Principles of Programming Languages

Lecture 02

Criteria for Language Design

Criteria for Language Design

1. Simplicity

— mnemonic

— clear easily mastered semantics

— as few basic concepts as possible

— feature/concepts limited enough to master entire language (discourages "dialecting")

— effects of feature *combinations* easily predictable

— simple rules of combination

ex:

(1) PL/I: default coercion rules among `fixed bin`, `fixed dec`, `float bin`, `float dec`, when modified by *scale* attributes, become very complex. Each rule is reasonable by itself—the combination yields strange results.

```
dcl M fixed dec(10,5), N fixed bin(5,4);
    N = 0;
    M = N + .1;

expr attr.      repn.                     val.
 .1  dec(1,1)   0.1 (dec)                 1/10
N+.1 bin(5,4)   0.0001|100110011 ..       1/16
```

(binary conversion, then truncation)

```
M    dec(10,5) 00000.06250 (dec)          1/16
```

(2) ALGOL 60:

— **own** static

— **array** dynamic size (known at block entry)

ex: **own boolean array** B[M:N];

— created on first entry to block

— retained between entries to block (with values at block exit)

— seemingly could *vary* in size

— conflicts with stack implementation

— meaning?

(3) PASCAL: fairly simple

(4) ADA:

Entia non sint multiplicanda praeter necessitatem
—William of Ockham

— procedure calls: keyword *or* positional for actual/formal correspondence

— *but* positional parameters must occur first, and once a keyword is used, rest of the call must use keyword parms

```
REORDER_KEYS(NUM_OF_ITEMS,
                      KEY_ARRAY :=: RESULT_TABLE);
```

2. Well-defined Syntactic/Semantic Description

- syntax — not a problem to *specify* once designed
- semantics — big problem; still often informal
  - ALGOL 68 Report (1968)
  - Revised Report on ALGOL 68 (1975)
  - *Informal Introduction to ALGOL 68*, Lindsey & Van der Meulen (1977)
    - Chapter 0: Very Informal Introduction to ALGOL 68
- Techniques
  - Interpretive (operational): PL/I (VDL, Wegner, 1972)
  - Axiomatic: PASCAL (Hoare-Wirth, 1971)
  - Denotational: Scheme (Steele & Sussman, 1978)

ex:

(1) ALGOL 60: Lack of complete syntax and semantics (especially I/O) harmed adoption

(2) PASCAL: **forward** left out of grammar; demands definition before use but makes exceptions for recursive types

(3) PASCAL: When are types equivalent? The *Report* is ambiguous

type *t* = **array**[1..100] **of** *real*;
var *a*, *b* : **array**[1..100] **of** *real*;
c : **array**[1..2] **of** *t*;
d : **array**[1..2] **of** **array**[1..100] **of** *real*;
e : *t*;
f : **array**[1..100] **of** *real*;
g : *t*;

— *structural equivalence*: equivalent if have same type structure

- {*a*, *b*, *e*, *f*, *g*} : **array**[1..100] **of** *real*;
- {*c*, *d*} : **array**[1..2] **of** **array**[1..100] **of** *real*;

— *declaration equivalence*: variables type compatible $\Leftrightarrow$ have same type *name* (built-in or user-defined) *or* appear in the same declaration (ex: PASCAL)

- $\{a, b\}$ (same declaration)
- $\{f\}$ (distinct declaration)
- $\{e, g\}$ (both of type t)
- $\{c\}$, $\{d\}$ (distinct declaration)

— *name equivalence*: variables type compatible $\Leftrightarrow$ have same type *name* (built-in or user-defined) (ex: ADA)

- *only* $\{e, g\}$ (both of type t)
- all others inequivalent

(4) `DO` loops in early FORTRAN: is the index value available after transfer out?

(5) ALGOL 60: Semantics of assignments

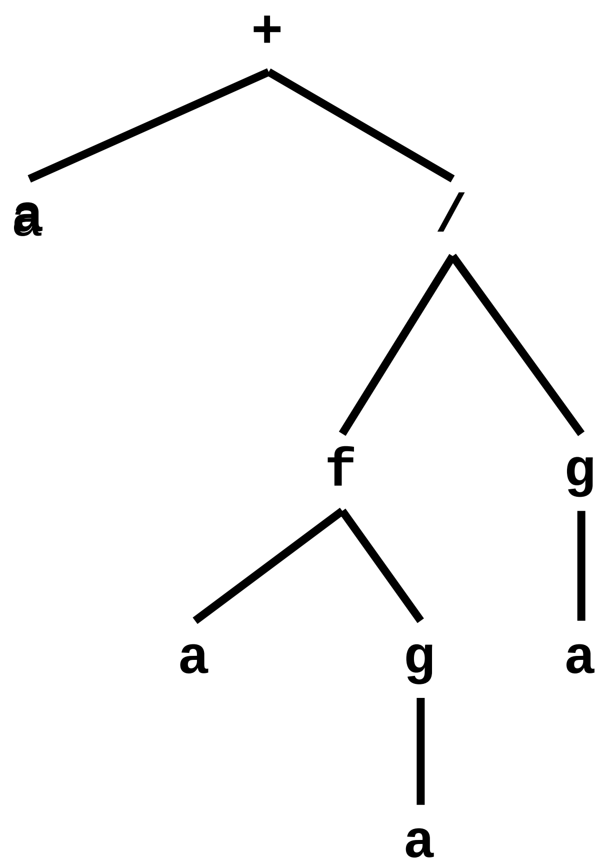
From the *Algol 60 Report*:

> ‘‘4.2.3. Semantics [of assignment statement]
> Assignment statements serve for assigning the value of an expression to one or several variables ... The process will in the general case be understood to take place in three steps as follows:
> 4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.
> 4.2.3.2. The expression of the statement is evaluated.
> 4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.’’

```
begin integer a;
    integer procedure f(x, y);  value y,x;  integer y,x;
        a := f := x + 1;
    integer procedure g(x);  integer x;
        x := g := a + 2;
    a := 0;  outreal(1, a + (f(a,g(a)) / g(a)) )
end
```

