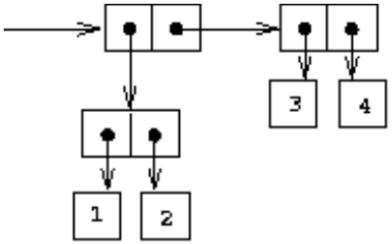
# W. 4.2
# Listy a programowanie funkcyjne

# Abstrakcja

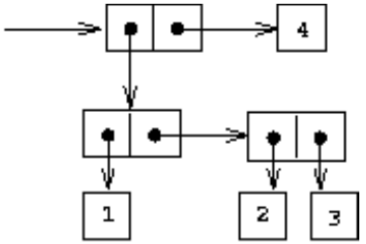
# Programowanie funkcyjne a listy

- Paradygmat funkcyjny polega na pisaniu programów przy użyciu funkcji (matematycznych – dla danego argumentu zawsze jedna określona wartość) nie posiadających efektów ubocznych.
- Lista jest jedną z podstawowych sekwencyjnych struktur danych w programowaniu funkcyjnym.
- Pierwszym użyciem list w programowaniu funkcyjnym jest język LISP (LISt Processor), w którym sam program był listą.
- Listy w programowaniu funkcyjnym są posiadają znacznie bardziej rozbudowaną strukturę i składają się z par.
- Para zawiera dwie komórki. Każda z komórek może magazynować konkretną daną lub wskazywać na kolejną parę.
- Pary możemy łączyć nie tylko w listę ale również w rozgałęzione struktury, które mogą naśladować drzewa lub grafy.
- Na początku opiszemy abstrakcyjny interfejs listy dla programowania funkcyjnego, a następnie podamy implementację w konkretnym języku programowania.

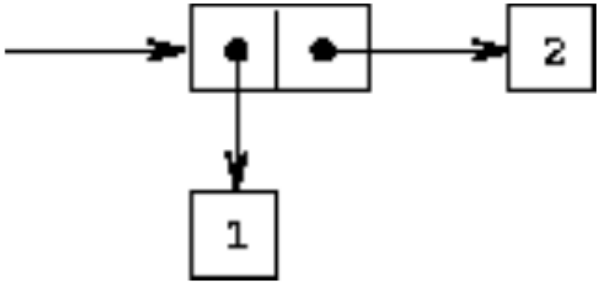
# Przykłady konstrukcji przy użyciu pary

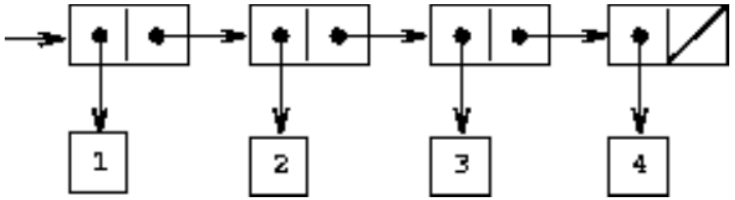



```
(cons (cons 1 2)
      (cons 3 4))
```




```
(cons (cons 1
            (cons 2 3))
      4)
```



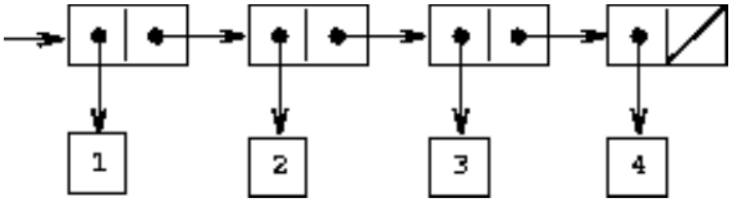



# Interfejs pary

- **Null –** jest symbolem „pustym”
- **Null? -** predykat (funkcja o wartościach logicznych) sprawdzająca, czy obiekt jest:
  - Null – predykat zwraca prawdę:  Null?(Null)=True
  - Nie null – predykat zwraca fałsz
- **pair**(M,N)  (często **cons**(M,N) ) **-** para o pierwszym elemencie M i drugim N
- **First** (często **car** od historycznego „**C**ontents of the **A**ddress part of the **R**egister”)  **-** funkcja zwraca pierwszy element pary:
  - first(pair(M,N)) = M
- **Rest** (często **cdr** od historycznego „**C**ontents of the **D**ecrement part of the **R**egister”) **-** zwraca drugi element (resztę) pary:
  - rest(pair(M,N)) = N
- Null?(pair(M,N)) = False

