Concepts in Programming Languages

Alan Mycroft[a]

Computer Laboratory
University of Cambridge

2014–2015 (Easter Term)

http://www.cl.cam.ac.uk/teaching/1415/ConceptsPL/

[a]Notes largely due to Marcelo Fiore—but errors are my responsibility.

Practicalities

- Course web page:

  `www.cl.cam.ac.uk/teaching/1415/ConceptsPL/`

  with *lecture slides*, *exercise sheet*, and *reading material*.

- One exam question.

Main books

- 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.
- ⋆ M. L. Scott. *Programming language pragmatics* (2ND EDITION). Elsevier, 2006.
- R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013.

Context : so many programming languages

Peter J. Landin: “The Next 700 Programming Languages”, CACM >>>>1966<<<<.

Some programming-language ‘family trees’ (too big for slide):

`http://www.oreilly.com/go/languageposter`

`http://www.levenez.com/lang/`

`http://rigaux.org/language-study/diagram.html`

`http://www.rackspace.com/blog/infographic-evolution-of-computer-languages/`

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

Topics

I. Introduction and motivation.

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

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

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

V. *Object-oriented* languages — Concepts and origins: Simula (1964–67), Smalltalk (1971–80).

VI. Languages for *concurrency and parallelism*.

VII. *Types* in programming languages: ML (1973–1978).

VIII. *Data abstraction* and *modularity*: SML Modules (1984–97).

IX. A *modern language design*: Scala (2007)

X. Miscellaneous concepts

~ Topic I ~

Introduction and motivation

References:

- **Chapter 1** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.
- **Chapter 1** of *Programming languages: Design and implementation* (3RD EDITION) by T. W. Pratt and M. V. Zelkowitz. Prentice Hall, 1999.
- **Chapter 1** of *Programming language pragmatics* (2ND EDITION) by M. L. Scott. Elsevier, 2006.

Goals

- Critical *thinking* about programming languages.

  $\boxed{\textbf{?}}$ What is a programming language!?

- *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 improve the use of familiar languages.
- To increase the vocabulary of useful programming constructs.
- To allow a better choice of programming language.
- To make it easier to learn a new language.
- To make it easier to design a new language.
- To simulate useful features in languages that lack them.
- To make better use of language technology wherever 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.

Influences

- Computer capabilities.
- Applications.
- Programming methods.
- Implementation methods.
- Theoretical studies.
- Standardisation.

Applications domains

Era	Application	Major languages	Other languages
1960s	Business	COBOL	Assembler
	Scientific	FORTRAN	ALGOL, BASIC, APL
	System	Assembler	JOVIAL, Forth
	AI	LISP	SNOBOL
Today	Business	COBOL, SQL, spreadsheet	C, PL/I, 4GLs
	Scientific	FORTRAN, C, C++ Maple, Mathematica	BASIC, Pascal
	System	BCPL, C, C++	Pascal, Ada, BASIC, MODULA
	AI	LISP, Prolog	
	Publishing	TEX, Postscript, word processing	
	Process	UNIX shell, TCL, Perl	Marvel, Esterel
	New paradigms	Smalltalk, SML, Haskell, Java Python, Ruby	Eiffel, C#, Scala

$\boxed{?}$ Why are there so many languages?

- Evolution.
- Special purposes.
- No one language is good at expressing all programming styles.
- Personal preference.

Motivating application in language design

A specific purpose provides *focus* for language designers; it helps to set criteria for making design decisions.

A specific, motivating application also helps to solve one of the hardest problems in programming language design: deciding which features to leave out.

Examples: Good languages designed with a specific purpose in mind.

- LISP: symbolic computation, automated reasoning
- FP: functional programming, algebraic laws
- BCPL: compiler writing
- Simula: simulation
- C: systems programming
- ML: theorem proving
- Smalltalk: Dynabook
- Clu, SML Modules: modular programming
- C++: object orientation
- Java: 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

- Imperative

procedural	C, Ada, **Pascal**, **Algol**, **FORTRAN**, ...
object oriented	**Scala**, C#,Java, **Smalltalk**, **SIMULA**, ...
scripting	Perl, Python, PHP, ...

- Declarative

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

Theoretical foundations

Examples:

- Formal-language theory.
- Automata theory.
- Algorithmics.
- λ-calculus.
- Semantics.
- Formal verification.
- Type theory.
- Complexity theory.
- Logic.

Standardisation

- Proprietary standards.
- Consensus standards.
  - ANSI (American National Standards Institute)
  - IEEE (Institute of Electrical and Electronics Engineers)
  - BSI (British Standard Institute)
  - ISO (International Standards Organisation)

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.
- **!** Try it and see!
- **!** Read the ANSI C Standard.

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.

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

Deprecated features.

Language standards :
Accidental misspecification

Various examples (we'll see "function types in Algol" later).

In language PL/1 the type `DEC(p,q)` means `p` digits with `q` after the decimal point.

$\boxed{?}$ So what value does the following expression have:

```
9 + 8/3
```

Suggestions:

- `11.666...` ?
- Overflow ?
- `1.666...` ?

`DEC(p,q)` means `p` digits with `q` after the decimal point.

Type rules for `DEC`IMAL in PL/1:

```
DEC(p1,q1) + DEC(p2,q2)
  => DEC(MIN(1+MAX(p1-q1,p2-q2)+MAX(q1,q2),15),MAX(q1,q2))
DEC(p1,q1) / DEC(p2,q2)
  => DEC(15,15-((p1-q1)+q2))
```

For `9 + 8/3` we have:

```
DEC(1,0) + DEC(1,0)/DEC(1,0)
  => DEC(1,0) + DEC(15,15-((1-0)+0))
  => DEC(1,0) + DEC(15,14)
  => DEC(MIN(1+MAX(1-0,15-14)+MAX(0,14),15),MAX(0,14))
  => DEC(15,14)
```

So the calculation is as follows

```
9 + 8/3
  -> 9 + 2.66666666666666
  ->    11.66666666666666    // out of range for DEC(15,14)
  ->       (OVERFLOW)
  ->     1.66666666666666    //  if OVERFLOW disabled
```

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.

1986–90: C++, **SML**, Haskell.

1991–95: Ada 95, TCL, Perl.

1996–2000: Java.

2000–05: C#, Python, Ruby, **Scala**.

1990– : Open/MP, MPI, Posix threads, Erlang, X10, MapReduce, Java 8 features.

For more information:

en.wikipedia.org/wiki/History_of_programming_languages

Language groups

- Multi-purpose languages
  - Scala, C#, Java, C++, C
  - Haskell, SML, Scheme, LISP
  - Perl, Python, Ruby
- Special-purpose languages
  - UNIX shell
  - SQL
  - $\LaTeX$

Things to think about

- What makes a good language?
- The role of
  1. motivating applications,
  2. program execution,
  3. theoretical foundations

  in language design.
- Language standardisation.

~ Topic II ~

FORTRAN : A simple procedural language

References:

- **Chapter 10(§1)** of *Programming Languages: Design and implementation* (3RD EDITION) by T. W. Pratt and M. V. Zelkowitz. Prentice Hall, 1999.

- ***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.
- The first high-level programming language to become widely used. (At the time the utility of any high-level language was open to question!)

> The main complaint was the efficiency of compiled code. This heavily influenced the design, orienting it towards providing execution efficiency.

- Standards:

  1966, 1977 (FORTRAN 77), 1990 (Fortran 90), . . .
  2010 (Fortran 2008).
- Remains main language for scientific computing.
- Easier for a compiler to optimise than C.

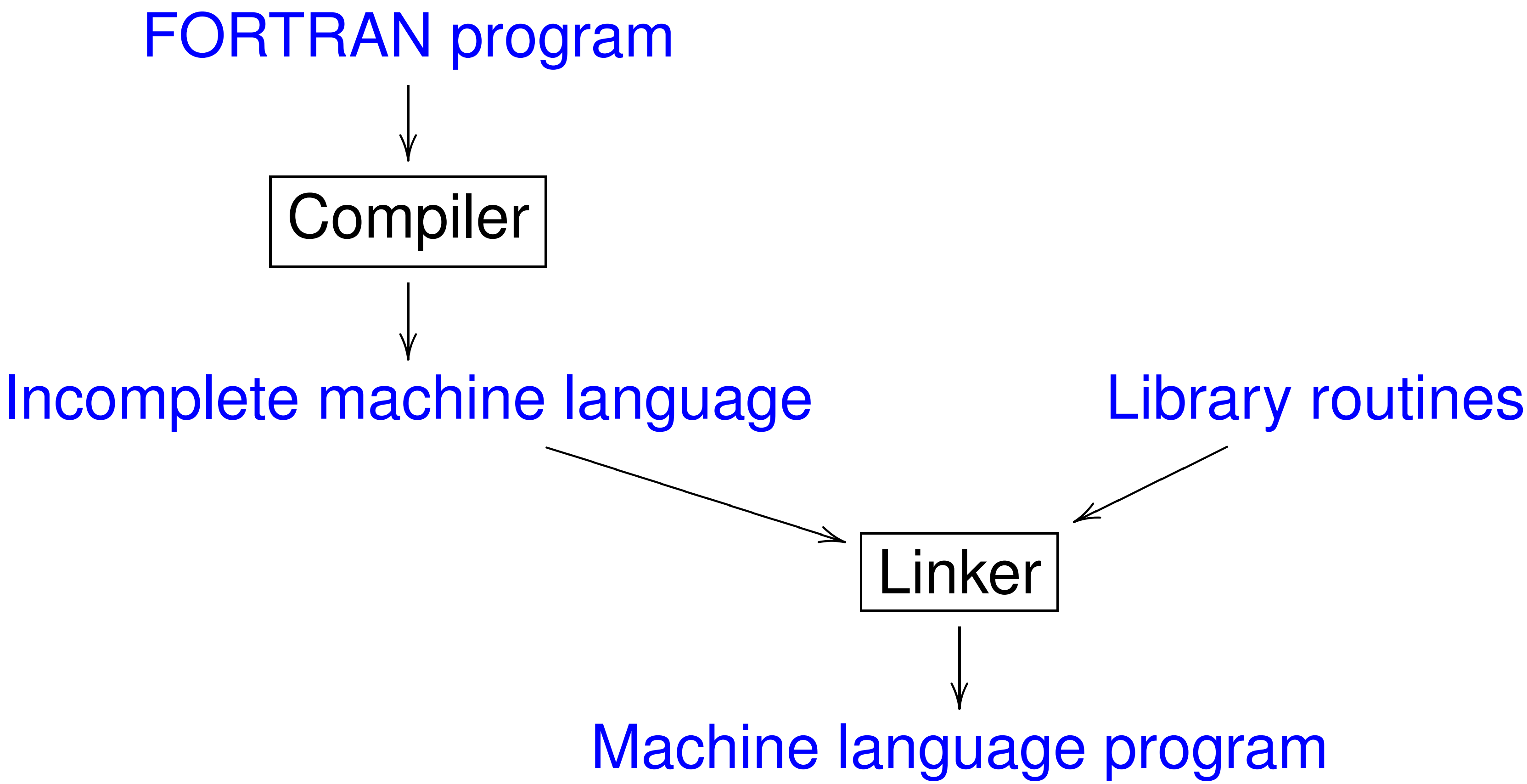
Overview

Execution model (traditional Fortran)

- FORTRAN program = main program + subprograms
  - Each is *compiled separately* from all others.
  - Translated programs are linked into final executable form during loading.
- All *storage is allocated statically* before program execution begins; no run-time storage management is provided.
- Flat register machine. *No stacks, no recursion*. Memory arranged as linear array.

