Cambridge University Computer Laboratory

Part Ib, Part II(General) and Diploma

Comparative Programming Languages

by
Martin Richards

mr@cl.cam.ac.uk

http://www.cl.cam.ac.uk/users/mr/

December 4, 2002

University Computer Laboratory
New Museum Site
Pembroke Street
Cambridge, CB2 3QG

Chapter 1

Introduction

1.1 Teaching and Learning Guide

1.1.1 Aims

The aims of this course are to introduce the principles and constraints that the affect the way programming languages are designed. Linguistic features of several languages will be explored and compared, so that both the mistakes and good ideas from the past can contribute to improved design and use of computer languages in the future.

1.1.2 Lectures

- **Introduction.** A brief history of programming languages. Comparing languages: programming paradigms, language features, execution styles. What makes a good good language.
- **Why study programming languages.** History, Changing influences, What makes a good programming language, application domains.
- **Language standardisation.** Language standards, examples from C and PL/I.
- **Fortran and COBOL.** History, example.
- **ALGOL.** History, Examples, Argument calling, Trouble spots, Algol W, Arguments, Records.
- **BCPL** History, Summary, Examples, Functions, Memory model, Objects, Coroutines.
- **C** History and importance, Example, Types, Operators, Statements, Declarations, Casts, Functions, Arrays and Pointers. Strings, Space allocation, Structures and Unions.

- **Representation of trees** In ML, BCPL and C.
- **Object Oriented Languages.** Simula-67 example, Class hierarchy and inheritance, SIMULA primitives.
- **Smalltalk.** History, Syntax, Classes, Declaration, Example showing the definition of new classes and methods
- **C++** Introduction to some of the features of C++, Design philosophy, Data hiding, data abstraction, Initialisation and cleanup, Parameterized types, iterators, Multiple inheritance.
- **Standards.**

1.1.3 Objectives

At the end of the course students should:

- Be familiar with several language paradigms and how they can be effective in different areas of application.
- Have a reasonable understanding of the compromises that have to be made in a standard specification of a language in relation to machine independence and efficieny.
- Appreciate the similarities and differences between various approaches taken by object oriented languages.
- Be sufficiently familiar with C and C++ to be able to read and undertand programs written in these languages.

1.2 The Course

- This course was given for the first time last year.
- It will cover language design issues of a variety of programming languages.
- It will give thumb nail sketches of several languages.
- Some languages, particularly C and C++, will be covered in slightly nore detail, since any computer scientist should be able to read code in these two.

1.3 Books

- Pratt, T.W. & Zelkowitz, M.V. (1996). *Programming Languages, Design and Implementation.* Prentice-Hall (3rd ed.).
- Appleby, D. & VandeKopple, J.J. (1997). *Programming Languages, Paradigm and Practice.* McGraw-Hill (2nd ed.).

- Stroustrup, B. *The C++ Programming Language.* Addison-Wesley.
- Stroustrup, B. (1994). *The Design and Implementation of C++.* Addison-Wesley.
- Mössenböck, H. (1993). *Object-Oriented Programming in Oberon-2*, Springer-Verlag.
- Antonakos, J.L. & Mansfield Jr., K.C. (1998). *Reference Guide to C and C++.* Prentice-Hall.
- Kernighan, B.W. & Ritchie, D.M. (1988). *The C Programming Language.* Prentice-Hall (2nd ed.).
- Banahan, M. *The C Book*, Addison Wesley

1.4 Why Study Programming Languages?

- To improve your ability to develop effective algorithms.
- To improve your use of your existing language.
- To increase your 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.

1.5 History

1951-55: Experimental use of expression compilers.

1956-60: FORTRAN, ALGOL 60, COBOL, LISP

1961-65: APL notation, ALGOL 60 (revised), SNOBOL, CPL

1966-70: APL, SNOBOL 4, FORTRAN 66, SNOBOL 4, BASIC, SIMULA 67, ALGOL 68, ALGOL-W, BCPL

1971-75: Pascal, PL/I (Standard), C, Scheme, Prolog

1976-80: Smalltalk, Ada, FORTRAN 77, ML

1981-85: Smalltalk-80, Growth of Prolog, Ada 83

1986-90: C++, SML

1991-95: Ada 95, TCL, Perl

1996-00: Java

1.6 Changing Influences

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

1.7 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.

1.8 Application Domains

- Business processing.
- Scientific.
- System.
- AI.
- Publishing.
- Process control
- Embedded systems.
- New paradigms.

Chapter 2

Language Standardisation

Consider the following program pragment.

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

Is it valid C and what is the value of `i`?

How do we answer such questions?

1. Read the reference manual.
2. Try it and see!
3. Read the ANSI C Standard.

2.1 Language Standards

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

2.2 Language Standards

- Timeliness.
- Conformance.
- Obsolescence.
- Ambiguity and freedom to optimise.

- Machine dependence.
- Undefined.
- Deprecated.

2.3 Language Standards

What do the following statements mean?

```
x = y + z;

x = z + y;

x = a + b + c;

x = (a + b) + c;

x = a + (b + c);

a = a + b + c;

b = a + b + c;

x = x + g()

x = g() + x;

x = g() + g();
```

What output should the following C program generate?

```
int x=1, y=1;

int g() { return ++y; }

int main() {
  x = ++x + ++x;
  y = g() + g();
  printf("x=%d y=%d\n", x, y);
  return 0;
}
```

Answer:

```
Linux              (gcc):   x=6 y=5
Mips               (gcc):   x=5 y=5
                   (cc):    x=6 y=5
DEC Alpha and Sun4 (gcc):   x=5 y=5
                   (cc):    x=5 y=5
```

2.4 Language Standards - PL/I

What does the following mean?

```
9 + 8/3
```

Is it?

1. 11.6666666....
2. Overflow
3. 1.6666666....

What does the following mean?

```
IF (1=1B) THEN ...
```

2.4.1 PL/I Decimal Types

The data type DEC(p,q) in PL/I describes decimal numbers containing p decimal digits of which q are after the decimal point.

Type rules for DECIMAL in PL/I:

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

So, 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(1+MAX(1-0,15-14)+MAX(0,14), 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
```

2.4.2 PL/I bit strings

Evaluation of: `IF (1=1B) ...`

```
      1   =   1B  -- DEC(1,0) and BIT STRING
=>  0001B = 1000B -- two BIT STRINGs
=>        0       -- FALSE
```

Chapter 3

FORTRAN

3.1 History

- The first high level language to become widely used.
- First developed by IBM for the IBM 704 computer in 1957.
- Emphesis on efficiency (for that machine).
- Static storage allocation and no recursion.
- Standards in 1966, 1977 and 1990.

3.2 FORTRAN 66 Example Program

```
PROGRAM TRIVIAL
INTEGER I
I=2
IF(I .GE. 2) CALL PRINTIT
STOP
END
SUBROUTINE PRINTIT
PRINT *,'Hello World'
RETURN
END
```

3.3 FORTRAN 77 Example Program

```
      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)
        GOTO 10
999   PRINT *,'All Done'
      STOP
      END

C  SUMMATION PROGRAM
       FUNCTION SUM(V,N)
       REAL :: V(N) ! New style declaration
       SUM = 0.0
       DO 20 I = 1,N
         SUM = SUM + V(I)
20       CONTINUE
       RETURN
       END
```

3.4 Features

- Static storage.
- No recursion (in FORTRAN 66), so no runtime stack needed.
- Separate compilation – extensive numerical libraries, e.g. NAG.
- Shared COMMON data areas.
- The EQUIVALENCE statement.
- Subroutine arguments passed by reference.
- Efficient compiled code.
- Extensively used for scientific work.
- Archaic syntax.

Chapter 4

COBOL

4.1 History

- COBOL (COmmon Business Oriented Language) has been widely used since the early 1960s.
- First version in 1960, revisions in 1974 and 1984.
- Uses nouns and verbs to describe actions.
- Complete separation of data descriptions from commands.
- Compilers used to be complex and slow.
- Programs somewhat easier to read than to write.
- Often blamed for the Y2K problem!

4.2 COBOL Example

