Kontakter under eksamen:
Per Holager, tlf. 99 61 78 36
Ole Edsberg tlf. 95 28 15 86

Språk: bokmål

EKSAMEN I FAG
TDT4165 PROGRAMMERINGSSPRÅK
Lørdag 15. mai, 2004, 0900–1300

Hjelpemidler: D (Ingen trykte eller håndskrevne hjelpemidler er tillatt. Godkjent lommekalkulator er tillatt.)

Les hele oppgavesettet før du begynner å besvare det. Svar kort og klart: Er svaret uklart eller lengre enn nødvendig, trekker dette ned.

Sensuren faller 5. juni 2004.

Oppgaven er kvalitetssikret av faglærer og en ekstern kontrollør.

Per Holager (faglærer):

Håvard Engum (ekstern kontrollør):

Kommentar/løsningsforslag

Hvis det dukker opp ting i besvarelser som ikke er dekket i retteveiledningen, så må den sensor som oppdager det sende en epost til de andre sensorene slik at forslaget kan vurderes og evt. legges til retteveiledningen.

Del 1 Flervalg (hver underoppgave teller 2.86%)

- Denne oppgaven skal besvares på vedlagt avkrysningsskjema.
- Sett bare ett kryss for hver oppgave. Oppgaver med mer enn et kryss vil få null poeng. På avkrysningsskjemaet er det flere svaralternativer enn i oppgavesettet. Pass på å ikke krysse av for alternativer som ikke finnes i oppgavesettet.
- Kryss av med svart eller blå kulepenn. La krysset gå helt ut til hjørnet av ruta, men ikke utenfor den. Hvis du krysser feil, kan krysset slettes ved å fylle hele ruta med blekk. Når du skal slette et kryss på denne måten er det viktig at ruta blir helt dekket slik at ikke noe av det hvite papiret synes inne i ruta. Korrekturlakk er forbudt.
- Ikke brett svarskjemaet på noen som helst måte.
- Skriv studentnummeret pent to ganger.
- Ikke skriv noe i feltet merket "Eventuell ekstra ID".
- For hver oppgave, velg det ene svaret du mener er mest riktig.

1) Hvilke deler består en enkel kompilator av?

   A: En tolker og en parser.

   B: En leksikalsk analysator (eng: *tokenizer*, *lexical analysator*), en parser og en kodegenerator.

   C: En prekompilator og en parser.

   D: En prekompilator, en parser, en leksikalsk analysator og en virtuell maskin.

   Kommentar/løsningsforslag

   B er riktig.

2) Denne funksjonen bruker mønstergjenkjenning (eng: *pattern matching*) for å finne det første elementet i en liste. Hvilken linje er det korrekt å sette inn?

```
fun {First Xs}
   % Sett inn riktig linje her. <---
end
```

A: `case Xs of Xs.1 then true end`

B: `case Xs of Xs.1 then Xs.1 end`

C: `case Xs of X|_ then X end`

D: `case X of Xs.1 then X end`

Kommentar/løsningsforslag

C er riktig.

3) Hva ville være en konsekvens av å forandre enkelt-tilordningslageret (eng: *single-assignment store*) i det deklarative kjernespråket (DKL) slik at det tillot mer enn én tilordning (eng: *assignment*) til en variabel?

A: Høyere ordens programmering ville bli umulig.

B: Objekt-orientert programming ville bli umulig.

C: Programmering av deklarative komponenter ville bli umulig.

D: Programmerte komponenter ville ikke lenger være garantert deklarative.

Kommentar/løsningsforslag

D er riktig.

4) Hva skjer i den deklarative beregningsmodellen i (Oz når en semantisk setning `(raise <x> end, E)` utføres? (I hele denne oppgaven bruker vi samme notasjon for utførelsestilstand som i tolk-erøvingen.)

A: Ingenting. Den semantiske setningen er ugyldig.

