Concepts in Programming Languages

Marcelo Fiore

Computer Laboratory
University of Cambridge

Lent 2007

⟨http://www.cl.cam.ac.uk/Teaching/2006/ConceptsPL/⟩

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.
- R. Sethi. *Programming languages: Concepts & constructs* (2ND EDITION). Addison-Wesley, 1996.

∼ Lecture 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.

Practicalities

- Lectures.

[1] Introduction and motivation.

[2] The first *procedural* language: FORTRAN (1954–58).

[3] The first *declarative* language: LISP (1958–62).

[4] *Block-structured* procedural languages: Algol (1958–68), BCPL (1967), Pascal (1970), C (1971–78).

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

[6,7] *Types*, *data abstraction*, and *modularity*: C++ (1983–98), SML (1984–97).

[8] The *state of the art*: Java (1996), C# (2000). (⋆ Andrew Kennedy ⋆)

- 2 exam questions.
- Course web page:

  ⟨`www.cl.cam.ac.uk/Teaching/2006/ConceptsPL/`⟩

  with *lecture slides* and *reading material*.

Goals

- Critical *thinking* about programming languages.

  ? 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.

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.

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, Postcript, word processing	
	Process	UNIX shell, TCL, Perl	Marvel, Esterel
	New paradigms	Smalltalk, ML, Haskell, Java Python, Ruby	Eifell, C#

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, ML module system: 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

Theoretical foundations

Examples:

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

Standardisation

- Proprietary standards.
- Consensus standards.
  - ANSI.
  - IEEE.
  - BSI.
  - ISO.

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 C

? What does the following mean?

```
#include <stdio.h>
main() {

int t = 1 ;
int t0 = 0 ;
t0 = (t=t+1) + ++t ;
printf("t0=%d t=%d\n",t0,t) ;

int u = 1 ;
int u0 = 0 ;
u0 = ++u + (u=u+1) ;
printf("u0=%d u=%d\n",u0,u) ;
```

```
int w = 1 ;
int w0 = 0 ;
w0 = (w=w+1) + (w=w+1) ;
printf("w0=%d w=%d\n",w0,w) ;


int x = 1 ;
int x0 = 0 ;
x0 = ++x + ++x ;
printf("x0=%d x=%d\n",x0,x) ;
```

```
int y = 1 ;
int y0 = 0 ;
int ppy() { return ++y; } ;
y0 = ppy() + ppy() ;
printf("y0=%d y=%d\n",y0,y) ;


int z = 1 ;
int z0 = 0 ;
z0 = ++z ;
z0 += ++z ;
printf("z0=%d z=%d\n",z0,z) ; }
```

Answer:

Linux	(gcc, cc)	t0=5	u0=6	w0=5	x0=6	y0=5	z0=5
Mips	(gcc)	t0=5	u0=5	w0=5	x0=5	y0=5	z0=5
	(cc)	t0=5	u0=6	w0=5	x0=6	y0=5	z0=5
DEC Alpha and Sun4	(gcc,cc)	t0=5	u0=5	w0=5	x0=5	y0=5	z0=5

Language standards PL/1

? What does the following

`9 + 8/3`

mean?

- `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 - OVERFLOW
  =     1.66666666666666 - 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.

Language groups

- Multi-purpose languages
  - C#, Java, C++, C
  - Haskell, ML, Scheme, LISP
- Scripting languages
  - Perl, TCL, UNIX shell
- Special-purpose languages
  - 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.

∼ Lecture 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.

FORTRAN = FORmula TRANslator (1957)

- The first high-level programming language to become widely used.
- Developed in the 1950s by an IBM team led by John Backus.
- At the time the utility of any high-level language was open to question!

  The main complain was the efficiency of compiled code. This heavily influenced the designed, orienting it towards providing execution efficiency.
- Standards:
  1966, 1977 (FORTRAN 77), 1990 (FORTRAN 90).

John Backus

As far as we were aware, we simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler which could produce efficient programs.[a]

[a]In R. L. Wexelblat, *History of Programming Languages*, Academic Press, 1981, page 30.

Overview
Execution model

- FORTRAN program = main program + subprograms
  - Each is *compiled separate* 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
Data types

- Numeric data: Integer, real, complex, double-precision real.
- Boolean data. called logical
- Arrays. of fixed declared length
- Character strings. of fixed declared length
- Files.

Overview
Control structures

- FORTRAN 66

  Relied heavily on statement labels and `GOTO` statements.
- FORTRAN 77

  Added some modern control structures (*e.g.*, conditionals).

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

- Columns and lines are relevant.
- Blanks are ignored (by early FORTRANs).
- Variable names are from 1 to 6 characters long, begin with a letter, and contain letters and digits.
- Programmer-defined constants.
- Arrays: when sizes are given, lower bounds are assumed to be 1; otherwise subscript ranges must be explicitly declared.
- Variable types may not be declared: implicit naming convention.

- Data formats.
- FORTRAN 77 has no `while` statement.
- Functions are compiled separately from the main program. Information from the main program is not used to pass information to the compiler. Failure may arise when the loader tries to merge subprograms with main program.
- Function parameters are uniformly transmitted by reference (or value-result).

  Recall that allocation is done statically.
- `DO` loops by increment.
- A value is returned in a FORTRAN function by assigning a value to the name of a function.

On syntax

A misspelling bug . . .

`do 10 i = 1,100` *vs.* `do 10 i = 1.100`

. . . that is reported to have caused a rocket to explode upon launch into space!