Overview

Compilation



Overview

Data types

- Numeric data: Integer, real, complex, double-precision real.
- Boolean data. $\boxed{\text{called logical}}$
- Arrays. $\boxed{\text{of fixed declared length}}$
- Character strings. $\boxed{\text{of fixed declared length}}$
- Files.
- Fortran 90 added 'derived data types' (like C structs).

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

```
      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 (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. Failure may arise when the loader tries to merge subprograms with main program.
  Fortran 90 provides a module system.
- Function parameters are uniformly transmitted by reference (or value-result). Traditionally all allocation is done statically.
  But Fortran 90 provides dynamic allocation.
- A value is returned in a FORTRAN function by assigning a value to the name of a function.

Types

- Traditional FORTRAN had no user-defined types.
  Fortran 90 added ‘derived data types’ (like C structs).

- *Static type checking* is used in FORTRAN, but the checking is traditionally incomplete.

  Constructs that could not be statically checked were often left unchecked at run time.
  (An early preference for speed over ease-of-bug-finding still visible in languages like C.)

  Fortran 90 added a module system which enables checking across separately compiled subprograms.

Storage

Representation and Management

- Storage representation in FORTRAN is *sequential*.
- Only two levels of referencing environment are provided, *global* and *local*.

The sequential storage representation is critical in the definition of the `EQUIVALENCE` and `COMMON` declarations.

- EQUIVALENCE

```
REAL X
INTEGER Y
EQUIVALENCE (X,Y)
```

- COMMON

```
COMMON/BLK/X,Y,K(25)            in MAIN
COMMON/BLK/U,V,I(5),M(4,5)      in SUB
```

Aliasing

Aliasing occurs when two names or expressions refer to the same object or location.

- Aliasing raises serious problems for both the user and implementer of a language.
- Because of the problems caused by aliasing, new language designs sometimes attempt to restrict or eliminate altogether features that allow aliases to be constructed.

Parameters

There are two concepts that must be clearly distinguished.

- The parameter names used in a function declaration are called *formal parameters*.
- When a function is called, expressions called *actual parameters* are used to compute the parameter values for that call.

FORTRAN subroutines and functions

- *Actual parameters* may be simple variables, literals, array names, subscripted variables, subprogram names, or arithmetic or logical expressions.

  The interpretation of a *formal parameter* as an array is done by the called subroutine.

- Traditionally each subroutine is compiled independently and no checking is done for compatibility between the subroutine declaration and its call.

  Fortran 90 fixed this, including allowing `IN` and `OUT` specificiers on parameters.

- The language specifies that if a formal parameter is assigned to, the actual parameter must be a variable. This is a traditional source of bugs as this needs cross-module compilation checking:

  Example:

```
SUBROUTINE SUB(X,Y)
  X = Y
  END

CALL SUB(-1.0,1.0)
```

  Solution: use the Fortran 90 features.

- Parameter passing is uniformly by *reference*.

FORTRAN *lives!*

- Fortran is one of the first languages, and the only early language still alive.
- Lots of CS people will tell you about all the diseases of Fortran based on 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

References:

- **Chapter 3** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.
- **Chapters 5(§4.5) and 13(§1)** of *Programming languages: Design and implementation* (3RD EDITION) by T. W. Pratt and M. V. Zelkowitz. Prentice Hall, 1999.

LISP = LISt Processing
(±1960)

- Developed in the late 1950s and early 1960s by a team led by John McCarthy in 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).

Programming-language phrases

- *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.

Innovation in the design of LISP

- LISP is an expression-based language.

  Conditional expressions that produce a value were new in LISP.

Some contributions of LISP

- Lists.
- Recursive functions.
- Garbage collection.
- Programs as data.

Overview

- LISP syntax is extremely simple. To make parsing easy, all operations are written in prefix form (*i.e.*, with the operator in front of all the operands).
- LISP programs compute with *atoms* and *cells*.
- LISP is an *untyped* programming language.

- Most operations in LISP take list arguments and return list values.

Example:

`( cons '(a b c) '(d e f) )` cons-cell representation

Remark: The function `(quote x)`, or simply `'x`, just returns the literal value of its argument.

? 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 parentheses!

Static and dynamic scope

There are two main rules for finding the declaration of a global identifier:

- *Static scope*. A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text.
- *Dynamic scope*. A global identifier refers to the identifier associated with the most recent environment.

Abstract machines

The terminology *abstract machine* is generally used to refer to an idealised computing device that can execute a specific programming language directly. Systems people use *virtual machine* (as in JVM) for a similar concept.

LISP abstract machine

The abstract machine for Pure LISP has four parts:

1. A *LISP* expression to be evaluated.
2. A *continuation*, which is a function representing the remaining of the program to evaluate when done with the current expression.
3. An *association list*, also know as the *A-list*.
4. A *heap*, which is a set of *cons cells* (or *dotted pairs*) that might be pointed to by pointers in the A-list.

Garbage collection

McCarthy (1960)

. . . When a free register is wanted, and there is none left on the free-storage list, a reclamation cycle starts.

$\boxed{\text{Garbage collection}}$

In computing, *garbage* refers to memory locations that are not accessible to a program.

Garbage collection is the process of detecting garbage during the execution of a program and making it available.

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`.

Parameter passing in LISP

The *actual parameters* in a function call are always expressions, represented as lists structures.

LISP provides two main methods of *parameter passing*:

- *Pass/Call-by-value.* The most common method is to evaluate the expressions in the actual-parameter list, and pass the resulting values.

- *Pass/Call-by-name.** A less common method is to transmit the expression in the actual parameter list *unevaluated*, and let the call function evaluate them as needed using `eval`.

  The programmer may specify transmission by name using `nlambda` in place of `lambda` in the function definition.

Strict and lazy evaluation

Example: Consider the following function definitions with parameter-passing by value.

```
( defun CountFrom(n) ( CountFrom(+ n 1) ) )

( defun FunnyOr(x y)
    ( cond ( x 1) ( T y ) )
  )

( defun FunnyOrelse(x y)
    ( cond ( (eval x) 1) ( T (eval y) ) )
  )
```

$\boxed{\textbf{?}}$ What happens in the following calls?

```
( FunnyOr T (CountFrom 0) )
( FunnyOr nil T )

( FunnyOrelse 'T '(CountFrom 0) )
( FunnyOrelse 'nil 'T )
```

~ Topic IV ~

Block-structured procedural languages
Algol and Pascal

References:

- **Chapters 5 and 7,** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.

- **Chapters 10(§2) and 11(§1)** of *Programming languages: Design and implementation* (3RD EDITION) by T. W. Pratt and M. V. Zelkowitz. Prentice Hall, 1999.

Parameter passing

The way that actual parameters are evaluated and passed to procedures depends on the programming language and the kind of *parameter-passing mechanisms* it uses.

The main distinction between different parameter-passing mechanisms are:

- the time that the actual parameter is evaluated, and
- the location used to store the parameter value.

NB: The *location* of a variable (or expression) is called its *L-value*, and the *value* stored in this location is called the *R-value* of the variable (or expression).

Parameter passing

Pass/Call-by-value

- In *pass-by-value*, the actual parameter is evaluated. The value of the actual parameter is then stored in a new location allocated for the function parameter.

- Under *call-by-value*, a formal parameter corresponds to the value of an actual parameter. That is, the formal `x` of a procedure `P` takes on the value of the actual parameter. The idea is to evaluate a call `P(E)` as follows:

  `x := E`;

  execute the body of procedure `P`;

  if `P` is a function, return a result.

Parameter passing

Pass/Call-by-reference

- In *pass-by-reference*, the actual parameter must have an L-value. The L-value of the actual parameter is then bound to the formal parameter.

- Under *call-by-reference*, a formal parameter becomes a synonym for the location of an actual parameter. An actual reference parameter must have a location.

Example:

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

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

The difference between call-by-value and call-by-reference is important to the programmer in several ways:

- **Side effects.**
- **Aliasing.**
- **Efficiency.**

Parameter passing

Pass/Call-by-value/result

Call-by-value/result is also known as *copy-in/copy-out* because the actuals are initially copied into the formals and the formals are eventually copied back out to the actuals.

Examples:

- 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(in: Integer; var out: Real);
  ```

- The only parameter-passing method in C is call-by-value; however, the effect of call-by-reference can be achieved using pointers. In C++ true call-by-reference is available using *reference parameters*.

- 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.

Parameter passing

Pass/Call-by-name

The Algol 60 report describes *call-by-name* .

Block structure

- In a *block-structured language*, each program or subprogram is organised as a set of nested blocks.

  A *block* is a region of program text, identified by begin and end markers, that may contain declarations local to this region.

