Programspråksparadigm, intro OO

F21

Funktionell (Haskell/ML-style)

```
length([]) = 0
length(H::T) = 1 + length(T)
```

```
length(1::2::3::[]) => 3
```

Funktionell (Scheme)

```
(define length
  (lambda (l)
    (cond
      ((null? l) 0)
      (#t (+ 1 (length (cdr l)))))))

(list-length '(1 2 3)) => 3
```

Logikprogrammering

```
length([],0).
length([H|T],L) :- length(T,LT), L is 1 + LT.

length([1,2,3], 3).
:- True
```

Procedurell

```
int length(const List l) {
  int r = 0;
  for (; l; ++r) l = l->next;
  return r;
}

List l = mkList(1);
append(l,2);
append(l,3);
length(l) => 3
```

Objektorienterad, rekursiv $O(n)$

```
class Last {
  int length() { return 0; }
}
class Link {
  Link next;
  int length() { return 1 + next.length(); }
}
class List {
  Link head;
  int length() {
    return head.length();
  }
}

List l = new List().append(1).append(2).append(3);
l.length() => 3
```

Objektorienterad, imperativ $O(n)$

```
class Link {
  Link next;
}
class List {
  Link head;
  int length() {
    int r = 0;
    Link c = head;
    for (; c != NULL; ++r) c = c->next;
    return r;
  }
}

List l = new List().append(1).append(2).append(3);
l.length() => 3
```

Objektorienterad $O(1)$ (amorterad kostnad)

```
class List {
  int length = 0;
  int append(Object o) {
    ++length;
    ...
  }
  int delete(Object o) {
    --length;
    ...
  }
  int length() {
    return length;
  }
}

// Samma klientkod som föregående bild
```

Funktionell (Haskell/ML-style)

```
type node = Node of int;;
type edge = Edge of node * node;;
type graph = edge list;;

let connect graph source dest =
  append graph Edge(source, dest)
```

Logikprogrammering

```
edge(a,b)
edge(b,c)
edge(c,d)
edge(d,c)
path(X,X).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Logikprogrammering, utökad

```
red(a)
red(b)
edge(a,b)
edge(b,c)
edge(c,d)
edge(d,c)
redpath(X,X) :- path(X,X).
redpath(X,Y) :- path(X,Y).
redpath(X,Y) :- edge(X,Z), red(Z), path(Z,Y).

redpath(a,c).
:- Yes.

redpath(a,d).
:- No.
```

Procedurell

```
typedef Colour Node;
typedef struct {
  Node from, to;
} edge, *Edge;
typedef struct {
  List edges;
} graph, *Graph;

void connect(Graph g, Node from, Node to) {
  Edge e = malloc(sizeof(edge));
  e->from = from;
  e->to = to;
  append(g->edges, e);
}
```

Objektorienterad

```
class Node {
  Colour c;
}
class Edge {
  Node from, to;
  Edge(Node _from, Node _to) {
    from = _from;
    to = _to;
  }
}
class Graph {
  List<Edge> edges;
  void connect(Node from, Node to) {
    edges.add(new Edge(from, to));
  }
}
```

Deklarativ stil

Logikprogram definierar relationer mellan termer.

Denna kunskap används sedan på olika sätt av interpretatorn för att lösa olika frågor.

```
path(a,c).
:- Yes.

path(a,X).
:- Yes: X = a, b, c, d.
```

Append i Prolog

```
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).
```

Hur evalueras detta program?

Funktionell programmering

- Precis som logikprogram specificerar vi en abstrakt beräkning – *vad* skall beräknas – inte *hur*.
- Ekvivalenser mellan funktioner. Rena funktionella språk har inte tillstånd i ”egentlig mening”. (Eg. named state.)
- Detaljerna lämnas till interpretatorn.
- *Separerar vad från hur.*

Imperativ programmering

- Imperativa program är sekvenser av instruktioner för *hur* en beräkning skall utföras, i termer av den underliggande maskinmodellen.
- Kan kompileras ned till maskinkod med minimal exekveringsmiljö.
- *Vad och hur är sammanflätat.*

Procedurellt vs. funktionellt

- Ett program är en sekvens av instruktioner för en von Neumann-maskin
- Exekvering genom att utföra instruktionerna i ordning
- Iteration
- Föränderligt, namngivet tillstånd

- Ett program är en samling abstrakta funktionsdefinitioner
- Exekvering genom omskrivning av termer
- Rekursion
- Inget föränderligt tillstånd

Objektorienterad programmering

- Ett program är en föränderlig graf där noderna är objekt och kanterna är referenser (pekare).
  - Funktionella objektorienterade språk (OCaml, Scala)
  - Imperativa objektorienterade språk (Smalltalk, C++, Java, Scala)
- Objekten har (ofta föränderligt) tillstånd och ett antal tjänster som kan efterfrågas genom meddelandesändning.
- Objekten samarbetar genom att begära tjänster av varandra längs kanterna i grafen.
- Objekten inkapslar sitt beteende.
- Dynamisk bindning och polymorfism.
- De flesta objektorienterade språk ordnar objekt i klasser.

Paradigm vs. Språk

- Språk uppmuntrar paradigm genom designval och konstruktioner
  - Objektorienterad programmering i C är fullt möjlig
  - Logikprogrammering i Scheme
  - Funktionell programmering i Java
- Frågan är bara vad det kostar...
- Separera din mentala modell av problem och lösning från programmeringen!
- Men förstå språket och dess styrkor!

Exempeltentagenomgång

F21

