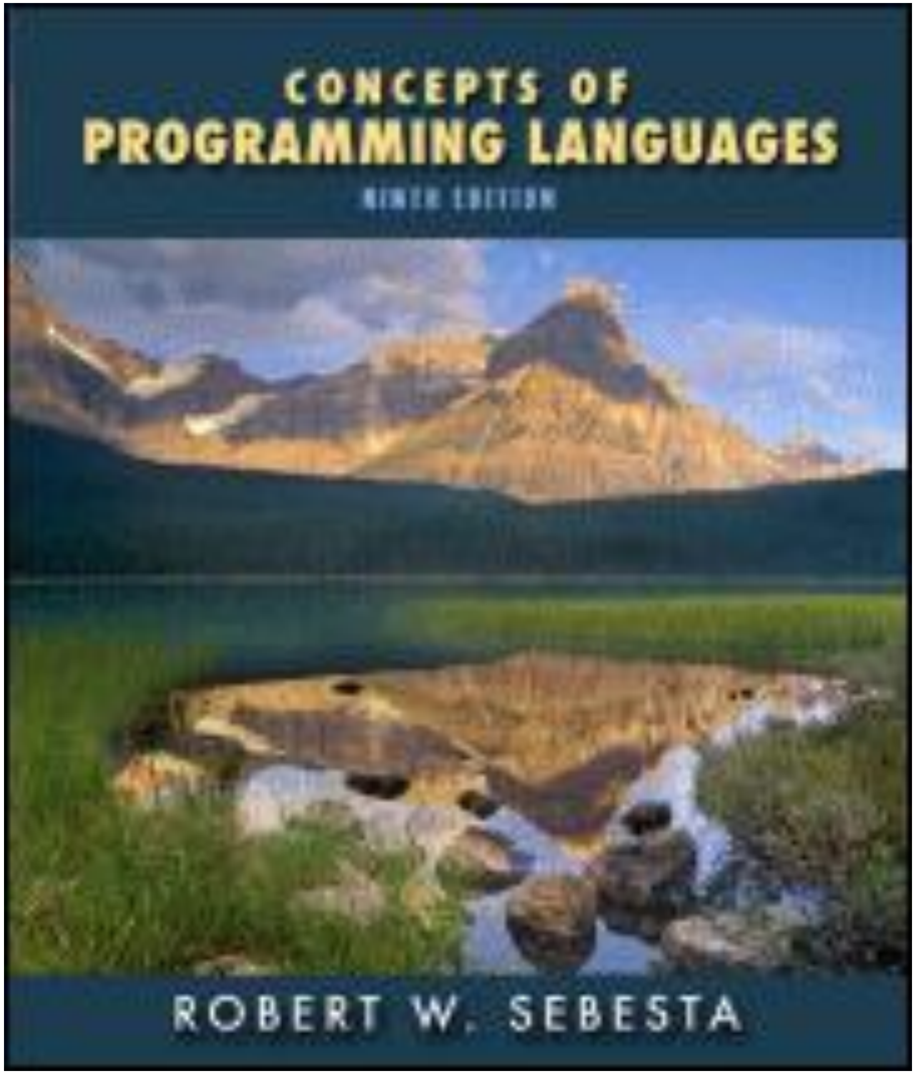
Chapter 9

Subprograms



Ch09 – Subprograms

9.6 Parameters That Are Subprogram Names

- Example
  - In C and C++, pointers or references to functions can be passed as parameters, but not functions.
  - int y = 2;

    int f(int x) { return x+y; }   // look up y in global data area

    Method 1 – Pointers to functions as parameters

    int p(int (*f)(int)) { return f(7); } // (*f) → f; f(7) → (*f)(7)

    cout << p(f) << p(&f);   // only code pointer is passed

    Method 2 – References to functions as parameters

    int p(int (&f)(int)) { return f(7); }   // f(7) → (*f)(7)

    cout << p(f);   // only code pointer is passed

9.6 Parameters That Are Subprogram Names

- Referencing environment for passed subprograms
  - Shallow binding
    The passed subprogram is executed in the calling env.
  - Deep binding
    The passed subprogram is executed in the defining env.
  - Ad hoc binding
    The passed subprogram is executed in the passing env.

9.6 Parameters That Are Subprogram Names

- A Javascript example
  - function sub1()

```
{
    var x = 1;                                    static distance = 1
    function sub2() { alert(x); }
    function sub3() { var x = 3; sub4(sub2);}
    function sub4(subx) { var x = 4; subx(); }
    sub3();
}
```

