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

Eksamen i emne
TDT4165 / SIF8028
Programmeringsspråk

Mandag 8. august 2005, kl. 0900 – 1300

BOKMÅL

Faglig kontakt under eksamen:
Ole Edsberg (mob.tlf. 95281586)

Oppgavesettet er kvalitetssikret av:
Jon Arvid Børretzen
Per Holager (faglærer)

Hjelpemidler: C – ingen trykte eller håndskrevne hjelpemidler er tillatt
Bestemt, enkel kalkulator er tillatt

Les hele oppgavesettet før du begynner å besvare det. Legg merke til at det sist i settet er et svar-ark for oppgave 1.
Det er angitt i prosent hvor mye hver oppgave og hver deloppgave teller ved sensur.
Svar kort og klart, og skriv tydelig: Er svaret uklart eller lengre enn nødvendig, trekker dette ned.
Hvis du mener en oppgave er uklar eller ufullstendig, skriv ned antagelser du gjør når du svarer.
Hvis ikke annet er oppgitt for en oppgave, gjøres all programmering i oz.

Oppgave 1 (20%) Flervalgsoppgave/ Teori

(Alle deloppgavene teller likt ved sensuren.) Fyll inn svarene på denne oppgaven på svararket på siste side.
Kun ett av svaralternativene på hver deloppgave er ment å være riktig. Er du i tvil, så velg svaret du tror fagstaben hadde tenkt seg. (Hvis det ikke står noe annet, er koden som følger skrevet i oz.)

a)

En Lisp-makro brukes først og fremst for å -
1: generere en liste verdier
2: generere Lisp-kode
3: optimalisere funksjonell programmering
4: pakke inn (eng.: *encapsulate*) en gruppe definisjoner

b)

Hvilken av de følgende operasjonene gjøres ikke i engangs-lageret (eng.: *single assignment store*)?
1: skape verdier
2: binde verdier til variable
3: binde variable til variable
4: skape variable

c)

Hvilken av de følgende utsagnene er del av det deklarative kjernespråket?
1: **thread** <*s*>$_1$ **end**
2: **case** <*x*> **of** <*pattern*> **then** <*s*>$_1$ **else** <*s*>$_2$
3: **for** <*x*> **in** <*y*>$_1$ **..**<*y*>$_2$ **do** <*s*>$_1$ **end**
4: **{SolveAll** <*f*>$_1$ **}**

d)

Hvilket programmeringsspråk er følgende programfragment skrevet i:

```
class Expr {public: static Expr* primary(void);
```

1: Prolog
2: Haskell
3: Java
4: C++

e)

Uttrykket *prosedyral abstraksjon* betyr:

1: prosedyrer som kan brukes til mange formål, i forskjellige sammenhenger

2: objekt-orientert

3: et hvilket som helst utsagn kan ekvivalent skrives i en prosedyre (funksjon) som deretter kalles

4: prosedyrer kan (bør) lages slik at mange interne detaljer ikke trengs i spesifikasjonen av dem

f)

I tilfelle du ikke husker dem, så er:

```
fun{FoldR X F S}
        case X
        of E|Xr then {F E {FoldR Xr F S}}
        else S
        end
end
fun{FoldL X F Ac}
        case X
        of E|Xr then {FoldL Xr F {F Ac E}}
        else Ac
        end
end
```

Gitt to funksjoner

```
declare fun{G1 L R} L|R end
declare fun{G2 L R} R|L end
```

Hvilket av de følgende kallene vil gi resultatet **[3 2 1]**:

1: **{FoldL [1 2 3] G1 nil}**

2: **{FoldL [1 2 3] G2 nil}**

3: **{FoldR [1 2 3] G1 nil}**

4: **{FoldR [1 2 3] G2 nil}**

g)

Gitt

```
declare fun{H L R} R-L end
```

Hva er da verdien av

```
{FoldL [1 2 3] H 1}
```

1: -5

2: -1

3: 1

4: 2

h)

Rent funksjonelle språk har ikke -

1: partielle verdier

2: mer enn 1 parameter i et funksjons-kall

3: statisk, sterk typing

4: lat utførelse (eng.: *lazy evaluation*)

i)