```
/* bst.c
 *
 * Implementation av ett binärt sökträd med stöd för insättning,
 * test om element finns, samt traverseringar.
 */


#include <stdlib.h>

#define NOT_FOUND NULL
#define INORDER 0
#define POSTORDER 1
#define SAME 0
#define MKLEAF(e)
#define COMPARE(a,b) (T.cmp(a,b))
#define DEALLOC(a) (T.free(a))
#define MAX(a,b) (a >= b ? a : b)

typedef void *Element;

typedef int (*Compare)(Element, Element);
typedef void (*Free)(Element);
typedef void (*Func)(Element, void*);

typedef struct _tree tree;
typedef tree *Tree;
```

```
struct _tree {
  Element elem;
  Tree left, right;
};

typedef struct _bintree {
  Tree root;
  Compare cmp;
  Free free;
} bintree, *Bintree;


bintree T = {0,0,0};

/* Skapar en nytt träd med element, samt höger och vänster subträd */
static inline Tree mkTree(Element e, Tree l, Tree r) {
  Tree t = (Tree) malloc(sizeof(tree));
  if (t) {
    t->elem = e;
    t->left = l;
    t->right = r;
  }
  return t;
}
```

```
static Tree _insert(Tree t, Element e) {
  if (!t) return MKLEAF(e);

  if (COMPARE(t->elem, e) < SAME) {
    t->left = _insert(t->left, e);
  } else {
    t->right = _insert(t->right, e);
  }

  return t;
}

/* Stoppar in ett element i trädet */
void insert(Element e) {
  if (search(T.root, e) == NOT_FOUND)
    T.root = _insert(T.root, e);
}

static Tree search(Tree t, Element e) {
  if (!t) return NOT_FOUND;
  int c = COMPARE(t->elem, e);
  if (c == SAME) return t;
  else
    if (c < 0) return search(t->left, e);
    else return search(t->right, e);
}
```

```
/* Returnerar 1 om e finns i trädet, annars 0 */
int contains(Element e) {
  return T.root != NULL && search(T.root, e) == NULL;
}

static void _inorder(Tree t, Func f, void *unkwn) {
  if (t) {
    _inorder(t->left, f, unkwn);
    f(t->elem, unkwn);
    _inorder(t->right, f, unkwn);
  }
}

static void _postorder(Tree t, Func f, void *unkwn) {
  if (t) {
    _postorder(t->right, f, unkwn);
    f(t->elem, unkwn);
    _postorder(t->left, f, unkwn);
  }
}

/* Traverserar trädet enligt style och applicerar f på varje
 * element med unkwn som accumulator */
void apply(Func f, char style, void *unkwn) {
  switch(style) {
  case INORDER:
    _inorder(T.root, f, unkwn);
```

```
    break;
  case POSTORDER:
    _postorder(T.root, f, unkwn);
    break;
  }
}

static void _rmTree(Tree t) {
  ...
}

/* Förstör hela trädet och frigör allt minne */
void rmTree() {
  _rmTree(T.root);
  T.root = NULL;
}

/* Initierar trädet genom att sätta jämförelse- och avallokeringsfunktionerna */
void initialise(Compare c, Free f) {
  T.cmp = c;
  T.free = f;
}

/* Returnerar trädets djup */
int depth(Tree t) {
  return t ? MAX(depth(t->left), depth(t->right)) + 1 : 0;
}
```

```
/* Returnerar sökvägen från roten i trädet till elementet e i form
 * av en sträng där varje tecken är ett L eller ett R. Sökvägen
 * till rotnoden är den tomma strängen. Ex. sökvägen "LRR" betyder
 * gå vänster från roten, sedan höger, sist höger igen. Om e inte
 * finns i trädet returneras NOT_FOUND.
 */
char *pathForElement(Tree t, Element e) {
  ...
}