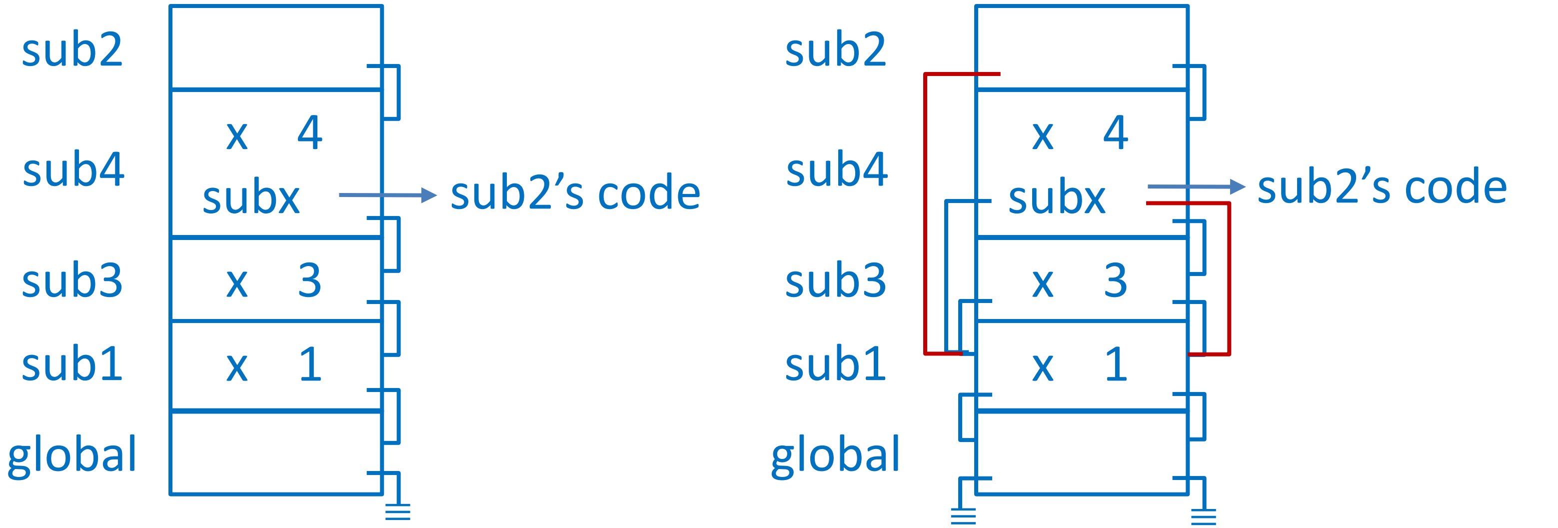
alert(4) – shallow binding, dynamic scoping

alert(1) – deep binding, static scoping

alert(3) – ad hoc binding; never used

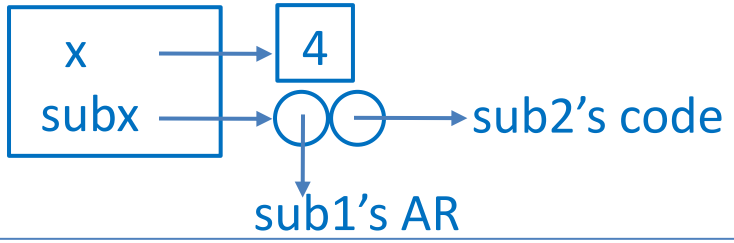
9.6 Parameters That Are Subprogram Names

- Shallow binding Deep binding



- Note: The r.h.s. diagram is for typed languages that don't allow returning functions as values (e.g. Pascal).

An AR for untyped languages:



Supplementary: Scheme Implementation

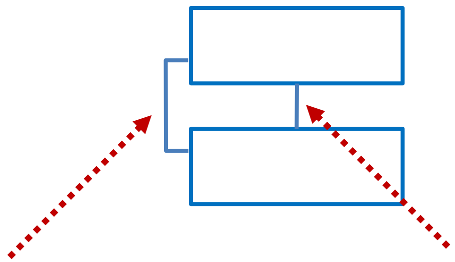
- First-class objects
  - First-class objects satisfy 3 properties:
    - Storable
    - Can be passed as arguments
    - Can be returned as function values
  - Example

    In C/C++, arrays and functions aren’t first-class.

    In Scheme, SML, Javascript, and Perl, functions are first-class

Supplementary: Scheme Implementation

- Initial run-time stack



top-level environment, initially empty

global environment, for predefined bindings

static pointer

dual purposes: dynamic pointer and list pointer

global

top-level

```
> cons
#<procedure cons>
> car
#<procedure car>
```

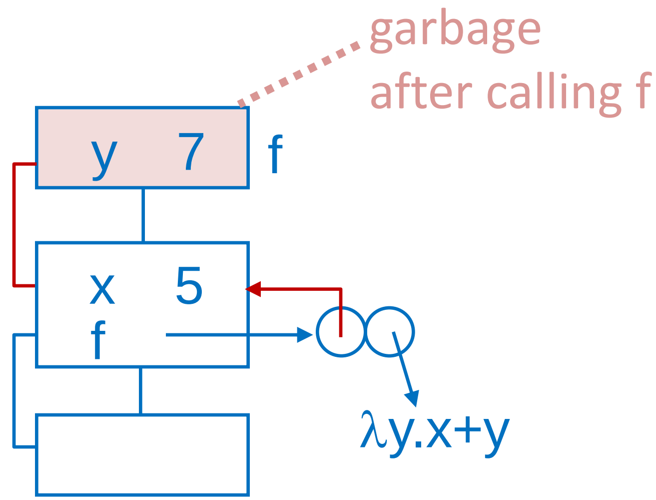
Supplementary: Scheme Implementation

- Evaluation of λ-expressions

(define x 5)

(define f (lambda (y) (+ x y)))

(f 7) ⇒ 12



A λ-expression evaluates to a closure

(code pointer, environment pointer)

where the environment pointer points to the stack top in which the λ-expression is evaluated (i.e. the definition environment of the λ-expression).