- Block structure was first defined in Algol. Pascal contains nested procedures but not in-line blocks; C contains in-line blocks but not nested procedures; Ada supports both.

- *Block-structured languages* are characterised by the following properties:
  - New variables may be declared at various points in a program.
  - Each declaration is visible within a certain region of program text, called a block.
  - When a program begins executing the instructions contained in a block at run time, memory is allocated for the variables declared in that block.
  - When a program exits a block, some or all of the memory allocated to variables declared in that block will be deallocated.

- An identifier that is not declared in the current block is considered global to the block and refers to the entity with this name that is declared in the closest enclosing block.

Algol

HAD A MAJOR EFFECT ON LANGUAGE DESIGN

- 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 DESCENDANTS LIVE ON!

Algol 60

Features

- Simple statement-oriented syntax.
- Block structure.
- 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.
  - 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

- One contribution of Algol 68 was its *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`, `structure`, `procedure`, `set`, `pointer`).

  Type constructions could be combined without restriction. This made the type system seem more systematic than previous languages.

- Algol 68 memory management involves a *stack* for local variables and *heap* storage. Algol 68 data on the heap 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.)

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.

Pascal

- 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.

- Problematically, in Pascal, the index type of an array is part of its type. The Pascal standard defines *conformant array parameters* whose bounds are implicitly passed to a procedure. The Ada programming language uses so-called *unconstrained array types* to solve this problem.

  The subscript range must be fixed at compile time permitting the compiler to perform all address calculations during compilation.

```
procedure Allowed( a: array [1..10] of integer ) ;
procedure
  NotAllowed( n: integer;
              a: array [1..n] of integer ) ;
```

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                        { datatype                    }
UBtree = record             {      'a UBtree = record of   }
  value: integer ;          {           'a * 'a UBkind     }
  case k: kind of           { and 'a UBkind =              }
    unary: ^UBtree ;        {           unary of 'a UBtree }
    binary: record          {         | binary of          }
      left: ^UBtree ;       {             'a UBtree *      }
      right: ^UBtree        {             'a UBtree ;      }
      end
end ;
```

~ Topic V ~

Object-oriented languages : Concepts and origins SIMULA and Smalltalk

References:

- ⋆ **Chapters 10 and 11** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.

- ♦ **Chapters 8, and 12(§§2 and 3)** of *Programming languages: Design and implementation* (3RD EDITION) by T. W. Pratt and M. V. Zelkowitz. Prentice Hall, 1999.

Objects in ML !?

```
exception Empty ;
fun newStack(x0)
  = let val  stack = ref [x0]
    in ref{  push = fn(x)
               => stack := ( x :: !stack )     ,
             pop = fn()
               => case !stack of
                    nil => raise Empty
                  | h::t => ( stack := t; h )
    }end ;
```

```
exception Empty
val newStack = fn :
      'a -> {pop:unit -> 'a, push:'a -> unit} ref
```

NB:

- $\boxed{\textbf{!}}$ The *stack discipline of Algol* for activation records fails!
- $\boxed{\textbf{?}}$ Is ML an object-oriented language?

  $\boxed{\textbf{!}}$ Of course not!

  $\boxed{\textbf{?}}$ Why?

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* means that when a message is sent to an object, the method to be executed is selected dynamically, at run time, according to the implementation of the object that receives the message.

There is a family of object-oriented languages that is based on the *"run-time overloading"* view of dynamic lookup. The most prominent design of this form is CLOS (= Common Lisp Object System), which features *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.

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 `a` has all the functionality of another object `b`, then we may use `a` in any context expecting `b`.

- The basic principle associated with subtyping is *substitutivity*: If `A` is a subtype of `B`, then any expression of type `A` may be used without type error in any context that requires an expression of type `B`.

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.

History of objects
SIMULA and Smalltalk

- Objects were invented in the design of SIMULA and refined in the evolution of Smalltalk.

- SIMULA: The first object-oriented language.

  The object model in SIMULA was based on procedures activation records, with objects originally described as procedures that return a pointer to their own activation record.

- 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.

SIMULA

- Extremely influential as the first language with classes objects, dynamic lookup, subtyping, and inheritance.
- Originally designed for the purpose of *simulation* by O.-J. Dahl and K. Nygaard at the Norwegian Computing Center, Oslo, in the 1960s.
- SIMULA was designed as an extension and modification of Algol 60. The main features added to Algol 60 were: class concepts and reference variables (pointers to objects); pass-by-reference; input-output features; coroutines (a mechanism for writing concurrent programs).

- A generic event-based simulation program

```
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. $\leadsto$ subtyping
- The selection of the simulation operation according to the kind of event being processed. $\leadsto$ dynamic lookup
- Ways in which to structure the implementation of related kinds of events. $\leadsto$ inheritance

SIMULA

Object-oriented features

- *Objects*: A SIMULA object is an 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 but was added later and used as the basis for C++.

- *Subtyping*: Objects are typed according to the classes that create them. Subtyping is determined by class hierarchy.

- *Inheritance*: A SIMULA class may be defined, by class prefixing, as an extension of a class that has already been defined including the ability to redefine parts of a 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***
```

SIMULA

Subclasses and inheritance

SIMULA syntax for a class C1 with subclasses C2 and C3 is

```
CLASS C1
  <DECLARATIONS1>;
C1 CLASS C2
  <DECLARATIONS2>;
C1 CLASS C3
  <DECLARATIONS3>;
```

When we create a C2 object, for example, we do this by first creating a C1 object (activation record) and then appending a C2 object (activation record).

Example:

```
POINT CLASS COLOREDPOINT(C); COLOR C;
BEGIN
  BOOLEAN PROCEDURE EQUALS(Q); REF(COLOREDPOINT) Q;
    ...;
END***COLOREDPOINT**
```

```
REF(POINT) P;  REF(COLOREDPOINT) CP;
P :- NEW POINT(1.0,2.5);
CP :- NEW COLOREDPOINT(2.5,1.0,RED);
```

NB: SIMULA 67 did not hide fields. Thus,

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

changes the color (colour) of the point referenced by `CP`.

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.

Examples:

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. An error in the original SIMULA type checker.

For `CLASS A; A CLASS B`, SIMULA subclassing produces the subtype relation `B <: A`.

But SIMULA also uses the semantically incorrect principle that, if `B<:A` then `REF(B)<:REF(A)`.

So, this code . . .

```
REF(A) a;  REF(B) b;

PROCEDURE ASSIGNa( REF(A) x )
  BEGIN  x :- a  END;

ASSIGNa(b);
```

. . . will statically type check, but may cause a type error at run time.

Smalltalk

- The object metaphor was extended and refined.
  - 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.
  - Objects and classes were shown useful organising concepts for building an entire programming environment and system.

Smalltalk

Terminology

- *Object*: A combination of private data and functions. Each object is an *instance* of some class.
- *Class*: A template defining the implementation of a set of objects.
- *Subclass*: Class defined by inheriting from its superclass.
- *Selector*: The name of a message (analogous to a function name).
- *Message*: A selector together with actual parameter values (analogous to a function call).
- *Method*: The code in a class for responding to a message.
- *Instance variable*: Data stored in an individual object (instance class).

Smalltalk

Classes and objects

class name	`Point`
super class	`Object`
class var	`pi`
instance var	`x, y`
class messages and methods	
<. . . names and codes for methods . . .>	
instance messages and methods	
<. . . names and codes for methods . . .>	

Definition of `Point` class

A class message and method for point objects

```
newX:xvalue Y:yvalue ||
   ^ self new x: xvalue y: yvalue
```

A new point at coordinates $(3, 4)$ is created when the message

```
newX:3 Y:4
```

is sent to the `Point` class.

For instance:

```
p <- Point newX:3 Y:4
```

Smalltalk

Inheritance

class name	`ColoredPoint`
super class	`Point`
class var	
instance var	`color`
class messages and methods	
`newX:xv Y:yv C:cv`	<. . . code . . .>
instance messages and methods	
`color`	`||^color`
`draw`	<. . . code . . .>

Definition of `ColoredPoint` class

Smalltalk

Abstraction

Smalltalk rules:

- *Methods are public*.
- *Instance variables are protected*.

Smalltalk
Dynamic lookup

The run-time structures used for Smalltalk classes and objects support *dynamic lookup* in two ways.

1. Methods are selected through the receiver object.

2. Method lookup starts with the method dictionary of the class of the receiver and then proceeds upwards through the class hierarchy.

Smalltalk

Self and super

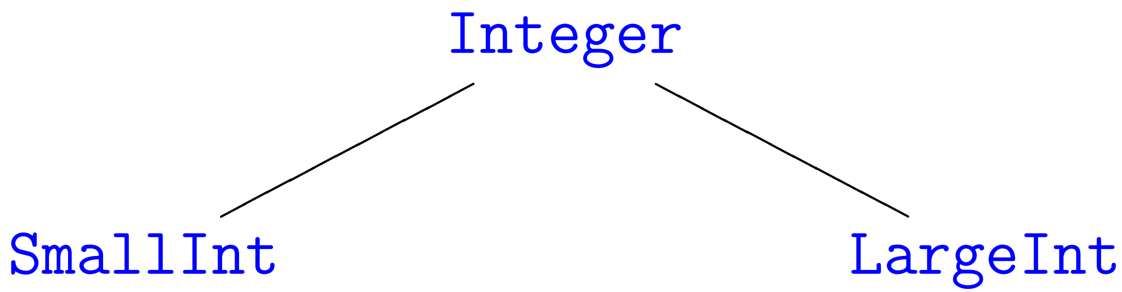
- The special symbol `self` may be used in the body of a Smalltalk method. The special property of `self` is that it always refers to the object that contains this method, whether directly or by inheritance.

- The special symbol `super` is similar to `self`, except that, when a message is sent to `super`, the search for the appropriate method body starts with the superclass of the object instead of the class of the object. This mechanism provides a way of accessing a superclass version of a method that has been overridden in the subclass.

Example: A factorial method