Types

- FORTRAN has no mechanism for creating user types.
- *Static type checking* is used in FORTRAN, but the checking is incomplete.

  Many language features, including arguments in subprogram calls and the use of `COMMON` blocks, cannot be statically checked (in part because subprograms are compiled independently).

  Constructs that cannot be statically checked are ordinarily left unchecked at run time in FORTRAN implementations.

Storage
Representation and Management

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

  The global environment may be partitioned into separate *common* environments that are shared amongst sets of subprograms, but only data objects may be shared in this way.

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

- `EQUIVALENCE`

  This declaration allows more than one simple or subscripted variable to refer to the same storage location.

  $\boxed{?}$ Is this a good idea?

  Consider the following:

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

- `COMMON`

  The global environment is set up in terms of *sets of variables and arrays*, which are termed *`COMMON` blocks*.

  A `COMMON` block is a named block of storage and may contain the values of any number of simple variables and arrays.

  `COMMON` blocks may be used to isolate global data to only a few subprograms needing that data.

  $\boxed{?}$ Is the `COMMON` block a good idea?

  Consider the following:

```
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 implementor 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.[a]

[a] In what regards to the problem of checking for aliasing, interested students may wish to investigate the work on the programming language *Euclid* and/or the work of John Reynolds and followers on *Syntactic Control of Interference*.

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

- The language specifies that if a formal parameter is assigned to, the actual parameter must be a variable, but because of independent compilation this rule cannot be checked by the compiler.

  Example:

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

  CALL SUB(-1.0,1.0)
  ```
- Parameter passing is uniformly by *reference*.

∼ Lecture 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.

- J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, **3**(4):184–195, 1960.[a]

[a] Available on-line from ⟨http://www-formal.stanford.edu/jmc/recursive.html⟩.

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.

- *Pure* LISP has no statements and no expressions with side effects. However, LISP also supports *impure* constructs.

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*.
- The basic data structures of LISP are *dotted pairs*, which are pairs written with a dot between the components. Putting atoms or pairs together, one can write symbolic expressions in a form traditionally called *S-expressions*. Most LISP programs use *lists* built out of S-expressions.
- 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 )                          val x = 1 ;
( defun g(z) (+ x z) )                  fun g(z) = x + z ;
( defun f(y)                            fun f(y)
    ( + ( g y )                           = g(y) +
        ( let                               let
            ( ( x y) )                        val x = y
            (                               in
              g x )                           g(x)
          ) ) )                             end ;
( f (+ x 1) )                           f(x+1) ;
```

! It is full of parentheses!

Historically, LISP was a *dynamically scoped* language . . .

```
( defvar x T )
( defun test(y) (eq x y) )
( cond
   ( x ( let ( (z 0) ) (test z) ) )
  )
```

vs.

```
( defvar x T )
( defun test(y) (eq x y) )
( cond
   ( x ( let ( (x 0) ) (test x) ) )
  )
```

. . . when *Scheme* was introduced in 1978, it was a *statically scoped* variant of LISP.

Static and dynamic scope

Static scope rules relate references with declarations of names in the program text; *dynamic scope* rules relate references with associations for names during program execution.

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.

Renaming of local variables

Lexical scope is deeply related to renaming of variables. It should not matter whether a program uses one name or another one for a local variable. Let us state this supposition as a principle:

> Consistent renaming of local names in the source text has no effect on the computation set up by a program.

This *renaming principle* motivates static scope because a language that obeys the renaming principle uses lexical scope. The reason is that the renaming principle can be applied to rename local variables until each name has only one declaration in the entire program. This one declaration is the one obtained under lexical scope.

The importance of static scope

Static scope rules play an important part in the design and implementation of most programming languages.

- Static scope rules allow many different sorts of connections to be established between references to names and their declarations during translation.

  For instance, relating a variable name to a declaration for the variable and relating a constant name to a declaration for the constant.

  Other connections include relating names to type declarations, relating formal parameters to formal

parameter specifications, relating subprogram calls to subprogram declarations, and relating statement labels referenced in `goto` statements to labels on particular statements.

In each of these cases, a different set of simplications may be made during translation that make execution of the program more efficient.

- Static scope rules are also important for the programmer in reading a program because they make it possible to relate a name referenced in a program to a declaration for the name without tracing the course of program execution.

Abstract machines

The terminology *abstract machine* is generally used to refer to an idealised computing device that can execute a specific programming language directly.

Typically an abstract machine may not be fully implementable. However, an abstract machine should be sufficiently realistic to provide useful information about the real execution of programs.

An important goal in discussing abstract machines is to identify the mental model of the computer that a programmer uses to write and debug programs.

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*.

   The purpose of the A-list is to store the values of variables that may occur either in the current expression to be evaluated or in the remaining expressions in the program.
4. A *heap*, which is a set of *cons cells* (or *dotted pairs*) that might be pointed to by pointers in the A-list.

Recursion
McCarthy (1960)

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

In general . . . , the routine for a recursive function uses itself as a subroutine. For example, the program for `subst x y z` uses itself as a subroutine to evaluate the result of substituting into the subexpression `car z` and `cdr z`. While `subst x y (cdr z)` is being evaluated, the result of the previous evaluation of `subst x y (car z)` must be saved in a temporary storage register. However, `subst` may need the same register for evaluating

`subst x y (cdr z)`. This possible conflict is resolved by the SAVE and UNSAVE routines that use the *public push-down list*[a]. The SAVE routine has an index that tells it how many registers in the push-down list are already in use. It moves the contents of the registers which are to be saved to the first unused registers in the push-down list, advances the index of the list, and returns to the program form which control came. This program may then freely use these registers for temporary storage. Before the routine exits it uses UNSAVE, which restores the contents of the temporary registers from the push-down list and moves back the index of this list. The result of these conventions is described, in programming terminology, by saying that the recursive subroutine is transparent to the temporary storage registers.

[a]1995: now called a stack

Garbage collection
McCarthy (1960)

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

[†] We already called this process "garbage collection", but I guess that I chickened out of using it in the paper—or else the Research Laboratory of Electronics grammar ladies wouldn't let me.

Garbage collection

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

> At a given point in the execution of a program P, a memory location ℓ is *garbage* if no completed execution of P from this point can access location ℓ. In other words, replacing the contents of ℓ or making this location inaccessible to P cannot affect any further execution of the program.

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

Programs as data

- LISP data and LISP program have the same syntax and internal representation. This allows data structures to be executed as programs and programs to be modified 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) ) )
  )
```