```
IDENTIFICATION DIVISION
PROGRAM-ID. SUM-OF-PRICES.
AUTHOR. T-PRATT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. SUN.
OBJECT-COMPUTER. SUN.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INP-DATA ASSIGN TO INPUT.
    SELECT RESULT-FILE ASSIGN TO OUTPUT.
DATA DIVISION.
FD INP-DATA LABEL RECORD IS OMITTED.
01 ITEM-PRICE.
    02 ITEM PICTURE X(30)
    02 PRICE PICTURE 9999V99.
WORKING-STORAGE SECTION.
77 TOT PICTURE 9999V99, VALUE 0, USAGE COMPUTATIONAL.
01 SUM-LINE.
```

```
    02 FILLER VALUE ' SUM =' PICTURE X(12).
    02 SUM-OUT PICTURE $$,$$$,$$9.99.
    02 COUNT-OUT PICTURE ZZZ9.
    ... More data
PROCEDURE DIVISION.
START.
    OPEN INPUT INP-DATA AND OUTPUT RESULT-FILE.
READ-DATA.
    READ INP-DATA AT END GO TO PRINT-LINE.
    ADD PRICE TO TOT.
    ADD 1 TO COUNT.
    MOVE PRICE TO PRICE-OUT.
    MOVE ITEM TO ITEM-OUT.
    WRITE RESULT-LINE FROM ITEM-LINE.
    GO TO READ-DATA.
PRINT-LINE.
    MOVE TOT TO SUM-TOT.
    ... More statements
    CLOSE INP-DATA AND RESULT-FILE.
    STOP RUN.
```

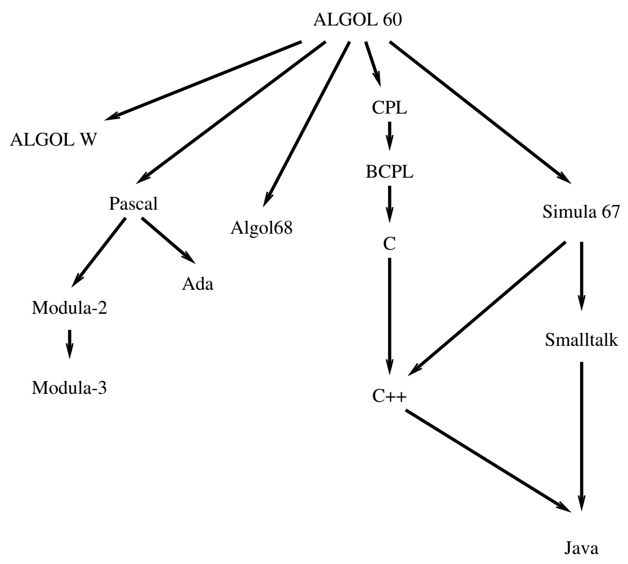
4.3 COBOL

- Data often held in a form that requires little conversion during input/output.
- Arithmetic operations simple and less important than data movement.
- Static data allocation and no recursion.
- Originally the only subroutine mechanism was the PERFORM statement which allowed specified paragraphs to be executed, possibly repeated.
- Such subroutines were parameterless.

Chapter 5

ALGOL 60

5.1 ALGOL Influence



5.2 History

- Designed by a committee in 1958
- ALGOL 60 (1960), Revised (Rome 1962)
- Eclipsed by FORTRAN because
  - No I/O statements.
  - No separate compilation.
  - No library.
  - Not supported by IBM.
- But had a major effect on language design.

5.3 ALGOL Innovations

- Block structure.
- Explicit type declarations for variables.
- Scope rules for local variables.
- Dynamic (not static) lifetimes for variables.
- Nested if-then-else expressions and statements.
- Call-by-value and call-by-name arguments.
- Recursive subroutines.
- Arrays with dynamic bounds.
- Use of BNF syntax description.

5.4 ALGOL Example

```
comment  The following procedure
         will transpose a square matrix;

procedure transpose(A, m)
  value m;
  real array A; integer m;
begin integer i, k;
  for i := 1 step 1 until m do
    for k := i+1 step 1 until m do
      begin real w;
            w := A[i,k];
            a[i,k] := a[k,i];
            a[k,i] := w
      end
end transpose
```

5.5 Call by Name

```
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)
```

The call:

```
sum(A[j], j, 1, 10)
```

is equivalent to:

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

5.6 Nested Procedures

```
integer procedure f(n);
  integer n;
begin
  integer a;

  integer procedure g(m);
    integer m;
  begin g := n + a + m;
        a := a+1
  end;

  if n=0 then a := 0;

  f := g(a)
end
```

5.7 Trouble spots 1

11 different possible answers to:

```
begin
  integer a;

  integer procedure f(x, y);
      value x, y;
      integer x, y;
    a := f := x+1;

  integer procedure g(x);
      integer x;
    x := g := a+2;


  a := 2;
  outreal(1, a + f(a, g(a))/g(a))
end
```

5.8 Trouble spots 2

- The types and mode of calling of procedure arguments could not be specified.

- The goto statement could cause a jump out to a label outside the current procedure.

- Labels were numeric (as in FORTRAN) and could be muddled with integers when passed in function arguments.

- Automatic type conversions were not fully specified, for example `x := x/y` was not properly defined when `x` and `y` were integers. (Is it allowed, and if so was `x` rounded or truncated.

- Own variables were a disaster.

- No precision specified for real numbers.

5.9 ALGOL W

- Designed by Niklaus Wirth (about 1968).

- More pragmatic with repect to efficiency.

- One of the first languages to have structures in the form of records with named fields.

- Allowed pointers to records.

- It had four modes of calling procedure arguments.

5.10 ALGOL W Procedure Arguments

```
PROCEDURE SUB1(INTEGER A;
               INTEGER VALUE B;
               INTEGER RESULT C;
               INTEGER VALUE RESULT D);
BEGIN
      B:= 7;
      A:=A+B;
      C:=B+D;
      D:=5
END

PROCEDURE SUB1(INTEGER A;
               INTEGER B;
               INTEGER C;
               INTEGER D);
BEGIN
      INTEGER BB;
      INTEGER CC;
      INTEGER DD;
      BB:=B;
      DD:=D;

      BB:= 7;
      A:=A+BB;
      CC:=BB+DD;
      DD:=5;

      C:=CC;
      D:=DD
END
```

5.11 ALGOL W Records

```
RECORD CARD (INTEGER VAL;
             INTEGER SUIT);
RECORD HAND (REFERENCE(CARD) CARD;
             REFERENCE(HAND) NEXT);

REFERENCE(HAND) NORTH;

NORTH := HAND(CARD(1,1),
         HAND(CARD(12,1),NULL));

... VAL(CARD(NEXT(NORTH))) ...
```

5.12 ALGOL W Problems

What does `X(Y)` mean?

It depends on the types of `X` and `Y`.

It could be:

- An array subscription.
- A procedure call.
- A field selection.
- A record constructor application.

Unfortunately, since formal procedures do not have their argument types specified, the types of variable are not always known.

Chapter 6

BCPL

An implementation is available on the Linux machines `gloves` and `belt`. To use the BCPL Cintcode system on these machines:

1. Set the PATH and BCPLPATH shell variables

   ```
   export BCPLPATH=/group/clteach/mr/BCPL/cintcode
   export PATH=$PATH:$BCPLPATH
   ```

   (If you plan to make much use of BCPL, put the above two commands in your .bash_profile (or .profile) file)

2. Copy some demonstration BCPL programs into a directory called bcplprogs

   ```
   mkdir bcplprogs
   cd bcplprogs
   cp -rp /group/clteach/mr/BCPL/bcplprogs/demos .
   cd demos
   ls
   ```

3. Choose one (eg primes.b), type it and compile and run it:

   ```
   cinterp
   type primes.b
   c b primes
   primes
   ```

4. To leave the BCPL cintcode system, type CTRL-c or:

   ```
   logout
   ```

The entire source of the BCPL Cintcode system can be found in:

```
/group/clteach/mr/BCPL/cintcode
```

The BCPL Cintcode system for Linux, Windows 95/98/NT and other systems is also available via my Home Page:

```
http://www.cl.cam.ac.uk/~mr
```

A manual for the BCPL Cintcode System will soon be available via my home page.

6.1 BCPL History

- It was designed and implemented in early 1967 when I was at at MIT.
- It is a very cut down version of CPL(1962-68) that is easy to implement.
- It was used extensively for systems research, particularly for the development of the Tripos Operating System and early developments of the Cambridge Ring.
- It is still used as a testbed for compiler research.

6.2 BCPL Summary

- It was designed as a systems programming language suitable for writing compilers and operating systems.
- It is typeless. All values are the same size (typically 32 bits).
- Values can be used to represent characters, integers, truth values, and pointers to data or code.
- Any operation is allowed on any value even though the result is sometimes meaningless.
- Pointers can be used to represent:
  - character strings,
  - vectors,
  - structures,
  - functions and
  - program labels.
- Memory was modelled as a collection of equal sized words, addressed by consecutive integers. This allows machine independent pointer arithmetic.

6.3 BCPL Example

```
GET "libhdr"

LET f(n) = n=0 -> 1, n*f(n-1)

LET start() = VALOF
{ FOR i = 1 TO 9 DO
    writef("f(%n) = %i5*n", i, f(i))
  RESULTIS 0
}
```

6.4 BCPL Functions

- Function arguments are called by value and stored in consecutive memory locations.
- Functions are variadic (allow a variable number of arguments, as in `writef`).
- Functions may be passed as arguments, returned as results, or assigned.
- Although function definitions may be nested, they may not contain variables referring to either arguments or locals of enclosing functions.
- Functions can be be correctly represented by just their entry addresses.
- Separate compilation was allowed using the global vector for inter module referencing.

6.5 BCPL Example

```
GET "libhdr"

GLOBAL { count:200; all:201  }

LET try(ld, row, rd) BE
  TEST row=all
  THEN count := count + 1
  ELSE { LET poss = all & ~(ld | row | rd)
         UNTIL poss=0 DO
         { LET p = poss & -poss
           poss := poss - p
           try(ld+p << 1, row+p, rd+p >> 1)
         }
       }

LET start() = VALOF
{ all := 1

  FOR i = 1 TO 12 DO
  { count := 0
    try(0, 0, 0)
    writef("%i2-queens solutions %i5*n",
            i,                   count)
    all := 2*all + 1
  }

  RESULTIS 0
}
```

6.6 BCPL Example

```
GET "libhdr"

LET coins(sum) = c(sum, (TABLE 50, 20, 10, 5, 2, 1))

AND c(sum, t) = sum<0 -> 0,
                sum=0 | !t=1 -> 1,
                c(sum, t+1) + c(sum-!t, t)
```

```
LET start() = VALOF
{  writes("Coins problem*n")

   t(0)
   t(1)
   t(2)
   t(5)
   t(21)
   t(100)
   t(200)
   RESULTIS 0
}

AND t(n) BE
  writef("Sum = %i3  number of ways = %i6*n",
                 n,                 coins(n))
```

6.7 BCPL Memory

```
         Address      Contents

           1000:         13
           1001:       2000
           1002:          0
           ...
           2000:         61
           2001:          0
           2002:         24
```

The expression: `1000!1!2` evaluates as follows:

```
         1000 ! 1 ! 2
    =   (!(1000+1)) ! 2
    =   (!(1001)) ! 2
    =     2000 ! 2
    =     !(2000+2)
    =     !2002
    =     24
```

6.8 BCPL Streams

```
LET next(s) = (s!0)(s)

LET out(s, x) = (s!1)(s, x)

LET close(s) = (s!2)(s)
```

The functions `s!0`, `s!1`, `s!2` are analogous to methods of the object `s` in an object oriented language. The functions defined above were used by Christopher Strachey in the operating system OS/6 in 1970.

6.9 BCPL Objects

In an object oriented language, classes have fields and methods. Often the fields hold different values for different instances of a class, but the methods are usually the same for each instance. This suggests an implementation in which the methods are held in a vector (of method functions) which is referenced by each instance of the class. The fields vector must be accessible to each method and so is normally passed as a (hidden) extra argument in a method call.

6.10 BCPL Objects

The following scheme works well for BCPL. In the scope of the declaration:

```
MANIFEST { F=0; G=1; H=2 }
```

The expression

```
               G#(obj, 13, 63)
```

means:

```
        (obj!0!G)(obj, 13, 63)
         (fns!1) (obj, 13, 63)
         methodG (obj, 13, 63)

obj -> 0:  fns -------------> 0: methodF
       1: FieldA             1: methodG
       2: FieldB             2: methodH
          ...
       n: FieldZ
```

6.11 BCPL Coroutines

Coroutines are somewhat like threads (or processes).

- They simulate parallel execution.
- They share the same address space.
- But they only giveup control voluntarily (no pre-emption).

Each coroutine needs its own runtime stack. When control passes from one BCPL coroutine to another, a value is passed. This value looks like an argument in the coroutine than is suspending itself, and like a result in the coroutine that is resuming control.

6.12 BCPL Coroutine Implementation

The implementation is as follows:

```
  --------------------->-----------------
 |                                       |
 ---------------------------------------------------------
|p  parent nxt  f  size  c  | .. stack .. suspended frame ..
 ---------------------------------------------------------
 |    |     |   |   |    |
 |    |     |   |   |    System work variable
 |    |     |   |   Size of the coroutine stack
 |    |     |   Coroutine main function
 |    |     Next coroutine
 |    The parent coroutine
 The save stack frame pointer
```

```
cptr := createco(f, size)   -- Create a coroutine.
deleteco(cptr)              -- Delete a coroutine.
res := callco(cptr, val)    -- Call another coroutine (cptr).
res := cowait(val)          -- Transfer control to the parent.
res := resumeco(cptr, val)  -- Suspend the current coroutine
                               and transfer to another.
```

The body of a coroutine behaves like:

```
   c := f(cowait(c))    // Repeat for ever
   REPEAT
```

6.13 Coroutines Example

```
GET "libhdr"

MANIFEST { upb = 4000 }

LET prime1() = VALOF

{ LET isprime = getvec(upb)
  FOR i = 2 TO upb DO
    isprime!i := TRUE // Until disproved
  FOR p = 2 TO upb IF isprime!p DO
  { LET i = p*p
    UNTIL i>upb DO
    { isprime!i := FALSE; i := i + p }
    cowait(p)
  }
  freevec(isprime)
  RESULTIS 0
}

AND prime2() = VALOF
{ FOR n = 2 TO upb DO
  { LET a, b = 2, 1
    FOR i = 1 TO n DO
```

```
    { LET c = (a+b) REM n; a := b; b := c }
    IF a=1 DO cowait(n)
  }
  RESULTIS 0
}

LET start() = VALOF
{ LET P1 = createco(prime1, 100)
  LET P2 = createco(prime2, 100)
  LET n1, n2, min = 0, 0, 0
  { IF n1=min DO n1 := callco(P1)
    IF n2=min DO n2 := callco(P2)
    min := n1<n2 -> n1, n2
    UNLESS n1=n2 DO
      writef(" %i4 from P%c*n", min, n1<n2 -> '1', '2')
  } REPEATUNTIL min=0
  deleteco(P1)
  deleteco(P2)
  RESULTIS 0
}
```

Chapter 7

C

7.1 History

- Designed by Dennis Ritchie and Ken Thompson at Bell Labs in 1972.
- It was based in B, a squeezed down minimal subset of BCPL implemented on an 8K PDP-7 by Ken Thompson.
- In 1970 the Unix project acquired a 24K PDP-11 and B was expanded to include structures and a more operators. This language became known as C.
- It was first used to write the kernel of Unix and has been closely associated with Unix ever since.
- ANSI Standard (1989)
- ISO Standard (1990)

7.2 C Example program