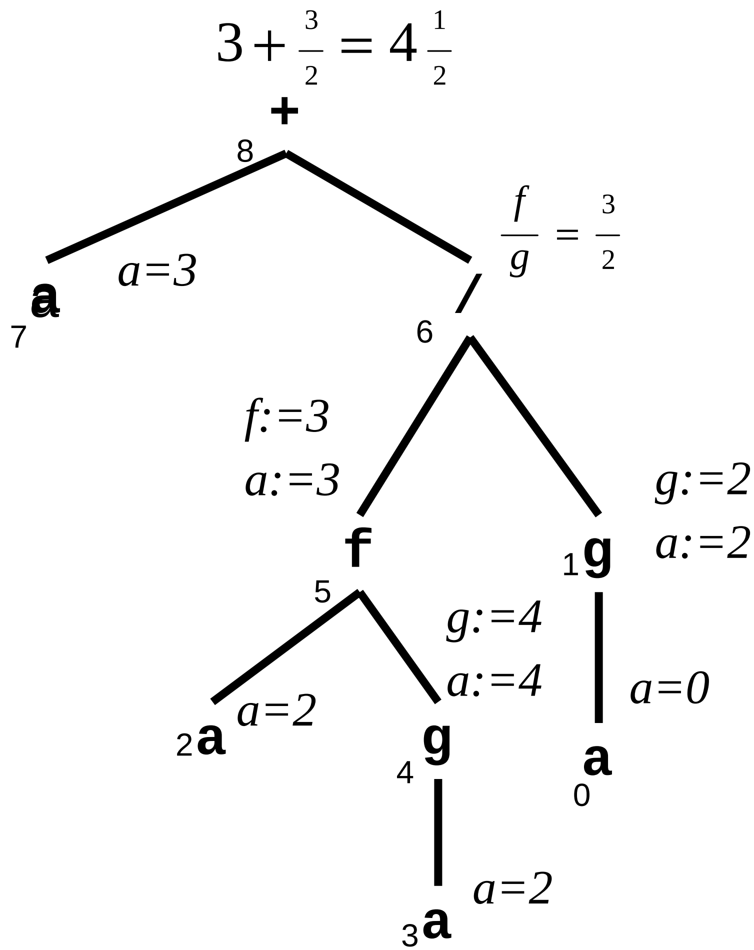
Many possible evaluations of: **`a + ( f(a,g(a)) / g(a) )`**

a + (f(a,g(a)) / g(a))



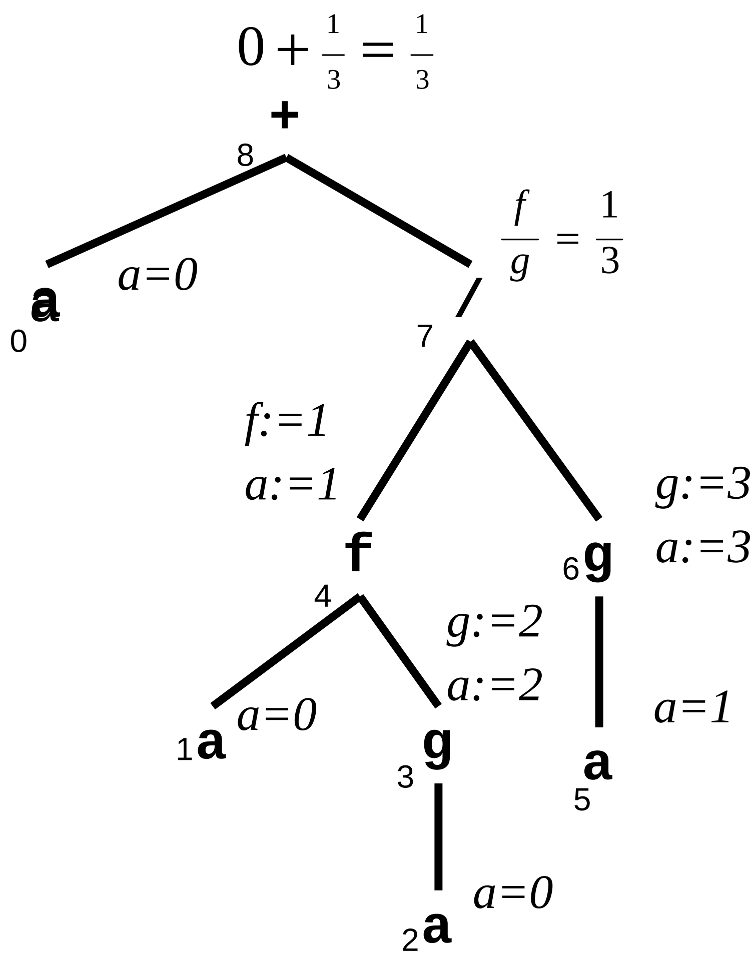

```
g(x) by name:
  x:=g:=a+2

f(x,y) by value:
  a:=f:=x+1

main:
  a:=0;
  print expr
```

(a) Eval denom. 1st, then numer. L-R




```
g(x) by name:
  x:=g:=a+2
f(x,y) by value:
  a:=f:=x+1
 main:
   a:=0;
   print expr
```

(b) Eval standard topological sort (L-R)




```
g(x) by name:
  x:=g:=a+2

f(x,y) by value:
  a:=f:=x+1

 main:
   a:=0;
   print expr
```

(c) Other possible results (11 total)

$$\frac{3}{5} \quad \frac{3}{2} \quad \frac{5}{2} \quad \frac{4}{3} \quad 3\frac{3}{5} \quad 3\frac{1}{3} \quad 5\frac{3}{5} \quad 3\frac{1}{2} \quad 7\frac{1}{2}$$

Another Example:

- Bad:

```
A[ a + B[f(a)] + g(a) ] := C[a] := 0;
```

- Algol examples from D.E. Knuth, ``The Remaining Trouble Spots in ALGOL 60'', *CACM 1967.*

(6) C, C++: Semantics of expressions & assignments

```
a = f(i); a += g(i);
```

```
a = f(i) + g(i);
```

NOT equivalent. Result of one is known; result of the other is unknown.