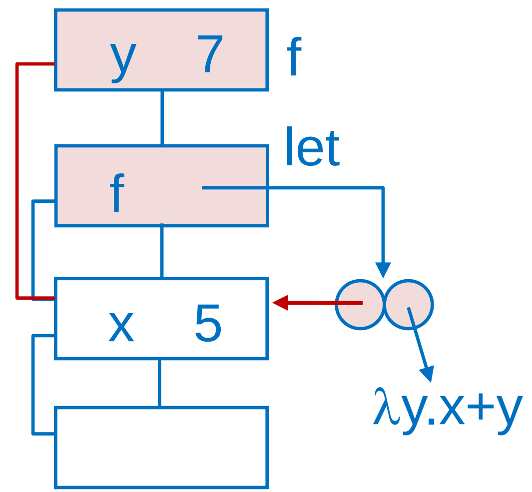
Supplementary: Scheme Implementation

- Evaluation of let expressions

(define x 5)

(let ((f (lambda (y) (+ x y))))

(f 7)) ⇒ 12

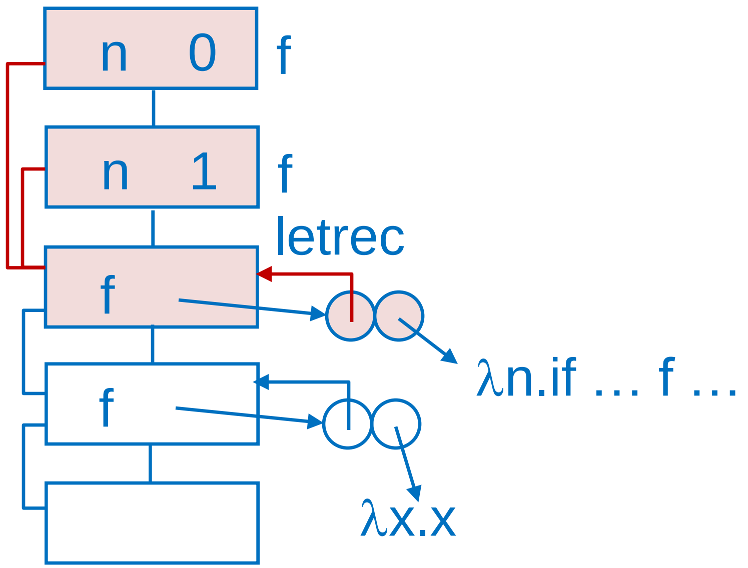


λy.x+y is evaluated before allocating let's AR.

Supplementary: Scheme Implementation

- Evaluation of letrec expressions

```
(define f (lambda (x) x))
(letrec ((f (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))))
   (f 1)) ⇒ 1
```



λn.if…f… is evaluated after allocating letrec's AR.

Supplementary: Scheme Implementation

- Functions as arguments

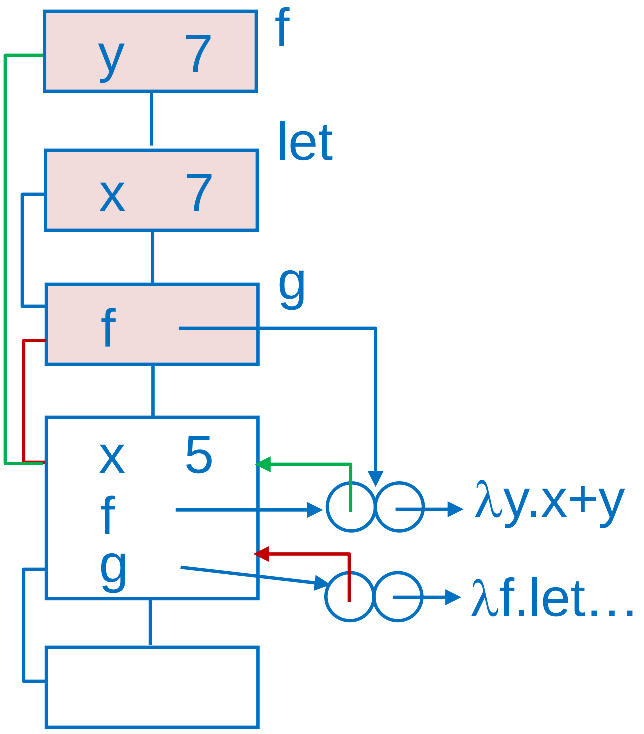
(define x 5)