```
clove$  cat example.c
#include <stdio.h>

const int maxsize=9;

int convert(char ch) { return ch-'0'; }

int sum(int v[], int n) {
  int res=0;
  int j;
  for(j=0; j<n; j++) res+=v[j];
  return res;
}

int main() {
  int a[maxsize];
  int j, k;
  while(k=convert(getchar())) {
    for(j=0; j<k; j++) a[j]=convert(getchar());
    for(j=0; j<k; j++) printf("%d ", a[j]);
    printf("; SUM=%d\n", sum(a, k));
    while(getchar() != '\n');
  }
  return 0;
}
```

7.3 Compiling and Running

```
clove$  cc -o example example.c
clove$  example
41234
1 2 3 4 ; SUM 10
512345
1 2 3 4 5 ; SUM 15
0
clove$
```

7.4 C Primitive Types

Numeric types:

```
unsigned char
unsigned short
unsigned int
unsigned long
char
short
int
long

float
double
long double
```

Numeric constants:

```
'A'            -- int
123U           -- unsigned int
1234567890UL   -- unsigned long

123            -- int
0xFF0037       -- int
1234567890L    -- long

123.456F       -- float
123.456        -- double
123.456e-5     -- double
123.456E12     -- double
123.456L       -- long double
```

7.5 Monadic Expression Operators

```
e(e,e,...,e)   -- function call
e[e,e,...,e]   -- subscripted expression
e->name        -- structure selector
e.name         -- field selector

!e             -- boolean not
~e             -- bitwise not
++e    e++     -- pre/post increment
```

```
--e    e--     -- pre/post decrement
+e             -- monadic plus
-e             -- monadic minus
(type)e        -- cast (or type conversion)
*e             -- indirection
&e             -- address of
sizeof e       -- size of the given value
```

7.6 Infixed Expression Operators

```
e * e          -- multiplication
e / e          -- division
e % e          -- remainder
e + e          -- plus
e - e          -- minus

e << e         -- left shift
e >> e         -- right shift
```

7.7 Infixed Expression Operators

```
e<e  e>e  e<=e  e>=e    -- relations
e==e  e!=e              -- relations

e & e          -- bitwize and
e ^ e          -- bitwize xor
e | e          -- bitwize or
e && e         -- boolean and
e || e         -- boolean or
e ? e : e      -- conditional expression
e=e  e+=e ...  -- assignments
e,e,...,e      -- expression list
```

7.8 Statements

```
{ d d ... d s s ... s }  -- a block
e;
;
if (e) s
if (e) s else s
if (e) s else s
while (e) s
do s while(e);
for(e;e;e)s  -- eg for(i=0;i<k;i++)v[i]=0;
switch(e) { case k: s
            ...
            case k: s
            default: s
          }
break;
continue;
return e;
goto name;
```

```
name: s
```

7.9 Declarations

```
int i, j=1, k;
float x;
long a, b;

char v[10];
int w[4] = { 0, 1, 2, 3};

int *p, **q;
int * tab[256];    -- vector of pointers

int f(int,int);    -- function declaration
int (*f)(int,int); -- pointer to a function
```

7.10 Casts

```
declaration         cast

unsigned int i;     (unsigned int)
float x;            (float)

char v[10];         (char [10])
int w[4];           (int [4])

int *p;             (int *)
int **q;            (int **)
int * tab[256];     (int *[256])

int f(int,int);     (int(int,int))
int (*f)(int,int);  (int (*)(int,int))
void (*g)(void);    (void(*)(void))
```

7.11 Functions

7.11.1 Declarations

```
/* a declaration  */
int mymax(int, int);
```

7.11.2 Definitions

```
/* a definition */
int mymax3(int a, int b, int c) {
  return mymax(a, mymax(b, c));
}

/* another definition */
int mymax(int a, int b) {
  return a>b ? a : b;
```

```
}
```

7.12 Arrays and Pointers

```
int a[20], *p;

p = & a[5];
*p = 0;  /* equivalent to a[5] = 0;  */

for (p=&a[0]; p<&a[20]; p++) *p = 0;
```

In older compilers the above is more efficient than:

```
int a[20], i;

for (i=0; i<20; i++) a[i]=0;
```

7.13 Pointer Arithmetic

An integer may be added to a pointer. It returns a pointer to the adjacent location of the appropriate size.

If `p` points to a 32 bit integer on a byte addressed machine, then `p+1` or `1+p` points to the adjacent 32 bit integer. In machine code terms this typically involves adding 4 to the byte address held in `p`.

```
  C                  BCPL

 int a[20];          LET a = VEC 19
 int *p;             LET p = ?

 a[1]                a!1
 *(a+1)              !(a+1)
 *(1+a)              !(1+a)
 1[a]                1!a
 &a[1]               @ a!1
 & *(a+1)            @ !(a+1)
 a+1                 a+1
```

7.14 Strings

C character strings are zero terminated char vectors. BCPL strings are byte vectors with the length held in byte zero.

```
  C                  BCPL

 char s[5]="ABCD";   LET s = "ABCD"

 s[0] == 'A'         s%0 =  4
 s[1] == 'B'         s%1 = 'A'
 s[2] == 'C'         s%2 = 'B'
 s[3] == 'D'         s%3 = 'C'
 s[4] ==  0          s%4 = 'D'
```

7.15 String Copying

7.15.1 C:

```
void copystring(char *from, char *to) {
  while (*from) *to++ = *from++;
  *to=0;
}
```

7.15.2 BCPL:

```
LET copystring(from, to) BE
  FOR i = 0 TO from%0 DO to%i := from%i
```

7.16 Space Allocation

7.16.1 C:

```
int *p = (int *)malloc(sizeof(int[100]));
if(p==0) .... /* no space  */
...
free(p);
```

7.16.2 BCPL:

```
LET p = getvec(99)
IF p=0 DO  ... // no space
...
freevec(p)
```

7.17 Structures, Version 1

```
#include <stdio.h>

struct Inode {
  int val;
  struct Inode* next;
} *p, *q;

struct Inode *mk(int x, struct Inode *rest) {
  struct Inode *res =
   (struct Inode*)malloc(sizeof(struct Inode));
  res->val = x;
  res->next = rest;
  return res;
}

int main(int argc, char** argv) {
  p = mk(13, 0);
  q = mk(541, p);
  printf("%d\n", q->next->val);
  return 0;
```

```
}
```

7.18 Structures, Version 2

```
#include <stdio.h>

typedef struct Inode {
  int val;
  struct Inode *next;
} *Ilist;

Ilist mk(int x, Ilist rest) {
  Ilist res = (Ilist)malloc(sizeof(struct Inode));
  res->val = x;
  res->next = rest;
  return res;
}

int main(int argc, char** argv) {
  Ilist p = mk(13, 0);
  Ilist q = mk(541, p);
  printf("%d\n", q->next->val);
  return 0;
}
```

7.19 Structures, Version 3

```
#include <stdio.h>

typedef struct Inode Inode;

struct Inode {
  int val;
  Inode *next;
};

Inode *mk(int x, Inode *rest) {
  Inode *res = (Inode *)malloc(sizeof(Inode));
  res->val = x;
  res->next = rest;
  return res;
}

int main(int argc, char** argv) {
  Inode *p = mk(13, 0);
  Inode *q = mk(541, p);
  printf("%d\n", q->next->val);
  return 0;
}
```

7.20 Field Selection

The field selection operator is dor (`.`) as in Java. The arrow notation is a shortcut for indirection followed by field selection.

`e -> name` means `(* e).name`.

7.21 Unions

Sometimes it is useful to have a variable whose value is one type at one moment and another type later. This is possible using unions. Syntactically a union declaration is just like a structure declaration, only the `struct` is replaced by `union`. It causes all the fields to be overlaid at the same position, rather than being given distinct locations.

7.22 Unions