B: Utførelsen består av følgende handlinger:

- Hvis stakken er tom, stopp utførelsen med en feilmelding. Ellers, popp neste element av stakken. Hvis dette elementet ikke er en `catch`-setning, stopp utførelsen med en feilmelding.
- La `(catch <y> then <s> end Ec)` være `catch`-setningen som ble funnet.
- Dytt `(<s>,Ec+{<y>->E(<x>)})` på stakken.

C: Utførelsen består av følgende handlinger:

- Hvis stakken er tom, stopp utførelsen med en feilmelding. Ellers, popp neste element av stakken.
  - Hvis dette elementet ikke er en `catch`-setning, dytt `(raise( <y> end, E)`, hvor `<y>` er elementet som ble poppet, på stakken.
  - Hvis dette elementet er en `catch`-setning `(catch <z> then <s> end Ec)`, dytt `(<s>,Ec+{<z>->E(<x>)})` på stakken.

D: Utførelsen består av følgende handlinger:

- Elementer poppes av stakken på leting etter en `catch`-setning.
  - Hvis en `catch`-setning `(catch <y> then <s> end Ec)` ble funnet, popp den fra stakken. Dytt så `(<s>,Ec+{<y>->E(<x>)})` på stakken.
  - Hvis stakken ble tømt uten at en `catch`-setning ble funnet, stopp utførelsen med en feilmelding.

Kommentar/løsningsforslag

Det skulle vært komma etter end i svaralternativene. D var ment korrekt. P.g.a de manglende kommaene blir vi nødt til å godkjenne A som riktig også.

5) Hvilke av programmeringspråkene Java, Prolog, det deklarative kjernespråket i Oz (DKL) og kjernespråket i Oz med eksplisitt tilstand (KLES) bruker statisk typesjekking?

A: DKL og KLES.

B: Alle.

C: Alle unntatt Prolog.

D: Java.

Kommentar/løsningsforslag

D er riktig.

6) Studer følgende program:

```
local X Y Z in
   fun {X Y} Y+Z end
   Y = 2
   Z = 3
   local Y Z in
      Y = 5
      Z = 7
      {Browse {X Y}}
   end
end
```

Hva ville vises i Browser-vinduet hvis Oz brukte dynamiske skop-regler? (Oz bruker til vanlig statiske skop-regler.)

A: 5

B: 8

C: 9

D: 12

Kommentar/løsningsforslag

D er riktig.

7) Hva vil vises i Browser-vinduet etter at følgende buffer er blitt matet inn i Emacs-grensesnittet til Oz?

```
declare

fun {FoldL L F U}
   case L
   of nil then U
   [] X|L2 then
      {FoldL L2 F {F U X}}
   end
end

{Browse {FoldL [1 2 3] fun{$ X Y} Y-X end 0}}
```

A: -6

B: 2

C: 6

D:
```
%************************ syntax error ************************
%**
%** expression at statement position
%**
%** in file ``Oz'', line 11, column 34
%** ------------------ rejected (1 error)
```

Kommentar/løsningsforslag

B er riktig.

8) Hvilke av de følgende to funksjonene vil kjøre med konstant stakk-størrelse?

```
fun {F1 A B}
   if B<0 then raise domainError end
   elseif B==0 then 1
   else A*{F1 A B-1}
   end
end

fun {F2 A B C}
   if B<0 then raise domainError end
   elseif B==0 then C
   else {F2 A B-1 A*C}
   end
end
```

A: Ingen av dem

B: F1 men ikke F2.

C: F2 men ikke F1.

D: Begge.

Kommentar/løsningsforslag

C er riktig.

9) Hvilket utsagn om følgende produsent/konsument-program er sant?

```
declare
fun {Produce N Limit}
   if N<Limit then
      N|{Produce N+1 Limit}
   else nil end
end
fun {Consume Xs A}
   case Xs
   of X|Xr then {Consume Xr A+X}
   [] nil then A
   end
end
local Xs S in
```

```
   thread Xs={Produce 0 500} end
   thread S={Consume Xs 0} end
   {Browse S}
end
```

A: Dette er et synkront program siden den ene tråden utføres etter den andre.