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

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

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

⁓ Lecture 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.

♦ **Chapter 5** of *Programming languages: Concepts & constructs* by R. Sethi (2ND EDITION). Addison-Wesley, 1996.

♦ **Chapter 7** of *Understanding programming languages* by M Ben-Ari. Wiley, 1996.

Parameters

There are two concepts that must be clearly distinguished:

♦ A *formal parameter* is a declaration that appears in the declaration of the subprogram. (The computation in the body of the subprogram is written in terms of formal parameters.)

♦ An *actual parameter* is a value that the calling program sends to the subprogram.

Example: Named parameter associations.

Normally the actual parameters in a subprogram call are just listed and the matching with the formal parameters is done by

position:

```
procedure Proc(First: Integer; Second: Character);
Proc(24,'h');
```

In Ada it is possible to use *named association* in the call:

```
Proc(Second => 'h', First => 24);
```

$\boxed{?}$ What about in ML? Can it be simulated?

This is commonly used together with *default parameters*:

```
procedure Proc(First: Integer := 0; Second: Character := '*');
Proc(Second => 'o');
```

Named associations and default parameters are commonly used in the command languages of operating systems, where each command may have dozens of options and normally only a few parameters need to be explicitly changed. However, there are dangers with this programming style. The use of default parameters can make a program hard to read because calls whose syntax is different actually call the same subprogram. Name associations are problematic because they bind the subprogram declaration and the calls more tightly than is usually needed. If you use only positional parameters in calling subprograms from a library, you could buy a competing library and just recompile or link. However, if you use named parameters, then you might have to do extensive modifications to your program to conform to the new parameter names.

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

1. 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.
2. 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.** Assignments inside the function body may have different effects under pass-by-value and pass-by-reference.
- **Aliasing.** Aliasing occurs when two names refer to the same object or location.

  Aliasing may occur when two parameters are passed by reference or one parameter passed by reference has the same location as the global variable of the procedure.

- **Efficiency.** Pass-by-value may be inefficient for large structures if the value of the large structure must be copied. Pass-by-reference maybe less efficient than pass-by-value for small structures that would fit directly on stack, because when parameters are passed by reference we must dereference a pointer to get their value.

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.

Actuals that do not have locations are passed by value. Actuals with locations are treated as follows:

1. *Copy-in phase*. Both the values and the locations of the actual parameters are computed. The values are assigned to the corresponding formals, as in call-by-value, and the locations are saved for the copy-out phase.
2. *Copy-out phase*. After the procedure body is executed, the final values of the formals are copied back out to the locations computed in the copy-in phase.

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* as follows:

1. Actual parameters are textually substituted for the formals. Possible conflicts between names in the actuals and local names in the procedure body are avoided by renaming the locals in the body.
2. The resulting procedure body is substituted for the call. Possible conflicts between nonlocals in the procedure body and locals at the point of call are avoided by renaming the locals at the point of call.

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.

- *In-line (or unnamed) blocks* are useful for restricting the scope of variables by declaring them only when needed, instead of at the beginning of a subprogram. The trend in programming is to reduce the size of subprograms, so the use of unnamed blocks is less useful than it used to be.

Nested procedures can be used to group statements that are executed at more than one location within a subprogram, but refer to local variables and so cannot be external to the subprogram. Before modules and object-oriented programming were introduced, nested procedures were used to structure large programs.

- 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 delcared 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 Algol-like programming languages evolved in parallel with the LISP family of languages, beginning with Algol 58 and Algol 60 in the late 1950s.
- The most prominent Algol-like programming languages are Pascal and C, although C differs from most of the Algol-like languages in some significant ways. Further Algol-like languages are: Algol 58, Algol W, Euclid, *etc.*

- The main characteristics of the Algol family are:
  - the familiar semicolon-separated sequence of statements,
  - block structure,
  - functions and procedures, and
  - static typing.

Algol 60

- Designed by a committee (including Backus, McCarthy, Perlis) between 1958 and 1963.
- Intended to be a general purpose programming language, with emphasis on scientific and numerical applications.
- Compared with FORTRAN, Algol 60 provided better ways to represent *data structures* and, like LISP, allowed functions to be called recursively.

  Eclipsed by FORTRAN because of the lack of I/O statements, separate compilation, and library; and because it was not supported by IBM.