```
union {
  float f;
  int i;
} var;

var.f = 23.5
printf("val = %f\n", var.f);

var.i = 5
printf("val = %d\n", var.i);
```

7.23 Example Problem

Construct a tree representation suitable for the expressions like `1+2*3`, and define a function `eval` to evaluate it. The required structure is something like:

```
      ADD
     /   \
    /     \
  NUM     MUL
   1     /   \
        /     \
      NUM     NUM
       2       3
```

7.24 ML Solution

```
datatype E = NUM of int
           | POS of E
           | NEG of E
           | MUL of E * E
           | DIV of E * E
           | ADD of E * E
           | SUB of E * E;
```

```
fun eval(NUM k)    = k
  | eval(POS e)    = eval e
  | eval(NEG e)    = ~ (eval e)
  | eval(MUL(x,y)) = eval x  *  eval y
  | eval(DIV(x,y)) = eval x div eval y
  | eval(ADD(x,y)) = eval x  +  eval y
  | eval(SUB(x,y)) = eval x  -  eval y;

eval( ADD(NUM 1,
          MUL(NUM 2, NUM 3)));
```

7.25 MCPL Solution

```
GET "mcpl.h"

MANIFEST
  NUM, POS, NEG, MUL, DIV, ADD, SUB

FUN eval                              // MCPL is like BCPL but
: [NUM, val]    => val                // allows pattern matching.
: [POS,   x]    => + eval x           // It is typeless,
: [NEG,   x]    => - eval x           // uses a BCPL-like stack,
: [MUL,   x, y] => eval x * eval y    // and has no garbage collector
: [DIV,   x, y] => eval x / eval y
: [ADD,   x, y] => eval x + eval y
: [SUB,   x, y] => eval x - eval y

FUN start : =>
  printf("val = %d\n",
          [ADD,[NUM,1],[MUL,[NUM,2],[NUM,3]]]
        )
```

7.26 BCPL Solution

```
GET "libhdr"

MANIFEST {
  Op=0; Rand1=1; Rand2=2
  NUM=1; POS=2; NEG=3; MUL=4; DIV=5; ADD=6; SUB=7
}

LET eval(e) = VALOF SWITCHON Op!e INTO
{ CASE NUM: RESULTIS Rand1!e
  CASE POS:  RESULTIS + eval(Rand1!e)
  CASE NEG:  RESULTIS - eval(Rand1!e)
  CASE MUL:  RESULTIS eval(Rand1!e) * eval(Rand2!e)
  CASE DIV:  RESULTIS eval(Rand1!e) / eval(Rand2!e)
  CASE ADD:  RESULTIS eval(Rand1!e) + eval(Rand2!e)
  CASE SUB:  RESULTIS eval(Rand1!e) - eval(Rand2!e)
}

LET mk1(op, a) = VALOF
{ LET r = getvec(1)
  Op!r, Rand1!r := op, a
```

```
  RESULTIS r
}

LET mk2(op, a, b) = VALOF
{ LET r = getvec(2)
  Op!r, Rand1!r, Rand2!r := op, a, b
  RESULTIS r
}

LET start() = VALOF
{ LET exp = mk2(ADD, mk1(NUM, 1),
                     mk2(MUL, mk1(NUM,2), mk1(NUM,3)))
  writef("eval(exp) = %n*n", eval(exp))
  RESULTIS 0
}
```

7.27 Union Example

```
#include <stdio.h>

enum Ops {NUM, POS, NEG, MUL, DIV, ADD, SUB};

typedef struct Num Num;
typedef struct Monad Monad;
typedef struct Dyad Dyad;

typedef union Expr {
  Num *N;
  Monad *M;
  Dyad  *D;
} Expr;

struct Num {
  int op;
  int val;
};

struct Monad {
  int op;
  Expr rand;
};

struct Dyad {
  int op;
  Expr left;
  Expr right;
};

Expr mknum(int n) {
  Expr res;
  Num *e = (Num *) malloc(sizeof(Num));
  e->op = NUM;
  e->val = n;
  res.N = e;
  return res;
}
```

```
Expr mk1(int operator, Expr a) {
  Expr res;
  Monad *e = (Monad *) malloc(sizeof(Monad));
  e->op = operator;
  e->rand = a;
  res.M = e;
  return res;
}

Expr mk2(int operator, Expr a, Expr b) {
  Expr res;
  Dyad *e = (Dyad *) malloc(sizeof(Dyad));
  e->op = operator;
  e->left = a;
  e->right = b;
  res.D = e;
  return res;
}

int eval(Expr e) {
  switch(e.M->op)
  { default:  return 0;

    case NUM: return e.N->val;
    case POS: return eval(e.M->rand);
    case NEG: return - eval(e.M->rand);
    case MUL: return eval(e.D->left) * eval(e.D->right);
    case DIV: return eval(e.D->left) / eval(e.D->right);
    case ADD: return eval(e.D->left) + eval(e.D->right);
    case SUB: return eval(e.D->left) - eval(e.D->right);
  }
  return 0;
}

int main() {
  Expr exp = mk2(ADD,
                 mknum(1),
                 mk2(MUL, mknum(2), mknum(3)));
  printf("eval(exp) => %d\n", eval(exp));
}
```

Chapter 8

Object Oriented Languages

Object Oriented Languages support:

- Data abstraction (the encapsulation of state with operations).
- Information hiding (encapsulation).
- Message passing and polymorphism.
- Inheritance, including dynamic binding.

Significant languages in the history of Object Orientation.

- Simula-67
- Smalltalk
- C++
- Java
- Oberon, Eiffel and many others

8.1 Simula-67

- Originated in Norway by Nygaard and Dahl in 1961, and was fully developed by 1967.
- Based on Algol 60.
- Designed for discrete event simulation.
- It extended Algol to contain objects that had state and a thread of control (similar to coroutine) that could simulate parallel activities.
- It had an inheritance mechanism.

8.2 Simula-67 Example

A simulation of a post office with a door, 4 counters, each with a clerk and a queue, and a random supplied of customers entering the door.

```
BEGIN

CLASS customer(tasks, oldlady)
  INTEGER tasks; BOOLEAN oldlady;     BEGIN ... END;

CLASS queue;                          BEGIN ... END;

CLASS clerk(q); ref(queue)q;          BEGIN ... END;

CLASS door;                           BEGIN ... END;

CLASS counter;                        BEGIN ... END;

REAL opentime, closetime;
REF(counter) c1, c2, c3, c4;

opentime := 8.00;
closetime := 17.00;

hold(opentime);
c1 :- NEW counter;
c2 :- NEW counter;
c3 :- NEW counter;
c4 :- NEW counter;
NEW door;

hold(closetime);
END
```

8.3 Simula-67 Example – More Detail

```
CLASS customer(tasks, oldlady)
  INTEGER tasks; BOOLEAN oldlady;
BEGIN
  REF(customer) next;
  REF(counter) service;

  WHILE tasks>0 DO
  BEGIN
    <assign value to service>;
    IF oldlady THEN <enter front of service.q>
               ELSE <enter tail of service.q>;
    IF service.postofficer is free
    THEN <activate service.postofficer>
    ELSE passivate;
    <participate in transaction>;
    <leave counter>
    tasks := tasks-1;
  END
END;
```

```
CLASS queue;
BEGIN
  REF(customer) first, last;
END;


CLASS clerk(q); REF(queue)q;
BEGIN
  REF(customer) person;
seviceing:
  WHILE q.first =/= NONE DO
  BEGIN person :- q.first;
        <take person out of the queue>;
        <engage with him in transactions>;
  END;
  <do other work until interrupted by the
   arrival of a new customer in the queue>;
  GOTO servicing;
END;

CLASS door;
BEGIN
  REAL arrtime;

  WHILE time<=closetime DO
  BEGIN arrtime := <time of next arrival>;
        hold(arrtime);
        NEW customer(<initial value of tasks>,
                     <initial value of oldlady>);
  END;
END;



CLASS counter;
BEGIN
  REF(queue) q;
  REF(clerk) postofficer;

  q :- NEW queue;
  postofficer :- NEW clerk(q);
END;
```

8.4 Simula-67 Class Hierachy

A hierachy of classes can be declared.