B: Variabelen `Xs` er delt mellom trådene, noe som gjør programmet uforutsigbart.

C: Konsument-tråden er synkronisert med produsent-tråden ved at konsumentens case-setning blokkerer sin utførelse inntil det neste strømelementet ankommer.

D: Dette programmet kommer ikke til å terminere.

Kommentar/løsningsforslag

C er riktig.

10) Kappløpsforhold (eng: *race conditions*) kan oppstå i parallellitetssmodellen (eng: *concurrency model*) til Oz hvis følgende er tilfelle:

A: Automatisk søppeltømming brukes.

B: Automatisk søppeltømming brukes ikke.

C: Eksplisitt tilstand brukes.

D: Eksplisitt tilstand brukes ikke og lat uførelse brukes.

Kommentar/løsningsforslag

C er riktig.

11) Hvilken metode for overføring av parametere simuleres i følgende prosedyre?

```
proc {IncrementAndBrowse A}
   B={NewCell A}
in
   B:=@B+1
   {Browse B}
end
```

A: "Kall-ved-variabel" (eng: *Call by variable*)

B: "Kall-ved-verdi" (eng: *Call by value*)

C: "Kall-ved-verdi/resultat" (eng: *Call by value-result*)

D: "Kall-ved-behov" (eng: *Call by need*)

Kommentar/løsningsforslag

B er riktig.

12) Hvilket av de følgende klassehierarkiene er lovlig?

A:
```
class AX meth m(...) ... end end
class BX meth n(...) ... end end
class A from AX meth m(...) ... end end
class B from BX end
class C from A B end
```

B:
```
class AX meth m(...) ... end end
class BX meth m(...) ... end end
class A from AX meth m(...) ... end end
class B from BX end
class C from A B end
```

C:
```
class AX meth m(...) ... end end
class BX meth m(...) ... end end
class A from AX end
class B from BX end
class C from A B end
```

D:
```
class AX meth m(...) ... end end
class BX meth n(...) ... end end
class A from AX meth n(...) ... end end
class B from BX meth o(...) ... end end
class C from A B end
```

Kommentar/løsningsforslag

A er riktig.

13) Studer følgende utsagn om ren Prolog (eng: *pure Prolog*) og det relasjonelle kjernespråket i Oz (RKL).

i): Alle programmer i ren Prolog kan oversettes til RKL.

ii): Alle programmer i RKL kan oversettes til ren Prolog.

iii): Prolog og RKL inneholder de samme programmeringskonseptene og er bare forskjellige i hvordan konseptene opptrer syntaktisk.

Hvilke av dem er sanne?

A: Kun i).

B: Kun ii).

C: Kun iii).

D: Alle tre.

Kommentar/løsningsforslag

A er riktig.

14) Studer følgende utsagn om den relasjonelle beregningsmodellen i Oz.

i) Det er mulig å kjøre flere søk samtidig.

ii) Det er mulig å kjøre et søk inne i et annet søk.

iii) Det er mulig å innkapsle et søk slik at resten av programmet blir beskyttet fra det.

Hvilke av dem er sanne?

A: Kun i).

B: Kun ii).

C: Kun iii).

D: Alle tre.

Kommentar/løsningsforslag

D er riktig.

Del 2 Spørsmål (hver underoppgave teller 6%)

Skriv ikke mer enn en side i normal skriftstørrelse som svar på denne delen.

a) Hva er lat utførelse (eng: *lazy execution*)? Nevn to mulige fordeler ved å bruke lat uførelse.

Kommentar/løsningsforslag

Lat utførelse er at en beregning ikke blir gjort før det er behov for resultatet. To eksempler på mulige fordeler:

- Man kan håndtere potensielt store eller uendelige datastrukturer.
- Man kan enkelt lage en produsent/konsument-situasjon som er drevet av konsumenten.

