| Machine code (hex): | Assembly language: |
|---|---|
| 23fc 0000 0001 0000 0040 | movl #0x1,n |
| | compare: |
| 0cb9 0000 000a 0000 0040 | cmpl #0xa,n |
| 6e0c | bgt end_of_loop |
| 06b9 0000 0001 0000 0040 | addl #0x1,n |
| 60e8 | bra compare |
| | end_of_loop: |

**Figure 1.1** Sample machine code and assembly language texts.

FORTRAN code:

```
   DO  2  N  =  1,10
2  CONTINUE
```

COBOL code:

```
        MOVE  1  TO  N.
AGAIN.
        IF  N  IS  EQUAL  TO  10
            GOTO  END−DEMO.
        ADD  1  TO  N.
        GOTO  AGAIN.
END−DEMO.
```

**Figure 1.2** The loop from Figure 1.1 in FORTRAN and COBOL.

C code:

```
for (n = 1; n <= 10; n++) {

}
```

Ada code:

```
for N in 1..10 loop
   null;
end loop;
```

**Figure 1.3** The loop from Figure 1.1 in C and Ada.

| | |
|---|---|
| 1940s | Any program that runs is a good program |
| 1950s | Any program can be done in one instruction less |
| 1960s | ‘Look what this one-line program can do!’ |
| 1970s | Algol 60 was a considerable improvement over all its successors |
| 1980s | Ada is a language that is doomed to succeed |
| 1990s | It’s not easy to make ten computers do the work of one |

**Figure 1.4** A one-liner history of programming languages.

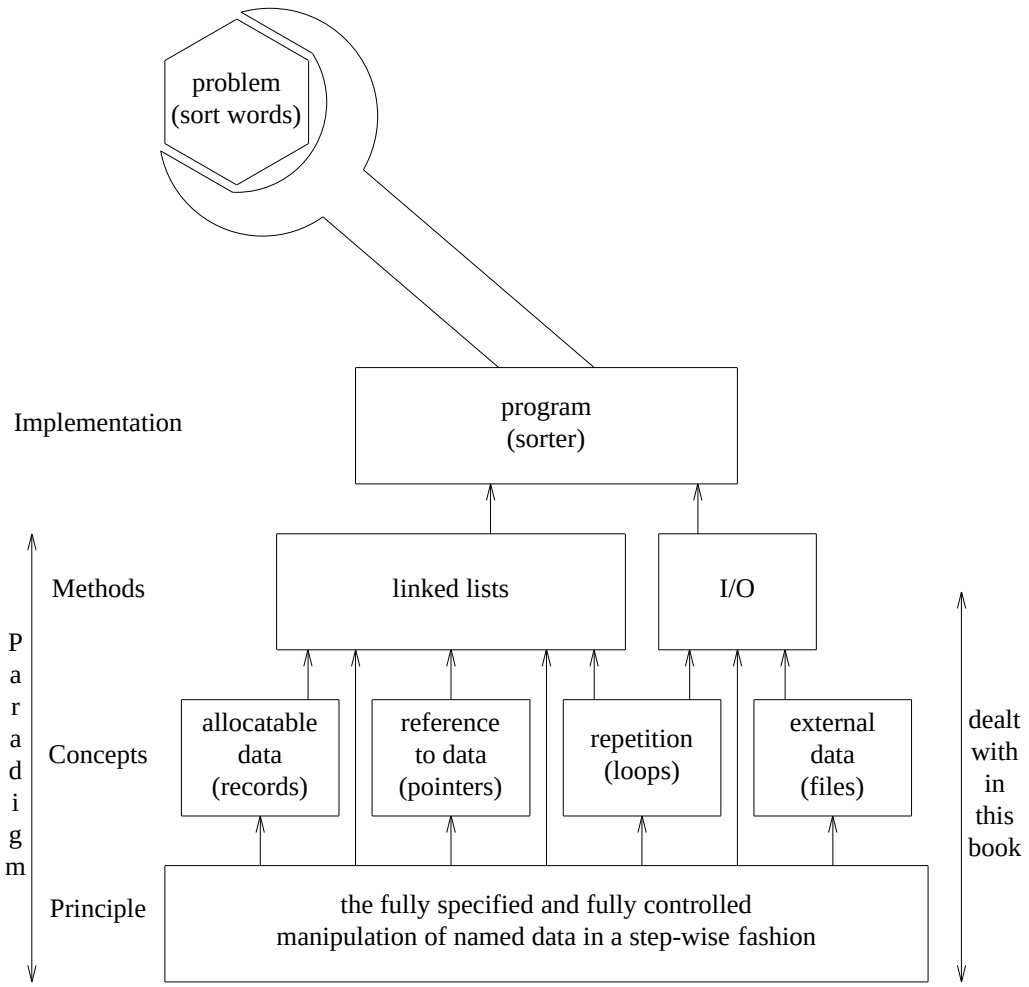


**Figure 1.5** The relationships between the components of a paradigm.

| | |
|---|---|
| 1st generation | machine languages |
| 2nd generation | assembly languages |
| 3rd generation | procedural languages |
| 4th generation | application languages |
| 5th generation | AI techniques, inference languages |
| 6th generation | neural networks |

**Figure 1.6** Language generations.

| Level | Instructions | Memory handling | Examples |
|---|---|---|---|
| *Low Level Languages (LLLs)* | simple machine-like instructions | direct memory access and allocation | assembly languages, Autocode |
| *High Level Languages (HLLs)* | expressions and explicit flow of control | memory access and allocation through operators (for example, new) | FORTRAN, the Algols, Pascal, C, Ada |
| *Very High Level Languages (VHLLs)* | fully abstract machine | fully hidden memory access and automatic allocation | SETL, Prolog, Miranda |

**Figure 1.7** Language levels.

| Level | Sample unit |
|---|---|
| lexical | begin |
| syntactic | if ... then ... else ... end if |
| contextual | a variable with its type, name, declaration and uses |
| semantic | flow of control during procedure call and return |

**Figure 1.8** Summary of the structure level of programs.

<program> ::= **program** <statement>* **end**
<statement> ::= <assignment> | <loop>
<assignment> ::= <identifier> := <expression> ;
<loop> ::= **while** <expression> **do** <statement>$^{+}$ **done**
<expression> ::= <value> | <value> + <value>
| <value> <= <value>
<value> ::= <identifier> | <number>

**Figure 1.9** A sample grammar in EBNF notation.

```
program
  n := 1;
  while n <= 10 do
    n := n + 1;
  done
end
```

**Figure 1.10** A simple program from the grammar in Figure 1.9.

<program>
**program** <statement>* **end**
**program** <statement> <statement> **end**
**program** <assignment> <loop> **end**
**program** <identifier> := <expression>; **while** <expression> **do** <statement>$^{+}$ **done end**
**program** n := <value>; **while** <value> <= <value> **do** <statement> **done end**
**program** n := <number>; **while** <identifier> <= <number> **do** <assignment> **done end**
**program** n := 1; **while** n <= 10 **do** <identifier> := <expression>; **done end**
**program** n := 1; **while** n <= 10 **do** n := <value> + <value>; **done end**
**program** n := 1; **while** n <= 10 **do** n := <identifier> + <number>; **done end**
**program** n := 1; **while** n <= 10 **do** n := n + 1; **done end**

**Figure 1.11** Producing the program from Figure 1.10.

sort the list $L$ with length $N$:
  case 1: $N < 2$:
    □ do nothing (a list of fewer than two items is already sorted)
  case 2: $N \geq 2$:
    □ split the list in two sublists, $L_1$ with length $N_1$ and
      $L_2$ with length $N_2$, such that $N_1 > 0$ and $N_2 > 0$
      (this can always be done since there are at least two items)
    □ sort the list $L_1$ with length $N_1$ and sort the list $L_2$ with length $N_2$
      (both problems are guaranteed to be easier to solve
      than the original since $N_1 < N$ and $N_2 < N$)
    □ merge $L_1$ and $L_2$, preserving the sorted order.