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:
  - Automatic type conversions were not fully specified (*e.g.*, `x := x/y` was not properly defined when `x` and `y` were integers—is it allowed, and if so was the value rounded or truncated?).
  - 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.
- There are some awkward issues related to control flow, such as memory management, when a program jumps out of a nested block.

Algol 60 procedure types[a]

In Algol 60, the type of each formal parameter of a procedure must be given. However, `proc` is considered a type (the type of `proc`edures). This is much simpler than the ML types of function arguments. However, this is really a type loophole; because calls to procedure parameters are not fully type checked, Algol 60 programs may produce run-time errors.

Write a procedure declaration for `Q` that causes the following program fragment to produce a run-time type error:

```
proc P ( proc Q )
  begin Q(true) end;

P(Q);
```

where `true` is a Boolean value. Explain why the procedure is statically type correct, but produces a run-time type error. (You may assume that adding a Boolean to an integer is a run-time error.)

[a] Exercise 5.1 of *Concepts in programming languages* by J. Mitchell, CUP, 2003.

Algol 60 pass-by-name Copy rule

```
real procedure sum(E,i,low,high); value low, high;
  real E; integer i, low, high;
  begin
    sum:=0.0;  for i := low step 1 until high do sum := sum+E;
  end

integer j; real array A[1:10]; real result;
for j:= 1 step 1 until 10 do A[j] := j;
result := sum(A[j],j,1,10)
```

By the *Algol 60 copy rule*, the function call to `sum` above is equivalent to:

```
  begin
    sum:=0.0;  for j := 1 step 1 until 10 do sum := sum+A[j];
  end
```

Algol 60 pass-by-name[a]

The following Algol 60 code declares a procedure `P` with one pass-by-name integer parameter. Explain how the procedure call `P(A[i])` changes the values of `i` and `A` by substituting the actual parameters for the formal parameters, according to the Algol 60 copy rule. What integer values are printed by the `program`? And, by using pass-by-value parameter passing?

```
begin
  integer i; i:=1;
  integer array A[1:2]; A[1]:=2; A[2]:=3;

  procedure P(x); integer x;
    begin  i:=x; x:=1  end

  P(A[i]);  print(i,A[1],A[2])
end
```

[a] Exercise 5.2 of *Concepts in programming languages* by J. Mitchell, CUP, 2003.

Algol 68

- Intended to remove some of the difficulties found in Algol 60 and to improve the expressiveness of the language.

  It did not entirely succeed however, with one main problem being the difficulty of efficient compilation (*e.g.*, the implementation consequences of higher-order procedures where not well understood at the time).
- 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.)
- The decision to allow independent constructs to be combined without restriction also led to some complex features, such as assignable pointers.

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

- Designed in the 1970s by Niklaus Wirth, after the design and implementation of Algol W.
- Very successful programming language for *teaching*, in part because it was designed explicitly for that purpose.

  Also designed to be compiled in one pass. This hindered language design; *e.g.*, it forced the problematic `forward` declaration.
- Pascal is a *block-structured* language in which *static scope* rules are used to determine the meaning of nonlocal references to names.

- A Pascal program is always formed from a single main program block, which contains within it definitions of the subprograms used.

  Each block has a characteristic *structure*: a header giving the specification of parameters and results, followed by constant definitions, type definitions, local variable declarations, other nested subprogram definitions, and the statements that make up the executable part.
- 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).

  A restriction that made Pascal simpler than Algol 68:

```
procedure Allowed( j,k: integer );
procedure AlsoAllowed( procedure P(i:integer);
                        j,k: integer );
procedure
NotAllowed( procedure MyProc( procedure P(i:integer) ) );
```

- 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 programmig 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 uses a mixture of *name* and *structural* equivalence for determining if two variables have the same type.

  Name equivalence is used in most cases for determining if formal and actual parameters in subprogram calls have the same type; structural equivalence is used in most other situations.
- Parameters are passed by value or reference.

  Complete static type checking is possible for correspondence of actual and formal parameter types in each subprogram call.

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 ;
```

Variant records introduce *weaknesses* into the type system for a language.

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

Summary

- The Algol family of languages established the command-oriented syntax, with blocks, local declarations, and recursive functions, that are used in most current programming languages.
- The Algol family of languages is statically typed, as each expression has a type that is determined by its syntactic form and the compiler checks before running the program to make sure that the types of operations and operands agree.

Appendix to Lecture IV

Block-structured procedural languages
BCPL and C

References:

- **Chapters 1 to 3** of *BCPL, the language and its compiler* by M. Richards and C. Whitby-Strevens. CUP, 1979.

BCPL

