the side effects of *e* and those in *e*′. But now all ‘effects’ like I/O are part of monadic values, and no longer ‘side effects’—so this doesn’t work
- Instead every monad `M` (including `IO` as a special case) has two operators: one to sequence computations and one to create an empty computation.

```
(>>=): 'a M -> ('a -> 'b M) -> 'b M
       (* infix, pronounced `bind' *)
return: 'a -> 'a M
```

# Digression – Monad Laws

The idea of monad originates in mathematics, so these operators have axioms relating `>>=` and `return` in the *mathematics*; these are seem as laws which all well-behaved *programming* monads satisfy.

```
[left unit]
  m >>= return      =  m
[right unit]
  return x >>= f    =  f x
[associativity]
  (m >>= f) >>= g   =  m >>= λx.(f x >>= g)
```

(The bind operator '`>>=`' syntactically groups to the left so the brackets in the final line are redundant.)

# Using the IO monad

In a system using monadic I/O, for example Haskell, the read-eval-print loop not only deals with pure values (integers, lists and the like), but has a *special treatment* for values in the `IO` monad. Given a value of type `t IO`, it:

- performs the side effects in the monad; then
- prints the resulting value of type `t`.

So the question is: how do we make a computation which (say) reads an integer, adds one to it, and then prints the result?

# Using the IO monad – examples

Read an integer, adds one to it, and print the result (using ML syntax):

```
rdint >>= (fun x -> wrint(x+1));
```

Do this twice:

```
let doit = rdint >>= (fun x -> wrint(x+1))
in doit >>= (fun () -> doit);;
```

Note that `doit` has type `unit IO`, so we use `>>=` to use `doit` twice.

NB: *computations* are not *called* as they are not functions; they are *combined* using `>>=` as in the above examples.

# Using the IO monad – examples (2)

Read pairs of numbers, multiply them, printing the sum of products so far, until the product is zero.

```
let rec foo s = rdint >>= fun x ->
                rdint >>= fun y ->
                if (x*y = 0) then return ()
                else wrint(s + x*y) >>= fun () ->
                  foo(s + x*y);
foo 0;;
```

Note the type of `foo` is `int -> unit IO`.
Note also the use of `return` to give a ‘do nothing’ computation.

# Practical use

The Mirage OS (`mirage.io`) is written in OCaml in a monadic style using the Lwt library. Haskell makes heavy use of monads. GHCi is just like the ML read-eval-print loop, but remember:

- Haskell syntax uses upper case for constants (types, constructors) and lower case for variables.
- Haskell swaps ':' and '::' relative to ML
- Type constructors are prefix
- `fun x->e` is written `\x->e`

Haskell also provides `do`-notation to allow programmers to write imperative-looking code which de-sugars to uses of `return` and `>>=`.

# A monad in Haskell

Common monads in Haskell are the *Maybe* monad (below), the State monad, or the IO monad (later).

```
data Maybe a = Nothing | Just a

return :: a -> Maybe a
return x  = Just x

(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
           Nothing -> Nothing
           Just x  -> g x
```

# Other monads

- Many other unary type constructors have natural monadic structure (at least as important as `IO`).
- For example, using Haskell syntax, `List t` and `Maybe t`. Another important one is `State s t` of computations returning a value of type `t`, but which can mutate a state of type `s`. (Subtlety: the monad is legally the unary type `State s` for some given s.)
- Haskell overloads `>>=` and `return` to work on all such types (Haskell's *type class* construct facilitates this).
- The common idea is 'threading' some idea of state implicitly through a calculation.

Haskell example using `List` in GHCi:

```
[1,2,3] >>= \x->if x==2 then return 5 else [x,x]
[1,1,5,3,3]
```

# Generalised Algebraic Data Types (GADTs)

OCaml data type (just like datatype in ML):

```
type 'a mylist = MyNil | MyCons of 'a * 'a mylist;;
```

Can also be written

```
type 'a mylist = MyNil : 'a mylist
               | MyCons : 'a * 'a mylist -> 'a mylist;;
```

Why bother (it's longer and duplicates info)?

How about this:

```
type _ exp = Val : 'a -> 'a exp
           | Eq : 'a exp * 'a exp -> bool exp
           | Add : int exp * int exp -> int exp
```

Allows `bool exp` values to be checked that `Add`, `Eq` etc. are used appropriately. E.g.