**Figure 1.12** Decomposing the problem of sorting a list.

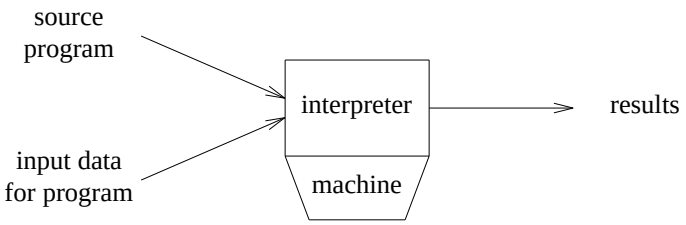


**Figure 1.13** Interpretation.

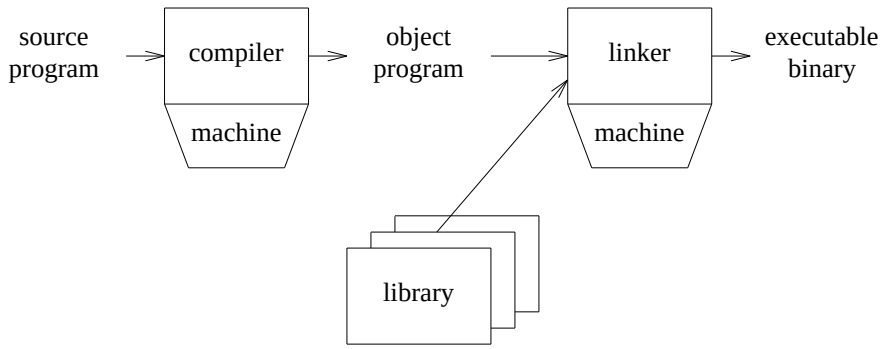


**Figure 1.14** Compilation.

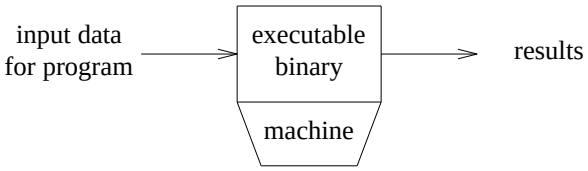


**Figure 1.15** Calling a compiled program.

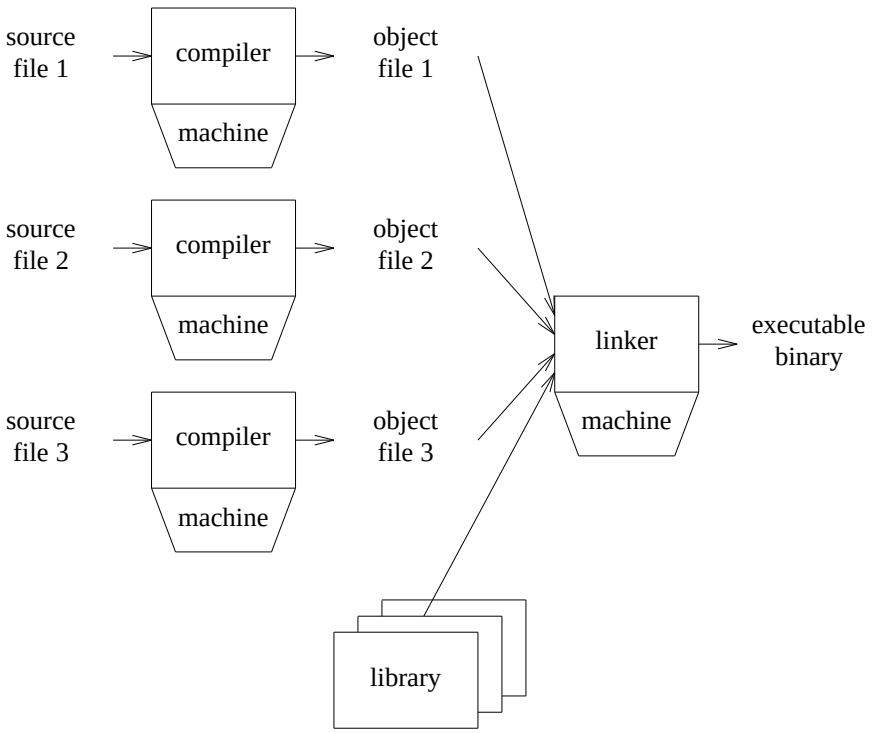


**Figure 1.16** Separate compilation of three source files.

```
#define   NUMBER_OF_TURNS   10

#define   For(var,from,to)   for (var = from; var <= to; var++)
#define   loop               {
#define   end_loop           }
#define   null

For (n, 1, NUMBER_OF_TURNS) loop
  null;
end_loop;
```

**Figure 1.17** Implementing an Ada-like for-loop in C by macro preprocessing.

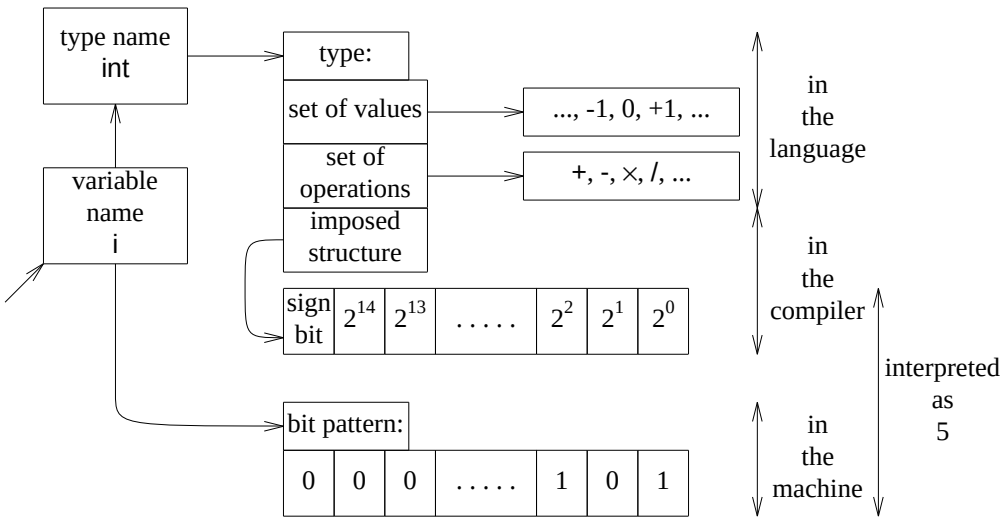


**Figure 2.1** The interrelationships between names, structure and values.

```
                                    type Small_Int is
                                      range 0 .. 255;
char c;                             C: Character;
int i;                              I: Integer;
unsigned int n;                     N: Natural;
float d;   /* a real number */      D: Float;   -- a real number
unsigned char s;                    S: Small_Int;
```

**Figure 2.2** Some declarations involving basic types.

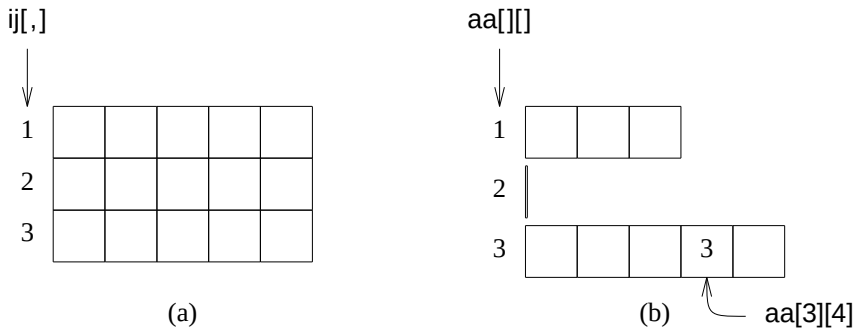


**Figure 2.3** A two-dimensional array (a) and an array of arrays (b).

```
typedef struct {
  brand_type brand;
  unsigned int
    number_of_doors;
  int price;
} car_type;
```

```
type Car_Type is
record
  Brand: Brand_Type;
  Number_Of_Doors:
    Natural;
  Price: Integer;
end record;
```

**Figure 2.4** A record definition describing a car.

```
                                    type Bin_Tree_Type;
                                    type Bin_Tree_Access_Type is
                                      access Bin_Tree_Type;

typedef struct _bin_tree_type {     type Bin_Tree_Type is record
  struct _bin_tree_type               Left, Right:
    *left, *right;                      Bin_Tree_Access_Type;
  item_type value;                    Value: Item_Type;
} bin_tree_type;                    end record;
```