(define f (lambda (y) (+ x y)))

(define g

(lambda (f) (let ((x 7)) (f x))))

(g f) ⇒ 12



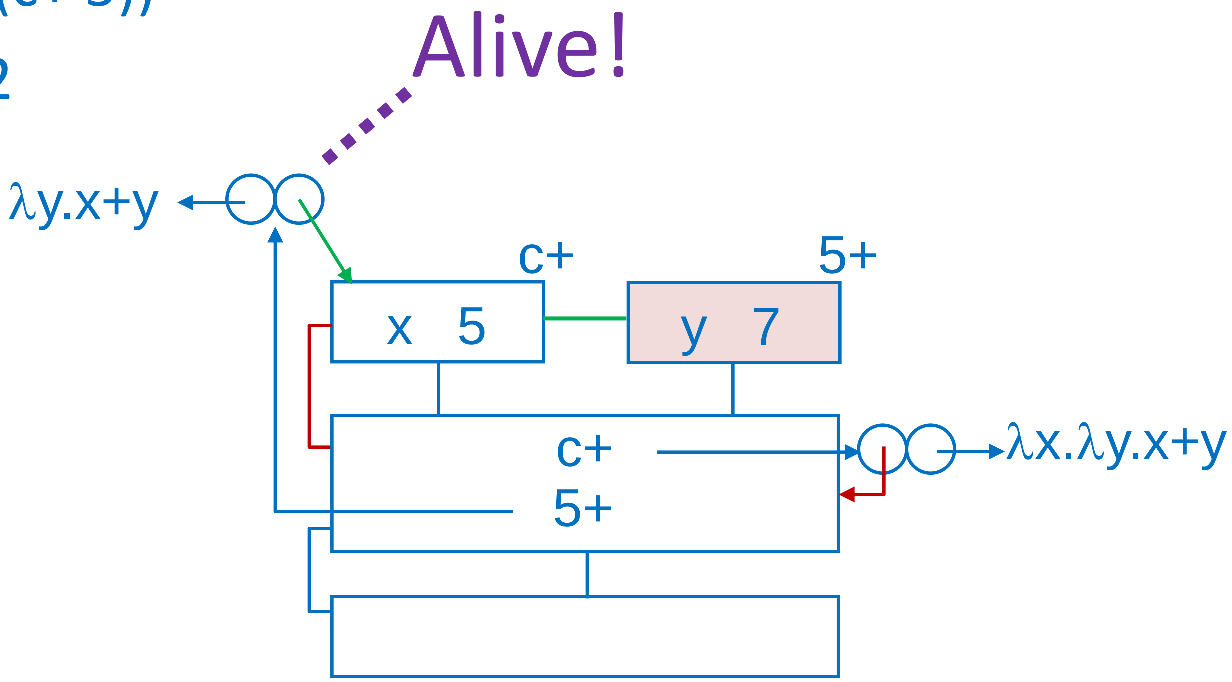
Supplementary: Scheme Implementation

- Functions as returned values

(define c+ (lambda (x) (lambda (y) (+ x y))))

(define 5+ (c+ 5))