```
LET BCPL BE
  $( LET CPL = "Combined Programming Language"
     WRITEF("Basic %S", CPL) $)
```

- Designed by Martin Richards in 1967 at MIT.
- Originally developed as a *compiler-writing* tool, has also proved useful as a *systems-programming* tool.
- BCPL adopted much of the syntactic richness of CPL; however, in order to achieve the *efficiency* necessary for systems-programming, its scale and complexity is far less than that of CPL.
- BCPL has only *one data type*: the bit-pattern.

BCPL[a]

Philosophy

- **Abstract machine.**

  The most important feature is the *store*: a set of numbered *storage cells* arranged so that the numbers labelling adjacent cells differ by one.

  All storage cells are of the same size and each of them holds a *value* (= bit-pattern). A value is the only kind of object that can be manipulated directly in BCPL, and every variable and expression in that language will always evaluate to one of these.

[a] Notes from Chapter 1 of BCPL, the language and its compiler by M. Richards and C. Whitby-Strevens. CUP, 1979.

Many basic operations on values are provided. One of these, of fundamental importance, is *indirection*. This operation takes one operand with is interpreted as an integer and yields the contents of the storage cell labelled by that integer.

- **Data types.**

  The design of BCPL distinguishes between two classes of data types.

  1. *Conceptual types.* The kind of abstract object the programmer had in mind.
  2. *Internal types.* Basic types for modelling conceptual types.

  Much of the flavour of BCPL is the result of the conscious design decision to provide only one internal type. The

most important effects on language design are:

1. There is no need for type declarations in the language. This helps to make programs concise and also simplifies problems such as the handling of actual/formal parameter correspondence and separate compilation.
2. It gives the language nearly the same power as one with dynamically varying types (as in LISP), and yet retains the efficiency of a language (like FORTRAN) with manifest types. In languages (such as Algol) where the elements of arrays must all have the same type, one needs some other linguistic device in order to handle dynamically varying data structures.
3. Since there is only one internal type in the language there can be no automatic type checking, and it is possible to

write nonsensical programs which the compiler will translate without complaint.

- **Variables.**

  The purpose of a *declaration* in BCPL is: to introduce a name and specify its scope; to specify its extent; to specify its initial value.

  In BCPL, variables may be divide into two classes:

  1. *Static variables.* The extent of a static variable is the entire execution time of the program. The storage cell is allocated prior to execution and continues to exist until execution is complete.
  2. *Dynamic variables.* A dynamic variable is one whose extent starts when its declaration is executed and continues until

execution leaves the scope of the variable. Dynamic variables are usually necessary when using routines recursively.

- **Recursion.**

  Procedures may be used recursively, and in order to allow for this and yet maintain very high execution efficiency, there is the restriction that the free variables of a procedure must be static.

- **Modularity.**

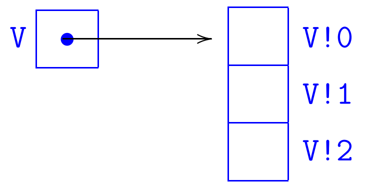
  BCPL uses a form of static storage, called *global vector*, which allows separately compiled modules to reference and call each other and to share data. This facility is not unlike the FORTRAN `COMMON` storage area.

Pointers and arrays

- A *pointer* in BCPL is the address of a word of store.
- The unary operator `@` is used to produce the *address* of a variable.
- The unary operator `!` is used to access the store cell pointed to by an address.
- The *array* declaration

  ```
  LET V = VEC 2
  ```

  establishes: (i) an array of three consecutive locations, and (ii) a separate variable `V` which is initialised to the address of the first location of the array:



$(\texttt{V!E} \overset{\text{def}}{=} \texttt{!(V+E)})$

Here `V` behaves like any other local variable, the main difference being that it is initialised by the compiler as a pointer. Hence its value can be copied into another variable (which as a result will also point to the same array), or passed as a parameter to a procedure.

Parameter passing

The BCPL procedure call uses the *call-by-value* technique for parameter passing.

As simple variables are passed by value, a copy is made of the actual parameters for the called procedure to use. Assigning to the formal parameters will not change the values of the original variables specified as actual parameters. This is similar to the Algol call-by-value mechanism, and in contrast to the FORTRAN parameter-passing mechanism.

The *effect* of the parameter-passing mechanism in BCPL is that simple variables are passed by value, and vectors by reference.

Procedures

```
LET COUNT(ARRAY, SIZE) = VALOF
$( LET NUMBER = 0
   FOR I = 0 TO SIZE DO ARRAY!I := 0
   $( LET C = READN()
      IF C < 0 RESULTIS NUMBER
      IF C > SIZE THEN C := SIZE
      ARRAY!C := ARRAY!C + 1
      NUMBER := NUMBER + 1
   $) REPEAT
$)
```

Note that there is no mention that `ARRAY` is an array. It is the programmer's responsibility to make sure that if a parameter is treated as an array inside a procedure, then an array is provided in the procedure call.

Procedures as values

- BCPL has been carefully designed so that it is possible to represent a procedure by a simple BCPL value, called the *procedure value*. The procedure value is placed in a variable bearing the name of the procedure.
- Procedure values can be assigned to ordinary variables.

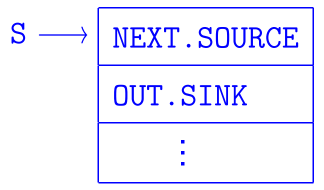
```
...
    LET CH = GETBYTE(FORMAT, P)
    SWITCHON CH INTO
    $( ...
       CASE 'S': F:= WRITES; GOTO L
       CASE 'C': F:= WRCH; GOTO L
       ...
    $)
...
L: F(ARG, N)
```

Thus, a procedure may be passed as a parameter to another procedure, or returned as the result of a function call.

Example: I/O streams