**Figure 2.5** The correct definition for bin_tree_type.

```
int i;                          I: aliased Integer;
                                type Int_Access is access Integer;
int *ip = &i;                   IP: Int_Access := I'Access;

*ip = 7;                        IP.all := 7;
printf("%d\n", i);              Put(I);
```

**Figure 2.6** Implicit assignment through a pointer.

```
int i;                          I: Integer;
int &ir = i;                    IR: Integer renames I;

ir = 7;                         IR := 7;
printf("%d\n", i);              Put(I);
```

**Figure 2.7** Implicit assignment through a reference (alias).

```
                                    type Contents is
                                      (Has_Car, Has_Bicycles);
typedef union {                     type Garage(Status: Contents) is
                                      record
                                        case Status is
                                          when Has_Car =>
  car_type car;                             Car: Car_Type;
                                          when Has_Bicycles =>
  unsigned int nmb_of_bicycles;             Nmb_Of_Bicycles:
                                              Natural;
                                        end case;
} garage;                             end record;
```

**Figure 2.8** A union declaration for a dual-purpose garage.

**Figure 2.9** The assignment S(4..6) := S(3..5);.

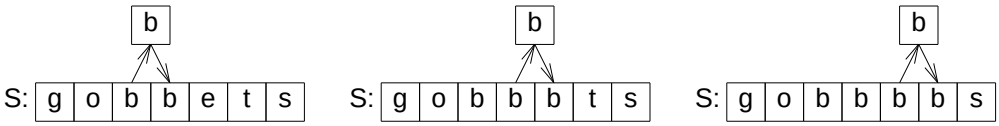


**Figure 2.10** Incorrect interpretation of S(4..6) := S(3..5);.

| Level | Operators | |
|---|---|---|
| 0 | names, literals, parenthesized expressions | |
| 1 | (), [], . | (function call, subscripting, selection) |
| 2 | +, − | (unary plus and minus) |
| 3 | ↑ | (exponentiation) |
| 4 | *, /, mod, div | (standard arithmetic) |
| 5 | +, − | (binary plus and minus) |
| 6 | <, >, <=, >=, =, ≠ | (comparison) |
| 7 | ¬ | (logical NOT) |
| 8 | &, \| | (logical AND and OR) |
| 9 | := | (assignment) |

**Figure 2.11** A simple set of operators with their levels.

| infix | prefix | postfix |
|---|---|---|
| 3 + 4 | + 3 4 | 3 4 + |
| 3 + 4 * 5 | + 3 * 4 5 | 3 4 5 * + |
| (3 + 4) * 5 | * + 3 4 5 | 3 4 + 5 * |

**Figure 2.12** Some expressions in infix, prefix and postfix notations.

```
switch (n) {
case 0:
  printf("empty set");
  break;
case 1:
  printf("singleton");
  break;
default:
  printf("multiple");
  break;
}
```

```
case N is
  when 0 =>
    Put("empty set");

  when 1 =>
    Put("singleton");

  when others =>
    Put("multiple");

end case;
```

**Figure 2.13** A selection statement.

```
case My_Garage.Status is
  when Has_Car =>
    Put(My_Garage.Car);
  when Has_Bicycles =>
    Put(My_Garage.Nmb_Of_Bicycles);
end case;
```

**Figure 2.14** Ada selection statement is similar to a union declaration.

```
int sum;                          Sum: Integer;
int i;                            −− No declaration of I needed

sum = 0;                          Sum := 0;
for (i = 0; i < n; i++) {         for I in 1..N loop
   sum += a[i];                      Sum := Sum + A(I);
}                                 end loop;
```

**Figure 2.15** Using a for-statement to iterate over an array.

```
while (!end_of_file()) {
  read_next_line();
  if (line_is_empty())
    break;

  process_line();
}
```

```
while not End_Of_File loop
  Read_Next_Line;
  if Line_Is_Empty then
    exit;
  end if;
  Process_Line;
end loop;
```

**Figure 2.16** A loop with two continuation conditions.

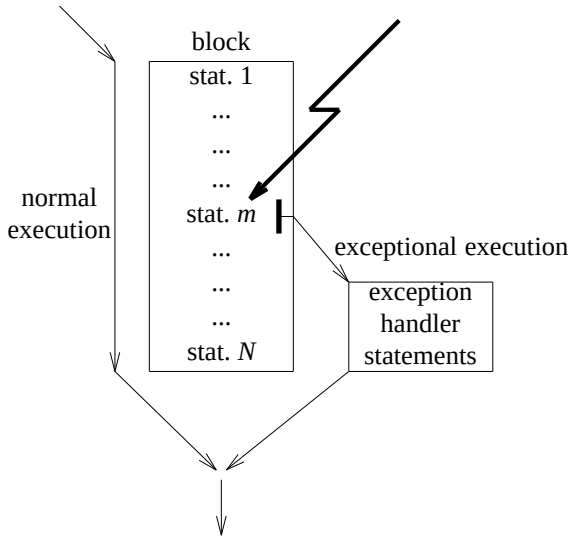


**Figure 2.17** Block with normal and exceptional execution.

```
begin
  Put(1/X);
exception
  when Constraint_Error =>
    Put("Division by zero!");
end;
```

**Figure 2.18** A block with a simple exception handler in Ada.

```
begin
  Put(1/X);
exception
  when EO: Constraint_Error =>
    Put("Division by zero!");
    Put("Details follow.");
    Put(Exception_Message(EO));
end;
```

**Figure 2.19** An exception handler which analyses the exception occurrence.

```
{                                   declare
  int t;                              T: Integer;
                                    begin
  /* swap x and y */                  -- swap x and y
  t = x; x = y; y = t;                T := X; X := Y; Y := T;
}                                   end;
```

**Figure 2.20** The use of a block to declare a temporary variable.

```
{                                   declare
  int x;                              X: Integer;
                                    begin
  ...                                 ...
  {                                   declare
    int y;                              Y: Integer;
                                      begin
    ...                                 ...
    {                                   declare
      int t;                              T: Integer;
                                        begin
      /* swap x and y */                  -- swap x and y
      t = x; x = y;                       T := X; X := Y;
      y = t;                              Y := T;
    }                                   end;
    ...                                 ...
  }                                   end;
}                                   end;
```

**Figure 2.21** Identification in nested blocks.

```
{                                   declare
  int height = 6;                     Height: Integer := 6;
  int width = 7;                      Width: Integer := 7;
  int area =                          Area: Integer :=
      height*width;                       Height*Width;
                                    begin
  ...                                 ...
}                                   end;
```

**Figure 2.22** Immediate use of declared names.

```
void f(void);
void g(void);

void f(void)
{
  ...
  g();      /* use g() */
  ...
}

void g(void)
{
  ...
  f();      /* use f() */
  ...
}
```

```
procedure F;
procedure G;

procedure F is
begin
  ...
  G;       -- use G
  ...
end F;

procedure G is
begin
  ...
  F;       -- use F
  ...
end G;
```

**Figure 2.23** The use of forward declarations.

```
A: {                                  A: declare
     int i, j;                             I, J: Integer;
     typedef struct {                      type Rec_IJ is record
     A           A                         A
       int i, j;                             I, J: Integer;
       A    R
     } rec_ij;                             end record;
       A
     rec_ij s;                             S: Rec_IJ;
     A       A                             A   A
                                         begin
     ...                                   ...
     j = i;                                J := I;
     A   A                                 A    A
     j = s.i;                              J := S.I;
     A   A  R                              A    A   R
   B: {                                  B: declare
   A                                     A
       int i, j;                             I, J: Integer;
       A    B
       rec_ij t;                             T: Rec_IJ;
       B       B                             B   B
                                           begin
       ...                                   ...
       j = i;                                J := I;
       B   B                                 B    B
       j = s.i;                              J := S.I;
       B   B  R                              B    B   R
       j = t.i;                              J := T.I;
       B   B  R                              B    B   R
       ...                                   ...
   }                                       end B;
   ...                                     ...
}                                      end A;
```

**Figure 2.24** Scope and visibility of the is.

```
declare
  -- Cents working on Float values:
  function Cents(V: Float) return Integer is
  begin
    return Integer(V*100.0);
  end Cents;

  -- Cents working on String values:
  function Cents(S: String) return Integer is
  begin
    -- convert a string like "$19.95" to
    -- 1995 and return that value
  end Cents;
  -- the name Cents is overloaded now
begin
  Put(Cents(18.95));
    -- identifies first Cents and prints 1895
  Put(Cents("$8.99"));
    -- identifies second Cents and prints 899
end;
```