```
factorial ||
   self <= 1
      ifTrue: [^1]
      ifFalse: [^ (self-1) factorial * self]
```

in the `Integer` class for



Smalltalk

Subtyping

Type `A` is a *subtype* of type `B` if any context expecting an expression of type `B` may take any expression of type `A` without introducing a type error.

Semantically, in Smalltalk, it makes sense to associate *subtyping* with the *superset* relation on class interfaces.

Smalltalk

Object-oriented features

- *Objects*: A Smalltalk object is created by a class.
- *Classes*: A Smalltalk class defines variables, class methods, and the instance methods that are shared by all objects of the class.
- *Abstraction*: Abstraction is provided through protected instance variables. All methods are public but instance variables may be accessed only by the methods of the class and methods of subclasses.

- *Subtyping*: Smalltalk does not have a compile-time type system. Subtyping arises implicitly through relations between the interfaces of objects. Subtyping depends on the set of messages that are understood by an object, not the representation of objects or whether inheritance is used.

- *Inheritance*: Smalltalk subclasses inherit all instance variables and methods of their superclasses. Methods defined in a superclass may be redefined in a subclass or deleted.

~ Topic VI ~

Languages for Concurrency and Parallelism

Sources of parallel computing

Five main sources:

1. Theoretical models: PRAM, BSP (complexity theory), CSP, CCS, π-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, π-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, Hadoop, Skywriting.
8. [On Chip] Verilog, Bluespec.

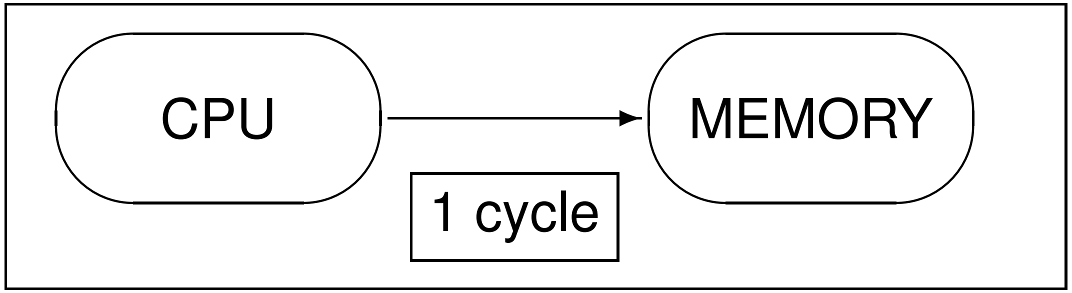
NB: Language features may fit multiple architectures.

Painful facts of parallel life

1. Single-core clock speeds have stagnated at around 3GHz for the last ten 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.)

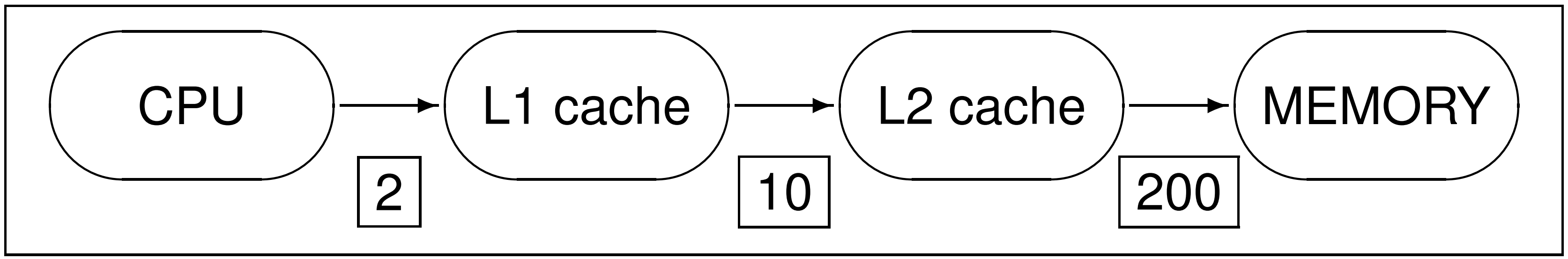
Takeaway: optimising performance requires exploiting parallelism, you'll have to program this yourself, and getting it wrong gives slow-downs and bugs due to races.

A programmer's view of memory



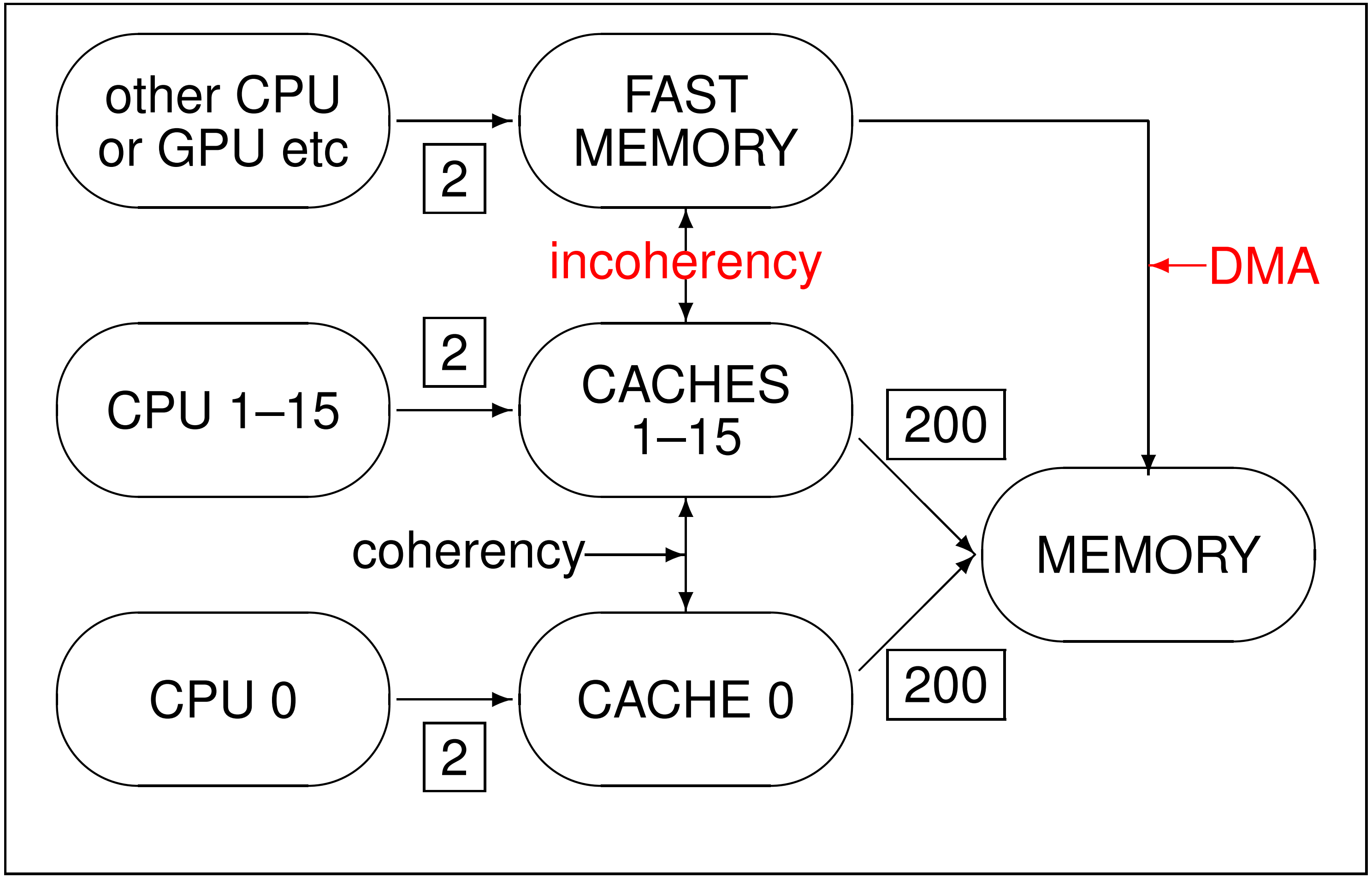
(This model was pretty accurate in 1985.)

A 2004-era single-core view of memory and timings

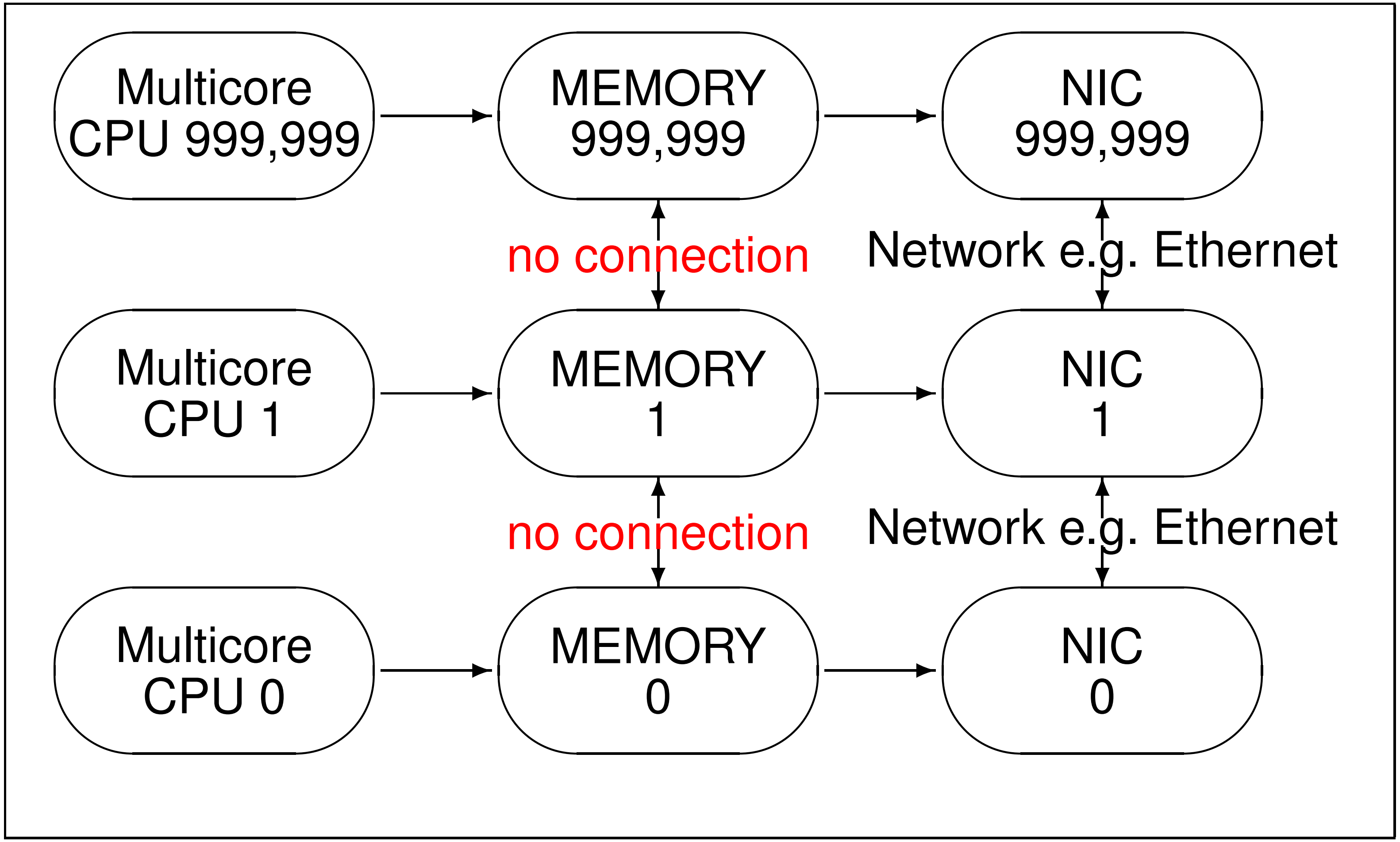


Multi-core-chip memory models

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



A Compute Cluster or Cloud-Computing Server



(The sort of thing which Google uses.)

Lecture topic: 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"
- 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: rest of this lecture.

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').

Ask: language $\Rightarrow$ programmer 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 α, $\overline{\alpha}$, can communicate with each other or the world (non-deterministically if multiple partners offered). Internal communication gives special internal action τ.

Behaviour $p ::= 0 \mid \alpha.p \mid p + p \mid p|p \mid X \mid \mathit{rec}\ X.p$
(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”

Part II course.

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.

New Part II course in 2014/15.

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.

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;
   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”.

Clusters/Cloud Computing

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?

More emphasis on message-passing . . .

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 (1)

Can mean either “doing one computer’s worth of work on a server instead of locally”. Google Docs. Or . . .

Cloud Computing (2)

. . . massively parallel combinations of computing, e.g. MapReduce invoked by a search engine.

MapReduce can match a search term against many computers (Map) each holding part of Google index of words, and then combine these result (Reduce).

Reduce here means parallel reduce (tree-like, logarithmic cost), not *foldl* or *foldr* from ML.

Functional style (idempotency) useful for error resilience (errors happen often in big computations). Try to ensure computation units are larger than cost of transmitting arguments and results. (also: Skywriting project in Cambridge)

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 X

- 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 which the mutator (user's program) is running. Hard?
- Incremental? Track imported/exported pointers?

Java 8: Internal vs External iteration

Can't trust users to iterate over data. They start with

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

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

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

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.toStream().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!). The Java 8 API ensures that a Stream pipeline like the above only traverses the data once.

~ Topic VII ~

Types in programming languages

References:

- **Chapter 6** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.
- **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.
- There are 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.

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.

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, pointer arithmetic
Almost safe	Pascal	Explicit deallocation; dangling pointers
Safe	LISP, SML, Smalltalk, Java	Type checking

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 type checking: The compiler generates code so that, when an operation is performed, the code checks to make sure that the operands have the correct types.

Examples: LISP, Smalltalk.

Compile-time type checking: The compiler checks the program text for potential type errors.

Example: SML.

NB: Most programming languages use some combination of compile-time and run-time type checking.

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;` $\surd$(upcast)
`a = (A)b;` $\surd$(explicit upcast)
`b = a;` $\times$(implicit downcast—illegal Java)
`b = (B)a;` $\surd$(but needs run-time type-check)

Mixed static and dynamic type checking!

Static vs. dynamic type checking

Main trade-offs between compile-time and run-time checking:

Form of type checking	**Advantages**	**Disadvantages**
Run-time	Prevents type errors	Slows program execution
Compile-time	Prevents type errors Eliminates run-time tests Finds type errors before execution and run-time tests	May restrict programming because tests are *conservative*

Type checking in ML

Idea

Given a context Γ, an expression e, and a type τ, decide whether or not the expression e is of type τ in context Γ.

Examples:

- $$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \ \texttt{orelse} \ e_2 : \texttt{bool}}$$

$$\mathsf{TC}(\Gamma, e_1 \ \texttt{orelse} \ e_2, \tau) = \begin{cases} \mathsf{TC}(\Gamma, e_1, \texttt{bool}) \wedge \mathsf{TC}(\Gamma, e_2, \texttt{bool}) & \text{, if } \tau = \texttt{bool} \\ \text{false} & \text{, otherwise} \end{cases}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\mathsf{TC}(\Gamma, (e_1, e_2), \tau) = \begin{cases} \mathsf{TC}(\Gamma, e_1, \tau_1) \wedge \mathsf{TC}(\Gamma, e_2, \tau_2) & \text{, if } \tau = \tau_1 * \tau_2 \\ \text{false} & \text{, otherwise} \end{cases}$$

Type equality

The question of *type equality* arises during type checking.

$\boxed{?}$ What does it mean for two types to be equal!?

Structural equality. Two type expressions are *structurally equal* if and only if they are equivalent under the following three rules.

SE1. A type name is structurally equal to itself.

SE2. Two types are structurally equal if they are formed by applying the same type constructor to structurally equal types.

SE3. After a type declaration, say `type n = T`, the type name `n` is structurally equal to `T`.

Name equality:

Pure name equality. A type name is equal to itself, but no constructed type is equal to any other constructed type.

Transitive name equality. A type name is equal to itself and can be declared equal to other type names.

Type-expression equality. A type name is equal only to itself. Two type expressions are equal if they are formed by applying the same constructor to equal expressions. In other words, the expressions have to be identical.

Examples:

- **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 equality in C/C++.** C uses structural equivalence for all types except for records (`struct`s). `struct` types are named in C and C++ and the name is treated as a type, equal only to itself. This constraint saves C from having to deal with recursive types.

Type declarations

There are two basic forms of type declarations:

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.

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 ML

Idea

Typing rule:

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

Inference rule:

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

Typing rule:

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

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:

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

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:

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

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

Solution: $\alpha_0 = (\alpha_3 \to \alpha_3) \to \alpha_3 \to \alpha_3$

Polymorphism

Polymorphism, which literally means "having multiple forms", refers to constructs that can take on different types as needed.

Forms of polymorphism in contemporary programming languages:

Parametric polymorphism. A function may be applied to any arguments whose types match a type expression involving type variables.

Parametric polymorphism may be:

Implicit.

Explicit.

Ad hoc polymorphism or overloading. Two or more implementations with different types are referred to by the same name.

Subtype polymorphism. The subtype relation between types allows an expression to have many possible types.

let-polymorphism

- The standard sugaring

$$\texttt{let val } x = v \texttt{ in } e \texttt{ end} \quad \mapsto \quad (\texttt{fn } x \texttt{ => } e)(v)$$

  does not respect ML type checking.

  For instance

$$\texttt{let val f = fn x => x in f(f) end}$$

  type checks, whilst

$$(\texttt{fn f => f(f)})(\texttt{fn x => x})$$

  does not.

- Type inference for let-expressions is involved, requiring *type schemes*.

Polymorphic exceptions

Example: Depth-first search for finitely-branching trees.

