KTH Information and Communication Technology

ID1019
Johan Montelius

Programmering II (ID1019)
2018-03-13 08:00-12:00

Namn: ______________________________

Instruktioner

- Svaren skall lämnas på dessa sidor, använd det utrymme som finns under varje uppgift för att skriva ner ditt svar.
- Svar skall skrivas på svenska eller engelska.
- Du skall lämna in hela denna tentamen.
- Inga ytterligare sidor skall lämnas in.

Betyg

Tentamen har ett antal uppgifter där några är lite svårare än andra. De svårare uppgifterna är markerade med en stjärna, *[p*]*, och ger poäng för de högre betygen. Vi delar alltså upp tentamen in grundpoäng och högre poäng. Se först och främst till att klara de normala poängen innan du ger dig i kast med de högre poängen.

- E: 12 grundpoäng
- D: 15 grundpoäng
- C: 18 grundpoäng
- B: 20 grundpoäng och 8 högre poäng
- A: 20 grundpoäng och 10 högre poäng

De som skriver 4.5hp versionen så skall bara svara på frågorna 1-6. Gränsen för E, D och C är då 8, 10 och 12 poäng. Gränserna för B och A är 3 respektive 5 poäng.

Gränserna kan komma att justeras nedåt men inte uppåt.

Namn:________________________________ Persnr:________________________________

1 Lambdakalkyl [2p]

Evaluera följande lambdauttryck:

- $(\lambda x \to x + 5)4$ **Svar:** 9
- $(\lambda x \to (\lambda y \to x + 2 * y)3)5$ **Svar:** 11
- $(\lambda x \to (x)5)(\lambda z \to z + z)$ **Svar:** 10

2 Operationell semantik [2p]

Givet de regler för en operationell semantik som finns i appendix, visa steg för steg vilka regler som används och evaluera följande uttryck:

Svar:

$$\frac{y/a \in \{y/a\}}{E\{y/a\}(\mathtt{y}) \to a}$$

$$\frac{E\{\}(\mathtt{:a}) \to a \qquad S(\{y/b\}, \mathtt{y}) \to \{\} \qquad P\{\}(\mathtt{y}, a) \to \{y/a\} \qquad E\{y/a\}(\mathtt{y}) \to a}{E\{y/b\}(\mathtt{y\ =\ :a;\ y}) \to a}$$

$$\frac{E\{\}(\mathtt{:b}) \to b \qquad S(\{\}, \mathtt{y}) \to \{\} \qquad P\{\}(\mathtt{y}, b) \to \{y/b\} \qquad E\{y/b\}(\mathtt{y\ =\ :a;\ y}) \to a}{E\{\}(\mathtt{\ y\ =\ :b;\ y\ =\ :a;\ y}) \to a}$$

$$\frac{y/a \ in \ \{y/a\} \qquad a \not\equiv b}{P\{y/a\}(\mathtt{y}, b) \to \text{fail}}$$

$$\frac{P\{\}(\mathtt{y}, a) \to \{y/a\} \qquad P\{y/a\}(\mathtt{y}, b) \to \text{fail}}{P\{\}(\mathtt{\{y,y\}}, \{a, b\}) \to \text{fail}}$$

$$\frac{E\{\}(\mathtt{\{:a,\ :b\}}) \to \{a, b\} \qquad S(\{\}, \mathtt{\{y,y\}}) \to \{\} \qquad P\{\}(\mathtt{\{y,y\}}, \{a, b\}) \to \text{fail}}{E\{\}(\mathtt{\ \{y,y\}\ =\ \{:a,\ :b\};\ y}) \to \perp}$$

Namn:______________________________ Persnr:______________________________

3 Mönstermatchning [2 p]

Givet nedanstående uttryck, vad blir den resulterande omgivningen i de fall där möstermatchninging lyckas? (om du använder Erlang så skall vänster ledet vara `[X, Y | Z]`)

a: [x, y | z] = [1, 2, 3] **Svar:** x = 1, y = 2, z = [3]

b: [x, y | z] = [1, [2, 3]] **Svar:** x = 1, y = [2,3] z = []

c: [x, y | z] = [1 | [2, 3]] **Svar:** x = 1, y = 2, z = [3]

d: [x, y | z] = [1 | [2, 3] | [4]] **Svar:** syntaxfel

e: [x, y | z] = [1, 2, 3, 4] **Svar:** x = 1, y = 2, z = [3, 4]

Svar: Det var inte meningen att det skulle vara ett syntaxfel i fjärde exemplety så den utgår.