**Figure 2.25** Overloading resolution in Ada.

```
int i;                          I: Integer;

test_arg(int n) {               procedure Test_Arg(N: Integer) is
                                begin
   printf("%d", n);               Put(N);
   n = 3;                         N := 3;          -- <<<< error
   printf("%d", n);               Put(N);
}                               end Test_Arg;

i = 1;                          I := 1;
test_arg(i);                    Test_Arg(I);
printf("%d", i);                Put(I);
```

**Figure 2.26** Assigning to a formal parameter in C and Ada.

```
VAR i: INTEGER;          (* declares i of type integer *)

PROCEDURE TestVar(VAR x: INTEGER);
     (* x is a call by reference parameter *)
BEGIN
  WriteInt(i, 1);        (* print i using 1 position *)
  x := 3;
  WriteInt(i, 1);        (* print i again *)
END TestVar;

i := 1;
TestVar(i);
```

**Figure 2.27** An aliasing problem caused by a call by reference parameter.

```
get(int *np) {
  ...
  some code to read an int value
  ...
  *np = the value read;      /* assignment under int pointer np */
}
```

**Figure 2.28** Simulating an output parameter in C.

```
HOW TO PRINT.STOCK book.list:
   PUT 0, 0 IN n.bks, s.val        \ number.of.books, sales.value
   FOR book IN book.list:
      PRINT.ENTRY
   PRINT.TOTAL
PRINT.ENTRY:
   PUT book IN number, title, quantity, price
   WRITE number, quantity, price, title /
   PUT n.bks + quantity, s.val + quantity*price IN n.bks, s.val
PRINT.TOTAL:
   WRITE "We have ", n.bks, " books, worth ", s.val /
```

**Figure 2.29** A HOW TO unit with two refinements in ABC.

```
package Date is

  type Date_Type is private;

  function Create_Date(Y, M, D: Integer)
    return Date_Type;
  function Last_Of_The_Month(Date: Date_Type)
    return Boolean;

private
  type Date_Type is record
    Year, Month, Day: Integer;
  end record;
end Date;
```

**Figure 2.30** Package specification for handling calendar dates in Ada.

```
package body Date is

  Normal_Month_Length: constant array (1..12) of Integer :=
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);

  function Is_Leap_Year(Y: Integer) return Boolean is
  begin
    return ((Y mod 4 = 0) and (Y mod 100 /= 0))
      or (Y mod 400 = 0);
  end Is_Leap_Year;

  function Month_Length(Y, M: Integer)
    return Integer is
  begin
    if M = 2 and Is_Leap_Year(Y) then return 29;
    else return Normal_Month_Length(M);
    end if;
  end Month_Length;

  function Create_Date(Y, M, D: Integer)
    return Date_Type is
  begin
    -- we should check the values of M and D here
    return (Year => Y, Month => M, Day => D);
  end Create_Date;

  function Last_Of_The_Month(Date: Date_Type)
    return Boolean is
  begin
    return Date.Day =
      Month_Length(Date.Year, Date.Month);
  end Last_Of_The_Month;

end Date;
```

**Figure 2.31** Package body for handling calendar dates in Ada.

```
typedef struct {
   int dt_year, dt_month, dt_day;
} date_type;

extern date_type create_date(int, int, int);
extern int last_of_the_month(date_type);
```

**Figure 2.32** Header file date.h for handling calendar dates in C.

```
#include   "date.h"

static int normal_month_length[12] =
  {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

static int is_leap_year(int y)
{
  return ((y % 4 == 0) && (y % 100 != 0))
    || (y % 400 == 0);
}

static int month_length(int y, int m)
{
  if (m == 2 && is_leap_year(y)) return 29;
  else return normal_month_length[m−1];
}

date_type create_date(int y, int m, int d)
{
  date_type dt;

  /* we should check the values of m and d here */
  dt.dt_year = y, dt.dt_month = m, dt.dt_day = d;
  return dt;
}

int last_of_the_month(date_type date)
{
  return date.dt_day ==
    month_length(date.dt_year, date.dt_month);
}
```

**Figure 2.33** Implementation file date.c for handling calendar dates in C.

```
with Text_IO;    -- import package Text_IO
with Date;       -- import package Date
use Date;        -- make the names in it visible

procedure Use_Date is
  Date: Date_Type;
begin
  Date := Create_Date(1992, 2, 29);   -- Feb. 29, 1992
  if Last_Of_The_Month(Date) then
    Text_IO.Put("Pay day!");
  end if;
end Use_Date;
```

**Figure 2.34** Use of the Date package in Ada.

```
#include   <stdio.h>
#include   "date.h"

use_date(void)
{
   date_type date;

   date = create_date(1992, 2, 29); /* Feb. 29, 1992 */
   if (last_of_the_month(date)) {
      printf("Pay day!");
   }
}
```

**Figure 2.35** Use of the date package in C.

```
abstract type Date_Type(Y, M, D: Integer) is
  private Year, Month, Day: Integer;

  -- private definitions of Normal_Month_Length,
  -- Is_Leap_Year and Month_Length from the earlier
  -- package Date

  function Last_Of_The_Month return Boolean is
  begin
    return Day = Month_Length(Year, Month);
  end Last_Of_The_Month;

begin
  -- we should check the values of M and D here
  Year := Y, Month := M, Day := D;
end Date_Type;
```

**Figure 2.36** An abstract data type for Date_Type in pseudo-Ada.

```
generic
  type Item_Type is private;
package Bin_Tree is
  type Bin_Tree_Type is private;
  function New_Bin_Tree return Bin_Tree_Type;
  function Add_Item(T: Bin_Tree_Type; I: Item_Type)
    return Bin_Tree_Type;
  procedure Print_Bin_Tree(T: Bin_Tree_Type);
  -- more Bin_Tree handling subprograms ...
private
  type Bin_Tree_Access_Type is access Bin_Tree_Type;

  type Bin_Tree_Type is record
    Left, Right: Bin_Tree_Access_Type;
    Value: Item_Type;
  end record;
end Bin_Tree;
```

**Figure 2.37** An (incomplete) generic package specification.

```
with Date;        -- import Date package
use Date;         -- make the names in it visible
with Bin_Tree;    -- import Bin_Tree generic package

procedure Test_Bin_Tree is
  Dt: Date_Type;
  package Date_Tree is new Bin_Tree(Date_Type);
  use Date_Tree;
begin
  Dt := Create_Date(1992, 2, 29);
  Print_Bin_Tree(Add_Item(New_Bin_Tree, Dt));
end Test_Bin_Tree;
```

**Figure 2.38** A procedure using the generic package Bin_Tree.

```
procedure qroots(a, b, c)
   local det
   det := b*b − 4*a*c
   if det < 0 then fail
   else {
      suspend (−b + sqrt(det)) / (2*a)
      suspend (−b − sqrt(det)) / (2*a)
   }
end
```

**Figure 2.39** A generator yielding the roots of a quadratic equation.

```
declare
  Magtape_1, Magtape_2: Magtape_Type;
begin
  if Random > 0.5 then raise Numeric_Error; end if;

  Magtape_1 := Grab_Magtape_Unit(1);
  if Random > 0.5 then raise Numeric_Error; end if;

  Magtape_2 := Grab_Magtape_Unit(2);
  if Random > 0.5 then raise Numeric_Error; end if;


  Release_Magtape_Unit(Magtape_2);
  if Random > 0.5 then raise Numeric_Error; end if;

  Release_Magtape_Unit(Magtape_1);
  if Random > 0.5 then raise Numeric_Error; end if;

exception
  when Numeric_Error =>
    null;   -- (dummy statement for syntactic correctness)
    -- insert code here to release all magtape units grabbed
end;
```

**Figure 2.40** Incomplete code for obtaining and releasing access to magnetic tape units.

```
class stack {
  protected:
    int buffer[100];      // variables; invisible to users
    int top;
  public:
    void push(int v);     // operations; visible to users
    int  pop(void);
};
```

**Figure 3.1** Declaration of class stack in C++.

```
// Implementation of the operation pop for class stack
int stack::pop(void)
{
  return buffer[−−top];
}

// Implementation of the operation push for class stack
void stack::push(int v)
{
  buffer[top++] = v;
}
```