(5+ 7) ⇒ 12



Supplementary: Scheme Implementation

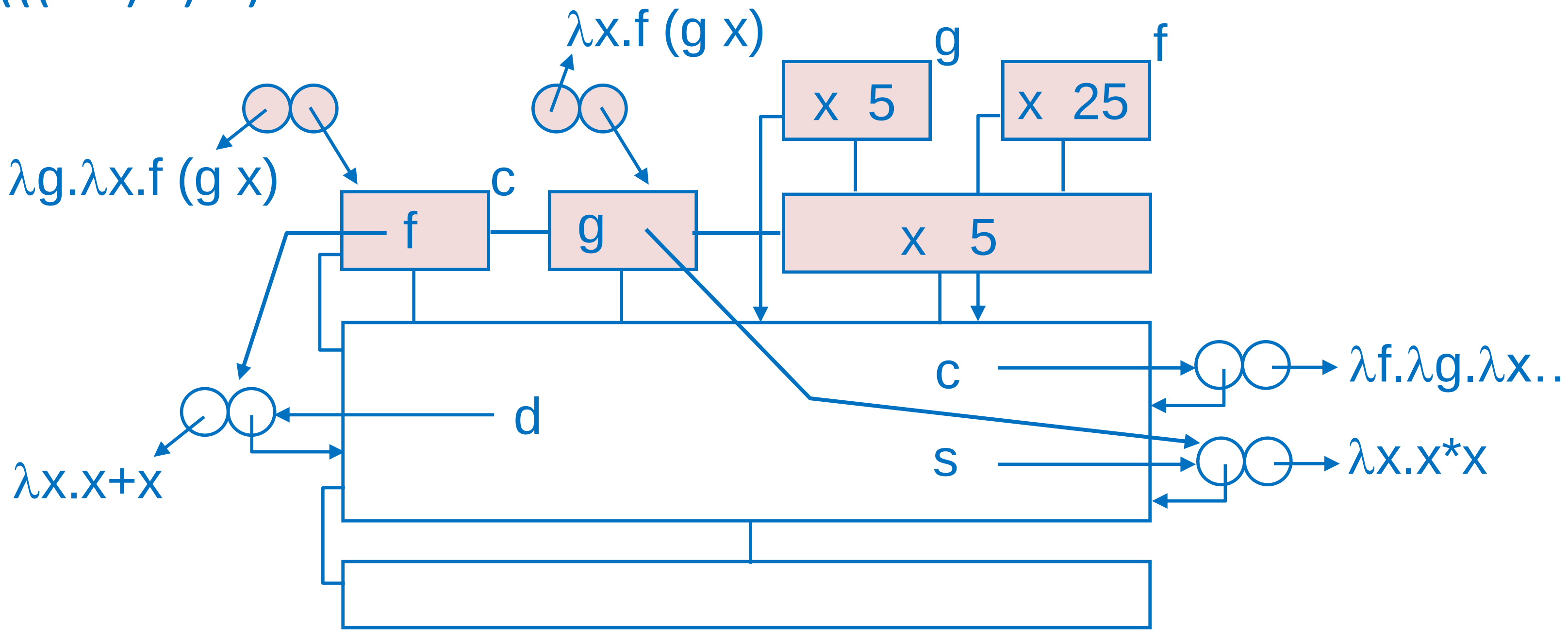
- Functions as returned values

  (define c (lambda (f) (lambda (g) (lambda (x) (f (g x))))))

  (define d (lambda (x) (+ x x)))

  (define s (lambda (x) (* x x)))

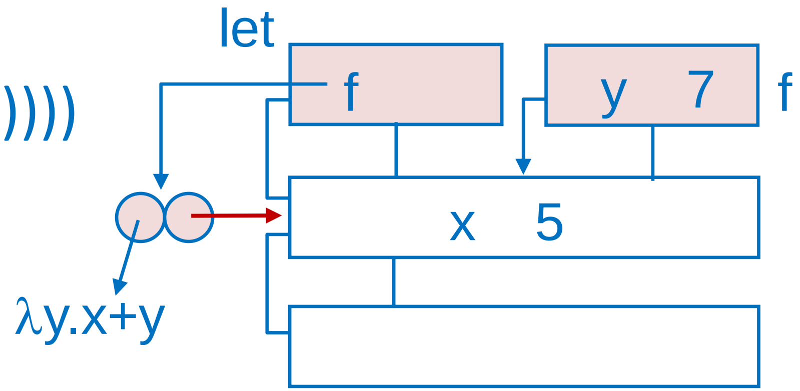
  (((c d) s) 5) ⇒ 50



Supplementary: Scheme Implementation

- Conclusions on functions as returned values
  - The run-time stack must be implemented as a linked list.
  - The run-time stack grows like a tree.
  - All objects (in particular, local objects) in Scheme have unlimited extent – they continue to exist as long as they aren't garbage collected.

```
(define x 5)
((let ((f (lambda (y) (+ x y))))
    f)
  7)
⇒ 12
```



Supplementary: Scheme Implementation

- Note on C/C++
  - Pointers/references to functions may be returned as function values, but not functions.
  - Only code pointers are returned.
    Free variables of a function must be global and can be accessed directly in the global/static data area.

```
int f(int x) { return x+y; }         // free y is global
int (*p())(int) { return &f; }       // or, return f;
int (&q())(int) { return f; }
cout << p()(7) << (*p())(7) ;
cout << q()(7) << (*q())(7);
```

9.7 Overloaded Subprograms

- Overloaded subprograms
  - An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
  - Properties of overloaded subprograms
    - Have distinct definitions
    - Needn’t behave similarly
    - Subprogram calls are resolved by the compiler
  - Example

```
int square(int n) { return n*n; }
double square(double n) { return n*n; }
long square(long n) { return 8825252*n; }
```

9.7 Overloaded Subprograms

- Two kinds of overloading
  - Context-independent overloading
    - Overloaded subprograms must differ in parameters, e.g. C++
  - Context-dependent overloading
    - Overloaded subprograms must differ in parameters or function value's type, e.g. Ada
  - Example

```
int square(int n) { return n*n; }
double square(int n) { return n*n; }
int x = square(2);          // call int → int
double x = square(2);       // call int → double
```

9.8 Generic Subprograms

- Generic subprograms
  - A *generic subprogram* takes parameters of different types on different activations.
- Generic subprograms in C++
  - Properties of generic subprograms in C++
    - abbreviations for sets of overloaded subprograms
    - template instantiations (macro-expansion) driven by the template arguments at compile time
  - Example

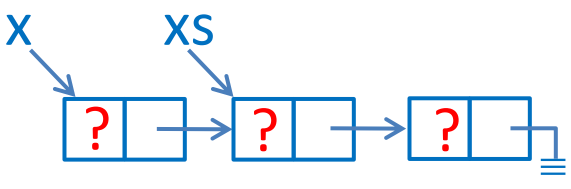
```
template<typename T> T square(T n) { return n*n; }
template<> long square(long n) { return 8825252*n; }
cout << square(2) << square(2.0) << square(2L);
```

