Imperative languages

- Von Neumann model:
  - store with addressable locations
- machine code:
  - effect achieved by changing contents of store locations
  - instructions executed in sequence, flow of control altered by jumps
- imperative language:
  - variable corresponds to store location
  - instructions executed in sequence, flow of control altered by conditional and loop statements
  - efficient implementation since close to design of conventional computers

© M. Winter

Functional languages

- computational model: lambda calculus
- mathematical functions: domain, range
- functional languages achieve effect by applying functions
- functional vs. imperative languages
  - store location
  - assignment statement vs. application of a function
    - side-effects
    - aliasing
    - referential transparency
      - (you can replace function calls with their results without changing the functionality)
      - (allows for *memoization*: basically caching for the sake of future function calls)

© M. Winter

Program structure

- "what", not "how"
- primitive functions
- mechanism to define new functions from old

```
max a b = if a < b then b else a
min a b = if a < b then a else b
difference a b c = max a (max b c) - min a (min b c)
```

- considerations:
  - synchronization
  - simple solutions
  - symbolic manipulation
  - list-processing

© M. Winter

Features of functional languages

- higher-order functions
  - can accept functions as parameters
  - can return functions as results

```
comp f g = \x -> f (g x)
```

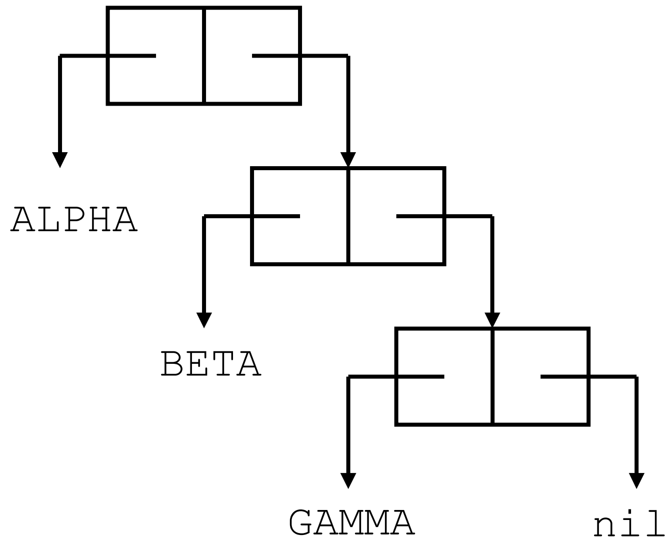
- recursion as a basic principle
- application of rewrite rule:
  - function call replaced by code body
- run-time overhead ⇒ garbage collection

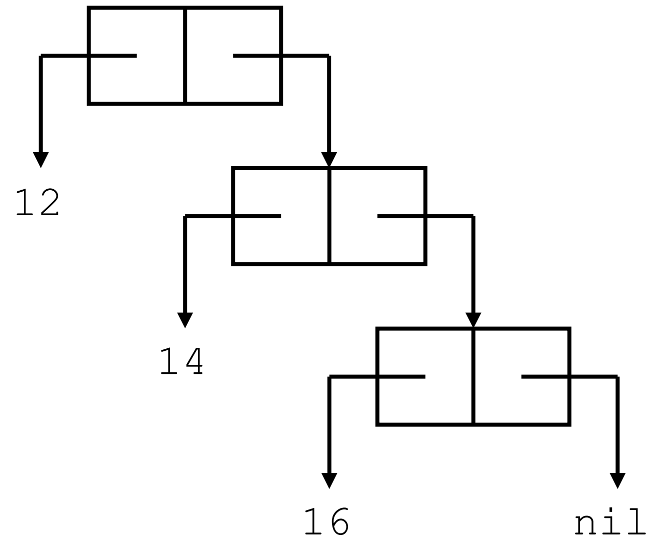
© M. Winter

Lists

- Lisp: atoms, lists, functions
- atoms
  - symbolic
  - numeric
- lists
  - each element is an atom or another list

```
(ALPHA BETA GAMMA)
(12 14 16)
((A B) (HELLO THERE) 94)
```