**Figure 3.2** Implementation of class stack.

```
// Declaration of class stack
class stack {
  protected:
    int *buffer;
        // space for buffer allocated dynamically
    int top;
    int stack_size;
        // number of elements the stack can contain
  public:
    // constructor:
    stack(int n)
       {stack_size = n;
        top = 0;
        buffer = new int[stack_size];}
    // destructor:
    ~stack(void)
       {delete buffer;}
    void push(int v);
    int  pop(void);
};
```

**Figure 3.3** A declaration of class stack with constructor and destructor operators.

```
main(void)
{
   stack myStack(500);
   // Create stack object with stack size 500. The operation
   // 'myStack.stack(500)' is invoked automatically.

   myStack.push(23);   // use the stack
   ....
   // End of function main(); 'myStack.~stack()' is
   // invoked here automatically.
}
```

**Figure 3.4** A procedure using the class stack.

```
// Declaration of class CalculatorStack
class CalculatorStack : public stack {
  public:
    void add(void);
      // Add two top elements, leave result on the stack
    CalculatorStack(int n): stack(n) {};
      // Constructor just calls stack(n)
};

// Implementation of operation add
void CalculatorStack::add(void)
{
  buffer[top−2] += buffer[top-1];
            // Add top element to second element
  top−−;  // Adjust top
}
```

**Figure 3.5** A class CalculatorStack.

```
CalculatorStack c(500);          // Create object

c.push(2);                       // Push the two numbers
c.push(3);                       // to be added,
c.add();                         // add them,
x = c.pop();                     // pop result, yields 5
```

**Figure 3.6** Using the class CalculatorStack.

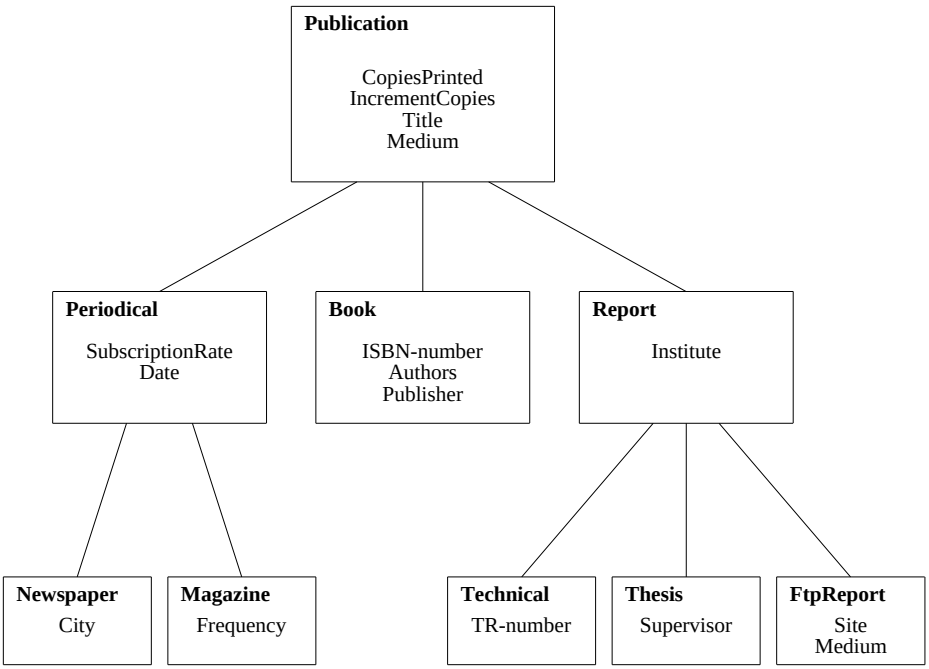


**Figure 3.7** A hierarchy of classes, each defining one or more operations.

```
Report *r;              // r points to object of type Report
Thesis *t;              // t points to object of type Thesis

r = new Report;         // allocate r
t = new Thesis;         // allocate t
sv = t->Supervisor();   // legal operation invocation
r = t;                  // legal assignment
inst = r->Institute();  // legal operation invocation
sv = r->Supervisor();   // illegal, rejected by compiler
```

**Figure 3.8** C++ code illustrating subclassing and subtyping.

```
for (ip = head; ip != 0; ip = ip−>next) {
  Publication *pub = ip−>item;
  FtpReport *ftp;

  switch (pub−>MyClass()) {
    case FTP_REPORT:
      ftp = (FtpReport *) pub;    // cast pub to right type
      printf("%s %s", ftp−>Title(), ftp−>FtpMedium());
      break;
    default:
      printf("%s %s", pub−>Title(), pub−>Medium());
      break;
  }
}
```

**Figure 3.9** Printing a list of publication items using static binding.

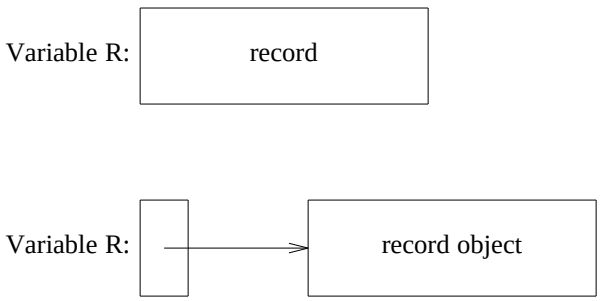


**Figure 3.10** Difference between a variable naming a record (used in imperative languages) and a variable containing a reference to a record object (used in object-oriented languages).

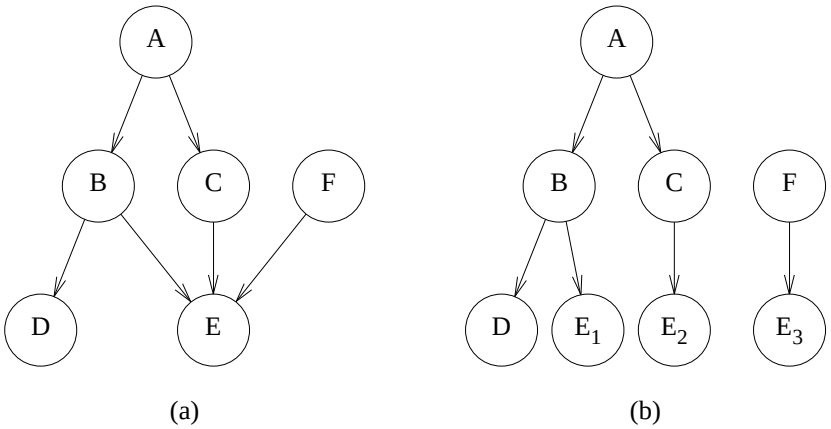


**Figure 3.11** Objects containing references to each other. (a) Object E is shared by three other objects; (b) three objects each contain a private copy of E.

```
class CalculatorStack {
  protected:
    stack *cs;    // invisible outside class CalculatorStack
  public:
    void push(int v)              {cs->push(v);}
    int  pop(void)                {return cs->pop();}
    void add(void)                {cs->push(cs->pop() + cs->pop());}
    CalculatorStack(int n)        {cs = new stack(n);}
};
```

**Figure 3.12** An alternative implementation of class CalculatorStack.

```
class STACK [G] creation make
feature
  push (v: G) is                        -- Push 'v' onto top.
    require                             -- A precondition
      not full
    do
      representation.put (v, last);
      last := last + 1
    ensure                              -- Postconditions
      not empty; last = old last + 1;
      top = v
    end;
  pop is                                -- Remove top element.
    require
      not empty
    do
      last := last -1
    end;
  top: G is                             -- Top element.
    require
      not empty
    do
      Result := representation.item (last);
    end;
  empty: BOOLEAN is                     -- Is the stack without elements?
    do
      Result := (last = 0)
    end;
  full: BOOLEAN is                      -- Is the stack full?
    do
      Result := (last = capacity)
    end
feature                                 -- Initialization
  make (n: INTEGER) is                  -- Create a stack with 'n' elements.
    do
      last := 0; capacity := n;
      !! representation.make (0, n)     -- Create an array
    end
feature {NONE}                          -- Internal variables
  last, capacity: INTEGER;
  representation: ARRAY [G];
end -- class STACK
```