(7) C++: There are 6 *different* kinds of scope:

— block (local) scope: names declared in a block & formal parameters defined in any enclosed block/function (except if redeclared)

— function scope: labels defined throughout function in which declared

— function prototype scope: names in the parameter list extend to the end of prototype definition

— file (global) scope: name declared outside all blocks/classes is defined throughout the translation unit

— class scope: class member name `n` is local to its class `C` and can be used only

(1) as `n` in another member function of class `C`

(2) as `c.n` where `c` is an object of class `C` (or a derived class of `C`)

(3) as `pc->n` where `pc` is a pointer to an object of class `C` (or a derived class of `C`)

(4) as `X::n` (scope resolution operator) where *X* is *C* (or a derived class of `C`)

— namespace scope: names defined only *inside* the namespace (a named collection of classes) or of*outside* using scope resolution operator `::` (namespaces support multivendor libraries with potential name conflicts)

```
namespace std {
   #include <iostream.h>
   #include <math.h>
}
namespace SOFTinc {
   class stack ( ... };
   class queue { ... };
}
    // here use std::<<
using namespace std;
    // here use << for std::<<
using namespace SOFTinc;
    // here use stack for SOFTinc::stack
```

3. Reliability/Safety

— Reliability: syntactic errors not easy to introduce and *discouraged* (related to *readability*)

— Safety: semantic errors detectable, preferably at compile time

ex:

(1) ALGOL 60: missing **;** after **comment** absorbs next statement ⇒ strange errors

(2) PL/I:

```
FOO: PROC OPTIONS(MAIN);
 DCL ...
 ON ENDFILE(SYSIN) BEGIN;
                 .
                 .
 (forgot END;)
                 .
                 .
 DO I=1 BY 1;
                 .
                 .
 (forgot END;)
END FOO;
```

Last `END` creates end for `FOO` and *any other enclosed structures with forgotton* `END`*s—with no warning!*

(3) PASCAL: **;** is a *separator*, not a *terminator* as in PL/I, ADA

PASCAL (separator bad):

```
if  C    then   S1
         else   S2 ;
                S3 ;
                {error without  begin  S2 ;  S3  end ; }
```

ADA (terminator good):

```
if  C    then   S1
         else   S2 ;
                S3 ;      (* ok *)
end  if  ;
```

- [Ripley & Druseikis, *Computer Languages 3* (1978) 227-240] [Gannon, *CACM 20* (1977) 584-595]
  - 20% of all PASCAL syntax errors are **;** terminator errors
  - 7.5% are missing **begin** , **end**

(4) Use unique delimiters for different constructs

ADA:

if	**end if** ;
for	**end loop;**
case	**end case;**

ALGOL 68:

do	**od**
if	**fi**
case	**esac**

?!:

comment	**tnemmoc**

(5) PASCAL *variant records*: type compatibility cannot be checked at compile time

```
type scale = (large, huge) ;
     measurement =
          record
               unit : string ;
               case size : scale of
                    large : (exact : integer) ;
                    huge  : (approx : real)
               end ;
var dist : measurement ;
```

unit →	|________|
size →	|________|(tag field)
exact, *approx* →	|________|

(∗) *dist.exact* is meaningful ⇔ *dist.size* = *large*
dist.approx is meaningful ⇔ *dist.size* = *huge*

— Programmer can alter *dist.size*

— Cannot enforce (∗) at compile time

— To enforce at runtime *every* reference to a variant field must check the tag field value (*large* or *huge*). Never done in practice.

(6) C, C++: `==` vs. `=`

```
if (n = 0)
    exception();
else
    norm();
```

(7) C, C++: Implicit pointer conversions

```
char* mem;
void* gen_ptr;

gen_ptr = mem;  // C and C++
mem = gen_ptr;  // assignt. compatible in C
                // illegal in C++
```

(8) Comments in C++

```
// FILE: rat.h
// ...
  class Rat
  {
  public:
     // CONSTRUCTORS
     Rat(int n = 0, int d = 1);   // default constructor: Rat r
                           // Rat(5) ==> 5/1
     Rat(const Rat& r);              // copy constructor
     Rat(double d);                   // init by a double value
     // DESTRUCTOR
     ~Rat(){ // nothing to do }
```

```
    // SELECTORS &  CONST MEMBER FUNCTIONS
    int numer() const { return num; }
    double decimal() const { return double(num)/double(den); }
    int denom() const { return den; }
    int sign()  const { if (num < 0) return -1; else return 1
    // ...
  private:
    int num;            // numerator
    int den;            // denominator > 0
  };

  // NONMEMBER functions for Rat Class
  Rat operator +(const Rat& x, const Rat& y);
  Rat operator *(const Rat& x, const Rat& y);
```

```
opu> make
g++ -Wall -c -g testrat.cxx
In file included from testrat.cxx:7:
rat.h:58: ‘Rat::operator +(const Rat &, const Rat &)’
      must take either zero or one argument
rat.h:59: ‘Rat::operator (unary *)(const Rat &, const Rat &)’
    must take either zero or one argument
testrat.cxx:79: parse error at end of input
*** Exit 1
Stop.
opu>
```

— *Why?*

Fixed as follows: In `rat.h` file, change `DESTRUCTOR` definition to:

```
        // DESTRUCTOR
        ~Rat(){ // nothing to do
              }
opu>
```

— Now it compiles!

(9) Static type checking vs. Run-time type checking

- ML: static

```
opu> sml
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit

- fun stringadd x = x + "astring";
std_in:2.21 Error: overloaded variable not defined at type
   symbol: +
   type: string

- fun S(x) = x;
val S = fn : 'a -> 'a

- (S 3);
val it = 3 : int

- fun S(x) = (x x);
std_in:6.12-6.16 Error: operator is not a function
   operator: 'Z in expression: x x
- fun S f = (fn x => f ( x x ));
std_in:0.0-0.0 Error: operator is not a function
   operator: 'Z in expression: x x
- ^D
opu>
```

- Scheme: dynamic

```
opu> scheme
Scheme Microcode Version 10.2
MIT Scheme, unix [bsd (unknown)] version
^AH (CTRL-A, then H) shows help on interrupt keys.

Scheme saved on ...
   Release 6.1.2
   Microcode 10.2
   Runtime 13.91
   SF 3.13

1 ]=> ; scheme not strongly typed
        (define stringadd
    (lambda (x)
         (+ x "astring")    ))
STRINGADD

1 ]=> (stringadd 5)
Illegal datum in second argument position "astring"
within procedure #[PRIMITIVE-PROCEDURE &+]
There is no environment available;
using the current read-eval-print environment.

2 Error-> ^G
Quit!
```

```
1 ]=> ; self-apply functional
        (define S
    (lambda (x) (x x)) )
S

1 ]=> (S 1)
Application of Non-Procedure Object 1
There is no environment available;
using the current read-eval-print environment.

2 Error-> ^G
Quit!

1 ]=> (define omega
    (lambda ()
        (S S))   )
OMEGA

1 ]=> (omega)
^G
Quit!
;;;;;;;  above interrupt occurred after a long processing loop
;;;;;;;  next we see if S behaves like an honest "self-apply"
```

```
1 ]=> (define id
    (lambda (x)   x ) )
ID

1 ]=> (id S)
#[COMPOUND-PROCEDURE #x13EDEC]

1 ]=> (S id)
#[COMPOUND-PROCEDURE #x19FE4C]

1 ]=> ( (S id) 10)
10

1 ]=> ( (id S) 10)
Application of Non-Procedure Object 10
There is no environment available;
using the current read-eval-print environment.

2 Error-> ^G
Quit!

1 ]=> (%exit)
Moriturus te saluto.
opu>
```

4. Fast Translation

— simple syntax ⇒ simple parser

— LL(1) or LR(1) preferable

— "one-pass" compilation

ex:

(1) PL/I: 10 passes over source file

(2) ADA: types, modes and names of parameters, together with the result type of a procedure, are used to resolve procedure references—leads to complex "static semantic analysis".

