Norges Teknisk-Naturvitenskapelige Universitet
Fakultet for Informasjonsteknologi, Matematikk og Elektroteknikk
Institutt for Datateknikk og Informasjonsvitenskap

Eksamen i TDT4165
Programmeringsspråk

Onsdag, 8. August, 2007, 09:30–13:30

Språk: Bokmål

Faglig kontakt under eksamen:

- Wacek Kuśnierczyk, Tlf 94894894

Hjelpemidler: C. Ingen trykte eller håndskrevne hjelpemidler er tillatt. Typegodkjent kalkulator er tillatt.

Eksamen er laget av: Wacek Kuśnierczyk

Eksamen er kontrollert av: Ole Edsberg

Generell Informasjon og Tips

Les følgende generelle kommentarer og hint før **før** du begynner å svare.

- Les *alle* oppgavene før du begynner å svare.
- Svar kort og konsist. Uklare og unødig lange svar vil gi dårligere uttelling.
- Skriv tydelig. Utydelig skrift kan få negative konsekvenser for karakteren, siden deler av svarene dine da kan bli misforstått, eller ikke forstått i det hele tatt, av sensorerene.
- I oppgaver hvor du blir bedt om å *forklare* eller *diskutere* svarene, vil korrekte svar uten diskusjon/forklaring ikke gi maksimal uttelling.
- All programmering må gjøres i Oz. Noen steder vil det bli spesifisert at du bare kan bruke elementer som fins i kjernespråket.
- Du får lov til å bruke funksjoner og prosedyrer som er tilgjenglige i Oz, som `FoldL`, `Map`, `Append` osv., bortsett fra oppgavene der du er eksplisitt bedt om ikke å bruke dem.

Gradering

Det er seks oppgaver, hver bestående av 3–6 deloppgaver. Oppgavene har forskjellig vanskelighetsgrad og krever forskjellige typer svar (kode, diskusjon). Uttelling for oppgavene (men ikke for deloppgavene) er gitt ved tittelen til hver enkelt oppgave.

Oppgave 1 (15%)

Del 1a

Hva er *prosedyre-abstraksjon*? Hva er de viktigste fordelene ved å bruke prosedyre-abstraksjon? Diskuter minst to slike fordeler.

Del 1b

Prosedyrene `FoldRight` og `FoldLeft` folder (prosesserer) en liste henholdvis høyre- og venstre-assosiativt. Her er fragmenter av definisjonene:

```
fun {FoldRight List Combine Transform Null}
   case List
   ...
end

fun {FoldLeft List Combine Transform Null}
   case List
   ...
end
```

I et kall til `FoldRight` eller `FoldLeft`, må:

- argumentet `List` være en liste;
- argumentet `Combine` være en to-arguments funksjon som skal brukes til å kombinere de påfølgende elementene i `List`;
- argumentet `Transform` være en ett-arguments funksjon som skal brukes til å transformere hvert element i `List`;
- argumentet `Null` være en verdi som det siste (i tilfellet `FoldRight`) eller det første (i tilfellet `FoldLeft`) elementet i `List` skal kombineres med.

For eksempel, kan funksjonen `Sum` som beregner den artimetiske summen av alle elementer i en liste implementeres ved bruk av `FoldRight` eller `FoldLeft` som følger:

```
fun {Sum List}
   {FoldLeft List
    fun {$ X Y} X+Y end   % Combine-funksjonen
    fun {$ X} X end       % Transform-funksjonen
    0}                    % Null-verdien
end

{Browse {Sum [1 2 3 4]}}
```

der `10` vises i browse-vinduet.

Implementer `FoldRight` og `FoldLeft` ved hjelp av fragmentene gitt over; du *må* bruke rekursjon eller iterasjon eksplisitt, du kan *ikke* bruke prosedyrer som `FoldL` eller `FoldR` som er tilgjengelige i Oz. Hold deg innenfor tyve linjer kode totalt. Er prosedyrene iterative eller rekursive? Er anvendelser av prosedyrene iterative eller rekursive prosesser? Forklar.

Del 1c

Funksjonen `Map` tar som argumenter en liste `List` og en ett-arguments-funksjon `Function`, og returnerer en liste hvor hver element er resultatet av en anvendelse av funksjonen `Function` på det tilsvarende elementet av `List`:

```
{Browse {Map [1 2 3 4] fun {$ X} X*X end}}
```

viser `[1 4 9 16]` i browse-vinduet.

Implementer `Map` ved bruk av dine `Fold`-funksjoner etter følgende mønster:

```
fun {Map List Function}
   {FoldX List
    fun ... end
    Function
    ...}
end
```

hvor `FoldX` må erstattes av en av `Fold`-funksjonene.