**Figure 3.13** A class stack in Eiffel.

```
function fac(n: integer): integer;
  var i, r: integer;  % r will eventually contain the result
begin
  r := 1;
  % compute n * (n−1) * (n−2) * ... * 1:
  for i := n downto 1 do
    r := r * i;
  return r;
end;
```

**Figure 4.1** Computing a factorial without using recursion.

```
fac 1 ⇒
cond (1 = 0) 1 (1*fac (1−1)) ⇒                          || Apply fac
if (1 = 0) then 1 else(1*fac (1−1)) ⇒                   || Apply cond
if False then 1 else(1*fac (1−1)) ⇒                     || Eval 1=0
1*fac (1−1) ⇒
1*(cond (1−1 = 0) 1 ((1−1) * fac ((1−1)−1))) ⇒          || Apply fac
1*((if 1−1 = 0) then 1 else ((1−1) * fac ((1−1)−1)))    || Apply cond
1*((if True then 1 else ((1−1) * fac ((1−1)−1))))       || Eval 1−1=0
1*1 ⇒
1                                                       || Eval 1*1
```

**Figure 4.2** Lazy evaluation of (fac 1).

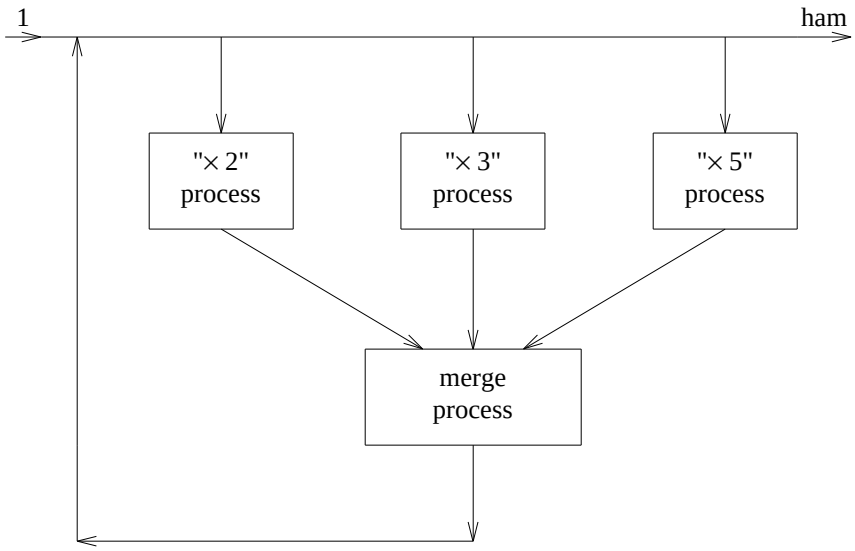


**Figure 4.3** Solving the Hamming problem with pseudo-parallel processes.

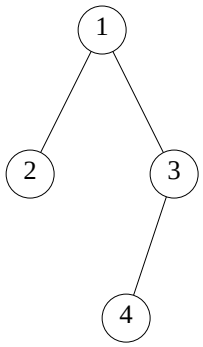


**Figure 4.4** Example tree.

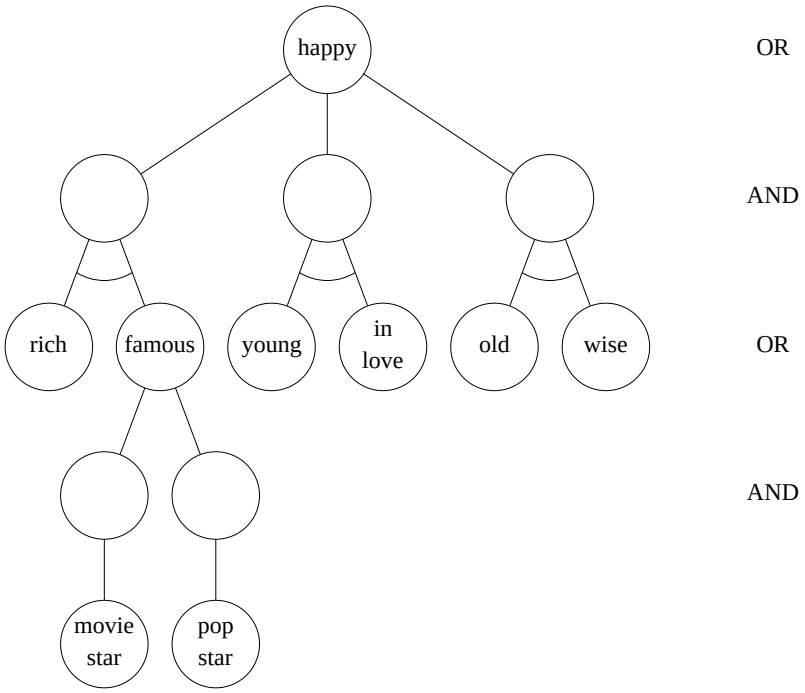


**Figure 5.1** An AND/OR search tree.

```
qsort([], []).
qsort([Pivot| Tail], Sorted) ←
   split(Pivot, Tail, SmallerEqual, Larger),
   qsort(SmallerEqual, SE),
   qsort(Larger, L),
   append(SE, [Pivot|L], Sorted).

split(Pivot, [], [], []).
split(Pivot, [X|Xs], [X|S], L) ←
   X <= Pivot, split(Pivot, Xs, S, L).
split(Pivot, [X|Xs], S, [X|L]) ←
   X > Pivot, split(Pivot, Xs, S, L).
```