```
datatype
  'a FBtree = node of 'a * 'a FBtree list ;
fun dfs P (t: 'a FBtree)
  = let
      exception Ok of 'a;
      fun auxdfs( node(n,F) )
        = if P n then raise Ok n
          else foldl (fn(t,_) => auxdfs t) NONE F ;
    in
      auxdfs t handle Ok n => SOME n
    end ;
val dfs = fn : ('a -> bool) -> 'a FBtree -> 'a option
```

When a *polymorphic exception* is declared, SML ensures that it is used with only one type. The type of a top level exception must be monomorphic and the type variables of a local exception are frozen.

Consider the following nonsense:

```
exception Poly of 'a ;   (*** ILLEGAL!!! ***)
(raise Poly true) handle Poly x => x+1 ;
```

∼ Topic VIII ∼

Data abstraction and modularity
SML Modules[a]

References:

- **Chapter 7** of *ML for the working programmer* (2ND EDITION) by L. C. Paulson. CUP, 1996.

---

[a]Largely based on an *Introduction to SML Modules* by Claudio Russo `<http://research.microsoft.com/~crusso>`.

The Core and Modules languages

SML consists of two sub-languages:

- The *Core* language is for *programming in the small*, by supporting the definition of types and expressions denoting values of those types.
- The *Modules* language is for *programming in the large*, by grouping related Core definitions of types and expressions 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 SML Modules language lets one split large programs into separate units with descriptive interfaces.

SML 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

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.

```
struct
  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))
end
```

The dot notation

One can name a structure by binding it to an identifier.

```
structure IntNat =
 struct
     type nat = int
     ...
     fun iter b f i = ...
 end
```

Components of a structure are accessed with the *dot notation*.

```
  fun 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.

Nested structures

Structures can be nested inside other structures, in a hierarchy.

```
structure IntNatAdd =
  struct
    structure Nat = IntNat
    fun add n m = Nat.iter m Nat.succ n
  end
  ...
fun mult n m =
IntNatAdd.Nat.iter IntNatAdd.Nat.zero (IntNatAdd.add m) n
```

Concrete signatures

Signature expressions specify the types of structures by listing the specifications of their components.

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

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

This signature fully describes the *type* of `IntNat`.

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

Opaque signatures

On the other hand, the following signature

```
sig type nat
    val zero : nat
    val succ : nat -> nat
    val 'a iter : 'a -> ('a->'a) -> nat -> '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*.

Example: Polymorphic functional stacks.