4 poeng for korrekt, kort og klar definisjon. (-1 for relativt små grader av uklarhet, -1 for lengde, -1 til -3 for relativt små grader av ukorrekthet)

For hvert av de to fordelene: 3 poeng for relevant fordel. (-1 for relativt liten mangel på relevans, -1 for lengde, -1 til -2 for relativt små grader av ukorrekthet)

b) Hva er en strøm (eng: *stream*)? Nevn to ting som kan gjøres med strømmer som ikke enkelt kan gjøres uten dem.

Kommentar/løsningsforslag

Oz-spesifikk definisjon: En strøm er en datastruktur som er som en liste bortsett fra at det siste tuppelet har en hale som er ubundet. Generell definisjon: En strøm er en datastruktur som representerer en sekvens av elementer og hvor man kan legge til elementer i den ene enden og hente dem ut i den andre enden.

Både Oz-spesifikke og generelle definisjoner må godtas.

Noen eksempler på ting som kan gjøres med strømmer:

- Deklarativt modellere tilstand som forandrer seg over tid.
- Deklarativ kommunikasjon mellom tråder.
- I Oz: Enkelt programmere køer.
- I Oz: Enkelt programmere høyrerekursjon uten å invertere akkumulatorverdi

Det er ikke så lett å argumentere for at noe er vanskelig å gjøre med *alle* andre teknikker enn strømmer. Derfor må studenter få full pott for å foreslå ting som er rimelig fornuftig bruk av strømmer.

4 poeng for korrekt, kort og klar definisjon. (-1 for relativt små grader av uklarhet, -1 for lengde, -1 til -3 for relativt små grader av ukorrekthet)

For hvert av de to ekseplene: 3 poeng for relevant eksempel. (-1 for relativt liten mangel på relevans, -1 for lengde, -1 til -2 for relativt små grader av ukorrekthet)

c) Hva er syntaktisk sukker? Nevn en fordel eller en ulempe ved bruk av syntaktisk sukker.

Kommentar/løsningsforslag

Syntaktisk sukker er syntaktiske elementer som kan defineres ved oversetting til resten av språket og som ikke tilbyr nye verktøy eller programmeringsteknikker men som kun har til hensikt å gjøre programmer kortere og/eller mer oversiktlige.

- Fordel: Kan øke lesbarhet av programmer.
- Fordel: Kortere kode.
- Ulempe: Gjør reglene for syntaks mer kompliserte og kan dermed tåkelegge sammenhengen mellom syntaks og semantikk.
- Ulempe: Mer komplisert parser.

Merk at det var spurt etter en fordel *eller* en ulempe. Derfor trenger man ikke å oppgi begge deler for å få full pott.

5 poeng for korrekt, kort og klar definisjon. (-1 eller -2 for relativt små grader av uklarhet, -1 for lengde, -1 til -3 for relativt små grader av ukorrekthet)

5 poeng for relevant fordel eller ulempe (-1 til -2 for relativt liten mangel på relevans, -1 for lengde, -1 til -3 for relativt små grader av ukorrekthet)

d) Hva er fordelen ved å gjøre feilhåndtering ved hjelp av unntak (eng: *exceptions*) framfor å gjøre det ved å kalle bruker-definerte prosedyrer når feil oppstår?