Kan både `FoldLeft` og `FoldRight` brukes? Hvis ikke, hvilken av dem kan ikke, og hvorfor?

Oppgave 2 (20%)

Gitt følgende fragment av et program hvor en dataabstraksjon blir brukt:

```
declare
X Y
Structure = {NewStructure}
UpdatedStructure = {Get {Get {Put {Put Structure 1} 2} X} Y}
```

Utførelse av programmet fører til at `X` blir bundet til `2` og `Y` blir bundet til `1`.

Del 2a

Hva er et *buntet* (eng: bundled) dataabstraksjon? Er strukturene produsert av `NewStructure` buntet eller ikke? Er ikke-buntede data absktraksjoner (abstrakte data typer) nyttige? Diskuter.

Del 2b

Hva er en *sikker* (eng: sikker) dataabstraksjon? Er strukturer produsert av `NewStructures` sikre eller ikke? Må en dataabstraksjon være buntet for å være sikker? Kan en sikker dataabstraksjon implementeres i en deklarativ beregningsmodell? Kan en deklarativ dataabstraksjon implementeres i en ikke-deklarativ beregningsmodell? Er ikke-sikre dataabstraksjoner nyttige? Diskuter.

Del 2c

Gi tre eksempler på lineære dataabstraksjoner og to eksempler på ikke-lineære dataabstraksjoner. Er dataabstraksjonen representert ved `NewStructure` lineær eller ikke? Hva slags dataabstraksjon kan den være?

Del 2d

Gi en implementasjon av `NewStructure` slik at `Structure` blir deklarativ, buntet og sikker. Bruk følgende mønster:

```
NewStructure = local
   ...
in
   ...
end

Put = fun
   ...
end

Get = fun
   ...
end
```

Løsningen må *ikke* overskride 30 kodelinjer totalt.

Del 2e

Diskuter alle mulige kombinasjoner av egenskapene buntet/ubuntet, åpen/sikker og deklarativ/ikke-deklarativ. Hvilke av dem er implementerbare, og hvilke er nyttige til hvilke formål? Hvilke av disse kombinasjonene kan ikke implementereres i en deklarativ beregningsmodell? Hvilke av disse kombinasjonene krever en samtidig (eng: concurrent) beregningsmodell, og hvilke av dem krever en samtidig beregningsmodell med delt tilstand (eng: shared state)? Forklar.

Oppgave 3 (20%)

Del 3a

Hva er *lat utførelse*? Hva er fordelene ved å bruke lat utførelse? Forklar.

Del 3b

I Oz kan late funksjoner enkelt defineres med nøkkelordet `lazy`. Men `lazy` er bare syntaktisk sukker: i kjernespråket brukes elementet `ByNeed`. Vis hvordan følgende definisjon kan uttrykkes i kjernespråket:

```
fun lazy {SumSquares N1 N2}
   N1*N1+N2*N2
end
```

Hint: pass på — i denne definisjonen er det ikke bare `lazy` som ikke tilhører kjernespråket.

Del 3c

Bruk lat utførelse og implementer funksjonen `ListEvenIntegers` som returnerer en liste over *alle* like heltall, fra og med 0 (det vil si, listen `[0 2 4 ...]`). `ListEvenIntegers` skal fungere som i følgende eksempel:

```
EvenIntegers = {ListEvenIntegers}
{Browse {ListItem EvenIntegers 1}}
{Browse {ListItem EvenIntegers 5}}
```

hvor `0` og `8` blir skrevet ut. Funksjonen `ListItem` tar som argumenter en liste og et positivt heltall, returnerer det elementet i listen som står på posisjonen gitt av heltallet (i en 1-basert listeindeksering, dvs. at det første elementet i listen har indeks 1). Implementer `ListItem`. (Du kan bruke `lazy` eller `ByNeed`, som du foretrekker. Du kan *ikke* bruke funksjoner tilgjengelige i Oz, som `List.nth` o.l.)

Del 3d

Hvilke elementer må inkluderes i den abstrakte maskinen (kjernespråktolkeren) for at programmer med bruk av lat utførelse skal kunne kjøres på den? Forklar følgende:

1. Hvordan blir den semantiske setningen $(\{\texttt{ByNeed}\ \langle x\rangle\ \langle y\rangle\},\ E)$ utført? (Her er $\langle x\rangle$ og $\langle y\rangle$ to identifikatorer, og E en omgivelse (eng: environment).)
2. Hva skjer hvis det oppstår behov (eng: need) for resultatet av en anvendelse av en lat funksjon? Hva betyr det at det er *behov* for en variabel? Gi to eksempler på kode der en variabel-identifikator er brukt, et hvor det er behov for den variabelen som er bundet til identifikatoren, og et hvor det ikke er det.
3. Hvilke to minnehåndteringsregler er nødvendige for å definere variabel-nåbarhet (eng: reachability) hvis beregningsmodellen inkluderer lat utførelse?