**Figure 5.2** A quicksort algorithm.

```
arc(1,2).
arc(1,3).
arc(3,4).
arc(4,2).
arc(4,5).
arc(5,2).
arc(5,6).
```

**Figure 5.3** Representation of a graph.

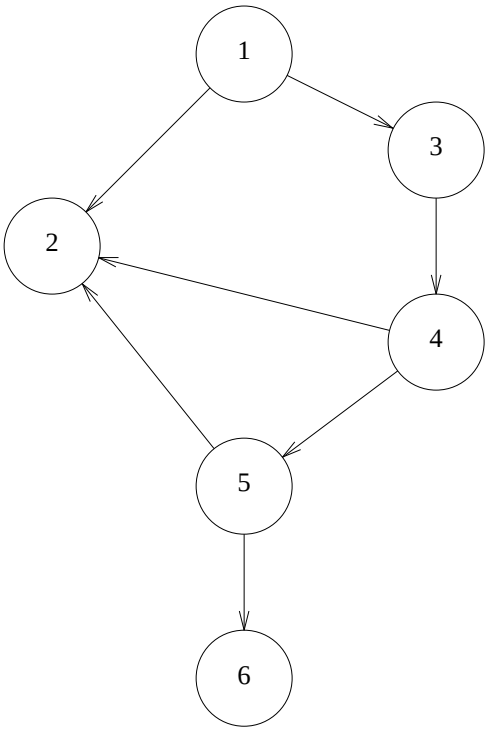


**Figure 5.4** An example directed acyclic graph.

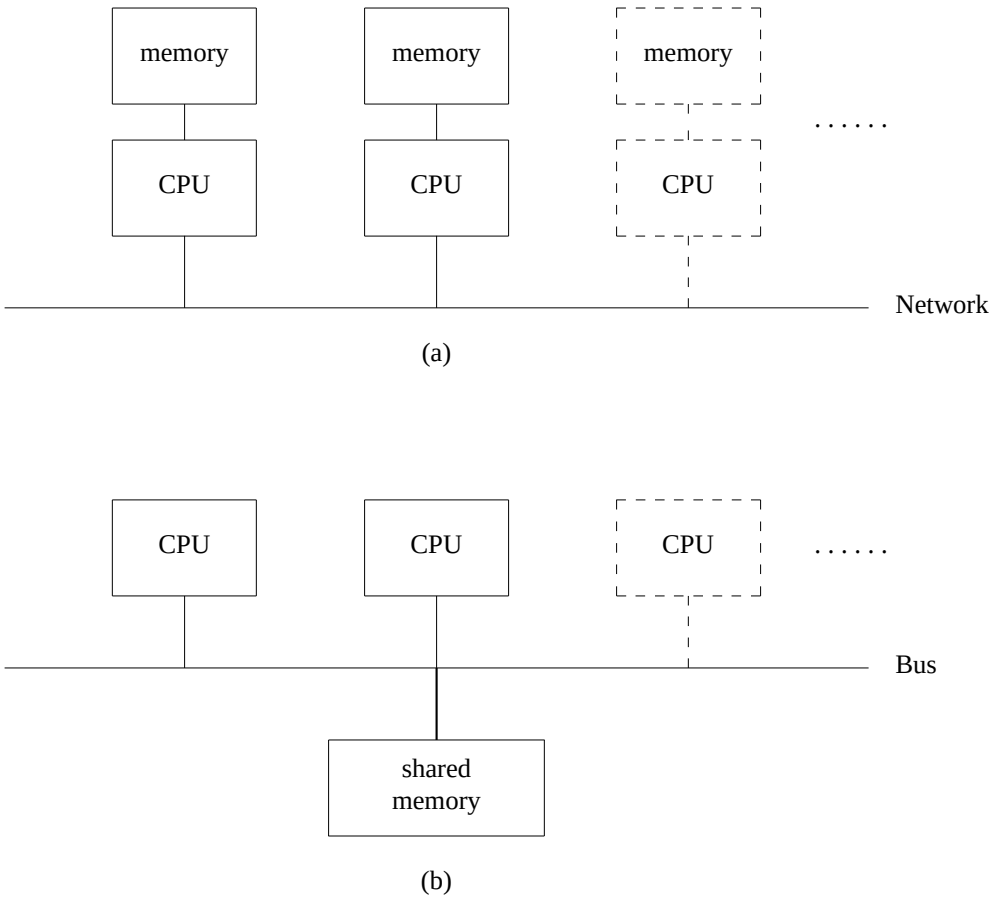


**Figure 6.1** (a) Multicomputer; (b) Shared-memory multiprocessor.

```
procedure Odd:
  var i: integer;
  for i := 1 to N by 2 do
    print i*i;
    resume Even;
  done

procedure Even:
  var j: integer;
  for j := 2 to N by 2 do
    print j*j;
    resume Odd;
  done
```

**Figure 6.2** Example co-routines.

```
A,B,C: array[1..N : 1..N] of real;

parfor i := 1 to N do
  parfor j := 1 to N do
    -- multiply A[i:*] and B[*:j];
    C[i,j] := 0;
    for k := 1 to N do
      C[i,j] := C[i,j] + A[i,k] * B[k,j];
    done
  done
done
```

**Figure 6.3** Parallel matrix multiplication using the **parfor** statement.

```
process compile(sourcefile, targetfile: string);  −− child
begin
  compile source file
  store generated object code in target file
  exit;       −− terminate process
end;

process main(void);  −− parent process
begin
  −− create two parallel processes
  fork compile("file1.c", "file1.o");
  fork compile("file2.c", "file2.o");
  join;       −− wait until all children have finished
  link file1.o and file2.o into an executable program file
end;
```

**Figure 6.4** Compiling two modules file1.c and file2.c in parallel.

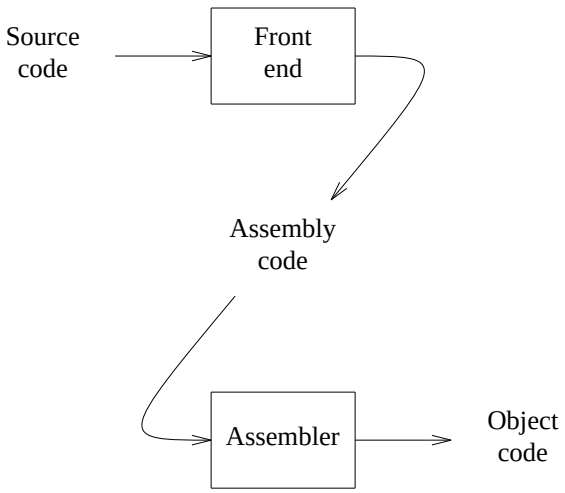


**Figure 6.5** Structure of a parallel compiler.

```
count: integer := 0;                          -- shared variable
buffer: array[0..MAX-1] of line_type;         -- shared buffer

process Front-end
  write_pos: integer := 0;                    -- local variable
  line: line_type;
begin
  while ... do
    generate a line of assembly code
    while count=MAX do done;                  -- busy waiting
    buffer[write_pos] := line;
    write_pos :=
      (write_pos+1) mod MAX;                  -- to wrap around
    count := count+1;
  done;
end;

process Assembler
  read_pos: integer := 0;                     -- local variable
  line: line_type;
begin
  while ... do
    while count=0 do done;                    -- busy waiting
    line := buffer[read_pos];
    read_pos :=
      (read_pos+1) mod MAX;
    count := count-1;
    process a line of assembly code
  done;
end;
```

**Figure 6.6** Code for the front-end and assembler (without mutual exclusion synchronization).

```
monitor buffer;
  −− shared variables encapsulated by the monitor
  count: integer := 0;
  buffer: array[0..MAX−1] of line;
  write_pos, read_pos: integer := 0;
  NotFull, NotEmpty: ConditionVariable;

  operation put(in line: line_type);
  begin
    if count = MAX then wait(NotFull); end if;
    buffer[write_pos] := line;
    write_pos := (write_pos+1) mod MAX;
    count := count + 1;
    if count = 1 then signal(NotEmpty); end if;
  end;

  operation get(out line: line_type);
  begin
    if count = 0 then wait(NotEmpty); end if;
    line := buffer[read_pos];
    read_pos := (read_pos+1) mod MAX;
    count := count − 1;
    if count = MAX−1 then signal(NotFull); end if;
  end;
end;
```

**Figure 6.7** A monitor implementing a shared buffer.

```
process Front-end
begin
  while ... do
    generate line of assembly code
    store line in message
    send message to Assembler
  done;
end;

process Assembler
begin
  while ... do
    receive M from Front-end
    get line from M
    assemble line
  done;
end;
```

**Figure 6.8** Message passing code for the front-end and assembler.

```
-- specification of the buffer process
task type Buffer_Task is
  entry Put(Line: Line_Type);              -- these three entries can
  entry Get(Line: out Line_Type);          -- be called by clients of
  entry Stop;                              -- the buffer task
end Buffer_Task;


-- implementation of the buffer process
task body Buffer_Task is
  Max: constant := 10;                     -- size of the buffer
  Buffer: array(Integer range 0..Max-1) of Line_Type;
                                           -- the buffer itself
  Write_Pos: Integer := 0;                 -- position for writing
  Read_Pos: Integer := 0;                  -- position for reading
  Nr_Lines: Integer := 0;                  -- number of lines in buffer
  Finished: Boolean := False;              -- used for termination
begin
  while not Finished loop
    select
      when Nr_Lines < Max =>               -- put line in buffer
      accept Put(Line: Line_Type) do
        Nr_Lines := Nr_Lines + 1;
        Buffer(Write_Pos) := Line;
        Write_Pos := (Write_Pos + 1) mod Max;
      end Put;
    or
      when Nr_Lines > 0 =>                 -- get line from buffer
      accept Get(Line: out Line_Type) do
        Nr_Lines := Nr_Lines - 1;
        Line := Buffer(Read_Pos);
        Read_Pos := (Read_Pos + 1) mod Max;
      end Get;
    or
      when Nr_Lines = 0 =>                 -- terminate process
      accept Stop do
        Finished := True;
      end Stop;
    end select;
  end loop;
end Buffer_Task;
```

**Figure 6.9** A buffer task in Ada.

```
object implementation IntObject;
  x: integer;            # local data of the object
  ....
  operation Value(): integer;
  begin
    return x;            # return the current value
  end;

  operation Assign(v: integer);
  begin
    x := v;              # assign a new value
  end;

  operation AwaitValue(v: integer);
  begin
    guard x = v do od;
  end;
end;
```

**Figure 6.10** Part of the implementation of object type IntObject.

```
protected body Int_Object is
  function Value return Integer is
  begin
    return X;
  end;

  procedure Assign(V: Integer) is
  begin
    X := V;
  end;

  entry Await_Zero when X = 0 is
  begin
    null;
  end;
end;
```