# Lista z par

- Konstrukcja listy [1,2,3, 4] z par:
  - **cons**(1,**cons**(2,**cons**(3,cons(4,Null))))
- Przypomina to raczej listę której drugim elementem jest lista, której drugim elementem jest ….
  - [1,[2,[3,[4]]]]



# Operacje na takiej liście

- Mamy listę:
  - **L= cons**(1,**cons**(2,**cons**(3,**cons**(4,Null)))) (=[1,[2,[3,[4]]]])
- Wówczas:
  - car(L) → 1
  - cdr(L) → **cons**(2,**cons**(3,**cons**(4,Null))) (=[2,[3,[4]]])
  - car(cdr(L)) → 2
  - cdr(cdr(L)) → **cons**(3,**cons**(4,Null)) (=[3,[4]])
  - car(cdr(cdr(L))) → 3
  - cdr(cdr(cdr(L))) → **cons**(4,Null) (=[4])
  - car(cdr(cdr(cdr(L)))) → 4
  - cdr(cdr(cdr(cdr(L)))) → Null

# Rekurencja na liście

- W programowaniu funkcyjnym nie ma pętli (for, while). Zamiast tych konstrukcji używamy rekurencji.
- Listy są idealne do przetwarzania przy pomocy rekurencji.
- Algorytm rekurencyjny:
  - **Przetwórz_Listę**(L):
    - Jeżeli null?(L) == true to przerwij rekurencję  # na liście nie ma już elementów
    - Zajmij się pierwszym elementem listy- car(L)
    - **Przetwórz_Listę**(cdr(L))  #przetwórz resztę listy

# Przykład – długość listy przy użyciu rekurencji ogonowej

- Algorytm (pseudokod) obliczający długość listy **L** przechodząc ją rekurencyjnie i zapamiętując jednocześnie w akumulatorze **n** liczbę elementów/wywołań rekurencyjnych:
  - **Lenght**(L, **n**=0):
    - If null?(L) == True:
      - return n
    - Else:
      - **Lenght**( cdr(L), **n+1**)
  - Lenght( L)  #wywołanie na liście L
- Więcej takich algorytmów poznamy na programowaniu funkcyjnym.

# Funkcje wyższego rzędy (higher order functions)

- W programowaniu funkcyjnym,funkcje wyższego rzędu są to funkcje które pobierają funkcje jako argumenty lub zwracają funkcje.
- Jedną z podstawowych funkcji występującą we wszystkich językach funkcyjnych i zastępującą pętlę for jest funkcja map. Służy ona do odwzorowania funkcji na listę:
  - map(2x , [1,2,3]) → [2,4,6]

# Założenia projektowe

- Lista będzie obiektem.
- Musimy zaimplementować operacje:
  - Cons()  - konstrukcja pary/listy
  - Car()  - pierwszy element listy
  - Cdr()  - pozostała część listy
  - Copy() - kopiowanie listy

# Implementacja - Python

# Python - lista

```
Null=None

def nullP( ls ):
    if (ls == Null):
        return(True)
    else:
        return(False)

def cons(a,b):
    return([a,b])

def car(ls):
    return(ls[0])

def cdr(ls):
    return(ls[1])

def Lenght(ls,n=0):
    if(nullP(ls)):
        return(n)
    else:
        return(Lenght(cdr(ls),n+1))
```

- l=cons(1, cons(2, cons(3, None)))
-
- print(car(l))
- print(cdr(l))
- print(car(cdr(l)))
- print(cdr(cdr(l)))
- print(Lenght(l))

# Python - Map

- Rekurencyjna implementacja Map:
  - def **Map**(f, ls):
  - print(ls)
  - if (ls != Null): return( cons(f(car(ls)), **Map**(f, cdr(ls))) )
  - else:
  - return(Null)
- Użycie:
  - l=cons(1, cons(2, cons(3, Null)))
  - def f(x):
    - return(2*x)
  - print(**Map**(f,l))

# Implementacja - C++

# Węzeł - C++

- class Node
- {
- public:
- Node();
- Node(const Node &);
- ~Node();
- int atom;
- char info;
- Node* next;
- Node* clink;
- };

- Atom – znacznik, czy węzeł ma wartość w info, czy wskaźnik clink;
- Info – dana
- Next – wskaźnik na kolejny węzeł
- Clink – central link  jest inicjowany not-null, gdy lewy element pary zawiera wskaźnik do węzła (wówczas to nie będzie lista).