(a) `F(X)` might be

- subprogram call: `F` subprogram, `X` parm
- array reference: `F` array variable, `X` index
- conversion: `F` type name, `X` expression

(b) *overloaded* operators ("*ad hoc* polymorphism")

```
function "+"(X, Y: VECTOR) return VECTOR;
function "+"(X, Y: MATRIX) return MATRIX;
```

specific operator identified by the operand types and return types

5. Efficient Object Code

- Small changes in usage should not result in huge changes in execution time
- Encouraged by early binding of information at compile-time and few run-time checks (e.g., *strong typing*)

ex:

(1) FORTRAN: very efficient

(2) ALGOL 60 **for** loop:

for $i := e_1$ **step** e_2 **until** e_3 **do** $\cdots$;

re-evaluates the step and bound expressions e_2, e_3 on *every* iteration; this penalizes most loops.

(3) SMALLTALK: elaborate bytecode to resolve `b + c` which abbreviates `b` **`add:`** `c`.

`c` might be string, vector, bignum, integer, ... Method to be used is determined by the message receiver object, using ‘‘dispatching’’ algorithm that searches dictionaries in `b`s instance dictionary, then class dictionary, then superclass dictionary, ... (*deferred binding* at run-time, for each message)

(4) C++: resolution of plus method in `b + c` made at compile time using known type of operand `c` and static code for `operator+` in known class (i.e., type) declaration of `b`. Efficient object code generated; no code for a run-time dictionary search.

6. Orthogonality

A language is *orthogonal* to the extent that one can separate it into elements that can be defined independently

— *regularity:* any combination of primitives should be allowed: no *ad hoc* restrictions on the use of certain constructs in certain places

— programmer should be able to infer legal constructs by generalization from instances

— independent functions should be controlled by independent language mechanisms

ex:

(1) Children's English: "I runned the program."
Actually "-*ed*" is not a true orthogonal *operator* in English because of verb irregularity

(2) FORTRAN (early):

(a) could initialize a constant with `'FOO'`, but could *not* assign string constants: `X = 'FOO'` illegal

(b) lacked relational operators (`<`, `=`) for strings

(c) expressions in array subscripts had a limited syntax: `n*I + m`, `n` and `m` integer constants

(3) PASCAL:

— components of a **packed array** cannot be passed by reference (**var**)

— **procedure**s, **function**s passed as parameters (actuals) cannot themselves have reference (**var**) parameters

— **function**s cannot return **array**, **record**, **set**, **file**, **function**

(4) ALGOL 60: the first language to focus on orthogonality

$X \ := \ Y \ + \ (\ \mathbf{if} \ A = B \ \mathbf{then} \ A + 1 \ \mathbf{else} \ A \ ) * B \ ;$
$\mathbf{goto} \ \ \mathbf{if} \ X = Y \ \mathbf{then} \ L1 \ \mathbf{else} \ L2 \ ;$

(5) ALGOL 68: highly orthogonal

— expressions on LHS as well as RHS of assignments

(**int** *home*, *away* ; *read*((*home*, *away*));
if *home* > *away* **then** *won*
elif *home* = *away* **then** *tied*
else *lost* **fi**)
+:= 1

— functions can return functions, etc.

(6) ADA: exponentiation `**` not defined for "fixed point" type

(7) ADA & PASCAL: **array** declarations can be given explicity without first naming a type. For any other kind of non-atomic type, one must use a *pre-defined* type. E.g., in ADA:

```
A : array(1..5) of CHARACTER;--legal
DAY : (M, T, W, TH, F);    --illegal
type WORKDAY is (M, T, W, TH, F);
DAY : WORKDAY;                 --legal
```

(8) PASCAL: size of an array is part of its type

⇒ writing a general sort routine is difficult (re-usable code discouraged)

⇒ an error routine must pad all error messages to the same length (since strings are stored as **packed arrays**)

⇒ files or pointers (heap allocation) must be used when size of a storage structure cannot be estimated before execution

(9) C: tries valiantly to integrate pointer manipulation with arithmetic, and arrays with pointers. However:

— ampersand is not a true operator:

```
int x;  int *px;
px = &x;               /* px gets addr of x */
px++;
&(x+1);                /* illegal */
&3                     /* illegal */
register x;
px = &x;               /* illegal */
```

— pointer arithmetic and array indexing are close, but enough different to be confusing:

```
int a[10];             /* array  a */
int *pa;               /* pointer pa */
a[i]  ≡  *(a+i)
pa[i]  ≡  *(pa+i)
pa = a;                /*sensible since pa*/
pa++;                  /*is a variable */
```

but . . .

```
a = pa;  /* illegal */
a++;      /* illegal */
p = &a; /*all illegal since a is a const */
```

Why do you seldom make these errors? Luck of style: array names are mostly actual *arguments* bound to formal *variables* (locals) in function calls. So *inside* the function body, an array parmeter with formal name `a` would be a local *variable*, and allow `a++` to be legal.

(10) C++: All functions have parameter types and result types—*except for class constructors and destructors*.

```
// FILE: stack1.h
#include <stdlib.h> // Provides size_t
#include <assert.h>   // Provides assert

    template <class Item>
    class Stack
    {
    public:
        // MEMBER CONSTANTS
        enum { CAPACITY = 64 };
        // CONSTRUCTOR
        Stack( ) { used = 0; }
        // DESTRUCTOR
        ~Stack( ) {     }
        // MODIFICATION functions
        void push(const Item& entry);
        Item pop( );
```

```
        // CONSTANT functions
        size_t size( ) const { return used; }
        bool is_empty( ) const { return used
        Item peek( ) const;
    private:
        Item data[CAPACITY];   // bottom 0
        size_t used;
    };
```

Q: Why doesn't `Stack()` return an object of the class `Stack`?