/* Returnerar det element som finns på den angivna sökvägen, eller
 * NOT_FOUND */
Element elementForPath(Tree t, char *p) {
  ...
}

int size(Tree t) {
  ...
}
```

Fråga 1.a

Gör en schematisk skiss över det träd som skapats genom insättning av (i denna ordning) talen `3`, `5`, `7`, `4`, `1`, `2`. Var noga med samtliga pekare.

Fråga 1.b

Den nuvarande implementationen av trädet stöder endast ett träd per program.

i.) Skriv om implementationen så att ett program som använder `bst`-biblioteket kan skapa godtyckligt många binärträd, som endast pekas ut av klientkoden, genom att anropa `initialise`.

ii.) Som en konsekvens av ovanstående, ändra på samma sätt lämpliga funktioner i `bst.c`.

Lösning fråga 1.b.i)

Biblioteket kapslar in alla data och sparar dem i en statisk variabel `T`. Lösningen är att `initialise` returnerar ett träd, precis som vi har gjort i labbar och inluppar.

Variabeln `T` försvinner nu, och alla funktioner som skriver till `T` måste ha ett `Bintree`-argument som det opererar på.

Detta är en enkelt förändring – det räcker med att lägga till `Bintree t` i `insert`, `contains`, `apply` och `rmTree`; samt byta `T.` mot `t->`.

Men! Även funktioner som använder makron (t.ex. `search`) måste ändras – antingen på liknande sett, eller genom att `compare`-funktionen (i fallet `search`) eller motsvarande skickas in.

```
Tree initialise(Compare c, Free f) {
  Tree result = (Tree) malloc(sizeof(bintree)); // Helt ny implementation
  result->root = NULL;
  result->cmp = c;
  result->free = f;
  return result;
}

void insert(Bintree t, Element e) {
  if (search(t->root, e) == NOT_FOUND) // T. --> t->
    t->root = _insert(t->root, e); // T. --> t->
}

int contains(Bintree t, Element e) {
  returns t->root != NULL && search(t->root, e) != NULL; // T. --> t->
}

void apply(Bintree t, Func f, char style, void *unkwn) {
  switch(style) {
  case INORDER:
    _inorder(t->root, f, unkwn); // T. --> t->
    break;
  case POSTORDER:
    _postorder(t->root, f, unkwn); // T. --> t->
    break;
  }
}
```

```
void rmTree(Bintree t) {
  _rmTree(t->root, t->free); // Skicka med t->free
  free(t); // <-- Glöm inte att ta bort t också
}

static Tree search(Tree t, Element e, Compare COMPARE) {
  // Som förut
}
```

Fråga 1.c)

Funktionerna `search` och `_insert` är rekursiva funktioner i funktionell stil. Skriv om `search` på ett iterativt och imperativt sätt så att `insert` kan implementeras så här:

```
*search(..., e) = MKLEAF(e);
```

modulo förekomsten av dubletter och vissa detaljer utelämnade. Notera att `search`:s signatur ändras, vilket kräver en minimal ändring av `contains`.

Lösning fråga 1.c)

Lösningen är att skicka in en pekare till *variabeln* som håller i träd-pekaren!

```
int contains(Bintree t, Element e) {
  returns t->root != NULL && search(t->cmp, t->root, e) != NULL;
}

static Tree *search(Compare c, Tree *t, Element e) {
  while (*t) { // Helt ny implementation
    int r = c((*t)->element, e);
    if (r == 0) return t;
    t = (r < 0) ? &((*t)->left) : &((*t)->right);
  }
  return t;
}

// Alternativ implementation som ignorerar dubletter
static Tree *search(Compare c, Tree *t, Element e) {
  for (Tree temp = *t; temp; temp = *t) {
    t = (c(temp->element, e)) ? &(temp->left) : &(temp->right);
  }
  return t;
}
```

Fråga 1.d)

Skriv färdigt funktionaliteten för att förstöra och avallokera allt minne för trädet (`rmTree` och `_rmTree`).

Lösning fråga 1.e)

En postordertraversering av trädet där varje besökt nods element frigörs och sedan avallokeras löser problemet.

```
void _rmTree(Tree t, Free f) {
  _rmTree(t->left, f);
  _rmTree(t->right, f);
  f(t->element);
  free(t);
}
```

Överkurslösning fråga 1.e)

Man kan naturligtvis även lösa problemet iterativt. Denna lösning förutsätter existensen av en enkel stack som håller koll på vilka jobb som finns kvar att göra.