Del 3e

Forklar forskjellen i utførelse mellom følgende to linjer kode:

```
thread {Procedure Argument} end
{ByNeed Procedure Argument}
```

Oppgave 4 (15%)

Scheme er en elegant dialekt av Lisp, for det meste brukt i undervisning. I denne oppgaven tar vi for oss σ, en egendefinert, meget begrenset delmengde av Scheme.

Følgende er et kort program i σ, bestående av tre instruksjoner:

```
(define one 1)
(define (double-q n) (q (q n)))
(double-q one)
```

Den første instruksjonen — en definisjon — definerer at `one` skal ha verdien `1`. Den andre instruksjonen definerer at `double-q` skal være en funksjon med en parameter (`n`), som, når den anvendes på et argument, returnerer resultatet av å anvende funksjonen `q` på verdien returnert av en anvendelse av den samme funksjonen `q` på argumentet til `double-q`. (Funksjonen `q` er ikke definert i koden over.) Den tredje instruksjonen — et uttrykk — anvender funksjonen `double-q` på `one`, som definert over.

Syntaksen til σ er ganske enkel:

- et program er en sekvens av én eller flere instruksjoner;
- en instruksjon er enten en definisjon eller et uttrykk;
- en definisjon er enten en variabel-definisjon (som `(define variable ...)`) eller en definisjon av en unær funksjon (som `(define (function argument) ...)`;

- et uttrykk er enten en identifikator (som `one`), et numerisk literal (som `1`), eller en anvendelse;
- en anvendelse er et uttrykk som involverer identifikatoren til en funskjon og et verdi-uttrykk (som `(function one)` eller `(function (function one))`).

Del 4a

Skriv en EBNF-grammatikk for σ i henhold til syntaks-beskrivelsen over, og slik at koden over kan genereres av grammatikken din. Du kan bruke ikke-terminalene ⟨*identifier*⟩ og ⟨*numeral*⟩ som om de allerede var definert. Alle andre ikke-terminaler brukt i grammatikken din må være definert i den. Definer grammatikken på ovenfra-og-ned-måten — begynn med ikke-terminalen ⟨*program*⟩:

$$\begin{array}{rcl} \langle \textit{program} \rangle & ::= & \ldots \\ \langle \ldots \rangle & ::= & \ldots \\ & \ldots & \end{array}$$

Del 4b

Er grammatikken din kontekst-sensitiv? Er den tvetydig? Forklar hva begrepene *context-sensitive* og *ambiguous* betyr i forbindelse med grammatikker. Gi et eksempel på en ikke kontekst-fri grammatikk og på en tvetydig grammatikk.

Del 4c

Basert på grammatikken din, implementer en syntaks-sjekker for σ. Bruk følgende mønster:

```
fun {CheckSyntax Tokens}
   ...
end
```

Anta at syntaks-sjekkeren mottar fra en tokenizer en sekvens av tokens representert som en liste. I denne sekvensen er hver identifikator pakket inn i en record med merkelappen '`identifier`' og hvert tall-ord pakket inn i en record med merkelappen '`number`'. Koden over, for eksempel, ville bli returnert fra tokenizeren som følgende sekvens:

```
['(' 'define' identifier('one') number('1') ')'
 '(' 'define' '(' identifier('double-q') identifier('n') ')'
     '(' identifier('q') '(' identifier('q') identifier('n') ')' ')' ')'
 '(' identifier('double-q') identifier('one') ')']
```

Denne sekvensen må godkjennes av syntaks-sjekkeren din, mao. må

```
{Browse {CheckSyntax Program}}
```

vise `true` (hvor `Program` har sekvensen vist ovenfor som verdi).

Hint: du kan implementere syntaks-sjekkeren som én prosedyre, eller som en samling av prosedyrer, én prosedyre for hver regel i grammatikken. Løsningen din må implementeres med relasjonell programmering, og koden må *ikke* overskride to sider.