A: Stroustrup wanted (run-time) declarations to look like C declarations:

```
Stack<int>   operand, offset;
Stack<char>   operator;
```

7. Language Objects First-Class

Defn: [Stoy, *Denotational Semantics*, MIT, 1989, p. 39]. An object is *first-class* if it can be

— returned as the result of a function call

— returned as the result of an expression evaluation

— assigned as the value of a variable

— entered into an array

— selected by a conditional expression

— passed as a parameter

(1) ALGOL 60: only first-class objects were values of type **real**, **integer** and **boolean**. **Procedure**s were really constants; could be called or passed as parameters only. There were **label** variables, but procedures could not return **label**s.

(2) LISP: often called a "functional language", but in classical LISP *functions are not first-class objects*:

```
e1   (+ 3 4)          ; an expr, evals to int
e2   (lambda (x y) (+ x y))
  ; does not eval to a function
  ; only has meaning applied to args

(e2 3 4)  ⇒  7

opu> kcl
AKCL (Austin Kyoto Common Lisp)  Version(1.505)

>(  (lambda (x y) (+ x y)) 7 5)
12

>(lambda (x y) (+ x y))

Error: The function LAMBDA is undefined.
Error signalled by EVAL.
Broken at EVAL.  Type :H for Help.
>>:r
Top level.
>(defun add (x y) (+ x y))
ADD

>(add 7 5)
12
```

```
>(defun ident(x) x)
IDENT

>(ident  3)
3

>(ident add)

Error: The variable ADD is unbound.
Error signalled by EVAL.
Broken at EVAL.  Type :H for Help.
>>:r
Top level.

>((ident add) 7 5)

Error: (IDENT ADD) is invalid as a function.
Error signalled by EVAL.
Broken at EVAL.  Type :H for Help.
>>:r
Top level.
>ˆDBye.
```

(3) ML ("Meta Language"): all functions are first-class;

```
opu> sml
Standard ML of New Jersey, Version 0.66
val it = () : unit
- 3+4;
val it = 7 : int
- fun add(x,y):int = (x:int) + (y:int);
val add = fn : int * int -> int
- add(7,5);
val it = 12 : int
- fun ident(x) = x;
val ident = fn : ’a -> ’a
- ident(add);
val it = fn : int * int -> int
- ident(add)(7,5);
val it = 12 : int
```

(4) Scheme functions are first class

```
opu> scheme
Scheme Microcode Version 10.2
MIT Scheme, unix [bsd (unknown)] version
^AH (CTRL-A, then H) shows help on interrupt keys.

1 ]=> (define add
    (lambda (x y)
        (+ x y) ))
ADD

1 ]=> (add 7 5)
12

1 ]=> (define ident
    (lambda (x)
        x    ))
IDENT

1 ]=> (ident add)
#[COMPOUND-PROCEDURE #x1691E0]

1 ]=> ( (ident add) 7 5 )
12

1 ]=> (%exit)
Moriturus te saluto.
```

(5) The equivalent example in Common Lisp requires computation of function *closures* like `#(quote ident)`

```
opu> kcl
AKCL (Austin Kyoto Common Lisp)  Version(1.505) Fri Dec 14 19

>(defun add (x y) (+ x y))
ADD

>(defun ident (x) x)
IDENT

>(funcall #'ident 3)
3

;;;;;  fun name != fun val (closure)
>(funcall #'ident add)

Error: The variable ADD is unbound.
Error signalled by EVAL.
Broken at EVAL.  Type :H for Help.
>>:r
Top level.
```

```
>(funcall #'ident #'add)
(LAMBDA-BLOCK ADD (X Y) (+ X Y))

;;;;;; looks like a functional argument but isn't

>( (funcall #'ident #'add) 7 5)

Error: (FUNCALL #'IDENT #'ADD) is invalid as a function.
Error signalled by EVAL.
Broken at EVAL.   Type :H for Help.
>>:r
Top level.

>(funcall (funcall #'ident #'add) 7 5)
12

>^DBye.
opu>
```

8. Transparent Data Types

— **Defn:** A data type is *transparent* when all values of that type can be represented as literals within the language.

— related to being first-class

(1) APL: arrays are both first-class objects and transparent

```
S ← 1 2 3 4 5
S ← ι5
```

(2) ALGOL 68: arrays are both first-class objects and transparent

[1 : 3, 1 : 2] **int** *A* := ((1, 2), (0, 0), (3, 4));

(row-major 3×2 array representation)

(3) ALGOL 68: structs are transparent:

struct (**string** *unit*, **int** *value*) *X* := *("lumens", 12);*

...

X := *("btu", 4);*

(4) PASCAL: **array**s are not transparent; no way to initialize or assign

(5) PASCAL: **set**s are transparent (except for limits on set size): *X* := [′*A*′..′*M*′]; *Y* := [2..12] *;* but *set* lacks orthogonality since **set of record** is disallowed.

9. Generality/Abstraction

- — all features should be composed of a few basic concepts
- — "natural" generalization of disparate concepts should be found

ex:

(1) PASCAL: enumeration types all support **pred**, **succ**, =, <, etc.

(2) PASCAL: **for** takes scalar types, sets

(3) ALGOL 68: **for from by to while do** · · · **od**. PL/I also has extremely generalized loop structure

(4) C almost unifies arrays and pointers

(5) Early C: structures could not be assigned to, copied, passed to or returned from functions

(6) APL: `A + B` for vectors, arrays; even `3 + A` defined for arrays

(7) PASCAL: dimensions of an array are part of its type:

function *sum*(*x* : **array**[1..100] **of** *real*) : *real* ;
. . .
end{*sum*} ;

But cannot be *re-used* on the argument
bigvec : **array**[1..100000] **of** *real* ;

(8) ALGOL 60: **begin** $\cdots$ **end** brackets do double, unrelated duty.

(a) They group statements into a compound statement

(b) They open and close *blocks*, thus manipulating the environment:

n := 10 ;
begin *integer* *n* ;
n := 3 ;
. . .
end

10. Machine Independence and Portability

— character set part of language definition

— language acknowledges variant machine arithmetics, word sizes, representation limits

ex:

(1) ADA: "representation attributes" make important implementation-dependent characteristics accessible to programmer; this makes it possible to program for version generators

```
INTEGER'FIRST          INTEGER'LAST
X'ADDRESS              X'SIZE
T'MACHINE_ROUNDS       T'MACHINE_EMAX
```

(2) PASCAL: unfortunate historical limit on size of sets CDC 6000 PASCAL: scalar types limited to ≤ 59 elements; subranges could only be indexed inside [0..58].