```
  CLASS a ...;
a CLASS b ...;
a CLASS c ...;
b CLASS d ...;
b CLASS e ...;
```

```
        Hierarchy              Objects

                         a     b     c     d     e
                        ---   ---   ---   ---   ---
            a          | a | | a | | a | | a | | a |
           / \          ---  |---| |---| |---| |---|
          /   \              | b | | c | | b | | b |
         b     c              ---   ---  |---| |---|
        / \                              | d | | e |
       /   \                              ---   ---
      d     e
```

8.5 Simula-67 Various Primitive

Componentes in a simulation which may be scheduled should be prefixed by the class `process`. The class outline is:

```
link CLASS process;
BEGIN BOOLEAN PROCEDURE idle; ...;
      REAL PROCEDURE evtime; ...;
      REF(process) PROCEDURE nextev; ...;
      detach;
      ...
      passivate;
END;
```

```
REF(process) PROCEDURE current; ...
```

returns the current process.

```
REAL PROCEDURE time; ...
```

returns the current time.

```
PROCEDURE hold(t); REAL t; ...
```

suspend the current process until time `t`..

```
PROCEDURE cancel(p); REF(process) p; ...
```

remove p from the event queue.

```
ACTIVATE p;
ACTIVATE p AT time;
ACTIVATE p DELAY time;
```

activate `p` at specified time.

Chapter 9

Smalltalk

Smalltalk was designed by Alan Kay in 1972 when he was working at Xerox PARC (Palo Alto) as part of his Dynabook project.

It was influenced by some of Papert's work at MIT on Logo, an interactive highly graphical computer learning environment intended to teach programming to children.

Dynabook was a far sighted early version of a handheld computer. The first was made of cardboard and filled with lead shot, to see what size and weight people would put up with.

Smalltalk was designed an an interpretive language, in which all names were looked up at runtime from and environment of declared names.

In more modern implementations code is compiled giving a speedup of 450 times! There are several Smalltalk implementations including a GNU version and one from the Disney Corporation (where Alan Kay now works).

This latter one is embedded in a system called Squeak and the version of Smalltalk is called Self. It consists of a complete program development system that is responsive and small enough (approx 1-2 Mbytes) to fit in a Palm Top or Handheld computer.

Smalltalk is a purely object-oriented language which cleanly supports the notions of classes, methods, messages and inheritance. All smalltalk code consists of chains of messages sent to objects.

9.1 Squeak Implementation

Squeak is a rather good (and fun) implementation of Smalltalk. It is free and runs on many different platforms. It has a home page on the web:

`http://squeak.cs.uicu.edu`

that contains much useful material. It contains links to downloads for various systems and a link to Wolfgang Kreutzer's good tutorial for the Squeak beginner entitled "Basic Aspects of Squeak and the Smalltalk-80 Programming Language" which was the source Cookie Monster example used in this chapter. This paper is also available via:

`http://www.cl.cam.ac.uk/Teaching/2000/CompProgLangs/smalltalk.html`

You can try out a read only version of the Linux vesrion of Squeak on belt or gloves by typing the following commands:

```
cd /group/clteach/mr/squeak
./sqk
```

Alternatively read the file `/group/clteach/mr/README.squeak` (on gloves or belt).

9.2 Smalltalk Syntax

Its syntax is almost trivial.

```
BODY -> | name ... name | E . E ... E

E    -> name                    x  Transcript   :=
      | # E                     #green
      | number                  123
      | string                  'hello'
      | ( E )
      | ^ E                     return with value of E
      | [ E . E ... E ]         [Transcript show: 'hello']
      | [ :name ... :name | E . E ... E ]  [: x | x + 1]
      | E name
      | E name E ... name E
      | E name E ... name E ; ... ; name E ...
```

9.3 Smalltalk Fragments

```
|count|
count := 0.
[count < 3]
  whileTrue:
    [ (count odd) ifTrue: [Transcript show: 'ODD']
                  ifFalse: [Transcript show: 'EVEN'].
       count := count+1.
       Transcript cr.
    ]

|canon|
canon := [ :singers :song |
           singers do:
             [:voice | Transcript
                         show: voice asString, ',', song;
                         cr]]


canon value: #(kermit jasper fred) value: 'croak'.
```

9.4 Smalltalk Classes

For programming convenience Smalltalk classes a grouped into *Categories* and their methods are also grouped into categories.

```
Morphic-Window                -- Class categories
Graphics-3D
```

```
...
Interface-Pluggable
Sesame-Street
   CookieMonster              -- a class
     initialization           -- method categories
     access
     queries
       isAsleep               -- methods
       isAwake
       isFull
           self isEmpty       -- definition of isFull
             ifFalse: [^ self tummy size >= self hunger]
             ifTrue: [^false]
     actions
     private
   Monster                    -- another class
Tools-Outlines
...
```

9.5 Declaring a new Class

A class is created by sending a message to the class that is going to be its parent. For example

```
!Object subclass: #Monster
  instanceVariableNames: 'colour tummy'
  classVariableNames: ' '
  poolDictionaries: ' '
  category: 'Sesame-Street'!
```

This create the class `Monster` which is a subclass of `Object`. It has two instance variables (`colour` and `tummy`) and no class variables, and `Monster` is one of the classes in the category `Sesame-Street`. Ignore `poolDictionaries`. Classes are created dynamically (during program execution).

9.6 Definition of CookieMonster

We now create a class `CookieMonster` as a subclass of `Monster`. It is also a member of `Sesame-Street`.

```
!Monster subclass: #CookieMonster
  instanceVariableNames: 'state hunger'
  classVariableNames: ' '
  poolDictionaries: ' '
  category: 'Sesame-Street'!
```

9.7 Defining Methods

Methods can be defined by sending a `methodsFor` message to a class. For example:

```
!Monster methodsFor: 'actions'!
eat: someItem
  self tummy add: someItem
!!
```

```
!Monster methodsFor: 'queries'!
isEmpty
 ^ self tummy isNil
!!
```

9.8 Defining Methods

```
!Monster methodsFor: 'access'!
colour
 ^ colour
!
colour: aSymbol
 colour := aSymbol
!
tummy
 ^ tummy
!
tummy: aCollection
 tummy := aCollection
!!
!Monster methodsFor: 'initialization'!
initialize
  self colour: #green.
  self tummy: Bag new
!!
```

9.9 Defining Class Methods

Class methods and variables in Smalltalk are like static methods and variables in Java.

There is one class method to define for `Monster`.

```
!Monster class  methodsFor: 'creation'!
new
  ^ super new initialize
!!
```

9.10 Cookie Monster Methods

Cookie Monsters inherit from Monster, but add more specific behaviour of their own.