Namn:________________________________ Persnr:________________________________

4 Rekursion

ett binärt träd [2 p]

Implementear en funktion, `sum/1`, som tar ett träd och returnerar summan av alla värden i trädet. Trädet är representerat som följer:

```
@type tree :: {:node, integer(), tree(), tree()} |  nil
```

Svar:

```
def sum(nil) do 0 end
def sum({:node, v, left, right}) do
  v + sum(left) + sum(right)
end
```

svansrekursion [2 p*]

Den normala definitionen av `append/2` är inte svansrekursiv. Implementera functionen `reverse/1` svansrekursivt och använd den för att implementerar `append/2` svansrekursivt.

Svar:

```
def reverse(a) do reverse(a, []) end

def reverse([], b) do b end
def reverse([h|t], b) do
  reverse(t, [h|b])
end

def append(a, b) do reverse(reverse(a), b) end
```

5 Tidskomplexitet

spegla ett träd [2 p]

Om vi har nedanstående implementation av en funktion som speglar ett träd, vad är den asymptotiska tidskomplexiteten för funktionen?

```
def mirror(nil) do nil end
def mirror({:node, left, right}) do
   {:node, mirror(right), mirror(left)}
end
```

Svar: Tidskomplexiteten är $O(n)$ där n är antalet noder i trädet.

en kö [2 p*]

Antag att vi representerar en kö med hjälp av två listor och har nedanstående implementering av `enqueue/2` och `dequeue/1`. Vad är den amorterade tidskomplexiteten för att lägga till och sedan ta bort ett element ur kön?

```
def enqueue({:queue, head, tail}, elem) do
  {:queue, head, [elem|tail]}
end

def dequeue({:queue, [], []}) do :fail end
def dequeue({:queue, [elem|head], tail}) do
  {:ok, elem, {:queue, head, tail}
end
def dequeue({:queue, [], tail}) do
  dequeue({:queue, reverse(tail), []})
end
```

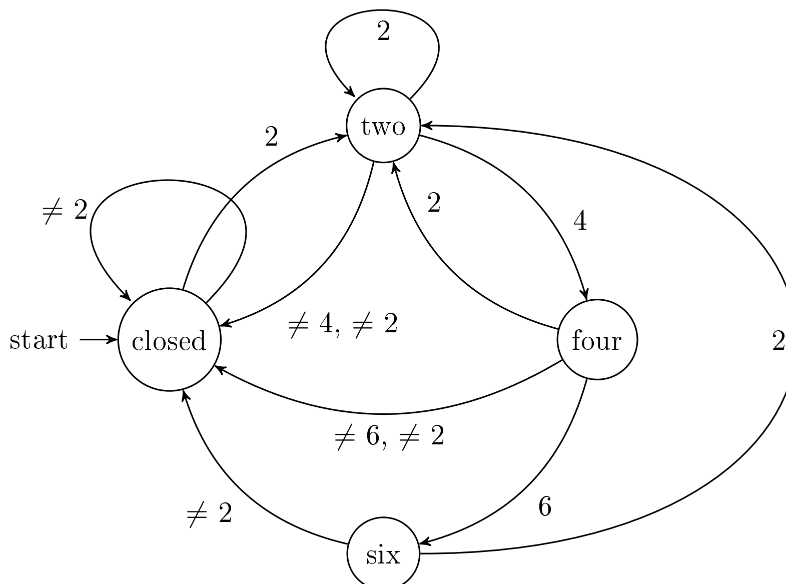
Svar: Den amorterade tidskomplexiteten är $O(1)$ eftersom vi har konstant tid att lägga till ett element, konstant tid för att flytta det till den främre listan (`reverese/1` är $O(n)$ men kostnaden kan amorteras på de n elementen) och konstant tid att ta ut elementet från den främre listan.