A

B nil HELLO 94 nil

THERE nil

Operations on lists

- list operations
  - `head` `ALPHA` is the head of `(ALPHA BETA GAMMA)`
  - `tail` `(BETA GAMMA)` is the tail of `(ALPHA BETA GAMMA)`
  - `cons`

```
cons(head(x), tail(x)) = x
head(cons(a,x)) = a
tail(cons(a,x)) = x
```

- recursion

```
isMember x a = if a==[]
                 then False
                 else if x==head a
                        then True
                        else isMember x (tail a)
```

© M. Winter

Higher order functions

```
function map (f, x) is
begin
   if x = nil then return nil;
               else return cons(f(head(x)), map (f, tail(x)));
   endif;
end map;

function timesTen (x) is
begin
   return 10 * x;
end timesTen;

aList                       (17 24 59)
map(timesTen, aList)               (170 240 590)
```

© M. Winter

Lisp - functional core

- function application
- symbolic expressions (S-expressions)
- lambda expressions
  - naming new functions: `DEF`
- interactive system
- prefix notation
  - all expressions are parenthesised

© M. Winter

- predefined functions
  - `CAR`, `CDR`
  - `QUOTE`
    - short form ‘
  - `ATOM`
  - `NULL`
  - `MAPCAR`
- special forms
  - `NIL`, `T`
  - `COND`

© M. Winter

```
(TIMES 10 2)

(LAMBDA (X) (TIMES 10 X))

(DEF TIMESTEN (LAMBDA (X) (TIMES 10 X)))

(TIMESTEN 2)   ⇒ ((LAMBDA (X) (TIMES 10 X)) 2)
   ⇒ (TIMES 10 2)
   ⇒ 20

(MAPCAR ‘TIMESTEN ‘(17 24 59))

(DEF ISMEMBER (LAMBDA (X A)
   (COND ((NULL A) NIL)
     ((EQ X (CAR A)) T)
     (T (ISMEMBER X (CDR A)))
   )
))
```

Lisp - Example

Derivative: Given an S-expression such as:

```
(TIMES 4 (POWER X 3))
```

the aim is to produce the S-expression that represent its derivative with respect to `X`, that is, in this case, to produce the list:

```
(TIMES 12 (POWER X 2))
```

```
(DEF THIRD (LAMBDA (Y)
    (CAR (CDR (CDR Y)))
))
```

© M. Winter

```
(DEF DERIVE (LAMBDA (Y)
   (COND
        ((EQ (CAR Y) ‘POWER)
        (LIST ‘TIMES (THIRD Y)
                (LIST ‘POWER ‘X (DIFFERENCE (THIRD Y) 1))
        ))
        ((EQ (CAR Y) ‘TIMES)
        (LIST ‘TIMES (TIMES (CAR (CDR Y)) (THIRD (THIRD Y)))
                (THIRD (DERIVE (THIRD Y)))
        ))
        (T ‘ERROR)
   )
))
```

© M. Winter

Scope rules

- dynamic scope: identifier bound to most recent definition
- free variable
- example:

```
(DEF BASE (LAMBDA (F A)
   (PLUS (F 10) A)
))
(DEF TWODIGIT (LAMBDA (A B)
   (BASE ‘(LAMBDA (C) (TIMES A C)) B)
))
```

The result of `(TWODIGIT 2 3)` is not `23`.

© M. Winter

Scope rules

```
(TWODIGIT 2 3)
   ⇒ (BASE '(LAMBDA (C) (TIMES A C)) 3)        most recent A = 2
   ⇒ (PLUS ((LAMBDA (C) (TIMES A C)) 10) 3)   most recent A = 3
   ⇒ (PLUS ((TIMES 3 10) 3)
   ⇒ (PLUS 30 3)
   ⇒ 33
```

This problem would not have arisen if the atom `A` had not been used in two different contexts; for example if `BASE` had been defined as:

```
(DEF BASE (LAMBDA (F X)
   (PLUS (F 10) X)
))
```

Be careful !!!