Gitt

```
declare E F G H
thread
        proc{X1 P Q S T}
            case T of more|Tr then
                R=P+Q  Sr in
                S=R|Sr
                {X1 Q R Sr Tr}
            end
        end
    in
        {X1 H 1 E F}
end
thread
        proc{X2 N P Q}
            Qr in
            Q=more|Qr
            case P of R|Pr then
                if N>0
                then {X2 N-1 Pr Qr}
                end
            end
        end
    in
        {X2 G E F}
end
G=11
H=1
for D in E do
    {Show D}
end
```

Denne koden vil skrive ut en serie tall som er

1: del av en aritmetisk serie

2: del av en geometrisk serie

3: del av Fibonacci-serien

4: en linje fra Pascals trekant

j)

Hva gjør setningen

{Exchange <*x*> <*y*> <*z*>**}**

1: binder *E*(<*z*>) til gammel verdi av *E*(<*x*>), setter ny verdi av *E*(<*x*>) til *E*(<*y*>)
2: binder *E*(<*x*>) til gammel verdi av *E*(<*y*>), setter ny verdi av *E*(<*y*>) til *E*(<*z*>)
3: binder *E*(<*z*>) til gammel verdi av *E*(<*y*>), setter ny verdi av *E*(<*y*>) til *E*(<*x*>)
4: binder *E*(<*y*>) til gammel verdi av *E*(<*x*>), setter ny verdi av *E*(<*x*>) til *E*(<*z*>)

Oppgave 2 (20%) Deklarativ programmering

Skriv en funksjon **Pasc** som returnerer Pascals trekant som en liste av lister. (Lag hjelpefunksjoner etter behov.) Funksjonen skal ta størrelsen på trekanten som parameter. Resultatet skal være litt rotert i forhold til det vi er vant med: **{Pasc 8}** skal for eksempel gi resultatet

```
[[1   1   1   1   1   1   1   1]
 [1   2   3   4   5   6   7]
 [1   3   6  10  15  21]
 [1   4  10  20  35]
 [1   5  15  35]
 [1   6  21]
 [1   7]
 [1]]
```

Oppgave 3 (15%) Høyere ordens & relasjonsmodellen

Lag et filter i relasjonsmodellen. Mer konkret, skriv en funksjon som tar en boolsk funksjon og ei liste som parametre, og som gir som resultat en funksjon uten parametre. Denne resultat-funksjonen skal altså gis som parameter til **SolveAll**, den skal bruke **choice**. Den skal returnere de elementene fra lista som gir **true** når de gis som parametre til den boolske funksjonen.

Oppgave 4 (25%) Parser

Gitt en syntax for lister av: lister og integer konstanter:

<*list*> ::= '**[**' <*seq*> '**]**' .
<*seq*> ::= | <*seq*> <*list*> | <*seq*> '**num**'.

her skal '**num**' forståes som et naturlig tall. Som vanlig antar vi at i innputten til lexer-modulen skal det på dette punktet være en serie siffer; ut av lexeren og inn i parseren kommer et tuppel **num(V)** hvor **V** er verdien til det naturlige tallet. En innputt-streng til leksikalanalysatoren kan f.x. være

[2 [3 []] 4 [[5] 6]]

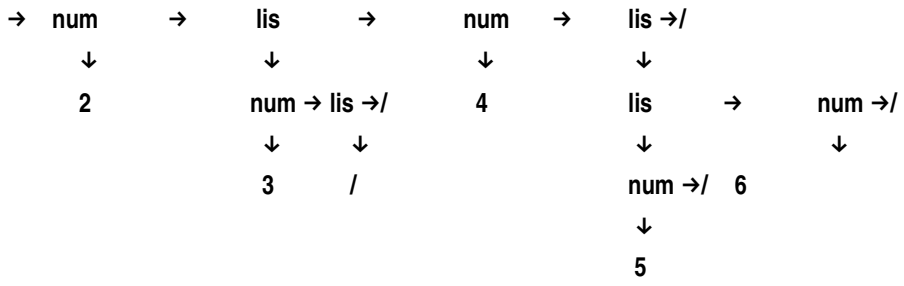
Resulterende innputt til parseren kan da være følgende liste av terminalsymboler:

['[' num(2) '[' num(3) '[' ']' ']' num(4) '[' '[' num(5) ']' num(6) ']' ']']