```
LET NEXT(S) = (S!0)(S)
LET OUT(S,X) BE (S!1)(S,X)
```

The relevant information concerning a particular stream `S` is stored in an array to which `S` points. The first few items in this array are procedure values. The array takes the following form



The procedure value held in the zeroth element of `S` represents the function which implements `NEXT`, *etc.*

C

- Designed and implemented from 1969 to 1973, as part of the Unix operating system project at Bell Labs.
- C was designed by Dennis Ritchie, as an evolution of Ritchie and Ken Thompson's language B, which was in turn based on BCPL.

  B was a pared-down version of BCPL, designed to run on the small computer used by the Unix project. The main difference between B and C is that B was untyped whereas C has types and type-checking rules.

- An important feature of C has been the tolerance of C compilers to type errors. This is partly because C evolved from typeless languages. As C evolved further and was later standardised by an ANSI committee in the mid-1980s, backward compatibility with the then-existing C code also prevented strong typing restriction. One of the most commonly cited advantages of C++ over C is the fact that C++ provides better type checking.

Summary

- The C programming language is similar to Algol 60, Algol 68, and Pascal in some respects: command-oriented syntax, blocks, local declarations, and recursive functions. However, C also shares some features with its untyped precursor BCPL, such as pointer arithmetic. C is also more restricted than most Algol-based languages in that functions cannot be declared inside nested blocks: All functions are declared outside the main program. This simplifies storage management.

~ Lecture 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.

- **Chapter 7** of *Programming languages: Concepts & constructs* by R. Sethi (2ND EDITION). Addison-Wesley, 1996.
- **Chapters 14 and 15** of *Understanding programming languages* by M Ben-Ari. Wiley, 1996.
- ⋆ B. Stroustrup. What is "Object-Oriented Programming"? (1991 revised version). Proc. 1st European Conf. on Object-Oriented Programming. (Available on-line from ⟨`http://public.research.att.com/~bs/papers.html`⟩.)

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
```

```
val BoolStack = newStack(true) ;

val BoolStack = ref {pop=fn,push=fn}
  : {pop:unit -> bool, push:bool -> unit} ref

val IntStack0 = newStack(0) ;

val IntStack0 = ref {pop=fn,push=fn}
  : {pop:unit -> int, push:int -> unit} ref

val IntStack1 = newStack(1) ;

val IntStack0 = ref {pop=fn,push=fn}
  : {pop:unit -> int, push:int -> unit} ref
```

```
IntStack0 := !IntStack1 ;

val it = () : unit

#pop(!IntStack0)() ;

val it = 1 : int

#push(!IntStack0)(4) ;

val it = () : unit
```

```
map ( #push(!IntStack0) ) [3,2,1] ;

val it = [(),(),()] : unit list

map ( #pop(!IntStack0) ) [(),(),(),()] ;

val it = [1,2,3,4] : int list
```

NB:

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

  $\boxed{!}$ Of course not!

  $\boxed{?}$ Why?

Basic concepts in object-oriented languages[a]

Four main language concepts for object-oriented languages:

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

[a]Notes from **Chapter 10** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.

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. In other words, the object "chooses" how to respond to a message.

  The important property of dynamic lookup is that different objects may implement the same operation differently, and so may respond to the same message in different ways.

- Dynamic lookup is sometimes confused with overloading, which is a mechanism based on *static types* of operands. However, the two are very different. $\boxed{?}$ Why?

  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.
- Abstraction based on objects is similar in many ways to abstraction based on abstract data types: Objects and abstract data types both combine functions and data, and abstraction in both cases involves distinguishing between a public interface and private implementation.

  Other features of object-oriented languages, however, make abstraction in object-oriented languages more flexible than abstraction with abstract data types.

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`.

- The primary advantage of subtyping is that it permits uniform operations over various types of data.

  For instance, subtyping makes it possible to have heterogeneous data structures that contain objects that belong to different subtypes of some common type.
- Subtyping in an object-oriented language allows functionality to be added without modifying general parts of a system.

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 form another, changes to one affect the other. This has a significant impact on code maintenance and modification.

Inheritance is not subtyping[a]

Subtyping is a relation on interfaces,
inheritance is a relation on implementations.

One reason subtyping and inheritance are often confused is that some class mechanisms combine the two. A typical example is C++, in which `A` will be recognized by the compiler as a subtype of `B` only if `B` is a public base class of `A`. Combining subtyping and inheritance is an elective design decision.

[a]Interested students may consult the POPL'90 paper *Inheritance is not subtyping* by W. R. Cook, W. L. Hill and P. S. Canning available on-line from ⟨http://www.cs.utexas.edu/users/wcook/publications.htm⟩.

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

Objects in SIMULA

Class: A procedure returning a pointer to its activation record.

Object: An activation record produced by call to a class, called an instance of the class. $\rightsquigarrow$ a SIMULA object is a closure

- SIMULA implementations place objects on the heap.
- Objects are deallocated by the garbage collector (which deallocates objects only when they are no longer reachable from the program that created them).

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 67 did not distinguish between public and private members of classes.

A later version of the language, however, allowed attributes to be made "protected", which means that they are accessible for subclasses (but not for other classes), or "hidden", in which case they are not accessible to subclasses either.

SIMULA
Further object-related features

- *Inner*, which indicates that the method of a subclass should be called in combination with execution of superclass code that contains the `inner` keyword.
- *Inspect* and *qua*, which provide the ability to test the type of an object at run time and to execute appropriate code accordingly. (`inspect` is a class (type) test, and `qua` is a form of type cast that is checked for correctness at run time.)