Kommentar/løsningsforslag

Mulige svar

- Når vi kaller en prosedyre vil utførelsen fortsette rett etter kallstedet når prosedyren er ferdig. Dette er ofte ikke ønskelig ved feilhåndtering siden kallet antakeligvis skyldes at det vi var i ferd med gjøre ikke lot seg gjennomføre. Med unntak vil utførelsen etter unntakshåndteringen i stedet fortsette utenfor koden som gikk feil. Da har vi muligheten til å starte koden som gikk feil fra begynnelsen av eller finne på noe annet. (Dette var svaret vi tenkte på da vi ga oppgaven.) Maks 10 poeng.
- Unntak lar oss håndtere feil på en enhetlig måte. Dette er i og for seg riktig, men brukerdefinerte prosedyrer lar oss gjøre det samme. Faktisk kan prosedyrer la feil håndteres på en mer enhetlig måteettersom man kan gjenbruke samme prosedyre flere steder. Ofte vil man i praksis kalle en feilhåndteringsprosedyre fra en catch-setning. Dessuten er det uklart hva som menes med enhetlig". Max 3 poeng.
- Unntak lar oss fange masse forskjellige exceptions med en catch-setning. Dette kan også gjøres med prosedyrer ved at samme prosedyre kalles fra forskjellige feilsituasjoner.
- Sparer arbeid som må gjøres ved eksplisitt feilhåndtering: Feilene maa i begge tilfeller må håndteres av brukeren. Exceptions sparer arbeidet med aa manuelt kvitte seg med stakken over neste catch-setning. At arbeid spares er et veldig generelt svar, og det ville vaert "riktig" nesten uansett hva fordelen med exceptions var. Maks 1 poeng.
- Unntak lar oss delegere ansvar for feilhåndtering til programmet som kaller komponenten som ga unntaket. Dette er lurt siden det kan være kaller som vet hva som må gjøres i tilfelle unntak. Dette skjer f.eks når mozart-systemet gir et system-unntak eller når en bibliotek-funksjon gir et unntak. Dette er i og for seg riktig og et godt poeng, men det samme kan gjøres med prosedyrer ved at kaller sender en prosedyreverdi for feilhåndtering som parameter til modulen eller funksjonen som skal delegere feilhåndtering. Max 5 poeng siden løsningen med brukerdefinerte prosedyrer ikke er helt opplagt og kanskje også litt kronglete.
- Mozart kaster unntak ved systemfeil og vi må bruke unntak for å håndtere disse feilene: Det var ikke dette vi tenkte på, og det stod ikke i oppgaven at spørsmålet var spesifikt for Oz. Vi maa anta hvis bruker-definerte prosedyrer brukes i stedet for unntak, saa gjelder dette for grunnsystemet ogsaa, f.eks ved at bruker sender en feilhaandteringsprosedyre til systemet ved oppstart av programmet. 0 poeng.
- Exceptions er allerede implementert i språket og det er lettere aa bruke disse enn aa lage ting selv: Prosedyrer er ogsaa stoettet i spraaket. 0 poeng.
- Forklaring av hvordan exceptions fungerer: Dette er ikke et relevant svar med mindre man ogsaa skriver hva som er fordelen framfor brukerdefinerte prosedyrer.

Poeng for denne oppgaven blir max-score for svaret gitt over (Hvis man tar med flere delvis riktige svar gis den høyeste scoren blant disse.) -1 til -2 poeng for lengde, -1 til -5 for ulike relativt små grader av uklarhet, -1 til -5 for ulike relativt små innslag av ukorrekthet (i tillegg til det som er tatt med i beregningen av max-poeng).

Del 3 Høyere ordens programmering (teller 6%)

Skriv en funksjon `Map` som tar som argumenter en liste `L` og en funksjon `F` og returnerer en liste hvor hvert element `E` i `L` er erstattet med {`F E`}.

For eksempel skal {`Map [1 2 3] fun{$ X} X+1 end`} returnere lista `[2 3 4]`.

Kommentar/løsningsforslag

Løsning:

Steg i	Miljø E_i		Lager σ_i
0			
1	`P1`	$\longrightarrow$	x_1
2	`P1`	$\longrightarrow$	$x_1 \rightarrow x_2 \rightarrow$ (`proc {$ T U V} T=U+V end`,$\emptyset$)
3	`X`	$\longrightarrow$	x_3
	`P1`	$\longrightarrow$	$x_1 \rightarrow x_2 \rightarrow$ (`proc {$ T U V} T=U+V end`,$\emptyset$)
$\vdots$	$\vdots$		$\vdots$
5	`X`	$\longrightarrow$	x_3
	`P1`	$\longrightarrow$	$x_1 \rightarrow x_2 \rightarrow$ (`proc {$ T U V} T=U+V end`,$\emptyset$)
	`P2`	$\longrightarrow$	$x_4 \rightarrow x_5 \rightarrow$ (`proc {$ T U V} T=X+V end`,{`X`$\rightarrow x_3$})
$\vdots$	$\vdots$		$\vdots$
10	`X`	$\longrightarrow$	$x_3 \rightarrow x_8 \rightarrow 1$
	`Y`	$\longrightarrow$	$x_6 \rightarrow x_9 \rightarrow 2$
	`Z`	$\longrightarrow$	x_7
	`P1`	$\longrightarrow$	$x_1 \rightarrow x_2 \rightarrow$ (`proc {$ T U V} T=U+V end`,$\emptyset$)
	`P2`	$\longrightarrow$	$x_4 \rightarrow x_5 \rightarrow$ (`proc {$ T U V} T=X+V end`,{`X`$\rightarrow x_3$})
11	`X, U`	$\longrightarrow$	$x_3 \rightarrow x_8 \rightarrow 1$
	`T`	$\longrightarrow$	x_7
	`V`	$\longrightarrow$	$x_6 \rightarrow x_9 \rightarrow 2$
12			$x_3 \rightarrow x_8 \rightarrow 1$
			$x_7 \rightarrow 3$
			$x_6 \rightarrow x_9 \rightarrow 2$

Tabell 1: Sekvens av utførelsestilstander for det hemmelige programmet.

```
fun {Map Xs F}
   case Xs
   of nil then nil
   [] X|Xr then {F X}|{Map Xr F}
   end
end

Retteveiledning:
10 poeng for 100% fungerende program. (-1 for Oz-spesifikt problem, syntaks), eller
5 poeng for nesten riktig men feilfungerende program, eller
3 poeng for forklaring av oppbyggingen av programmet.
```

Del 4 Semantikk (teller 6%)

Det finnes minst ett program i det deklarative kjernespråket (DKL) som vil bli utført på den abstrakte maskinen i 13 utførelsessteg

$$(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow \cdots (ST_{13}, \sigma_{13}),$$

hvor utførelsestilstandene (eng: *execution states*) etter noen av trinnene er de samme som i Tabell 1. Skriv et slikt program. (Tabell 1 hopper over enkelte steg. Steg som er resultat av oppdeling av sekvensiell sammensetting (eng: *sequential composition*) er hverken med i Tabell 1 eller i opptellingen som ga tallet 13.)

Kommentar/løsningsforslag

Det var en feil i tabellen. Prosedyreverdiene skal fortsatt være i lageret i de to siste trinnene.

Løsning:

```
local P1 in
   proc {P1 T U V} T = U+V end
   local X in
      local P2 in
         proc {P2 T U V} T = X+V end
         local Y in
            local Z in
               X = 1
               Y = 2
               skip
               {P2 Z X Y}
            end
         end
      end
   end
end
```

På eksamen ble det annonsert at det ikke var så viktig nøyaktig hvor mange steg utførelsen bestod av.

10 poeng for kjørbart program som gir riktige tilstander og 13 (+/- 1) trinn. (-1 for Oz-spesifikke problem, syntaks, -3 for lite avvik i tilstander, -5 for middels stort avvik i tilstander, -1 for feil i antall trinn større enn (+/- 1))

Studenter som ikke allerede har fått full pott for denne oppgaven får ett bonuspoeng hvis de påpeker de manglende prosedyreverdiene i de to siste trinnene.

Del 5 Grammatikker og parsing (hver underoppgave teller 6%)

I denne oppgaven skal du skrive i Oz en rekursiv nedstigningsparser for et lite subsett av `SELECT`-setninger i SQL definert ved følgende grammatikk:

```
<select stmt> ::= 'select' <name list> 'from' <name list> 'where' <expr>
<name list>   ::= <name> | <name> ',' <name list>
<expr>        ::= <name> '=' <value>
```

Anta at `'select'`, `'from'`, `'where'`, `','`, og `'='` har blitt redusert til atomer av en leksikalsk analysator (eng: *tokenizer*, *lexical analysator*) som også har laget tupler `name(L)` for `<name>` og `val(V)` for `<value>`, hvor `L` er et atom som svarer til navnet som brukeren skrev og `V` er et heltall likt konstant-verdien.