Resultatet fra parseren skal være et semantisk tre. Eksemplet vårt skal gi verdien:

```
num(    con:2
        nxt:lis(    con:num(    con:3       nxt:lis(    con:nil     nxt:nil ))
                    nxt:num(    con:4
                                nxt:lis(    con:lis(    con:num(    con:5       nxt:nil )
                                                        nxt:num(    con:6       nxt:nil ))
                                            nxt:nil ))))
```

Alternativt kan vi tegne dette:



Her har vi **/** i stedet for **nil**.

a) (5%)

Syntaxen over egner seg ikke når man skal lage en vanlig topp-ned, venstre-mot-høyre parser. Hvorfor ikke? Finn en ekvivalent syntax som er egnet. Prøv å gjøre den så enkel som mulig.

b) (20%)

Lag en rekursiv nedstigningsparser som aksepterer syntaksen du er kommet fram til under foregående deloppgave, og som produserer det semantiske treet beskrevet over. (Det semantiske treet er ordnet slik at det burde passe godt til syntaxen du er kommet fram til.)

Oppgave 5 (20%) Typesjekk

I denne oppgaven skal vi se på typesjekk for et veldig enkelt, strengt (eng. *strict*) typet språk. Riktig bruk av datatyper skal sjekkes etter parsing og før man begynner å utføre programmet. Den konkrete syntaksen (eng.: *concrete syntax*), dvs. syntaksen for det som går inn i lekser og parser, er ikke viktig for denne oppgaven: Vi antar at parseren genererer feilfrie semantiske trær (i læreboken: *abstract syntax trees*) på formen definert ved EBNF syntaksen som følger:

<*Program*> ::= '**[**' { <*Stmt*> } '**]**'

<*Stmt*> ::= **decl(ident** <*Type*> **)**
| **asg(ident** <*Expr*> **)**

<*Expr*> ::= **plus(**<*Expr*> <*Expr*>**)**
| **val(**<*Type*>**)**
| **ident**

<*Type*> ::= **int** | **bool**

Forklaring:

- **decl(ident <*Type*>)** betyr at en variabel med navn **ident** erklæres med type <*Type*>.
- **ident** er en identifikator, altså variabel-navn, representert som et atom i oz. Når en identifikator brukes i en <*Expr*> eller **asg(...)**, skal den tidligere ha forekommet i en erklæring slik at <*Type*>-n er kjent.
- **asg(ident <*Expr*>)** betyr at når dette utføres, vil verdien av <*Expr*> tilordnes til variablen **ident**. Dette er bare lov hvis typene til variablen og <*Expr*>-en er like.
- **plus(<*Expr*> <*Expr*>)** betyr at man under utførelse får en verdi ved å addere sammen verdiene av de to <Expr>-ene. Dette er bare lov hvis begge har type **int**, da er typen til resultatet **int**.
- **val(<*Type*> V)** representerer en konstant av **<*Type*>** med verdi **V.** Denne siste trenges ikke for typesjekk.

Et eksempel på et slikt semantisk tre:

[decl(a int) decl(b int) asg(b plus(a val(int 3)))]

Her erklærer man altså først variablene **a** og **b**, begge av type **int**. Så tilordner man verdien av **a + 3** til variabel **b**.
For å holde rede på identifikatorer og typer kan du bruke følgende funksjoner fra standard-biblioteket:

initiering:
```
declare D={NewCell nil}
D:={NewDictionary}
```

for å notere en ny variabel:
```
{Dictionary.put @D Ident Type}
```

for å finne typen til en variabel:
```
Type={Dictionary.get @D Ident}
```

a) (5%)
Forklar hvordan du vil organisere koden i følgende deloppgave.

b) (15%)
Lag et program som kontrollerer at slike semantiske trær bruker typene på riktig måte. Lag spesielt en prosedyre **Check** som tar treet og evt. 'dictionary' e.l. som parametre, og som reiser et unntak som beskriver den første feilen, (feilmelding i form av atom eller tuppel-label) om det er noen.

Studentummer: . . . . . .

Dato:. . . . . Side.:. . . . Antall ark: . . .

Svarark for Oppgave 1, Flervalgsoppgave

	svaralt.:	1	2	3	4
deloppg					
a)					
b)					
c)					
d)					
e)					
f)					
g)					
h)					
i)					
j)					

Husk å levere inn denne siden sammen med resten av besvarelsen din!