© M. Winter

Lisp - miscellaneous

- No language standard
- Scheme
  - popular dialect
  - meaningful identifiers, reserved words
  - block structure, static scope rules

```
(DEFINE TIMESTEN (LAMBDA (X) (* 10 X)))

(DEFINE ISMEMBER (LAMBDA (X A)
   (COND ((NULL? A) #F)
         ((EQ? X (CAR A)) #T)
           (ELSE (ISMEMBER X (CDR A)))
   )
))
```

© M. Winter

Lisp - miscellaneous

- Imperative features
  - assignment: SET, SETQ

    (SETQ A ‘B) = (SET ‘A ‘B)

    (SET A ‘C)
  - PROG,
  - LOOP, GO

```
(DEF SUMLIST (LAMBDA (X)
   (PROG (TOTAL)
        (SETQ TOTAL 0)
        LOOP
        (COND ((NULL X) (RETURN TOTAL))
              (T (SETQ TOTAL (PLUS TOTAL (CAR X)))
        ))
        (SETQ X (CDR X))
        (GO LOOP)
)))
```

© M. Winter

FP systems

- purely functional
- data values: atoms or sequences
- each element of a sequence is an atom or a sequence
- no variables
- functions can have only one parameter (may be a sequence)
- application of function `G:X`
- functional forms
  - `APPLYTOALL`
  - composition: `o`
  - condition: `(P -> H;G)`
  - construction: `[F1, F1, … , FN] : X`

© M. Winter

FP systems - example

```
DEF ISMEMBER =
   (NULL ° SEC -> F;
   (EQ ° [HEAD, HEAD ° SEC] -> T;
    ISMEMBER ° [HEAD, TAIL ° SEC]))


ISMEMBER : <3,<7,3,9>>
```

© M. Winter

```
NULL ° SEC : <3,<7,3,9>>
   ⇒ NULL : (SEC : <3,<7,3,9>>)
   ⇒ NULL : <7,3,9>
   ⇒ false

EQ ° [HEAD, HEAD ° SEC] : <3,<7,3,9>>
   ⇒ EQ : ([HEAD, HEAD ° SEC] : <3,<7,3,9>>)
   ⇒ EQ : <HEAD : <3,<7,3,9>>, HEAD ° SEC : <3,<7,3,9>>>
   ⇒ EQ : <3, HEAD : <7,3,9>>
   ⇒ EQ : <3, 7>
   ⇒ false

ISMEMBER ° [HEAD, TAIL ° SEC] : <3,<7,3,9>>
   ⇒ ISMEMBER : ([HEAD, TAIL ° SEC] : <3,<7,3,9>>)
   ⇒ ISMEMBER : <HEAD : <3,<7,3,9>>, TAIL ° SEC :
   <3,<7,3,9>>>
   ⇒ ISMEMBER : <3, TAIL : <7,3,9>>
   ⇒ ISMEMBER : <3, <3,9>>
```

Haskell

Features of Haskell:

- static typing
- type inference
- higher-order functions
- polymorphic functions
- pattern matching
- list comprehension
- lazy evaluation
- Classes (overloading)
- type operators
- monads
- modules

© M. Winter

Example

```
factorial :: Integer -> Integer
factorial 0          = 1
factorial (n+1)      = (n+1)*(factorial n)
```

- `Integer` (unbounded) vs. `Int` (bounded, at least $-2^{29}\ldots2^{29}-1$)

```
factorial 100

933262154439441526816992388562667004907159682643816
214685929638952175999932299156089414639761565182862
536979208272237582511852109168640000000000000000000
00000 :: Integer
```

Polymorphic functions

- Identity

```
id :: a -> a
id x = x
```

- Composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

- Curry/Uncurry

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p = f (fst p) (snd p)
```

© M. Winter

Lists

- `[a]` data type of lists with elements from `a`.
- `head, tail`
- `[], (:)`

```
length :: [a] -> Int
length []     = 0
length (_:l)  = 1 + length l

map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs)  = f x : map f xs

filter :: (a -> Bool) -> [a] -> [a]
filter p []                       = []
filter p (x:xs)     | p x         = x : filter p xs
                    | otherwise   = filter p xs
```

© M. Winter