(3) **`f77`** *et. seq.* has standardized character set; earlier FORTRANS did not (e.g., code was uncompilable due to identifies like `RATE_OF_PAY`)

(4) FORTRAN (early): could EQUIVALENCE a logical and real quantity, if knew the representation of reals

(5) C++: `short` and `int` could be implemented the same. `float`, `double`, `long` could be implemented the same. `char` might be more than 8 bits. All these are ANSI "spec".

(6) ADA: the package `SYSTEM` draws together implementation-dependent language characteristics

```
package SYSTEM is
  type ADDRESS is implementation defined;
  type NAME      is implementation defined enumeration type;

  SYSTEM_NAME : constant NAME := implementation defined;
  STORAGE_UNIT: constant := implementation defined;
  MEMORY_SIZE : constant := implementation defined;
  --  System Dependent Named Numbers;
  MIN_INT     : constant := implementation defined;
  MAX_INT     : constant := implementation defined;
  TICK        : constant := implementation defined;
        ...
  -- Other System Dependent Declarations
  subtype PRIORITY is INTEGER range implementation defined;
        ...
end SYSTEM;
```

11. Verifiability

- — language should make formal verification of program correctness easy
- — need an axiomatic/denotational description of all semantics
- — far from achieved

ex:

(1) PL/C (Cornell PL/I): **assert** feature was a "compilable comment"

(2) ADA: (Strawman 1975) Design was to be "amenable to verification of correctness".

(3) C++: `assert( used < CAP && 0 <= top <= used);` violation causes exception to be thrown at run-time. If user provides no handler, a run-time error message is printed, and program exits.

12. Consistency with Familiar Notations

— respect common expectations regarding established notation

— encouraged by *overloading*: the ability to define multiple meanings for the same syntax; resolved by context

ex:

(1) PASCAL: +: **integer**, **real**, **boolean**
<: any enumeration type

(2) ADA: **`package`** `textio`

```
GET( ITEM : out  CHARACTER );
                 STRING
                 INTEGER
                 FLOAT
                 FIXED
                 BOOLEAN
```

enumeration types

(3) COBOL: `ADD B TO C GIVING A`

(4) Smalltalk:

`x + 2*y` $\equiv$ `((x plus: 2) times: y)`

x sent $+2$ message; resulting object is sent $*y$ message; resulting object has value $(x + 2) \cdot y$.

(5) C++: flexible operator overloading for any class

```
bool operator==(const Table& x,
                const Table& y);
```

output insertion operator generalizes:

```
ostream& operator<<(ostream& outstream,
                const Table& source);
```

as does *input extraction* operator `>>`.
Each new object class can overload these with appropriate ‘‘output packing’’ and ‘‘input unpacking’’ code.

(6) C++ notation for *casting*:

```
x = (double) i;    // C
x = double(i);     // C++
x = static_cast<double>(i)// ANSI C++
```

(latter uses notation for *template arguments* in C++)

13. Uniformity

— similar things should have similar meanings

≡ dissimilar semantics ⇒ dissimilar appearance

— ‘‘semantic differences should appear as syntactic differences’’

— basic constructs should be applied consistently and universally

— "semantic similarities should appear as syntactic similarities"

(1) PASCAL: control-flow **case** ≠ variant record **case**

case *expression* **of**
 $value_1$: $stmt_1$;
 ⋮
 $value_n$: $stmt_n$;
end

record *<fixed part>;*
case *ident: typeid* **of**
 $value_1$: ($fields_1$) ;
 ⋮
 $value_n$: ($fields_n$) ;
end

case *f(c)* **of**
 apple : $a := a + 1$;
 banana : $b := b + 1$;
end

record *count: integer;*
case *c: fruit* **of**
 apple : (*diameter* : *integer*) ;
 banana : (*length* : *real*) ;
end

(2) ADA: `F(X)` could be an array reference or a function call. But arrays and functions are *very* different [Van der Linden, *SIGPLAN Notices 17*, 3(Mar 82), 93-94]

— arrays have to be initialized before use, but not functions

— arrays can receive as well as return values, but not ADA functions (PL/I has ‘‘pseudo-functions’’)

— arrays can be passed to a subprogram as a parameter, but not ADA functions (no **procedure** type in ADA)

— arrays may return different values at different times with the same arguments, but not *pure* ADA functions

— arrays are viewed as data areas, but functions are viewed as code

(3) ALGOL 60: double use of **begin** · · · **end** as statement compounder and block definer. Confuses sequencing, definition and allocation.

(4) PASCAL: **repeat** loops can have any number of statements in the body, but **for** loops can have only one (requiring multi-statement body to be bracketed).

14. Extensibility

— supports *abstract datatype*: a collection of data items and functions that create, access and mutate these data. Implemented by classes, modules (Modula2), clusters (CLU), packages (ADA), etc.

— encouraged by overloading, inheritance or type-parameters: the compiler "dispatches" the code that will be executed

(1) ADA: **package** concept and overloading of operators

(2) PASCAL: user-define types, but no way to group data types and associated procedures and functions together

(3) ALGOL 68: highly extensible via overloading

mode *vec* = **struct** *(***real** *xcoord, ycoord, zcoord);*
op *x* = *(vec u) real: xcoord* **of** *u;*
op *y* = *(vec u) real: ycoord* **of** *u;*
op *z* = *(vec u) real: zcoord* **of** *u;*
op * = *(real r, vec u) vec: (r* x u, r* y u, r* z u);*
op * = *(vec u,v) real: x u*x v + y u*y v + z u*z v;*
op *norm* = *(vec u) real: sqrt(u*u);*

can declare precedence of ops

(4) C++: Class definition: compile-time overloading & type checks, inheritance, and type-templating of classes

15. Supports Information Hiding

D. Parnas, "On the criteria to be used in decomposing systems into modules", *CACM 15*, 12 (Dec 72), 1053-58.

- modules should be designed with two types of security in mind (abstraction barrier)