Ikke bruk tid på feilhåndtering. Den innebygde feilhåndteringen i Oz er tilstrekkelig for denne oppgaven. Bruk unifikasjon og lignende teknikker der det kan gjøre programmet kortere.

a) Er strukturen i grammatikken egnet for en rekursiv nedstigningsparser? Forklar. Hvis den ikke er egnet, beskriv ekvivalens-bevarende transformasjoner som kan gjøres for å komme fram til en egnet grammatikk. Vis de endrede definisjonene med EBNF-notasjon.

Kommentar/løsningsforslag

`<name list>` har to alternativer som begge starter med `<name>`; denne må venstre-faktoriseres. Den blir da:

```
<name list> ::= <name> [',' <name list>]
```

b) Skriv i Oz en funksjon `{Expr In ?Out}` som kjenner igjen en instans av `<expr>` fra starten av leksem-lista (eng: *token list*) `In`, for eksempel lista `[name(jon) '=' val(4) ...]`. Funksjonen må binde resten av lista til `Out` og returnere et tuppel, for eksempel `equal(jon 4)`, som på en fornuftig måte inneholder informasjonen fra den delen av lista som ble gjenkjent.

Kommentar/løsningsforslag

```
declare
fun {Expr In Ut}
        case In
        of name(N)|'='|val(V)|R then Ut=R erLik(N V)
        end
end

declare N
{Browse {Expr [name(jon) '=' val(4)] N}}
{Browse N}
```

c) Skriv i Oz en funksjon `{NameList In ?Out}` som kjenner igjen en instans av `<name list>` fra starten av leksem-lista (eng: *token list*) `In`, for eksempel lista `[name(f1) ',' name(f2) ...]`, binder resten av lista til `Out` og returnerer en liste av tupler, for eksempel `[name(f1) name(F2)]`, som på en fornuftig måte inneholder informasjonen fra den delen av lista som ble gjenkjent.

Kommentar/løsningsforslag

Det er en feil i oppgaven. Det skal være `[name(f1) name(f2)]`, ikke `[name(f1) name(F2)]`. Dette ble annonsert under eksamen.

```
declare
fun {NamList In Ut}
        case In
        of name(N)|R then
                case R
                of ','|R2 then navn(N)|{NamList R2 Ut}
                else Ut=R navn(N)|nil
                end
        else Ut=In nil
        end
end

declare N
{Browse {NamList [name(olav) ',' name(jens) ',' name(helge)] N}}
{Browse N}
```

d) Skriv i Oz en funksjon `{SelectStmt In ?Out}` som kjenner igjen en hel instans av `<select stmt>`, fra leksem-lista (eng: *token list*) `In`, for eksempel

```
['select' name(f1) ',' name(f2)
 'from' name(tab)
 'where' name(f1) '=' val(4) ... ],
```

binder en parameter `Out` til resten av leksem-lista (markert ved `...`) og returnerer et syntakstre som inneholder informasjonen fra den delen av leksem-lista som ble gjenkjent, for eksempel `select([name(f1) name(f2)] [name(tab)] equal(f1 4))`.

Kommentar/løsningsforslag

```
declare Lx=['select' name(f1) ',' name(f2)
                'from' name(tab)
                'where' name(f1) '=' val(4)
                ]
```

```
declare
fun {Select I1 Ut} S I2 F I3 B in
        if I1.1==’select’ then
                S={NamList I1.2 ’from’|I2}
                F={NamList I2 ’where’|I3}
                B={Expr I3 Ut}
                select(S F B)
        end
end

declare N
{Browse {Select Lx N}}
{Browse N}
```