9.8 Generic Subprograms

- Generic subprograms in ML
  - Properties of generic subprograms in ML
    - Have a single definition
    - Behave similarly
  - Example

```
– fun length [] = 0
=     | length (x::xs) = 1+length xs;
val length = fn : 'a list -> int
– length [1,2,3];

– length [1.1,2.2,3.3];
```



9.8 Generic Subprograms

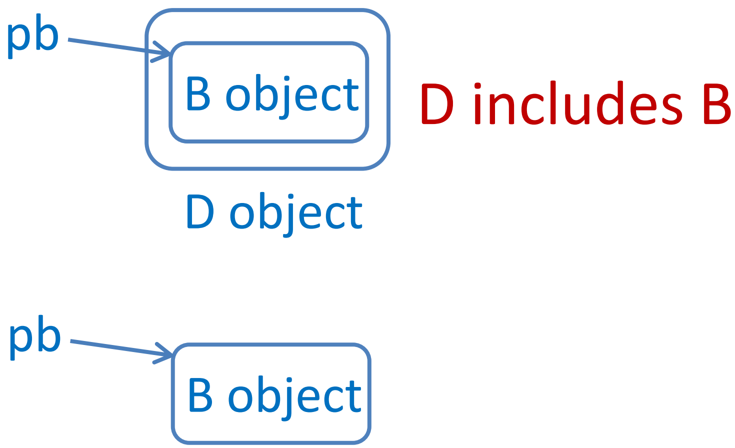
- Polymorphism
  - Ad hoc polymorphism (i.e. overloading)
  - Parameterized polymorphism
    - Usually refer to ML's generic subprograms
    - C++'s generic subprograms is usually regarded as a special kind of parameterized polymorphism
  - Inclusion polymorphism
    - Polymorphism in OOP
    - Late binding; run-time polymorphism

      (Ad hoc and parameterized polymorphisms are early-binding compile-time polymorphisms.)

9.8 Generic Subprograms

- Polymorphism
  - Inclusion polymorphism in C++

```
class B {
public: virtual void p() {}
};
class D : public B {
public: virtual void p() {}
};
B* pb=new D;
pb->p();            // D::p
pb=new B;
pb->p();            // B::p
```