```
!CookieMonster methodsFor: 'private'!
askForCookie
  ^ FillInTheBlank request: 'Give me a cookie'
!
complainAbout: anItem
  Transcript show: 'No want ', anItem printString.
  Transcript cr.
  self colour #red
!
isCookie: anItem
  ^ ((anItem = 'cookie') | (anItem = #cookie))
!!
```

9.11 Cookie Monster Methods

```
!CookieMonster methodsFor: 'actions'!
eat: aCookie
  super eat: aCookie.
  self colour: #green
!
nag
[self isAwake]
whileTrue:
[| item |
 item := self askForCookie.
 (self isCookie: item)
   ifTrue: [self eat: item]
   ifFalse: [self complainAbout: item].
   (self isFull) ifTrue: [self sleep]]
!
sleep
  self state: #asleep.
  self hunger: 0
!
wakeUp
  self tummy: Bag new.
  self state: #awake.
  self hunger: (Random new next * 13).
  self nag.
!!
```

9.12 Cookie Monster Methods

```
!CookieMonster methodsFor: 'queries'!
isAsleep
  ^ state = #asleep
!
isAwake
  ^ self isAsleep not
!
isFull
  self isEmpty
    ifFalse: [^ self tummy size >= self hunger]
    ifTrue: [^false]
!!
!CookieMonster methodsFor: 'access'!
hunger
  ^ hunger
!
hunger: anIntegerNumberOfCookies
  hunger := anIntegerNumberOfCookies
!
state
  ^ state
!!
```

Chapter 10

Object oriented Programming

This chapter covers the paper:

What is "Object-Oriented Programming"? by Bjarne Stroustrup (the designer of C++).

This paper is availabe via:

`http://www.cl.cam.ac.uk/Teaching/2000/CompProgLangs/bjarne.ps`

or

`http://cm.bell-labs.com/cm/cs/cstr.html`

10.1 Language Design Philosophy

- All features must be cleanly and elegantly integrated into the language.
- It must be possible to use features in combination to achieve solutions that would otherwise have required extra separate features.
- There should be as few spurious and "special purpose" features as possible.
- A feature should be such that its implematation does not impose significant overheads on programs that do not require it.
- A user need only know about the subset of the language explicitly used to write a program.

10.2 Programming Paradigms

10.2.1 Procedural Programming

```
double sqrt(double arg) {
  // code for calculating a square root
}

void some_function() {
  double root2 = sqrt(2);
  // ...
}
```

10.2.2 Data Hiding

The file `stack.h`

```
char pop();
void push(char);
const stack_size = 100;
```

The file `stack.cpp`

```
#include "stack.h"
static char v[stack_size]; // local to module
static char* p = v;

char pop() {
  // check for underflow and pop
}
void push(char ch) {
  // check for overflow and push
}
```

The file `prog.cpp`

```
#include "stack.h"

void some_function() {
  push('c');
  char c = pop();
  if (c != 'c') error("impossible");
}
```

10.2.3 Data Abstraction

The file `stack.h`

```
class stack_id; // stack_id is a type

stack_id create_stack(int size);
void destroy_stack(stack_id);
char pop(stack_id);
void push(stack_id, char);
```

The file `prog.cpp`

```
#include "stack.h"

void some_function() {
  stack_id s1;
  stack_id s2;
  s1 = create_stack(200);
  // Oops: forgot to create s2
  push(s1, 'a');
  char c1 = pop(s1);
  if (c1 != 'a') error("impossible");
  push(s2, 'b');
  char c2 = pop(s2);
  if (c2 != 'b') error("impossible");
  destroy_stack(s2);
  // Ooops: forgot to destroy s1
}
```

The above scheme is nearly good enough when no more than one object of a type is needed.

```
class complex {
  double re, im;
public:
  complex(double r, double i) { re=r; im=i; }
  complex(double r) { re=r; im=0; }

  friend complex operator+(complex, complex);
  friend complex operator-(complex, complex);
  friend complex operator-(complex);
  friend complex operator*(complex, complex);
  friend complex operator/(complex, complex);
};
```

The definition of `complex +` could be:

```
complex operator+(complex a1, complex a2) {
  return complex(a1.re+a2.re, a1.im+a2.im);
}
```

Typical use:

```
complex a = 2.3;
complex b = 1/a;
complex c = a+b*complex(1, 2.3);
c = - (a/b)+2;
```

10.3 Problems with Data Abstraction

```
class point { ... };
class color { ... };

enum kind { circle, triangle, square };

class shape {
  point center;
  color col;
  kind k;
  // representation of shape
public:
  point where() { return center; }
  void move{point to} { center=to; draw(); }
  void draw();
  void rotate(int s);
  // more operations
};

void shape::draw()
  switch (k) {
    case circle:   // draw a circle
                   break;
    case triangle: // draw a triangle
                   break;
    case square:   // draw a square
```

```
                   break;
  }
}
```

10.4 Object-Oriented Programming

```
class shape {
  point center;
  color col;
  ...
public:
  point where() { return center; }
  void move{point to} { center=to; draw(); }
  virtual void draw();
  virtual void rotate(int);
  ...
};
```

Typical use of class `shape` is:

```
void rotate_all(shape* v, int size, int angle) {
// rotate all members of a vector by given angle
  for (int i=0; i<size; i++) v[i].rotate(angle);
}
```

Typical definition of a particular shape.

```
class circle : public shape {
  int radius;
public:
  void draw() { ... }
  void rotate() {}  // yes, the null function
}
```

10.5 Initialization and Cleanup

```
class vector {
  int sz;
  int* v;
public:
  void init(int size);
  ...
};
```

```
vector v;
// don't use yet
v.init(10);
// ok to use v now
..
```

10.6 Initialization and Cleanup

It is better to use constructors and destructors.

```
class vector {
  int sz;
  int* v;
public:
  vector(int);                  // constructor
  ~vector();                    // destructor
  int& operator[] (int index); // subscript operator
```

Typical definitions:

```
vector::vector(int s) {
  if (s<=0) error("bad vector size");
  sz = s;
  v = new int[s]; // allocate an array
}

vector::~vector() {
  delete v;  // deallocate the vector
}
```

10.7 Assignment and Initialization

It is possible to control all copy operations in C++.

```
class vector {
  int sz;
  int* v;
public:
  vector(int);                  // constructor
  ~vector();                    // destructor
  int& operator[] (int index);   // subscript operator
  void operator=(const vector&); // assignment
  vector(const vector&);         // initialization
};
```

Typical definitions:

```
vector::operator=(const vector& a) {
  // check size and copy elements
  if (sz != a.sz) error("bad vector size for =");
  for (int i=0; i,sz; i++) v[i]=a.v[i];
}

vector::vector(const vector& a) {
  // initialize a vector from another vector
  sz = a.sz;
  v = new int[sz];
  for (int i=0; i,sz; i++) v[i]=a.v[i];
}
```

10.8 Parameterized Types

It would be more useful if the writer of the `vector` class did not know what the element type it is to use.

```
template<class T> class vector { // element type T
  T* v;
  int sz;
public:
  vector(int) {
    if (s <= 0) error("bad vector size");
    v = new T[sz=s];
  }
  T& operator[] (int index);   // subscript operator
  int size() { return sz; }
  ...
};
```

A template specifies a family of types.

```
vector<int> v1(100);
vector<complex> v2(200);

v2[i] = complex(v1[x], v1[y]);
```

Usually, only one version of `size()` is needed.

10.9 Iterators

Iterators can be provided by overloading the function call operator.

```
class vector_iterator {
  vector& v;
  int i;
public:
  vector_iterator(vector& r) { i=0; v=r; }
  int operator()() {
    return i<v.size() ? v.elem(i++) : 0;
  }
};
```

It can now be used, as in:

```
void f(vector& v) {
  vector_iterator next(v);
  int i;
  while (i=next()) print(i);
}
```

10.10 Iterators (alternative)

The iterator mechanism can be put in the “container” type, if preferred.

```
class vector {
  int* v;
  int i;
  int current;
public:
  int next() { return (++current < sz) ? v[current] : 0; }
  int curr() { return v[current];
  int prev() { return (0 <= --current) ? v[current] : 0; }
};
```

It can now be used, as in:

```
void f(vector& v) {
  vector v(sz);
  int i;
  while (i=v.next()) print(i);
}
```

10.11 Multiple Implementations

```
template<class T> class stack {
public:template<class T> class stack {
  virtual void push(T) = 0; // pure virtual function
  virtual T pop() = 0;
};
```

This declares an abstract class. It can be used, but not created.