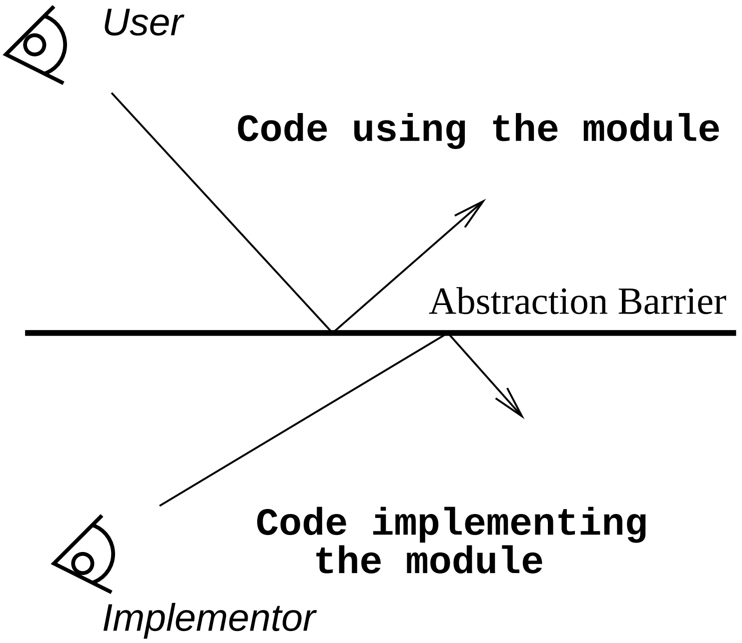
  (A) *secure abstraction*: the *user* has all the information needed to *use* the module correctly—and nothing more

  (B) *secure modification*: the *implementor* has all the information needed to *implement* the module correctly—and nothing more

- Two discoveries during the evolution of languages

  — name visibility rules need to be flexible: classical block structure is too rigid

  — passing information among modules via parameter lists is insufficient


User
Code using the module
Abstraction Barrier
Code implementing
the module
Implementor

ex: FORTRAN "labeled" COMMON provided a means for sharing data among subroutines/functions *without declaration in an enclosing block*

```
   DIMENSION A(99)  block not declared
10 READ(1,20) K
20 FORMAT(I2)
     .  .  .
   SUBROUTINE VBLE(ITOKEN)
   COMMON /SYMTAB/NAME(100)block SYMTAB
     .  .  .
   END
   SUBROUTINE CONST(ITOKEN)
   COMMON /SYMTAB/ENTRY(100)shared
     .  .  .
   END
```

- Lack of information hiding leads to two problems in some languages

  — *indiscriminate access*: when *inadvertent* access to variables cannot be *forbidden*

  — *vulnerability*: when access to *needed* variables cannot be *assured* *ex:* ALGOL 60: inadvertent access can occur with any attempt at abstract datatypes

```
begin
    integer array S[1 : 100] ;
    integer TOP;
    procedure Push(x); integer x;
        begin TOP:=TOP+1; S[TOP]:=x end ;
    procedure Pop(x); integer x;
        begin Pop:=S[TOP] ; TOP:=TOP-1 end ;
    boolean procedure Empty();
        begin ··· end ;
    TOP:=0 ;
    ⋮
        user code using ''stack'' abstraction
        Push/Pop calls
        inadvertent access to S and TOP!
end
```

- no way to place declaration in a block-structured language that prohibits indiscriminate access
- any change in implementation of datatype in effect changes the whole program

ex: ALGOL 60: cannot guarantee access to the same variables after local editing

begin integer *x*;
    ⋮   big code
    begin
            ← add new declaration **real** *x*;
        ⋮   big code
        $x := x + 1;$
        ⋮
    end
    ⋮
end

— Difficulty: ALGOL 60 has merged three things that should be decoupled

- *definition* of new object ‘‘class’’ (ADT)
- *allocation* of new object instance
- *access* by binding a new name to new instance

ex:

(1) ADA: package concept makes definition, allocation, name access *orthogonal*

```
generic package STACK is  --"template" def
 procedure PUSH(X: in INTEGER);
 procedure POP(X: out INTEGER);
 function EMPTY return BOOLEAN;
   . . .
end STACK;
package STACK1 is new STACK;  --"instance" allocation
package STACK2 is new STACK;
 --note: in ADA this is static—not done at run-time
declare
 use STACK1;    -- provide for name access
 I, N: INTEGER;
 begin
  .
  .
  .
  PUSH(I);    --access
  POP(N);
  .
  .
  .
  STACK1.PUSH(I);  --qualified name access
  STACK2.POP(N);
 end;
```

(2a) SNOBOL4: vulnerability avoidance (information hiding) violated by *dynamic binding* (*dynamic scoping*).

- *dynamic binding:* free names in each function activation are bound to the names in the *environment of the caller*, resolvable only at run-time.
- *static binding (static scoping, lexical scoping):* free names are bound to the names in lexically surrounding scopes, based on purely textual rules, and resolvable at compile-time.
  - — In the examples below, what value of `y` is printed by `fun`?
    - PASCAL has static binding
    - SNOBOL4 has dynamic binding

```
program bind(output);
const TAB = '    ';
var x,y,z: integer;
```

```
function fun: integer;
var x: integer;
begin (* fun *)
     x := 99;
     writeln('fun:' ,TAB, x ,TAB,y ,TAB,z);
     fun := y
end (* fun *) ;

procedure sub;
var y: integer;
begin (* sub *)
     y := 13;
     writeln('sub 1:' ,TAB, x  ,TAB, y ,TAB, z);
     z := fun;
     writeln('sub 2:' ,TAB, x  ,TAB, y ,TAB, z)
end (* sub *) ;

begin (* bind *)
     x := 1; y := 2; z := 3;
     writeln('bind 1:' ,TAB, x ,TAB, y ,TAB, z);
     sub;
     writeln('bind 2:' ,TAB, x ,TAB, y ,TAB, z);
     z := fun;
     writeln('bind 3:' ,TAB, x ,TAB, y ,TAB, z);
end (* bind *) .
```

```
bind 1:         1        2        3
sub 1:          1       13        3
fun:       99        2        3
sub 2:          1       13        2
bind 2:         1        2        2
fun:       99        2        2
bind 3:         1        2        2
```

```
 MACRO SPITBOL VERSION 3.5 1.1
1         BIND
2             TAB = '  '
3
4             DEFINE('FUN()X')            :(FUN.)
5         FUN X = 99
6             OUTPUT = 'FUN' TAB X TAB Y TAB Z
7             FUN = Y                     :(RETURN)
8         FUN.
9
10            DEFINE('SUB()Y')            :(SUB.)
11        SUB       Y = 13
12            OUTPUT = 'SUB 1' TAB X TAB Y TAB Z
13            Z = FUN()
14            OUTPUT = 'SUB 2' TAB X TAB Y TAB Z
15                                    :(RETURN)
16        SUB.
17
18            X = 1 ; Y = 2 ; Z = 3
21            OUTPUT = 'BIND 1' TAB X TAB Y TAB Z
22                   SUB()
23            OUTPUT = 'BIND 2' TAB X TAB Y TAB Z
24                   Z = FUN()
25            OUTPUT = 'BIND 3' TAB X TAB Y TAB Z
26        END
```

```
BIND 1   1    2    3
SUB 1    1    13   3
FUN  99  13   3
SUB 2    1    13   13
BIND 2   1    2    13
FUN  99  2    13
BIND 3   1    2    2
```