```
signature STACK =
sig
  exception E
  type 'a reptype    (* <-- INTERNAL REPRESENTATION *)
  val new: 'a reptype
  val push: 'a -> 'a reptype -> 'a reptype
  val pop: 'a reptype -> 'a reptype
  val top: 'a reptype -> 'a
end ;
```

```
structure MyStack: STACK =
struct
  exception E ;
  type 'a reptype = 'a list ;
  val new = [] ;
  fun push x s = x::s ;
  fun split( h::t ) = ( h , t )
    | split _ = raise E ;
  fun pop s = #2( split s ) ;
  fun top s = #1( split s ) ;
end ;
```

```
val MyEmptyStack = MyStack.new ;
val MyStack0 = MyStack.push 0 MyEmptyStack ;
val MyStack01 = MyStack.push 1 MyStack0 ;
val MyStack0' = MyStack.pop MyStack01 ;
MyStack.top MyStack0' ;
```

```
val MyEmptyStack = [] : 'a MyStack.reptype
val MyStack0 = [0] : int MyStack.reptype
val MyStack01 = [1,0] : int MyStack.reptype
val MyStack0' = [0] : int MyStack.reptype
val it = 0 : int
```

Named and nested signatures

Signatures may be *named* and referenced, to avoid repetition:

```
signature NAT =
  sig type nat
      val zero : nat
      val succ : nat -> nat
      val 'a iter : 'a -> ('a->'a) -> nat -> 'a
end
```

Nested signatures specify named sub-structures:

```
signature Add =
  sig structure Nat: NAT (* references NAT *)
      val add: Nat.nat -> Nat.nat -> Nat.nat
  end
```

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 specified value must have a more general type scheme;
  - ⋆ every specified structure must be enriched by a substructure.

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.

Subtyping

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.

Using signatures to restrict access

The following structure uses a *signature constraint* to provide a restricted view of `IntNat`:

```
structure ResIntNat =
  IntNat : sig type nat
               val succ : nat->nat
               val iter : nat->(nat->nat)->nat->nat
           end
```

NB: The constraint `str:sig` prunes the structure `str` according to the signature `sig`:

- `ResIntNat.zero` is *undefined*;
- `ResIntNat.iter` is *less* polymorphic that `IntNat.iter`.

Transparency of _:_

Although the `_:_` operator can hide names, it does not conceal the definitions of opaque types.

Thus, the fact that `ResIntNat.nat` = `IntNat.nat` = `int` remains *transparent*.

For instance the application `ResIntNat.succ(~3)` is still well-typed, because `~3` has type `int` ...but `~3` is negative, so not a valid representation of a natural number!

SML Modules

Information hiding

In SML, we can limit outside access to the components of a structure by *constraining* its signature in *transparent* or *opaque* manners.

Further, we can *hide* the representation of a type by means of an `abstype` declaration.

The combination of these methods yields abstract structures.

Using signatures to hide the identity of types

With different syntax, signature matching can also be used to enforce *data abstraction*:

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

The constraint `str :> sig` prunes `str` but also generates a new, *abstract* type for each opaque type in `sig`.

- 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.

Opaque signature constraints

```
structure MyOpaqueStack :> STACK = MyStack ;

val MyEmptyOpaqueStack = MyOpaqueStack.new ;
val MyOpaqueStack0 = MyOpaqueStack.push 0 MyEmptyOpaqueStack ;
val MyOpaqueStack01 = MyOpaqueStack.push 1 MyOpaqueStack0 ;
val MyOpaqueStack0' = MyOpaqueStack.pop MyOpaqueStack01 ;
MyOpaqueStack.top MyOpaqueStack0' ;
```

```
val MyEmptyOpaqueStack = - : 'a MyOpaqueStack.reptype
val MyOpaqueStack0 = - : int MyOpaqueStack.reptype
val MyOpaqueStack01 = - : int MyOpaqueStack.reptype
val MyOpaqueStack0' = - : int MyOpaqueStack.reptype
val it = 0 : int
```

Datatype and exception specifications

Signatures can also specify datatypes and exceptions:

```
structure PredNat =
  struct datatype nat = zero | succ of nat
         fun iter b f i = ...
         exception Pred
         fun pred zero = raise Pred
           | pred (succ n) = n
  end
  :> sig datatype nat = zero | succ of nat
         val iter:  'a->('a->'a)->(nat->'a)
         exception Pred
         val pred:  nat -> nat (* raises Pred *) end
```

This means that clients can still pattern match on datatype constructors, and handle exceptions.

SML Modules

Functors

- An SML *functor* is a structure that takes other structures as parameters.
- Functors let us write program units that can be combined in different ways. Functors can also express generic algorithms.

Functors

Modules also supports *parameterised structures*, called *functors*.

Example: The functor `AddFun` below takes any implementation, `N`, of naturals and re-exports it with an addition operation.

```
functor AddFun(N:NAT) =
        struct
          structure Nat = N
          fun add n m = Nat.iter n (Nat.succ) m
        end
```

- A functor is a *function* mapping a formal argument structure to a concrete result structure.

- The body of a functor may assume no more information about its formal argument than is specified in its signature.

  In particular, opaque types are treated as distinct type parameters.

  Each actual argument can supply its own, independent implementation of opaque types.

Functor application

A functor may be used to create a structure by *applying* it to an actual argument:

```
structure IntNatAdd = AddFun(IntNat)
structure AbsNatAdd = AddFun(AbsNat)
```

The actual argument must match the signature of the formal parameter—so it can provide more components, of more general types.

Above, `AddFun` is applied twice, but to arguments that differ in their implementation of type `nat` (`AbsNat.nat` $\neq$ `IntNat.nat`).

Example: Generic imperative stacks.

```
signature STACK =
 sig
  type itemtype
  val push: itemtype -> unit
  val pop: unit -> unit
  val top: unit -> itemtype
end ;
```

```
exception E ;
functor Stack( T: sig type atype end ) : STACK =
struct
  type itemtype = T.atype
  val stack = ref( []: itemtype list )
  fun push x
    = ( stack := x :: !stack )
  fun pop()
    = case !stack of [] => raise E
      | _::s => ( stack := s )
  fun top()
    = case !stack of [] => raise E
      | t::_ => t
end ;
```

```
structure intStack
  = Stack(struct type atype = int end) ;
```

```
structure intStack : STACK
```

```
intStack.push(0) ;
intStack.top() ;
intStack.pop() ;
intStack.push(4) ;
```

```
val it = () : unit
val it = 0 : intStack.itemtype
val it = () : unit
val it = () : unit
```

Why functors ?

Functors support:

Code reuse.

`AddFun` may be applied many times to different structures, reusing its body.

Code abstraction.

`AddFun` can be compiled before any argument is implemented.

Type abstraction.

`AddFun` can be applied to different types `N.nat`.

Are signatures types?

The syntax of Modules suggests that signatures are just the types of structures . . . but signatures can contain opaque types.

In general, signatures describe *families of structures*, indexed by the realisation of any opaque types.

The interpretation of a signature really depends on how it is used!

In functor parameters, opaque types introduce *polymorphism*; in signature constraints, opaque types introduce *abstract types*.

Since type components may be type constructors, not just types, this is really *higher-order* polymorphism and abstraction.

Structures as records

Structures are like Core records, but can contain definitions of types as well as values.

What does it mean to project a type component from a structure, e.g. `IntNatAdd.Nat.nat`?

Does one needs to evaluate the application `AddFun(IntNat)` at *compile-time* to simplify `IntNatAdd.Nat.nat` to `int`?

No! Its sufficient to know the *compile-time* types of `AddFun` and `IntNat`, ensuring a *phase distinction* between compile-time and run-time.

Type propagation through functors

Each functor application *propagates* the actual realisation of its argument's opaque type components.

Thus, for

`structure IntNatAdd = AddFun(IntNat) structure AbsNatAdd = AddFun(AbsNat)`

the type `IntNatAdd.Nat.nat` is just another name for `int`, and `AbsNatAdd.Nat.nat` is just another name for `AbsNat.nat`.

Examples:

`IntNatAdd.Nat.succ(0)` $\surd$

`IntNatAdd.Nat.succ(IntNat.Nat.zero)` $\surd$

`AbsNatAdd.Nat.succ(AbsNat.Nat.zero)` $\surd$

`AbsNatAdd.Nat.succ(0)` $\times$

`AbsNatAdd.Nat.succ(IntNat.Nat.zero)` $\times$

Generativity

The following functor almost defines an identity function, but *re-abstracts* its argument:

```
functor GenFun(N:NAT) = N :> NAT
```

Now, each application of `GenFun` generates a new abstract type: For instance, for

```
structure X = GenFun(IntNat) structure Y = GenFun(IntNat)
```

the types `X.nat` and `Y.nat` are *incompatible*, even though `GenFun` was applied to the *same* argument.

Functor application is *generative*: abstract types from the body of a functor are replaced by fresh types at each application. This is consistent with inlining the body of a functor at applications.

Why should functors be generative?

It is really a design choice. Often, the invariants of the body of a functor depend on both the types *and values* imported from the argument.