```
stack<cat> s; // error: stack is abstract

void some_fn(stack<cat> s, cat kitty) { // ok
  s.push(kitty);
  cat c2 = s.pop();
  ...
}

template<class T> class aStack : public stack<T> {
  // actual representation of a stack object
  // in this case an array
  ...
public:
  aStack(int size);
  ~aStack();
  void push(T);
  T pop();
};
```

Elsewhere we can create a list based stack.

```
template<class T> class lStack : public stack<T> { ...
};
```

We can now create stacks of both sorts

```
void g() {
  lstack<cat> s1(100);
  astack<cat> s2(100);

  cat ginger;
  cat snowball;

  some_fn(s1, ginger);
  some_fn(s2, snowball);
}
```

10.12 Multiple Inheritance

Multiple inheritance is when a class inherits from more than one base class. C++ allows multiple inheritance many languages (eg Java) do not.

```
class my_displayed_task : public displayed, public task {
  // my stuff
};

class my_task : public task { // not displayed
  // my stuff
};

class my_displayed : public displayed { // not a task
  // my stuff
};
```

Using only single inheritance leads to code duplication and loss of flexibility – typically both.

10.13 Ambiguity

Multiple inheritance cause ambiguity when two parents have methods with the same name. Such ambiguity can be solved at compile time.

```
class A { public: void f(); ... }
class B { public: void f(); ... }
class C : public A, public B { ... }

void g(C* p) {
  p->f(); // error: ambiguous
}
```

A solution is the following:

```
class C : public A, public B {
  ...
public:
  void f();
  ...
}
```

```
void C::f()
{ // C's own stuff
  A::f();
  B::f();
}
```

Chapter 11

Questions

1. a) Briefly discuss what developments have affected the design of computer programming languages over the period 1951 to the present day.

   b) List and give brief explanations of six features that you regard as important for a programming language to gain wide acceptance.

   c) Discuss the difference between a proprietary and a consensus language standard. To what extent can a programmer expect a program that conforms to a standard generate identical results when run using different conforming compilers.

   d) Discuss some of the compromises that have to be made when writing a programming language standard.

2. Both FORTRAN and COBOL are languages that were designed in the late 1950s. Briefly state the application areas that they were designed to support and discuss whether it would be better to have one language that covered boths application areas. Why are these two languages still widely used?

3. Outline some of the innovations in ALGOL that made it such a significant and influential language. Why was it never as widely used as FORTRAN?

4. ALGOL provided call by name and call by value, Pascal allowed call by value and call by reference, and ALGOL W provides call by name and a variety of others including call by result, and call by value result. Briefly describe the calling mechanisms mentioned above and discuss why most recent programming languages only provide one mode of parameter passing.

5. It is required to represent the parse tree of an expression such as 1+2*3 where the leaf nodes contain integers and the operators are ADD, SUB, MUL and DIV. Compare how you might do this in BCPL, ML and C, and illustrate your answer by outlining the code for an evaluation function eval that computes the value of such an expression represented as a parse tree. Syntactic errors will not be penalised, but your answers should be in the typical style of each language.

6. Write brief notes on the following topics relating to BCPL.

   a) The advantages and disadvantages of a typeless language.

   b) To what extent can programs in BCPL be written in an Object-Oriented style.

   c) The coroutine mechanism of BCPL,

   d) In BCPL implementations, a function can be represented by just its entry addresses, but in languages, such as ALGOL and CPL, a more complicated representation was necessary. Discuss the restriction in BCPL that makes this more efficient representation possible.

7. Briefly outline with the aid of small examples how Simula-67 Classes can be defined and used. How does the language support inheritance.

   What application area was Simula-67 aimed to support? Discuss whether the same area can now be supported equally well by more modern languaged such as C++ or Java.

8. Smalltalk is a language with an almost trivial syntax containing none of the usual constructs such as if-the-else and while-do found in many imperative languages. Give a brief outline of the main syntactic features of Smalltalk and show how conditional and repetitive constructs can be encoded. Illustrate your answer by giving an iterative version of the factorial function.

   Outline the differences between class and instance variables, and class and instance methods.

9. Many features in C++ were designed so that they could be implemented very efficiently. Select three such features that are present in C++ but not in Java. Discuss whether the extra efficiency obtained is worth the added complexity of the language.

11.1 Past Examination Questions

1998 paper 6 question 6

Many languages forbid explicit pointer arithmetic or restrict its use. What kinds of problems are they seeking to avoid? Why is it allowed (with some restrictions) in C and C++, and commonly used by programmers? [8 marks]

What ar the dangers inherent in allowing memory deallocation to be under the direct control of the programmer? Given these dangers, why has direct memory deallocation not been automated in C and C++? [12 marks]

1999 paper 5 question 5

Discuss what it means for a program written in C to conform with the ANSI Standard and for the compiler to conform to the standard. [4 marks]

Discuss the reasons why a program conforming to a standard may fail to yield exactly the same results when run on different conforming compilers. [10 marks]

Discuss why it is sometimes possible to write apparently simple expressions, such as 9+8/3 in PL/I, that yield unexpected results in languages that have a wide variety of numerical data types. To what extent is it possible to eliminate such problems in future languages? [6 marks]

1999 paper 6 question 5

Early programming languages has relatively poor facilities for type checking, data abstraction, data hiding and encapsulation. Explain the meaning of these terms and discuss their importance. [6 marks]

Outline the key features that a language must have to be called object-oriented and briefly discuss to what extent C++, Java and Smalltalk have them. [7 marks]

Briefly discuss some of the reasons why C++ programs typically run faster than equivalent programs run in Java or Smalltalk. [7 marks]

2000 paper 5 question 5

Outline how you implement complex numbers in C++. Your implementation should attempt to make complex numbers look as if they were built into the language by allowing new comples numbers be declared, initialised, assigned and operated on by the normal arithmetic operators. 13 marks]

Discuss to what extent a good C++ compiler could implement your version of complex number as efficiently as if they had been a primitive type in the language. [7 marks]

2000 paper 5 question 5

Outline how you implement complex numbers in C++. Your implementation should attempt to make complex numbers look as if they were built into the language by allowing new comples numbers be declared, initialised, assigned and operated on by the normal arithmetic operators. 13 marks]

Discuss to what extent a good C++ compiler could implement your version of complex number as efficiently as if they had been a primitive type in the language. [7 marks]

2000 paper 6 question 5

Give a brief summary of the main syntactic constructs found in the programming language Smalltak. Other languages often have the conditional constructs if-then-else and while. Show how the two constructs can be defined in Smalltalk. [8 marks]

Illustrate the use of Smalltalk by showing how you would define a method to compute the factorial of an integer. [8 marks]

Although Smalltalk was originally designed to be an interpretive language, modern implementations are dramatically more efficient. Briefly outline what techniques might have been used to make this improvement. [4 marks]

2001 paper 5 question 5

(a) Outline the main innovations that are in Simula 67 but were not in Algol 60, paying particular attention to Simula Classes. [6 marks]

(b) Illustrate how Simula can be used to simulate a small restaurant with six tables, two waiters and small groups of customers arriving at random intervals. You need only specify the classes that you would define. Most algorithmic details may be omitted. [6 marks]

(c) Discuss to what extent Simula has been made redundant by the development of modern object-oriented languages such as java. [8 marks]

2001 paper 6 question 5

(a) Briefly explain the concept of coroutines as used in BCPL and outline the effect of the library functions `create(f, size)`, `deleteco(cptr)`, `callco(cptr, val)` and `cowait(val)`. [10 marks]

(b) Outline how you would design a coroutine to merge, in increasing order, two infinite streams of increasing integers supplied by two other coroutines. [5 marks]

(c) Briefly outline how you would implement an analogous merging mechanism in an object-oriented language, such as Java, that does not provide a coroutine mechanism. [5 marks]

2002 paper 5 question 5

Guess what this question will be and answer it. [20 marks]

2002 paper 6 question 5

Guess what this question will be and answer it. [20 marks]