SIMULA
Sample code[a]

```
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***
```

[a]See Chapter 4(§1) of *SIMULA begin* (2nd edition) by G. Birtwistle, O.-J. Dahl, B. Myhrhug, and K. Nygaard. Chartwell-Bratt Ltd., 1980.

```
CLASS LINE(A,B,C); REAL A,B,C;
  COMMENT***Ax+By+C=0 REPRESENTATION
BEGIN
  BOOLEAN PROCEDURE PARALLELTO(L); REF(LINE) L;
    IF L =/= NONE THEN
      PARALLELTO := ABS( A*L.B - B*L.A ) < 0.00001;

  REF(POINT) PROCEDURE MEETS(L); REF(LINE) L;
    BEGIN REAL T;
      IF L =/= NONE and ~PARALLELTO(L) THEN
        BEGIN
          ...
          MEETS :- NEW POINT(...,...);
        END;
    END;***MEETS***
```

```
  COMMENT*** INITIALISATION CODE
  REAL D;
  D := SQRT( A**2 + B**2 )
  IF D = 0.0 THEN ERROR ELSE
    BEGIN
      D := 1/D;
      A := A*D;  B := B*D;  C := C * D;
    END;
END***LINE***
```

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;  P :- NEW POINT(1.0,2.5);

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

NB: SIMULA 67 did not hide fields. Thus,

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

changes the color 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.
   ```
   inspect a
     when B do b :- a
     otherwise ...
   ```
3. An error in the original SIMULA type checker surrounding the relationship between subtyping and inheritance:

   ```
   CLASS A;   A CLASS B;
   ```

   SIMULA subclassing produces the subtype relation `B<:A`.

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

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

This code . . .

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

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

P.S. The same type error occurs in the original implementation of Eiffel.[a]

[a] Interested students may consult *A proposal for making Eiffel type-safe* by W. Cook in *The Computer Journal*, 32(4):305-311, 1989. Available on-line from ⟨`http://www.cs.utexas.edu/users/wcook/publications.htm`⟩.

Smalltalk

- Developed at XEROX PARC in the 1970s.
- Major language that popularised objects; very flexible and powerful.
- 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.

Smalltalk

Motivating application : Dynabook

- Concept developed by Alan Kay.
- Influence on Smalltalk:
  - Objects and classes as useful organising concepts for building an entire programming environment and system.
  - Language intended to be the operating system interface as well as the programming language for Dynabook.
  - Syntax designed to be used with a special-purpose editor.
  - The implementation emphasised flexibility and ease of use over efficiency.

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
```

Some instance messages and methods

```
x || ^x
y || ^y
moveDx: dx Dy: dy ||
   x <- x+dx
   y <- y+dy
```

Executing the following code

```
p moveDX:2 Y:1
```

the value of the expressions `p x` and `p y` is the object `5`.

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

- `ColoredPoint` inherits instance variables `x` and `y`, methods `x`, `y`, `moveDX:Dy:`, *etc.*
- `ColoredPoint` adds an instance variable `color` and a method `color` to return the `color` of a `ColoredPoint`.
- The `ColoredPoint draw` method *redefines* (or *overrides*) the one inherited from `Point`.
- An option available in Smalltalk is to specify that a superclass method should be undefined on a subclass.

Example: Consider

```
newX:xv Y:yv C:cv ||
   ^ self new x:xv y:yv color:cv
cp <- ColoredPoint newX:1 Y:2 C:red
cp moveDx:3 Dy:4
```

The value of `cp x` is the object `4`, and the value of the expression `cp color` is the object `red`.

Note that even though `moveDx:Dy:` is an inherited method, defined originally for points without color, the result of moving a `ColoredPoint` is again a `ColoredPoint`.

Smalltalk
Abstraction

Smalltalk rules:

- *Methods are public*.

  Any code with a pointer to an object may send any message to that object. If the corresponding method is defined in the class of the object, or any superclass, the method will be invoked. This makes all methods of an object visible to any code that can access the object.
- *Instance variables are protected*.

  The instance variables of an object are accessible only to methods of the class of the object and to methods of its subclasses.

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

$$\begin{tikzcd}
 & \texttt{Integer} \arrow[dl, no head] \arrow[dr, no head] & \\
\texttt{SmallInt} & & \texttt{LargeInt}
\end{tikzcd}$$

Smalltalk

Interfaces as object types

Although Smalltalk does not use any static type checking, there is an implicit form of type that every Smalltalk programmer uses in some way.

The *type* of an object in Smalltalk is its interface, *i.e.* the set of messages that can be sent to the object without receiving the error “message not understood”.

The interface of an object is determined by its class, as a class lists the messages that each object will answer. However, different classes may implement the same messages, as there are no Smalltalk rules to keep different classes from using the same selector names.

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.

? Why?

- In Smalltalk, the interface of a subclass is often a subtype of the interface of its superclass. The reason being that a subclass will ordinarily inherit all of the methods of its superclass, possibly adding more methods.
- In general, however, subclassing does not always lead to subtyping in Smalltalk.
  1. Because it is possible to delete a method from a superclass in a subclass, a subclass may not produce a subtype.
  2. On the other hand, it is easy to have subtyping without inheritance.

Smalltalk

Object-oriented features

- *Objects*: A Smalltalk object is created by a class.

  At run time, an object stores its instance variables and a pointer to the instantiating class.
- *Classes*: A Smalltalk class defines variables, class methods, and the instance methods that are shared by all objects of the class.

  At run time, the class data structure contains pointers to an instance variable template, a method dictionary, and the superclass.

- *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.

∼ Lecture VI ∼

Types

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 organizing 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.

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 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 `t1` and `t2`, 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.

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. Programs do not need to contain types; types and instantiations of type variables are computed.
Example: SML.

Explicit. The program text contains type variables that determine the way that a construct may be treated polymorphically.

Explicit polymorphism often involves explicit instantiation or type application to indicate how type variables are replaced with specific types in the use of a polymorphic construct.

Example: C++ templates.

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.

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 ;
```

∼ Lecture VII ∼

Data abstraction and modularity
SML Modules

Reference:

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

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.