```
functor OrdSet(O:sig type elem
                    val compare:  (elem * elem) -> bool
                end) = struct
  type set = O.elem list (* ordered list of elements *)
  val empty = []
  fun insert e [] = [e]
    | insert e1 (e2::s) = if O.compare(e1,e2)
        then if O.compare(e2,e1) then e2::s else e1::e2::s
        else e2::insert e1 s
end :> sig type set
          val empty:  set
          val insert:  O.elem -> set -> set
       end
```

For

structure S

```
= OrdSet(struct type elem=int  fun compare(i,j)= i <= j  end)
```

structure R

```
= OrdSet(struct type elem=int  fun compare(i,j)= i >= j  end)
```

we want `S.set` $\neq$ `R.set` because their representation invariants depend on the `compare` function: the set $\{1, 2, 3\}$ is `[1,2,3]` in `S.set`, but `[3,2,1]` in `R.set`.

Why functors ?

- Functors let one decompose a large programming task into separate subtasks.
- The propagation of types through application lets one extend existing abstract data types with type-compatible operations.
- Generativity ensures that applications of the same functor to data types with the same representation, but different invariants, return distinct abstract types.

Sharing specifications

Functors are often used to combine different argument structures.

Sometimes, these structure arguments need to communicate values of a *shared* type.

For instance, we might want to implement a sum-of-squares function $(n, m \mapsto n^2 + m^2)$ using separate structures for naturals with addition and multiplication . . .

Sharing violations

```
functor SQ(
    structure AddNat:
        sig structure Nat:  sig type nat end
            val add:Nat.nat -> Nat.nat -> Nat.nat
        end
    structure MultNat:
        sig structure Nat:  sig type nat end
            val mult:Nat.nat -> Nat.nat -> Nat.nat
        end ) =
struct
    fun sumsquare n m
      = AddNat.add (MultNat.mult n n) (MultNat.mult m m) ×
end
```

The above piece of code is *ill-typed*: the types `AddNat.Nat.nat` and `MultNat.Nat.nat` are opaque, and thus different. The `add` function cannot consume the results of `mult`.

Sharing specifications

The fix is to declare the type sharing directly at the specification of `MultNat.Nat.nat`, using a concrete, not opaque, specification:

```
functor SQ(
    structure AddNat:
        sig structure Nat:  sig type nat end
            val add:  Nat.nat -> Nat.nat -> Nat.nat
        end
    structure MultNat:
        sig structure Nat:  sig type nat = AddNat.Nat.nat end
            val mult:  Nat.nat -> Nat.nat -> Nat.nat
        end ) =
struct
  fun sumsquare n m
    = AddNat.add (MultNat.mult n n) (MultNat.mult m m) √
end
```

Sharing constraints

Alternatively, one can use a post-hoc *sharing specification* to identify opaque types.

```
functor SQ(
    structure AddNat:
        sig structure Nat:  sig type nat end
            val add:Nat.nat -> Nat.nat -> Nat.nat
        end
    structure MultNat:
        sig structure Nat:  sig type nat end
            val mult:Nat.nat -> Nat.nat -> Nat.nat
        end
    sharing type MultNat.Nat.nat = AddNat.Nat.nat
) =
struct
  fun sumsquare n m
    = AddNat.add (MultNat.mult n n) (MultNat.mult m m)
end
```

√

Limitations of modules

Modules is great for expressing programs with a complicated static architecture, but it's not perfect:

- Functors are *first-order*: unlike Core functions, a functor cannot be applied to, nor return, another functor.
- Structure and functors are *second-class* values, with very limited forms of computation (dot notation and functor application): modules cannot be constructed by algorithms or stored in data structures.
- Module definitions are *too sequential*: splitting mutually recursive types and values into separate modules is awkward.

~ Topic IX ~

A modern language design: Scala

< www.scala-lang.org >

References:

- *Scala By Example* by M. Odersky. Programming Methods Laboratory, EPFL, 2008.
- *An overview of the Scala programming language* by M. Odersky *et al*. Technical Report LAMP-REPORT-2006-001, Second Edition, 2006.

Scala (I)

- Scala has been developed from 2001 in the Programming Methods Laboratory at EPFL by a group lead by Martin Odersky. It was first released publicly in 2004, with a second version released in 2006.
- Scala is aimed at the construction of components and component systems.

  One of the major design goals of Scala was that it should be flexible enough to act as a convenient host language for domain specific languages implemented by library modules.

- Scala has been designed to work well with Java and C#.

  Every Java class is seen in Scala as two entities, a class containing all dynamic members and a singleton object, containing all static members.

  Scala classes and objects can also inherit from Java classes and implement Java interfaces. This makes it possible to use Scala code in a Java framework.

- Scala's influences: Beta, C#, FamilyJ, gbeta, Haskell, Java, Jiazzi, $ML_{\leq}$, Moby, MultiJava, Nice, OCaml, Pizza, Sather, Smalltalk, SML, XQuery, *etc.*

A procedural language !

```
def qsort( xs: Array[Int] ) {
  def swap(i: Int, j:Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort(l: Int, r: Int) {
    val pivot = xs( (l+r)/2 ); var i = l; var j = r
    while (i <= j) {
      while ( lt( xs(i), pivot ) ) i += 1
      while ( lt( xs(j), pivot ) ) j -= 1
      if ( i<=j ) { swap(i,j); i += 1; j -= 1 }
    }
    if (l<j) sort(l,j)
    if (j<r) sort(i,r)
  }
  sort(0,xs.length-1)
}
```

NB:

- Definitions start with a reserved word.
- Type declarations use the colon notation.
- Array selections are written in functional notation.

  (In fact, arrays in Scala inherit from functions.)
- Block structure.

A declarative language !

```
def qsort[T]( xs: Array[T] )( lt: (T,T)=>Boolean ): Array[T]
= if ( xs.length <= 1 ) xs
  else {
    val pivot = xs( xs.length/2 )
    Array.concat( qsort( xs filter (x => lt(x,pivot)) ) lt ,
                         xs filter (x => x == pivot )       ,
                  qsort( xs filter (x => lt(pivot,x)) ) lt )
  }
```

NB:

- Polymorphism.
- Type declarations can often be omitted because the compiler can infer it from the context.
- Higher-order functions.
- The binary operation $e \star e'$ is always interpreted a the method call $e. \star (e')$.
- The equality operation == between values is designed to be transparent with respect to the type representation.

Scala (II)

Scala fuses (1) *object-oriented* programming and (2) *functional* programming in a statically typed programming language.

1. Scala uses a uniform and pure *object-oriented* model similar to that of Smalltalk: Every value is an object and every operation is a message send (that is, the invocation of a method).

   In fact, even primitive types are not treated specially; they are defined as type aliases of Scala classes.

2. Scala is also a *functional* language in the sense that functions are first-class values.

Mutable state

- Real-world objects with state are represented in Scala by objects that have variables as members.
- In Scala, all mutable state is ultimately built from variables.
- Every defined variable has to be initialised at the point of its definition.
- Variables may be `private`.

Blocks

Scala is an *expression-oriented* language, every function returns some result.

Blocks in Scala are themselves expressions. Every block ends in a result expression which defines its value.

Scala uses the usual block-structured scoping rules.

Functions

A function in Scala is a first-class value.

The anonymous function

```
( x1: T1, ... , xn: Tn ) => E
```

is equivalent to the block

```
{ def f ( x1: T1 , ... , xn: Tn ) = E ; f }
```

where `f` is a fresh name which is used nowhere else in the program.

Parameter passing

Scala uses call-by-value by default, but it switches to call-by-name evaluation if the parameter type is preceded by `=>`.

Imperative control structures

A functional implementation of while loops:

```
def whileLoop( cond: => Boolean )( comm: => Unit )
  { if (cond)  comm ; whileLoop( cond )( comm )  }
```

Classes and objects

- `class`es provide fields and methods. These are accessed using the dot notation. However, there may be `private` fields and methods that are inaccessible outside the class.

  Scala, being an object-oriented language, uses dynamic dispatch for method invocation. Dynamic method dispatch is analogous to higher-order function calls. In both cases, the identity of the code to be executed is known only at run-time. This similarity is not superficial. Indeed, Scala represents every function value as an object.

- Every class in Scala has a superclass which it `extends`.

  A class inherits all members from its superclass. It may also `override` (*i.e.* redefine) some inherited members.

  If class A extends class B, then objects of type A may be used wherever objects of type B are expected. We say in this case that type A *conforms* to type B.

- Scala maintains the invariant that interpreting a value of a subclass as an instance of its superclass does not change the representation of the value.

  Amongst other things, it guarantees that for each pair of types `S <: T` and each instance `s` of `S` the following semantic equality holds:

  `s.asInstanceOf[T].asInstanceOf[S]` = `s`

- Methods in Scala do not necessarily take a parameter list. These *parameterless* methods are accessed just as value fields.

  The uniform access of fields and parameterless methods gives increased flexibility for the implementer of a class. Often, a field in one version of a class becomes a computed value in the next version. Uniform access ensures that clients do not have to be rewritten because of that change.

- `abstract class`es may have *deferred* members which are declared but which do not have an implementation. Therefore, no objects of an abstract class may be created using `new`.

```
abstract class IntSet {
  def incl( x:Int ): IntSet
  def contains( x:Int ): Boolean
}
```

Abstract classes may be used to provide interfaces.

- Scala has `object` definitions. An object definition defines a class with a single instance. It is not possible to create other objects with the same structure using `new`.

```
object EmptySet extends IntSet {
  def incl( x: Int ): IntSet
    = new NonEmptySet(x,EmptySet,EmptySet)
  def contains( x: Int ): Boolean = false
}
```

An object is created the first time one of its members is accessed. (This strategy is called *lazy evaluation*.)

- A `trait` is a special form of an abstract class that does not have any value (as opposed to type) parameters for its constructor and is meant to be combined with other classes.

```
trait IntSet {
  def incl( x:Int ): IntSet
  def contains( x:Int ): Boolean
}
```

Traits may be used to collect signatures of some functionality provided by different classes.

Case study (I)

```
abstract class Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}
class Number( n: Int ) extends Expr {
  def isNumber: Boolean = true
  def isSum: Boolean = false
  def numValue: Int = n
  def leftOp: Expr = error("Number.leftOp")
  def rightOp: Expr = error("Number.rightOp")
}
```

```
class Sum( e1: Expr; e2: Expr ) extends Expr {
  def isNumber: Boolean = false
  def isSum: Boolean = true
  def numValue: Int = error("Sum.numValue")
  def leftOp: Expr = e1
  def rightOp: Expr = e2
}
def eval( e: Expr ): Int = {
  if (e.isNumber) e.NumValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
  else error("bad expression")
}
```

? What is good and what is bad about this implementation?

Case study (II)

```
abstract class Expr {
  def eval: Int
}
class Number( n: Int ) extends Expr {
  def eval: Int = n
}
class Sum( e1: Expr; e2: Expr ) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```

This implementation is easily extensible with *new types of data*:

```
class Prod( e1: Expr; e2: Expr ) extends Expr {
  def eval: Int = e1.eval * e2.eval
}
```

But, is this still the case for extensions involving *new operations* on existing data?

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’:

http://en.wikipedia.org/wiki/Expression_problem

Case study (III)

Case classes

```
abstract class Expr
case class Number( n: Int ) extends Expr
case class Sum( e1: Expr; e2: Expr ) extends Expr
case class Prod( e1: Expr; e2: Expr ) extends Expr
```

- Case classes implicitly come with a constructor function, with the same name as the class.