`Val 3: int exp` √

`Val true: bool exp` √

`Add(Val 3, Val 4): int exp` √

`Add(Val 3, Val true)` ×

`Eq(Val true, Val false): bool exp` √

`Eq(Val 3, Val 4): bool exp` √

`Eq(Val 3, Val true)` ×

- GADTs are a special case of *dependent types* (later this lecture).

Can even write `eval` where the ***type*** of the result depends on the ***value*** of its type:

```
let rec eval = function
 | Val(x) -> x
 | Eq(x,y) -> eval(x) = eval(y)
 | Add(x,y) -> eval(x) + eval(y);;

eval: 'a exp -> 'a
```

(Some type-checking dust being swept under the carpet here.)

# *Topic XII*

## Concepts growing in importance recently

- Type-managed storage [example: Rust]
- WebAssembly
- Resumable exceptions/algebraic effect handlers [examples: OCaml]

# Approaches to storage allocation

- *Manual*, e.g. C/C++
  Danger: user-incompetence (use after free)
- *Automatic*, e.g. Java, Python
  Danger: unexpected delays (GC in a flight controller???)
- *Ban it*, MISRA (motor industry embedded coding standard)
  Rule 18-4-1 “Dynamic heap memory allocation shall not be used.”
- *Type-managed* – Rust
  Upside: type system ensures memory- and thread-safety and automatically adds calls to deallocate storage. No GC.
  Downside: search online for “Rust hard to learn” – but this is perhaps an advantage for smart programmers!
  Fact: Rust is ranked 19 in the 2022 RedMonk language rankings – so there are plenty of smart programmers and interesting companies about!

# Storage management is more than allocation and deallocation

Regardless of manual or automatic deallocation there's a wider issue: *ownership* (related to the sweet spots on slides 222)

- If I pass a mutable object to be incorporated into a global datastructure, then logically the call transfers ownership from the caller to the callee, so I should never refer to it again – just like `free`!
- When an API call returns me a record (which I plan to mutate) is the caller or returner (callee) responsible for copying it?
- Can we check this sort of property at compile time? Types?

Reasoning about ownership is more general than reasoning about manual deallocation (freeing an object is just like passing ownership to the pool of free memory).

# Type Systems – weakness

In traditional type systems $\Gamma \vdash e : t$, variables have the same type throughout the scope that introduce them.

This means that the three errors in the following program can’t be detected by the type system:

```
{ char *x = malloc(10); // x has type char *
  foo(x);           // x still has type char *
  free(x);          // x has type char * (but should not??)
  foo(x);           // a use-after-free disaster...
  x = malloc(20);   // type char * is right again for x
  x = malloc(30);   // an unfaulted memory leak
}                   // and another one (x gone out-of-scope)
```

Replacing `free(x)` with `AddToGlobalDataStructure(x)` shows classical types are equally weak at controlling sharing of data in languages like Java (details on next slide).

# Type Systems – weakness (2)

```
{ char *x = malloc(sizeof(SomeRecord));
  x->field1 = 4; x->field2 = 5;
  AddToGlobalDataStructure(x);
  // x = malloc(sizeof(SomeRecord));
  x->field1 = 8; x->field2 = 9;
  AddToGlobalDataStructure(x);
  // free(x);
}
```

Is there a bug in this code? [Almost certainly.] But what is it?

- AddToGlobalDataStructure wants to take ownership of `x`, so the problem is due to commenting out line 4?
- AddToGlobalDataStructure merely reads the fields of `x`, so the problem is due to commenting out line 7?
- Common error in API-understanding (= bugs) – needs documentation (formal or informal).
- How can we document this formally? [Answer: “Use Rust”]

# Solutions

- Code it all in the type system: *linear types*, *substructural types*, *separation logic*.
- Treat reference types as having two attributes: *type* and *ownership*.
- Just different phrasing, Rust follows the latter.
- Rust performs type-checking then runs the *borrow checker*.

*Borrow*? Well, if we have owners we might also lend and borrow, right?

[Thanks to Brendan Coll (CST Part II 2021/22) for his comments on these notes on Rust.]

# Rust by example

(from the manual)

- Rust types look a bit like Java types with C-like qualifiers, but be careful.
- `Box<i32>` is like `ref` in ML or a boxed `int` in Java:

```
fn create_box() {
  let _box1 = Box::new(3i32);  // ref to heap int
  // '_box1' is destroyed ('dropped') and its memory freed
  // Resembles C++ RAII/destructors.
}
// destroy_box takes ownership of an item of
// heap-allocated memory (default call-by-value)
fn destroy_box(c: Box<i32>) {
  println!("Destroying a box that contains {}", c);
  // 'c' goes out of scope here and is deallocated.
}
```

- Passing an object by value passes its ownership, and owners can destroy things (think cars etc.)