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
```

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.

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
```

abstypes

```
structure MyHiddenStack: STACK =
struct
  exception E ;
  abstype 'a reptype = S of 'a list (* <-- HIDDEN         *)
  with                              (*     REPRESENTATION *)
    val new = S [] ;
    fun push x (S s) = S( x::s ) ;
    fun pop( S [] ) = raise E
      | pop( S(_::t) ) = S( t ) ;
    fun top( S [] ) = raise E
      | top( S(h::_) ) = h ;
  end ;
end ;
```

```
val MyHiddenEmptyStack = MyHiddenStack.new ;
val MyHiddenStack0 = MyHiddenStack.push 0 MyHiddenEmptyStack ;
val MyHiddenStack01 = MyHiddenStack.push 1 MyHiddenStack0  ;
val MyHiddenStack0' = MyHiddenStack.pop MyHiddenStack01  ;
MyHiddenStack.top MyHiddenStack0' ;
```

```
val MyHiddenEmptyStack = - : 'a MyHiddenStack.reptype
val MyHiddenStack0 = - : int MyHiddenStack.reptype
val MyHiddenStack01 = - : int MyHiddenStack.reptype
val MyHiddenStack0' = - : int MyHiddenStack.reptype
val it = 0 : int
```

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.

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
```

```
map ( intStack.push )  [3,2,1] ;
map ( fn _ => let val top = intStack.top()
              in  intStack.pop(); top  end )
    [(),(),(),()] ;
```

```
val it = [(),(),()] : unit list
val it = [1,2,3,4] : intStack.itemtype list
```

Appendix to Lecture VII

An introduction to SML Modules
by Claudio Russo[a]

References:

- *ML for the Working Programmer* by Larry Paulson, Cambridge University Press. [A textbook for undergraduates and postgraduates.]

[a]⟨http://research.microsoft.com/~crusso⟩

- *The Standard ML Basis Library* by Reppy *et al.*, Cambridge University Press. [A useful introduction to ML standard libraries, and a good example of Modular programming.]
- *The Definition of Standard ML* by Milner *et al.*, MIT Press. [A formal definition of SML, using structured operational semantics. Useful for language implementors and researchers.]
- *Purely Functional Data Structures* by Chris Okasaki, Cambridge University Press. [Contains clever functional data structures, implemented in Haskell and SML Modules.]
- ⟨http://www.standardml.org⟩

Outline

Aim: To provide a gentle introduction to SML Modules.

- Review Core features related to Modules.
- Introduce the Modules Language, using small examples.
- Briefly relate Modules constructs to the Core language.
- Highlight some limitations of Modules.

NB: Only the important features of Modules are covered.

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 Core language

The SML Core is a strongly-typed call-by-value functional language with impure features (state and exceptions).

Types are mostly implicit and inferred by the compiler.

SML programs must be statically well-typed before being evaluated.

The Core is *type sound*: evaluation of a well-typed expression is guaranteed to be free of run-time type errors.

Core features

The SML Core has a number of other features:

- a rich collection of primitive types (e.g. `int, real, Int16.int, Word32.word`);
- mutually recursive polymorphic functions and datatypes;
- dynamically allocated, mutable references (type `'a ref`);
- exceptions;
- pattern matching on values.

Most of these features have little or no interaction with Modules.

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.

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 =
  IntNat.Nat.iter IntNatAdd.Nat.zero (IntNatAdd.add m) n
```

The dot notation (`IntNatAdd.Nat`) accesses a nested structure.

Sequencing dots provides deeper access (`IntNatAdd.Nat.zero`).

Nesting and dot notation provides *name-space* control.

Structure inclusion

To avoid nesting structures and dot notation, one can also directly `open` a structure identifier, importing its components:

```
struct   open Nat
         fun add n m = iter m succ n   end
```

NB: This is equivalent to the following

```
struct   type nat = Nat.nat
         val zero = Nat.zero
         val succ = Nat.succ
         val iter = Nat.iter
         fun add n m = iter m succ n   end
```

Though convenient, it’s bad style: the origin of an identifier is no longer clear and bindings are silently re-exported.

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*.

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 inclusion

To avoid nesting, one can also directly `include` a signature identifier:

```
sig include NAT
    val add: nat -> nat ->nat
end
```

NB: This is equivalent to the following signature.

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

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!

Using signatures to hide types identities

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`.

Now, the actual implementation of `AbsNat.nat` by `int` is *hidden*, so that `AbsNat.nat` ≠ `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`.

For example, the application `AbsNat.succ(~3)` is ill-typed: `~3` only 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.

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.

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`).

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`.

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)` ✓
`IntNatAdd.Nat.succ(IntNat.Nat.zero)` ✓
`AbsNatAdd.Nat.succ(AbsNat.Nat.zero)` ✓
`AbsNatAdd.Nat.succ(0)` ✗
`AbsNatAdd.Nat.succ(IntNat.Nat.zero)` ✗

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.

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.

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.

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.

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.