  Hence one can construct expression trees as:

  ```
  Sum( Sum( Number(1) , Number(2) ) , Number(3) )
  ```

- Case classes and case objects implicitly come with implementations of methods `toString`, `equals`, and `hashCode`.

- Case classes implicitly come with nullary accessor methods which retrieve the constructor arguments.

- Case classes allow the constructions of *patterns* which refer to the case class constructor (see next slide).

(Case classes are essentially ML data types in an object-oriented language.)

Case study (III)

Pattern matching

The `match` method takes as argument a number of cases:

```
def eval( e: Expr ): Int
  = e match
    {  case Number(x) => x
       case Sum(l,r) => eval(l) + eval(r)
       case Prod(l,r) => eval(l) * eval(r)
    }
```

If none of the patterns matches, the pattern matching expression is aborted with a `MatchError` exception.

Generic types and methods

- Classes in Scala can have type parameters.

```
abstract class Set[A] {
  def incl( x: A ): Set[A]
  def contains( x: A ): Boolean
}
```

- Scala has a fairly powerful type inferencer which allows one to omit type parameters to polymorphic functions and constructors.

Generic types

Variance annotations

The combination of type parameters and subtyping poses some interesting questions.

$\boxed{\textbf{?}}$ If `T` is a subtype of a type `S`, should `Array[T]` be a subtype of the type `Array[S]`?

$\boxed{!}$ No, if one wants to avoid run-time checks!

Example:

- For `ColPoint <: Point` and `a: Array[ColPoint]`,

  `(a.apply(0)).color: Col`

  type checks.

- Suppose that `Array` is covariant:

  `ColPoint <: Point` $\Longrightarrow$ `Array[ColPoint] <: Array[Point]`

  so that `a: Array[Point]`.

- Then, for `p: Point`, we have that `a.update(0,p)` type checks; and, as above, so does

  `(a.apply(0)).color: Col`

  But this is semantically equal to `p.color`; a run-time error.

In Scala, generic types like the following one:

```
class Array[A] {
  def apply( index: Int ): A
    ...
  def update( index: Int, elem: A )
    ...
}
```

have by default *non-variant* subtyping.

However, one can enforce *co-variant* (or *covariant*) subtyping by prefixing a formal type parameter with a `+`. There is also a prefix `-` which indicates *contra-variant* subtyping.

Scala uses a conservative approximation to verify soundness of variance annotations: a covariant type parameter of a class may only appear in covariant position inside the class. Hence, the following class definition is rejected:

```
class Array[+A] {
  def apply( index: Int ): A
    ...
  def update( index:Int , elem: A )
    ...
}
```

Functions are objects

Recall that Scala is an object-oriented language in that every value is an object. It follows that *functions are objects* in Scala.

Indeed, the function type

$$\texttt{(}\ A_1\texttt{,}\ \ldots\texttt{,}\ A_k\ \texttt{)}\ \texttt{=>}\ B$$

is equivalent to the following parameterised class type:

$$\texttt{abstract class Function}\mathbf{k}[-A_1,\ldots,-A_k,+B]$$
$$\{\ \texttt{def apply(}\ x_1{:}A_1,\ldots,x_n{:}A_k\ \texttt{):}\ B\ \}$$

(Since function types are classes in Scala, they can be further refined in subclasses. An example are arrays, which are treated as special functions over the type of integers.)

The function `x => x+1` would be expanded to an instance of `Function1` as follows:

```
new Function1[Int,Int] {
  def apply( x:Int ): Int = x+1
}
```

Conversely, when a value of a function type is applied to some arguments, the `apply` method of the type is implicitly inserted; *e.g.* for `f` and object of type `Function1[A,B]`, the application `f(x)` is expanded to `f.apply(x)`.

NB: Function subtyping is contravariant in its arguments whereas it is covariant in its result. **?** Why?

Generic types

Type parameter bounds

```
trait Ord[A] {
  def lt( that: A ): Boolean
}
case class Num( value: Int ) extends Ord[Num] {
  def lt( that: Num ) = this.value < that.value
}
trait Heap[ A <: Ord[A] ] {
  def insert( x: A ): Heap[A]
  def min: A
  def remove: Heap[A]
}
```

Generic types

Lower bounds

- A non-example:

```
abstract class Stack[+A]  // covariant declaration
{ def push( x:  A )  // A in contravariant position
                     // hence rejected
      : Stack[A]
     = new NonEmptyStack(x,this)
   def top: A
   def pop: Stack[A] }
```

that makes sense:

```
ColPoint <: Point

s : Stack[ColPoint] <: Stack[Point]

p : Point

s.push(p) : Stack[Point]   // OK

(s.push(p)).top : Point    // OK
```

- Covariant generic functional stacks.

  The solution:

```
abstract class Stack[+A] {
  def push[B >: A]( x: B ): Stack[B]
    = new NonEmptyStack(x,this)
  def top: A
  def pop: Stack[A]
}
class NonEmptyStack[+A]( elem: A, rest: Stack[A] )
extends Stack[A] {
  def top = elem
  def pop = rest
}
```

Implicit parameters and conversions

- **Implicit parameters**

  In Scala, there is an `implicit` keyword that can be used at the beginning of a parameter list.

```
def qsort[T]( xs: Array[T] )( implicit o: Ord[T] ): Array[T]
= if ( xs.length <= 1 ) xs
  else {
    val pivot = xs( xs.length/2 )
    Array.concat( qsort( xs filter (x => o.lt(x,pivot)) ) ,
                         xs filter (x => x == pivot )       ,
                  qsort( xs filter (x => o.lt(pivot,x)) ) )
  }
```

The principal idea behind implicit parameters is that arguments for them can be left out from a method call. If the arguments corresponding to implicit parameters are missing, they are inferred by the Scala compiler.

- **Implicit conversions**

As last resort in case of type mismatch the Scala compiler will try to apply an implicit conversion.

```
implicit def int2ord( x: Int ): Ord[Int]
  = new Ord[Int] { def lt( y: Int ) = x < y }
```

Implicit conversions can also be applied in member selections.

Mixin-class composition

Every class or object in Scala can inherit from several traits in addition to a normal class.

```
trait AbsIterator[T] {
  def hasNext: Boolean
  def next: T
}
trait RichIterator[T] extends AbsIterator[T] {
  def foreach( f: T => Unit ): Unit =
    while (hasNext) f(next)
}
```

```
class StringIterator( s: String )
  extends AbsIterator[Char] {
    private var i = 0
    def hasNext = i < s.length
    def next = { val x = s charAt i; i = i+1; x }
  }
```

Traits can be used in all contexts where other abstract classes appear; however only traits can be used as *mixins*.

```
object Test {
  def main( args: Array[String] ): Unit = {
    class Iter extends StringIterator(args(0))
                 with    RichIterator[Char]
    val iter = new Iter
    iter.foreach(System.out.println)
  }
}
```

The class `Iter` is constructed from a *mixin composition* of the parents `StringIterator` (called the *superclass*) and `RichIterator` (called a *mixin*) so as to combine their functionality.

The class `Iter` inherits members from both `StringIterator` and `RichIterator`.

NB: Mixin-class composition is a form of multiple inheritance, but avoids the ‘diamond’ problem of C++ and similar languages (where a class containing a field appears at multiple places in the inheritance hierarchy).

```
class A { public: int f;}
class B: public A { ... }
class C: public A { ... }
class D: public B,C { ... f ... }
// is this B::f or C::f?
// do B and C both have an A, or share a single one?
```

Scala language innovations

- Flexible syntax and type system.
- Pattern matching over class hierarchies unifies functional and object-oriented data access.
- Abstract types and mixin composition unify concepts from object and module systems.

∼ Topic X ∼

Miscellaneous (entertaining) concepts

Additional notes for lecture 8

Aspirational Overview

- Value types
- Generalised Algebraic Data Types (GADTs)
- Haskell classes
- Monadic IO
- Continuation-passing style (CPS) and call/cc
  `Wikipedia:Call-with-current-continuation`
- Access qualifiers vs. encapsulation.
- Futures
- Coq and Agda

Reified continuations

Make calling continuation appear to be a value in the language.

Reminder on continuation-passing style (CPS), perhaps mentioned in Compiler Construction. Can see a function of type $t_1 \to t_2$ as a function of type

$$(t_2 \to \mathit{unit}) \to (t_1 \to \mathit{unit})$$

Or uncurrying

$$(t_2 \to \mathit{unit}) \times t_1 \to \mathit{unit}$$

(One parameter of type t_1 and the other saying what to do with the result t_2 – like *argument* and *return address*!)

Instead of

```
fun f(x) = ... return e ...
print f(42)
```

we write

```
fun f'(k, x) = ... return k e' ...
f(print, 42)
```

In CPS all functions return $unit$ and all calls are now tail-calls (so the above isn't just a matter of adjusting a return statement).

Sussman and Steele papers from the 1970's ("Lambda the ultimate XXX").

Reified continuations (2)

A function with *two* continuation parameters rather than one can act as normal return vs. exception return. (Or Prolog success return vs failure return.)

But we don’t want to write all our code in CPS style. So: *call/cc* “call with current continuation”. Lots of neat programming tricks in a near-functional language.

Reified? The continuation used at the meta-level (semantics) to explain how a language operates is exposed as an object-level (run-time value).

Reified continuations (3)

Core idea (originally in Scheme, a form of Lisp):

```
fun f(k) = let x = k(2) in 3;
```

In ML this function ‘always returns 3’. E.g.

```
> f(fn x=>x);
```

But `callcc(f)` returns 2!

The return address/continuation used in the call to `f` is reified into a side-effecting function value `k` which represents the “rest of the computation after the call to `f`”.

Some similarity with `f(fn x => raise Foo);`

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 (OCaml GADT):

```
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) ✗
```

Can't do this in SML.

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

```
fun eval(Val(x)) = x
 |  eval(Eq(x,y)) = eval(x) = eval(y)
 |  eval(Add(x,y)) = eval(x) + eval(y);
```

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

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