Namn:________________________________ Persnr:________________________________

6 Processbeskriving

TRB: two-four-six ... [2 p]

Givet nedanstående tillståndsdiagram, implementera en process som har det önskade beteendet.



Svar:

```
def start() do
  spawn(fn() -> closed() end)
end
```

```
def closed() do
 receive do
   2 -> two()
   _ -> closed()
 end
end
```

```
def two() do
 receive do
   2 -> two()
   4 -> four()
   _ -> closed()
 end
end
```

```
def four() do
 receive do
   2 -> two()
   6 -> six()
   _ -> closed()
 end
end
```

```
def six() do
 receive do
   2 -> two()
   _ -> closed()
 end
end
```

Namn:________________________________ Persnr:________________________________

tic-tac-toe [2 p]

Antag att vi har följande definition av procedurerna `first/1`, `second/1` och `third/1`.

```
def first(p) do
  receive do
     :tic ->
       second(p, [:tic])
     :tac ->
       second(p, [:tac])
  end
end

def second(p,all) do
  receive do
    :tic -> third(p, [:tic|all])
    :tac -> third(p, [:tac|all])
    :toe -> third(p, [:toe|all])
  end
end

def third(p, all) do
  receive do
     x ->  send(p, {:ok, [x|all]})
  end
end
```

Vad blir resultatet när vi exekverar anropet `test/0`?

```
def test() do
   self = self()
   p = spawn(fn()-> first(self) end)
   send(p, :toe)
   send(p, :tac)
   send(p, :tic)
   receive do
     {:ok, res} -> res
   end
end
```

Svar: `[:tic, :toe, :tac]`

Namn:_________________________________ Persnr:_________________________________

parallel summering [2 p*]

Implementera en funktion `sum/1` som tar ett binärt träd med tal i löven och summerar alla talen i trädet parallellt.

Svar:

```
def sum({:leaf, n}) do n end
def sum({:node, left, right}) do
  self = self()
  spawn(fn() -> n = sum(left); send(self, n) end)
  spawn(fn() -> n = sum(right); send(self, n) end)
  receive do
    n1 ->
      receive do
        n2 ->
           n1 + n2
      end
   end
end
```

Namn:____________________________ Persnr:____________________________

7 Programmering

En heap är en trädstruktur där det högsta elementet återfinns i roten av trädet och vars vänstra och högra gren också är en heap.

en heap [2 p]

Definera en datastruktur som är lämplig till att representera en heap och implementera en funktion `new/0` som returnerar en heap. Antag att vi bara skall hantera heltal.

- `@spec new() :: heap()`

Svar:

```
 @type heap() :: nil | {:heap, integer(), heap(), heap()}

  def new() do
    nil
  end
```

add/2 [2 p]

Implementera funktionen `add/2` som adderar ett tal till en heap.

- `@spec add(heap(), integer()) :: heap()`

För att hålla heapen balanserad så skall man växla höger och vänster grenar dvs, när vi skall lägga till ett element i en undergren så lägger vi till det i högra grenen men gör resultatet till den nya heapens vänstra gren.

Svar:

```
def add(nil, v) do
  {:heap, v, nil, nil}
end
def add({:heap, k, left, right}, v) when k > v do
  {:heap, k, add(right, v), left}
end
def add({:heap, k, left, right}, v)  do
  {:heap, v, add(right, k), left}
end
```

Namn:________________________________ Persnr:________________________________

pop/1 [2 p]

Implementera funktionen `pop/1` som plockar ut det högsta elementet ur en heap och returnerar antingen `:fail`, om heapen är tom, eller `{:ok, integer(), heap()}`

- `@spec pop(heap()) :fail | {:ok, integer(), heap()}`

Svar:

```
def pop(nil) do :fail end
def pop({:heap, k, left, nil})  do
  {:ok, k, left}
end
def pop({:heap, k, nil, right})  do
  {:ok, k, right}
end
def pop({:heap, k, left, right})  do
  {:heap, l, _, _} = left
  {:heap, r, _, _} = right
  if l < r do
    {:ok, _, rest} = pop(right)
    {:ok, k, {:heap, r, left, rest}}
  else
    {:ok, _, rest} = pop(left)
    {:ok, k, {:heap, l, rest, right}}
  end
end
```

swap/2 [2 p]

Implementera funktionen `swap/2` som tar en heap och ett tal och returnerar `{:ok, integer(), heap()}` där talet är det högsta talet och heapen den resterande heapen. Funktionen skall ha samma betydelse som att först göra `add/2` på talet och sen göra `pop/1` för att ta ut det högsta elementet men vi skall göra detta i en funktion, inte anropa de båda funktionerna.