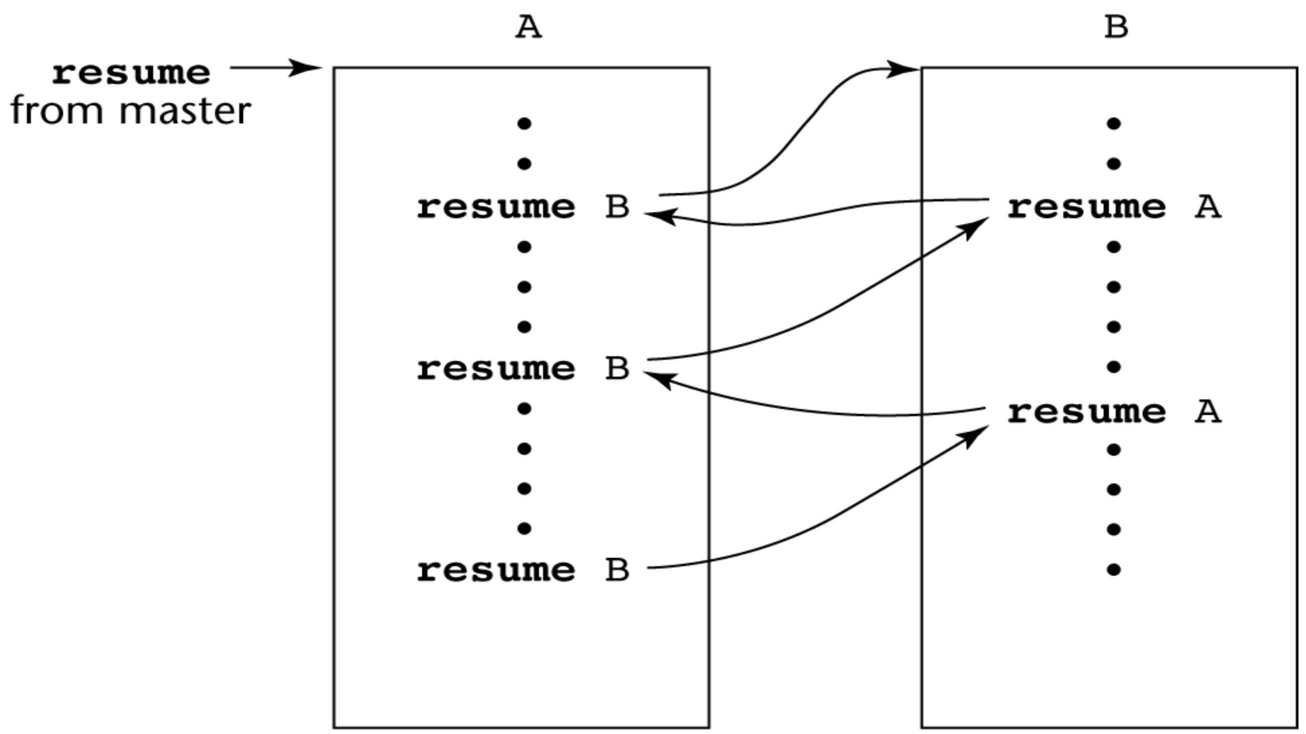


9.11 Coroutines

- Coroutine (Concurrent routine)
  - A *coroutine* is a subprogram that has multiple entries and controls them itself.
  - A coroutine call is named a *resume.*
  - The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine.
  - Coroutines repeatedly resume each other, possibly forever
  - Coroutines provide *quasi-concurrent execution* of program units (the coroutines)

9.11 Coroutines

- Coroutines illustrated: possible execution controls



(a)

9.11 Coroutines

- Coroutines in C/C++

```
// generate the infinite sequence 0,1,2,..., one at a time
int nats()
{
    static int state=0,k=0;
    switch (state) {
        case 0: goto start;
        case 1: goto resume;
    }
start: state=1;
    while (true) { return k; resume: k++; }
}
```

```
// ordinary function
int nats()
{
    static int k=0;
    return k++;
}
```

9.11 Coroutines

- Coroutines in C/C++

```
// Duff's device
int nats()
{
    static int state=0,k=0;
    switch (state) {
        case 0:  state=1;
                 while (true) {
                     return k;
        case 1:      k++;      // case within the nested block
                 }
    }
}
```

9.11 Coroutines

- Coroutines in C/C++

```
// Take the next n natural numbers from coroutine nats()
void takenext(int n)
{
    for (int i=1;i<=n;i++) cout << nats() << ' ';
    cout << endl;
}
int main(void)
{
    takenext(7); takenext(8);
}
```

9.5 Parameter Passing Methods

- Parameter passing methods
  - Call by value
  - Call by result
  - Call by value-result
  - Call by reference
  - Call by name
  - Call by need

call by copy

delayed evaluation

- Call by copy
  - Disadvantages
    - Space and time overhead for copying in/out large objects
    - Unsuitable for some types, e.g. functions.

9.5 Parameter Passing Methods

- Advantage
  - Time efficient when accessing formal parameters
- Call by reference
  - Disadvantage
    - Time inefficient when accessing formal parameters, due to indirection operations.
  - Advantages
    - Space and time efficient at the entry time when passing large objects.
    - Suitable for any type, e.g. functions

9.5 Parameter Passing Methods

- Parameter functionality and passing method
  - In some languages (e.g. Ada, Fortran 95), parameters are declared according to their *high-level* functionalities, rather than the *low-level* passing methods.
  - Three modes of functionalities

    in　　caller → callee

    out　　caller ← callee

    inout　　caller ↔ callee

9.5 Parameter Passing Methods

- Parameter functionality and passing method
  - For efficiency reasons, the passing methods used for each mode of functionality depend on the types of parameters.

	in	out	in out
Primitive type	value	result	value-result
Structured type	reference	reference	reference

Requirement: in mode parameters can't be modified.

Warning: out mode parameters should not be referenced before being assigned values.

9.5 Parameter Passing Methods

- A Fortran 95 example

```
subroutine fact(n,r)                        ! default is inout
implicit none
integer, intent(in) :: n
integer, intent(out) :: r
r=1
do while (n>1)
    r=n*r
    n=n-1      ! error
end do
end subroutine
```

⇒

```
integer :: i
r=1; i=n
do while (i>1)
    r=i*r
    i=i-1
end do
```

9.5 Parameter Passing Methods

More on call by copy and call by reference

- Design of call by result and value result
  - The order of copying-out the parameters is important.

```
program main                      subroutine p(a,b)
implicit none                     implicit none
integer :: x=7                    integer, intent(inout) :: a,b
call p(x,x)   ! unspecified       a=2            ! or, out
print *, x                        b=3
end program                       end subroutine
```

$a \rightarrow x$ and then $b \rightarrow x$, or $b \rightarrow x$ and then $a \rightarrow x$?

9.5 Parameter Passing Methods

- Design of call by result and value result
  - Where to copy out?

```
program main
implicit none
integer :: i=2
integer, dimension(3) :: a=[1,2,3]
call p(i,a(i))     ! unspecified
end program
```

```
subroutine p(a,b)
implicit none
integer, intent(inout) :: a,b
a=3                ! or, out
b=4
end subroutine
```

With left-to-right copying out process, should b be copied out to a[2] or a[3]?

9.5 Parameter Passing Methods

- Difference between call by reference and value result

  Call-by-reference may introduce aliases, due to sharing.