# Węzeł c.d. - C++

- Node::Node() { atom = 1; info = '\0'; next = clink = NULL; }
-
- Node::~Node() {}
-
- Node::Node(const Node& nd)
- {
- atom = nd.atom;
- info = nd.info;
- next = nd.next;
- clink = nd.clink;
- }

# Lista - C++

- class List
- {
- private:
- Node* head;
- public:
- // Constructor
- List();
- // Copy constructor
- List(const List &);
- // Destructor
- ~List();

- // overloading =
- List& operator = (const List&);
- Node* copy(Node*);
- List cons(char,List&);
- List cons(List&,List&);
- List car(List&);
- List cdr(List&);
- void printNode(Node*);
- void printList();
- void release(Node*);
- };
-
- List::List() { head = NULL; }

# Konstruktory/Destruktory, = - C++

- List::List(const List& larg)
- {
- if(larg.head == NULL) head = NULL;
- else head = copy(larg.head);
- }
-
- List::~List()
- {
- if(head) { release(head); }
- delete head;
- }

- List& List::operator = (const List& larg)
- {
- if(larg.head == NULL) head = NULL;
- else head = copy(larg.head);
- return *this;
- }

# List::copy - C++

- Node* List::copy(Node* narg)
- {
- Node* res = new Node;
- if(narg != NULL)
- {
- res->atom = narg->atom;
- if(res->atom)
- {
- res->info = narg->info;
- res->clink = NULL;
- }
- else res->clink = copy(narg -> clink);
- res->next = copy(narg -> next);
- }
- else res = NULL;
- return res;
- }
-

# List::cons - C++

- List List::cons(**char newatom, List& oldlist**)
- {
- List res(oldlist);
- Node* newNode;
- newNode = new Node;
- newNode -> atom = 1;
- newNode -> info = newatom;
- newNode -> clink = NULL;
- newNode -> next = res.head;
- res.head = newNode;
- return res;
- }

- List List::cons(**List& newList, List& oldList**)
- {
- List res(oldList);
- Node* newNode;
- newNode = new Node;
- newNode -> atom = 0 ;
- newNode -> clink = copy(newList.head);
- newNode -> next = res.head;
- res.head = newNode;
- return res;
- }

# Car, cdr - C++

- List List::car(List& oldList)
- {
- List res(oldList);
- List empty;
- List carList;
- if(res.head -> clink == NULL)
- carList=cons(res.head->info,empty);
- else carList.head = copy(res.head->clink);
- return carList;
- }

- List List::cdr(List& oldList)
- {
- List res(oldList);
- res.head = res.head->next;
- return res;
- }

# Print, release - C++

```
void List::printNode(Node* narg)
{
Node* tmp;
tmp = copy(narg);
cout << " ( " ;
while(tmp != NULL)
{
if(tmp->atom) cout << tmp->info << " ";
else printNode(tmp->clink);
tmp = tmp->next;
}
cout << " ) " ;
}

void List::printList() { printNode(head); }
```

```
void List::release(Node* ptr)
{  Node* tmp;
{
if((tmp = ptr -> next) != NULL)
{ release(tmp); delete tmp;
tmp = NULL;}
if((tmp = ptr -> clink) != NULL)
{ release(tmp); delete tmp;
tmp = NULL;
}}}
```

# Main - C++

- int main(void)
- {
- List empty;
- 
- List L;
- L = L.cons('G',empty);
- cout << "L = "; L.printList(); cout << endl;
- 
- List M;
- M= M.cons('L', empty);
- M= M.cons('E', M);
- cout << "M = "; M.printList(); cout << endl;
- List Mcar = M.car(M);
- cout << "car (M) = ";
- Mcar.printList() ;
- cout << endl;
- 
- return 0;
- }

# Zadanie

- Popraw cons aby było możliwe:
  - M= M.cons('E', M.cons( 'K', M.cons('L', empty)));
  - Zamiast:
    - M= M.cons('L', empty);
    - M= M.cons('E', M);
- Popraw analogicznie car i cdr, aby było możliwe:
  - List Mcdrcdr = M.cdr(M.cdr(M));

# Literatura dodatkowa

- Abelson, Sussman, and Sussman, „ Structure and Interpretation of Computer Programs”, MIT Press:
  - https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html
- Ruedi Stoop, Alexandre Hardy, Yorick Hardy, Willi-Hans Steeb, „Problems & Solutions In Scientific Computing”, World Scientific

Koniec