- `@spec swap(heap(), integer()) {:ok, integer(), heap()}`

Svar:

```
def swap(nil, v) do
  {:ok, v, nil}
end
```

```
  def swap({:heap, k, left, right}, v) when k > v do
    {:ok, v, left} = swap(left, v)
    {:ok, v, right} = swap(right, v)
    {:ok, k,  {:heap, v, left, right}}
  end

  def swap(heap, v) do
    {:ok, v,  heap}
  end
```

en generell heap [2 p]

Den heap vi har implementerat nu kommer ha det största talet överst och är i övrigt begränsat till att arbeta med tal (eller det som kan jämföras med <). Implementera en funktion `add/3` som tar en heap, ett element och en funktion som kan jämföra två element med varandra. Funktionen `add/3` skall som tidigare lägga in elementet i en heap men då använda den erhållna funktionen vid jämförelse mellan element.

- `@type cmp() :: (any(), any()) -> bool())`
- `@spec add(heap(), any(), cmp()) :: heap()`

Svar:

```
def add(nil, v, _) do
  {:heap, v, nil, nil}
end
def add({:heap, k, left, right}, v, cmp) do
  if cmp.(v, k) do
    {:heap, k, add(right, v, cmp), left}
  else
    {:heap, v, add(right, k, cmp), left}
  end
end
```

Specificera en typ `cheap()` som håller en funktion för jämförelse och en heap. Implementera en funktion funktion `new/1` som tar en funktion och retunerar en structur av typen `cheap()`.

- `@spec new(cmp()) :: cheap()`

Svar:

```
@type cheap() :: {:cheap, cmp(), heap()}
```

```
def new(f) do {:cheap, f, nil} edn
```

Implemtera en funktion `add/2` som tar en struktur av typ `cheap()`, och som anropar `add/3` med rätt argument och returnerar en struktur på samma form.

- `@spec add(cheap(), any()) :: cheap()`

Svar:

```
def add({:cheap, cmp, heap}, v) do
  {:cheap, cmp, add(heap, v, cmp)}
end
```

mittersta talet

Du skall implementera en process som har ett tillstånd bestående av en mängd tal. Mängden är initialt tom men processen skall sedan kunna acceptera följande meddelanden:

- `{:add, integer()}` : talet skall läggas till mängden
- `{:get, pid()}` : processen skall svara med antingen `:fail`, om mängden är tom, eller `{:ok, integer()}` där talet är det mellersta talet i mängden (vid jämnt antal kan en av de två mellersta returneras) som också plockas bort från mängden.

För att komplicera problemet så skall de båda operationerna kunnas göras på $O(lg(n))$ tid, där n är antalet element i mängden. Du får inte använda några bibliotek för att spara mängden tal utan skall använda den implementation av en heap som gjorts i föregående uppgifter. Processen kommer förutom att hålla koll på det mittersta elementet ha två heap:ar, en för mindre tal och en för större tal.

tillstånds diagram [2*]

Börja med att rita ett tillståndsdiagram. Processesn skall ha tre tillstånd: `empty` då vi inte har några element i mängden, `left` då vi har ett mittenelement och möjligtvis ett element mer till "vänster" (som är mindre) än vad vi har till "höger" och, `right` då vi har ett mittenelement och möjligtvis ett element mer till "höger" än "vänster".

Svar:

Namn:_________________________________ Persnr:_________________________________

inga element [2*]

Implementera hur vi startar processen och dess beteende i sitt tomma tillstånd.

Antag att vi har ned anstående funktion definerader i en module `Heap`.