  Call-by-value-result never introduces aliases, due to copying.

```
void swap(int& a,int& b)
{
    a = a+b; b = a-b; a = a-b;
}

swap(x,x);    // NO
```

```
subroutine swap(a,b)
implicit none
integer, intent(inout) :: a, b
a = a+b; b = a-b; a = a-b;
end subroutine

call swap(x,x)  ! OK
```

9.5 Parameter Passing Methods

- Difference between call by reference and value result

  Call-by-reference modifies the actual parameters immediately

  Call-by-value-result modifies the actual parameters on exit.

```
void p(int& x)
{
    x++;  throw 777;
}
a=2;
p(a);           // with call-by-reference, a becomes 3
                // with call-by-value-result, a is still 2.
```

9.5 Parameter Passing Methods

- Delayed evaluation (call by name, call by need)
  - An actual parameter isn't evaluated at the point of call. It is evaluated only when its value is needed (i.e. the corresponding formal parameter is referenced.)
  - Call by name
    - An actual parameter is evaluated *each time* its value is needed
  - Call by need
    - An actual parameter is evaluated *only* when its value is needed for *the first time*.
    - The value obtained is cached for further use.

9.5 Parameter Passing Methods

- Call by name
  - Used in Algol 60, Simula 67
  - Substitution rule: for the effect of call-by-name

    Substitute the actual parameter for every occurrence of the corresponding formal parameter.

    ```
    void swap (int x,int y) { int z=x; x=y; y=z; }
    swap(a,b)        ⇒ int z=a; a=b; b=z;
    swap(i,a[i])     ⇒ int z=i; i=a[i]; a[i]=z;
    ```
  - Difference between call by name and call by reference

    Call by name adopts flexible late binding

    Call by reference adopts fixed early binding

9.5 Parameter Passing Methods

- Call by name
  - How to resolve name conflict?
    - Rename the local variable

      void swap (int x,int y) { int z=x; x=y; y=z; }

      swap(a,z) ⇒ int z′=a; a=z; z=z′;

    - Evaluate the actual parameter in its environment

      swap(a,z) ⇒ int z=a; a=z; z=z

      evaluate in callee's env.

      evaluate in caller's env.

      Thus, the code z and its environment, i.e. a closure, must be passed.

9.5 Parameter Passing Methods

- Implementation of call by name
  - An actual parameter *exp* is passed as a parameterless function, called a ***thunk***.

    thunk() { *exp* }
  - A reference to the formal parameter becomes a call to the thunk.
- Implementation of call by need
  - Like call-by-name, except that a call-by-need thunk must memoize the value of the actual parameter computed for the 1st time.

9.5 Parameter Passing Methods

- Simulation of call-by-name in Scheme
  - ```
    (define-syntax freeze
        (syntax-rules ()
            ((freeze exp) (lambda () exp))))
    ```

    macro call   macro expansion

    ```
    (define thaw (lambda (promise) (promise)))
    ```
  - Since Scheme uses call by value, freeze can't be defined as a function.

    (define freeze (lambda (exp) (lambda () exp)))   ; NO!

    (freeze (+ 2 3)) ⇒ (freeze 5) ⇒ (lambda () 5)
    ```

9.5 Parameter Passing Methods

- Simulation of call-by-name in Scheme
  - Example

```
(define myif
    (lambda (e1 e2 e3) (if e1 e2 e3)))
(define f
    (lambda (n) (myif (= n 0) 1 (* n (f (- n 1))))))
(f 3)      ⇒ ???
(define myif
    (lambda (e1 e2 e3) (if e1 (thaw e2) (thaw e3))))
(define f
    (lambda (n)
        (myif (= n 0) (freeze 1) (freeze (* n (f (- n 1)))))))
```

9.5 Parameter Passing Methods

- Simulation of call-by-need in Scheme
  - Delay and force are built-in.

```
> (delay (+ 2 3))
#<procedure>
> (force (delay (+ 2 3)))
5
> (define x 1)
> (define f (lambda (x) (+ (force x) (force x))))
> (f (delay (begin (set! x (+ x 1)) x)))
4
> x
2
```

9.5 Parameter Passing Methods

- Simulation of call-by-need in Scheme
  - ```
    (define-syntax delay
        (syntax-rules ()
          ((delay exp)
           (let ((result-ready? #f)(result #f))
             (lambda ()
               (cond (result-ready? result)
                     (else (set! result exp)
                           (set! result-ready? #t) result))))))))
    (define force (lambda (promise) (promise)))
    ```
  - NB. This version of delay doesn’t work on recursive forcing.
    ```

9.5 Parameter Passing Methods

- Parameter-passing in functional languages

Parameter-passing	Evaluation	Semantics	Language	Implementation
call by value	applicative-order eager	strict	Scheme ML	environment
call by name	normal-order	non-strict	λ calculus	
call by need	lazy	non-strict	Haskell	graph reduction

Non-strict semantics allows one to bypass undefined values.

f x = 5

f (1/0) = ⊥ ⇐ strict semantics: f (⊥) = ⊥

f (1/0) = 5 ⇐ non-strict semantics: f (⊥) ≠ ⊥