Anta at prosedyrene og funksjonene nødvendige for relasjonell programmering er tilgjengelige, mao. kan programmet ditt starte som

```
\insert solve.oz

declare
fun {CheckSyntax Tokens}
   {SolveAll fun {$} ... true end \= nil}
end

...
```

Oppgave 5 (20%)

En dobbelt-sidig kø (en *dequeue* eller *deque*) er en lineær abstrakt datatype som støtter lagring, henting og fjerneing av elementer i begge ender. I denne oppgaven skal du implementere den dequeue som tilbyr de følgende fire operasjonene:

- *push*: legger til et element helt i slutten av dequeuen;
- *pop*: henter og fjerner det siste elementet i dequeuen;
- *shift*: legger til et element helt fremst i dequeuen;
- *unshift*: henter og fjerner det første elementet i dequeuen;[1]

Del 5a

Spesifiser dataabstraksjonen ved hjelp av *invariant assertions*, som følger:

{ d er udefinert }
gjør d til en ny tom dequeue
{ $d = \langle\rangle$ }

{ $d = \langle v_1, v_2, \ldots, v_n \rangle$, $e = v$ }
push verdien av e på d
{ $d = ?$, $e = ?$ }

og så videre for *alle* operasjonene, hvor v_i er verdier, og d og e er variabler. (I svaret ditt må du erstatte spørsmålstegnene med riktig innhold.)

Del 5b

Implementer en funksjon `NewDequeue` som produserer nye, tomme dequeuer. Implementasjonen din må være buntet med eksplisitt tilstand, og må bruke *procedure dispatching*. I følgende eksempel:

```
Dequeue = {NewDequeue}              % dequeuen er nå []
{Dequeue push(a)}                   % dequeuen er nå [a]
{Dequeue push(b)}                   % dequeuen er nå [a b]
{Dequeue shift(c)}                  % dequeuen er nå [c a b]
{Browse {Dequeue pop($)}}           % dequeuen er nå [c a]
{Browse {Dequeue unshift($)}}       % dequeuen er nå [a]
{Browse {Dequeue unshift($)}}       % dequeuen er nå []
```

[1] Navnene på operasjonene er tatt fra Perl, hvor slike operasjoner brukes til å aksessere elementer i arrays.

vil verdiene `b`, `c`, og `a` bli vist, i den rekkefølgen. Bruk følgende mønster:

```
fun {NewDequeue}
   Content = ...
   fun {Push Item}
      ...
   end
   ...
in
   ...
end
```

Koden din må *ikke* overstige 40 linjer.

Del 5c

Er dequeues, slik du har implementert dem, en sikker datatype?

Oppgave 6 (10%)

Forestill deg en situasjon hvor et antall klienter er tilkoblet en enkelt sørver. Hver klient kan be sørveren om å beregne resultatet av en anvendelse av en funksjon på et argument. Både argumentet og funksjonen sendes til sørveren, og sørveren sender tilbake det beregnede resultatet.

Gitt følgende scenario:

```
Server = {NewServer}

Square = {NewClient square fun {$ X} X*X end Server}
Cube = {NewClient cube fun {$ X} X*X*X end Server}
Double = {NewClient double fun {$ X} 2*X end Server}
...

{Square.compute 4}
{Cube.compute 5}
{Double.compute 6}
...
```

Dette programmet lager først en sørver og et antall tilkoblede klienter (sørveren vet ikke på forhånd hvor mange klienter som skal tilkobles). Deretter ber den klientene om å utføre en beregning. Dette resulterer i at `square(argument:4 result:16)`, `cube(argument:5 result:125)`, og `double(argument:6 result:12)` blir vist i browser-vinduet.

Del 6a

Klienter er implementert som følger:

```
fun {NewClient Name Function Server}
   proc {Compute Argument}
      Result in
      thread
         {Server.call message(Argument Function Result)}
         case Result of Result then
```

```
            {Browse Name(argument:Argument result:Result)}
         end
      end
   end
in
   client(compute:Compute)
end
```

Er klientene deklarative objekter? Hvis ikke, hvorfor ikke? Forklar.

Del 6b

Sørveren er implementert som følger (noen deler av koden er skjult):

```
fun {NewServer}
   InputStream
   ...
   proc {Process Stream}
      case Stream
      of message(Input Function Output)|Rest
      then Output = {Function Input}
         {Process Rest}
      end
   end
   proc {Call Message}
      ...
   end
in
   thread ... end
   server(call:Call)
end
```

Sørveren mottar beskjedene sendt til den fra klientene i en enkelt strøm. Fullfør koden slik at sørveren fungerer som i eksempelet over. Er sørveren et deklarativt objekt? Hvis ikke, forklar hvorfor.

Del 6c

Hva er den minimale mengden utvidelser nødvendig for å gjøre den deklarative, sekvensielle beregningsmodellen (presentert i Ch. 2 i læreboka) i stand til å implementere sørveren på en velfungerende måte? Kan denne klient-sørver konfigurasjonen implementeres i en deklarativ beregningsmodell? Forklar.