- `@spec new(cmp()) :: cheap()`

Svar:

```
defmodule Middle do

 def start() do
   spawn(fn() -> empty() end)
 end

 def empty() do
  receive do
    {:add, v} ->
      left = Heap.new(fn(x,y) -> x < y end)
      right = Heap.new(fn(x,y) -> x > y end)
      left(v, left, right)
    {:get, pid} ->
      send(pid, :fail)
      empty()
  end
 end
```

kanske fler till vänster [4*]

I processens "vänstra" tillstånd så har den ett mittersta element, ett antal element som är mindre (till vänster) och lika många eller ett färre som är större (till höger). Implementera processens betteende i dess vänstra tillstånd.

Antag att vi har ned anstående funktion definerader i en module `Heap`.

- `@spec add(cheap(), any()) :: cheap()`
- `@spec pop(cheap()) :: :fail | {:ok, any(), cheap()}`
- `@spec swap(cheap(), any()) :: {:ok, any(), cheap()}`

Svar:

```
 def left(m, left, right) do
 receive do
   {:add, v} when v < m ->
      {:ok, k, swaped} = Heap.swap(left, v)
      right(k, swaped, Heap.add(right, m))
   {:add, v}  ->
      right(m, left, Heap.add(right, v))
   {:get, pid} ->
      send(pid, {:ok, m})
      case Heap.pop(left) do
        {:ok, k, rest} ->
           right(k, rest, right)
        :fail
           empty()
      end
 end
end
```

Du behöver inte implementera processens högra tillstånd eftersom detta kommer likna dess vänstra tillstånd förutom att vi gör tvärt om.

Appendix - operational semantics

pattern matching

$$\frac{a \equiv s}{P\sigma(a, s) \rightarrow \sigma} \qquad \frac{a \not\equiv s}{P\sigma(a, s) \rightarrow \text{fail}}$$

$$\frac{v/t \notin \sigma}{P\sigma(v, s) \rightarrow \{v/s\} \cup \sigma} \qquad \frac{v/t \in \sigma \qquad t \not\equiv s}{P\sigma(v, s) \rightarrow \text{fail}}$$

$$\frac{v/s \in \sigma}{P\sigma(v, s) \rightarrow \sigma} \qquad \frac{}{P\sigma(_, s) \rightarrow \sigma}$$

$$\frac{P\sigma(p_1, s_1) \rightarrow \sigma' \wedge P\sigma'(p_2, s_2) \rightarrow \theta}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \rightarrow \theta}$$

$$\frac{P\sigma(p_1, s_1) \rightarrow \text{fail}}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \rightarrow \text{fail}} \qquad \frac{P\sigma(p_1, s_1) \rightarrow \sigma' \wedge P\sigma'(p_2, s_2) \rightarrow \text{fail}}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \rightarrow \text{fail}}$$

scoping

$$\frac{\sigma' = \sigma \setminus \{v/t \quad | \quad v/t \in \sigma \quad \wedge \quad v \quad \text{in} \quad p\}}{S(\sigma, p) \rightarrow \sigma'}$$

expressions

$$\frac{a \equiv s}{E\sigma(a) \rightarrow s} \qquad \frac{v/s \in \sigma}{E\sigma(v) \rightarrow s} \qquad \frac{v/s \notin \sigma}{E\sigma(v) \rightarrow \perp}$$

$$\frac{E\sigma(e_1) \rightarrow s_1 \qquad E\sigma(e_2) \rightarrow s_2}{E\sigma(\{e_1, e_2\}) \rightarrow \{s_1, s_2\}} \qquad \frac{E\sigma(e_i) \rightarrow \perp}{E\sigma(\{e_1, e_2\}) \rightarrow \perp}$$

$$\frac{E\sigma(e) \rightarrow t \qquad S(\sigma, p) \rightarrow \sigma' \qquad P\sigma'(p, t) \rightarrow \theta \qquad E\theta(\text{sequence}) \rightarrow s}{E\sigma(p = e; \text{sequence}) \rightarrow s}$$

$$\frac{E\sigma(e) \rightarrow t \qquad S(\sigma, p) \rightarrow \sigma' \qquad P\sigma'(p, t) \rightarrow \text{fail}}{E\sigma(p = e; \text{sequence}) \rightarrow \perp}$$

$$\frac{E\sigma(e) \rightarrow \perp}{E\sigma(p = e; \text{sequence}) \rightarrow \perp}$$