But but but . . .

# Copying and passing by value as similar

Rust calls `let b=a` below (and also passing by value) a *move*.

```
fn main() {
  let x = 5u32; // stack allocated int
  let y = x;    // copy x to y
  println!("x is {}, and y is {}", x, y); // use both
// BUT:
  let a = Box::new(5i32); // stack ref to heap-allocated int
  println!("a contains: {}", a); // (borrows the ref)

  let b = a; // copy a to b -- transfers ownership
  println!("a contains: {}", a); // error, ’a’ not owner

  destroy_box(b);
  println!("b contains: {}", b); // error, ’b’ not owner
}
```

Can we really write programs?

# Mutability

Mutability of data can be changed when ownership is transferred (think why). Note `*` is not quite like C:

```
fn main() {
  let immutable_box = Box::new(5u32);
  println!("immutable_box contains {}", immutable_box);

  *immutable_box = 4; // error

  // *Move* the box, changing the ownership (and mutability)
  let mut mutable_box = immutable_box;

  println!("mutable_box contains {}", mutable_box);  // 5
  *mutable_box = 4;
  println!("mutable_box now contains {}", mutable_box); // 4
}
```

# Borrowing

```
fn borrow_box(x: &Box<i32>) {
    println!("A borrowed box (see below): {}={}" &x, x); }

fn borrow_twice(x: &Box<i32>, y: &Box<i32>) {
    println!("Borrow two boxes: {}, {}", x, y); }

fn borrow_and_eat(x: &Box<i32>, y: Box<i32>) {
    println!("Borrow box {}, and destroy box {}", x, y); }

fn main() {
    let b = Box::new(5i32);
    let c = Box::new(6i32);
    borrow_box(&b); // 5=5
    borrow_twice(&b,&c); // 5,6
    borrow_twice(&b,&b); // 5,5
    borrow_and_eat(&b,c); // 5,6
    borrow_and_eat(&b,b); // error (borrow checker)
}
```

Subtlety: `println!` is a macro which implicitly inserts `&` before borrowed arguments, see `borrow_box` above using `x` twice.

# Borrowing – more details

- Can borrow (`&mut`) a mutable object once, or an immutable object many times (cf. Multiple Reader Single Writer (MRSW) in concurrent systems). Good for concurrent access, and also for the AddToGlobalDataStructure example earlier.
- Rust's borrowing discipline prevents unsafe uses of aliasing.
- Can borrow parts of an object
- All borrowing must be completed before ownership can be transferred
- Gives memory safety – no referencing freed memory, and pretty well avoids memory leaks. [Subtleties: you can break memory safety by using `unsafe` to cheat the type system; you can leak memory if you really try but not using examples we have seen.]
- Rust advocates claim that these rules are an acceptable the sweet spot to ensure memory safety.

# WebAssembly

- “Nothing to do either with the web, or with assembly.”
- A new bytecode that execute with near-native performance, with memory-safety guaranteed, in a hardware/language/platform independent way.
- Designed from day 1 with a formal executable specification: `https://github.com/WebAssembly/spec`.
- Has an efficient and portable representation that is compact, modular, efficient, streamable, parallelisable and portable.
- Is securely sandboxed by default with no ambient access to the computing environment in which code is executed.

Wasm was built as a *virtual instruction set architecture* that can then be compiled into the native CPU on demand.

# WebAssembly decompilation

Consider some simple C code:

```
typedef struct  float x, y, z;  vec3;

float dot(const vec3 *a, const vec3 *b) {
    return a->x * b->x + a->y * b->y + a->z * b->z;
}
```

Which compiles to the following:

```
(func $dot (type 0) (param i32 i32) (result f32)
  (f32.add (f32.add (f32.mul
    (f32.load (local.get 0))
        (f32.load (local.get 1)))
    (f32.mul (f32.load offset=4  (local.get 0))
        (f32.load offset=4 (local.get 1))))
    (f32.mul (f32.load offset=8 (local.get 0))
      (f32.load offset=8  (local.get 1))))))
```

See https://v8.dev/blog/wasm-decompile.