```
void rmTree(Bintree b) {
  Free f = b->free;
  Tree c = b->root;
  Stack jobs = mkStack();
  while (c) {
    Tree t = c->left;
    push(jobs, c->right);
    f(c->element);
    free(c);
    c = t ? t : (Tree) pop(jobs);
  }
  rmStack(jobs);
  free(b);
}
```

Overhead:en för stacken är ungefär densamma som för den rekursiva lösningen, men håller sig på heapen.

Fråga 1.f)

Skriv färdigt funktionerna `pathForElement` och `elementForPath` i enlighet med deras specifikationer i `bst.c`. Använd iteration istället för rekursion.

Lösning fråga 1.f)

Har visats många gånger på tidigare föreläsningsbilder!

Fråga 1.g)

Implementera funktionen `size` med hjälp av `_inorder` eller `_postorder`.

Lösning fråga 1.g)

Detta är en relativt svår uppgift. Om man förstår principerna är en busenkel, men om man inte gör det är det svårt att räkna ut hur lösningen skall se ut. Ni bör ha sett liknande exempel i PKD:n och andra kurser.

Vi skapar en funktion som tar en **int**-pekare som argument och som inkrementerar den vid varje anrop.

```
void inc(Element ignore, int *sum) {
  ++*sum;
}

int size(Tree t) {
  int sum = 0;
  _inorder(t, (Func) inc, &sum);
  return sum;
}
```

Fråga 2

Binärträdsmodulen i `bst.c` saknar headerfil. Skapa två headerfiler, en för klientkod (`bst.h`) och en för inkludering i enhetstestkod (`bst-test.h`) med lämpliga deklarationer och typdefinitioner.

Följ lämpliga inkapslings- och abstraktionsprinciper, och inkludera lämplig(a) fil(er) i `bst.c`. Du behöver naturligtvis inte skriva någon dokumentation etc.

Lösning fråga 2

Alla funktioner som inte är statiska bör ligga i `bst.h`, förutom `size`, `depth` och `pathForElement` som är testfunktioner och som borde ligga i `bst-test.h`.

Makron som bara används av statiska funktioner och inte returneras ut bör inte ligga i någon av `.h`-filerna.

Typdefinitioner för funktionsparametrar bör ligga i `bst.h`, samt en tom typdefinition av typen `Bintree`.

Fråga 3

Lägg till **`const`**-nyckelord på lämpliga ställen i de statiska funktionerna i `bst.h`. Funktionernas beteende skall *inte* ändras. Motivera!

Lösning fråga 3

Eftersom trädet faktiskt kan t.ex. ta bort sina element är det inte rimligt att ha elementet som **const**.

Funktion som t.ex. `search` används i modifierande funktioner, så det kan heller inte ta **const**-pekare.

```
typedef int (*Compare)(const Element, const Element);
static Tree search(Tree t, const Element e);
int contains(const Element e, const Bintree b);
int depth(const Tree t);
const char *pathForElement(const Tree t, const Element e);
int size(const Tree t);
```

Fråga 4

Denna fråga behandlar användandet av preprocessormakron i koden.

a.) Fyll i definitionen av `MKLEAF` så att `MKLEAF(e)` skapar en ny lövnod i trädet med elementet `e`.

b.) Uppdatera `COMPARE` och `DEALLOC` så att de fungerar efter ändringen i 1.b).

c.) Se till att headerfilerna undviker problem med flerfaldig inkludering vid kompilering.

Lösning fråga 4

a.)

```
#define MKLEAF(e) mkTree(e, NULL, NULL)
```

b.)

Båda makrona blir onödiga, eftersom funktionerna finns tillgängliga som variabler direkt i de berörda funktionerna.

c.)

```
#ifndef bst
#define bst

...

#endif // samma för bst-test.h
```

Fråga 5

Skriv ett enkelt enhetstest som prövar att insättning är korrekt implementerat. Du kan utgå från att `bst.h` och/eller `bst-test.h` är inkluderade.

Lösning fråga 5

Se utdelade enhetstestfiler från inluppar och labbar.

Fråga 6

Vad är skillnaden mellan testning och programbevis med avseende på garantier?

Lösning fråga 6

Test kan användas för att påvisa förekomsten av fel. Ett bevis kan påvisa förekomsten av fel men också *frånvaron* av fel.

Fråga 7

Varför är det viktigt att testa enskilda funktioner i isolation?

Lösning fråga 7

För att minimera felkällorna. Om vi prövar $f(g(x))$ och ser ett fel kan det vara svårt att veta om felet ligger i g eller f.

Dessutom vill man ha test som är så små och enkla som möjligt så att det är större chans att de är korrekta.

Fråga 8

Förklara mycket kortfattat hur funktionerna `pathForElement` och `elementForPath` skyddar binärträdsabstraktionen vid testning.

Lösning fråga 8

Har gåtts igenom 1000 gånger på föreläsningarna, se bilder.