(2b) LISP 1.5: vulnerability avoidance (information hiding) violated by *shallow binding*

- *shallow binding:* free names in body of *a function that is passed as a parameter* are bound to the names in the *environment of the caller*

- *deep binding:* free names in body of *a function that is passed as a parameter* are bound to the names in the environment *at the place where the function is defined*.

```
;;; LISP 1.5   -- shallow binding
> (defun add1 (x) (+ x 1))
add1
> (add1 5)
6
> (defun twice (f y) (f (f y)) )
twice
> (twice ’add1 5)
7
> (twice ’(lambda (x) (* 2 x)) 3)
12
> (setq y 2)
2
> (twice ’(lambda (x) (* y x)) 3)
27
;;; not 12!
```

- `y` interpreted dynamically in caller `twice`. Clash occurs with bound `y` in defn of `twice`. Bindings at call time are stored on an *association list*:

```
( ( f  . '(lambda (x) (* y x)) )  ( y . 3 ) )
(f (f y)) →
( (lambda (x) (* y x))
    ( (lambda (x) (* y x)) 3 ) )  →
( (lambda (x) (* y x))  (* y 3) )  →
( (lambda (x) (* y x))  9 )  →
( * y 9 )  →  27
```

- what happens in Common LISP? Deep binding

```
opu> kcl
AKCL (Austin Kyoto Common Lisp)  Version(1.505)
;;; illustrates deep binding in Common Lisp unless 'special'

>(defun add1 (x) (+ x 1) )
ADD1
>(defun twice (f y) (f (f y)) )
TWICE
>(twice #'add1 5)
Error: The function F is undefined.

>#'add1
(LAMBDA-BLOCK ADD1 (X) (+ X 1))
>'add1
ADD1
```

```
;;; forgot about FUNCALL!!

>(funcall #'+ 2 3)
5

>(funcall #'add1 5)
6

>(defun twice (f y) (funcall f (funcall f y) ) )
TWICE

>(twice #'add1 5)
7

>(twice #'(lambda (x) (* 2 x))  3)
12

>;;; set a variable named y in the top-level environment

>(set y 2)
2

>(twice #'(lambda (x) (* y x))   3)
12

>;;; note function closure # causes deep binding of "y"
 ;;; y is a "free name" in the last closure above
```

(3) C++: Widely practiced *style* of class definition puts *user interface* in a header file, and *implementation* in a separate file.

- declarations, prototypes in `.h` file and code in `.cxx`(`.cpp`) file. Programs *using* the class need only include the header.
- but C++ does not *require* this partition
- *in-lined* function definition within class declaration ‘‘exposes’’ implementation details, violating the ‘‘abstraction barrier’’
- *templated* class declarations *required to include* all implementation in same file as prototypes, and all to be included by class user. (Why?) This effectively breaks down the abstraction barrier.

```
// FILE: bag.h
// CLASS PROVIDED: Bag (a container class for a collection of i

#include <stdlib.h>    // Provides size_t and NULL
#include "link1.h"     // Provides Node struct

 class Bag
 {
 public:
  // TYPEDEF
  typedef Node::Item Item;
  // CONSTRUCTORS and DESTRUCTOR
  Bag( );
  ~Bag( );
  // MODIFICATION functions
  void insert(const Item& entry);
     ...
  // CONSTANT functions
  size_t size( ) const { return many_nodes; } // INLINED
  Item grab( ) const;
 private:
  Node *head_ptr;     // Head pointer for Item list
  size_t many_nodes; // Number of nodes on list
 };
```

```
// FILE: bag.cxx
// CLASS implemented: Bag (see bag.h)
#include <stdlib.h> // Provides NULL, rand, size_t
#include "link.h"   // Provides Node, list_clear, ...
#include "bag.h"

Bag::Bag( )
// Library facilities used: stdlib.h
{
      head_ptr = NULL;
      many_nodes = 0;
}

Bag::˜Bag( )
// Library facilities used: link1.h
{
      list_clear(head_ptr);
      many_nodes = 0;
}

void Bag::insert(const Item& entry)
// Library facilities used: link1.h
{
      list_head_insert(head_ptr, entry);
      many_nodes++;
}
   ...
```

16. Supports Development

— facilities for "programming in the large"

— full macro preprocessor; at least a define/include facility

— library extension and standard prolog facility

— information hiding across separately compiled modules

— separate compilation of modules (data and procedures)

— integrated with programming development tools

ex:

(1) PASCAL: major limitations ⇒ Turbo Pascal &c, Modula2, · · ·

(2) ALGOL 60: a major detriment to its use

(3) C, C++ plus Unix program development tools (`emacs`, `make`, `rcs`, debugger, ...) form successful program development environment. The language itself directly supports none of this.

(4) In `C#`, all major processors (compiler, linker) are available as class methods to be called within the language itself — allows for ‘‘on the fly’’ creation, translation, execution of code.

(5) In C,C++ the notion of *module* is identified with that of *file*, confounding two concepts. Not so in `modula3` and predecessors.

(6) Monolithic commercial development environments combine structure editors, preprocessors, compilers, libraries, linkers, project management, debuggers, multiple target code generators, document tools (e.g., MetroWerks *Code Warrior*, Visual *C++*)

(7) Smalltalk code is almost never actually written using its linearized ‘‘file-in’’ syntax. An elaborate development environment surrounds the user, with graphical user interface providing an interactive dialog for defining classes and entering code for class methods. Language itself is hard to distinguish from its programming environment. It is all in Smalltalk. (Only the virtual machine executing bytecode has to be ported to a new target.)