**Figure 6.11** Body of protected type Int_Object.

| Concept | Primitives | Goal |
|---|---|---|
| **Parallelism** | | |
| Co-routines | **resume** | pseudo-parallelism |
| Parallel statements | **parbegin** statement | true parallelism |
| Parallel loops | **parfor** statement | true parallelism |
| Processes | **fork**, **join** | true parallelism |
| **Shared variables** | | |
| Polling | test variable | condition synchronization |
| Locks | **lock**, **unlock** | mutual exclusion |
| Semaphores | **P**, **V** | mutual excl., cond. synchr. |
| Monitors | **wait**, **signal** | mutual excl., cond. synchr. |
| **Message passing** | | |
| Message passing | **send to**, **receive** [**from**] | 1-way message passing |
| Rendezvous | entry call, **accept** | 2-way message passing |
| RPC | call, shared var. synchr. | 2-way message passing |
| Select statement | **select** | nondeterminism |
| **Mixed models** | | |
| Tuple Space | **read**, **in**, **out** | general communication |
| Shared objects | abstract data types | general communication |

**Figure 6.12** Overview of language primitives for parallel and distributed programming.

| State | | manipulated by ... | | |
|---|---|---|---|---|
| | | commands | patterns | inference |
| accessed ... | as named variables and parameters | imperative | | |
| | by operations | object-oriented | | |
| | as named parameters only | | functional | logic |

**Figure 7.1** A simple classification of paradigms.

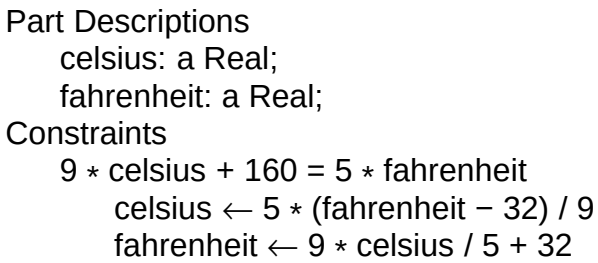


**Figure 7.2** Temperature conversion in ThingLab.

(a) (b)

**Figure 7.3** Two figures satisfying the same overly simple constraint.

```
          T    ⇒   'M'  'A'  'D'  'A'  'M'   ' '  'I'  'M'   ' '  'A'  'D'  'A'  'M'
        ⌽ T    ⇒   'M'  'A'  'D'  'A'   ' '  'M'  'I'   ' '  'M'  'A'  'D'  'A'  'M'
    T = ⌽ T    ⇒    1    1    1    1    0    0    1    0    0    1    1    1    1
∧ / T = ⌽ T    ⇒    1 ∧  1 ∧  1 ∧  1 ∧  0 ∧  0 ∧ 1 ∧  0 ∧  0 ∧  1 ∧  1 ∧  1 ∧  1
               ⇒    0
```

**Figure 7.4** Evaluation of ∧ / T = ⌽ T for T = 'MADAM IM ADAM'.

```
PROGRAM Quest;
  CONST door = { ['left', 'Evil Wizard'], ['right', 'Lost Baby'] };
    $ A constant SETL map with two "door"s in it.

  IF OK THEN move('left'); ELSE move('right'); END IF;
    $ This will always choose correctly and
    $ print 'Lost Baby found behind right door!!'.

  $ Procedure declarations come at the end of a SETL program:
  PROCEDURE move(dir);              $ Move in direction "dir".
    IF door(dir) = 'Lost Baby' THEN
      print('Lost Baby found behind', dir, 'door!!');
    ELSE
      FAIL;                         $ Cause backtrack.
    END IF;
  END PROCEDURE move;

END PROGRAM Quest;
```

**Figure 7.5** Demonstration program for the SETL OK/FAIL mechanism.

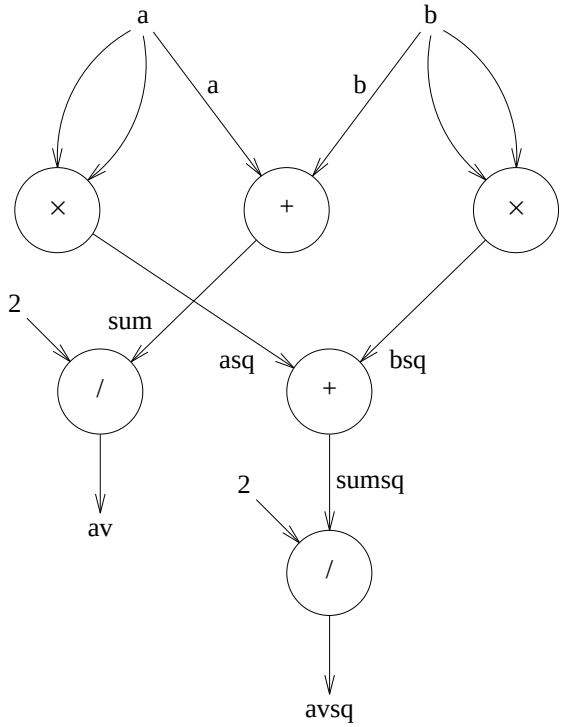


**Figure 7.6** A simple dataflow graph.

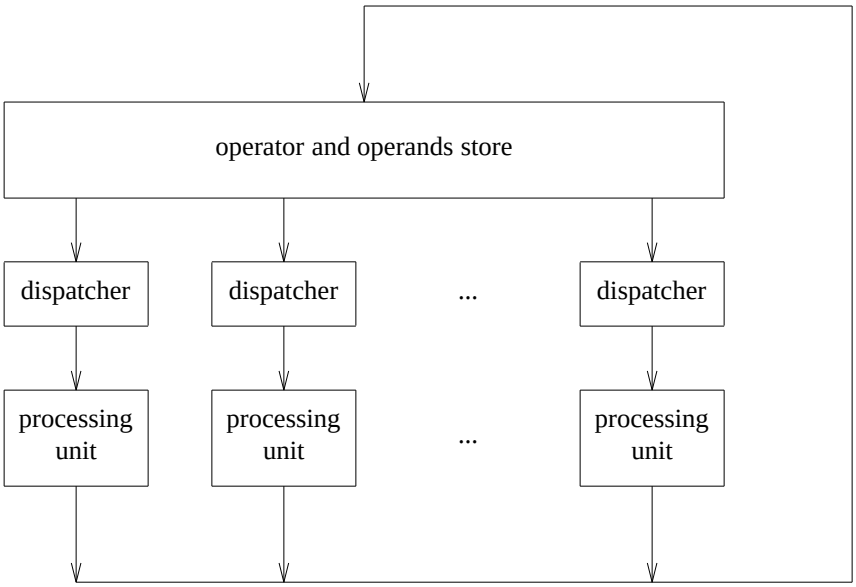


**Figure 7.7** A simplified diagram of a dataflow machine.

```
power :=
  FOR
    i: REAL := n;
    res: REAL := 1;
  DO
    IF i = 0 THEN res ELSE
      ITER i := i − 1; res := res * x; ENDITER
    ENDIF
  ENDFOR
```

**Figure 7.8** A VAL statement to calculate $x^n$.

```
i0 := 3;
res0 := 1;
test0 := i0 = 0;
power :=
  IF test0 THEN res0 ELSE
    ITER i1 := i0 − 1; res1 := res0 * 5; ENDITER
  ENDIF;
```

**Figure 7.9** The first cycle of the FOR loop of Figure 7.8.

```
i1 := i0 − 1;
res1 := res0 * 5;
test1 := i1 = 0;
power :=
  IF test1 THEN res1 ELSE
    ITER i2 := i1 − 1; res2 := res1 * 5; ENDITER
  ENDIF;
```

**Figure 7.10** The second cycle of the FOR loop of Figure 7.8.

```
protected type Buffer_Object is
  entry Put(Line: Line_Type);
  entry Get(Line: out Line_Type);
private
  Max: constant := 10;
  Buffer: array(Integer range 0..Max-1) of Line_Type;
  Write_Pos: Integer := 0;
  Read_Pos: Integer := 0;
  Nr_Lines: Integer := 0;
end Buffer_Object;

protected body Buffer_Object is

  entry Put(Line: Line_Type) when Nr_Lines < Max is
  begin
    Nr_Lines := Nr_Lines + 1;
    Buffer(Write_Pos) := Line;
    Write_Pos := (Write_Pos + 1) mod Max;
  end Put;

  entry Get(Line: out Line_Type) when Nr_Lines > 0 is
  begin
    Nr_Lines := Nr_Lines − 1;
    Line := Buffer(Read_Pos);
    Read_Pos := (Read_Pos + 1) mod Max;
  end Get;
end Buffer_Object;
```

**Figure 7.11** Type Buffer_Object.