INF-122 Obligatorisk oppgave 1, H-2016

Fremgangsmåte

- Løsningen leveres elektronisk til en katalog "Oblig-1" under "Innlevering/Vurderingsmappe/" på kursets hjemmeside.
- Løsningen skal leveres i en zip fil med navn som "FORNAVN.ETTERNAVN.zip".
- Innleveringsfristen er ***MANDAG, 17 Oktober, kl.12:00***. Det blir *ikke* mulig å forbedre karakteren – man må få innleveringen godkjent ved første forsøk.
- Eneste akseptable formatet er ren tekstfil (bruk helst kun ASCII tegn), som inneholder kjørbar kode. Kode som ikke kan kjøres gir et stryk/F. (***Ditt navn***, samt eventuelle tilleggsopplysning/-forklaring ***skal stå som kommentarer*** øverst i programteksten.)
- Programmet ditt trenger bare å virke på syntaktisk korrekt input, men du får ekstra uttelling for informative feilmeldinger under evaluering.
- Kommenter ikke-åpenbare "triks" i koden din (uten å gjenta oppgave-teksten).
- Kommenter også nøye alle eventuelle presiseringer du har gjort mht. evt. tvetydighet og synlighetskonvensjoner. Oppgaven forklarer grammatikken og regler for rekkevidden av deklarasjoner, men dersom du må gjøre ekstra antakelser, formuler de kort i kommentarer.
- Når oppgaven ber deg om å lage en funksjon med et gitt navn eller en bestemt type, er det ***vesentlig*** at du gir den akkurat det navnet/den typen.
- Tilnærmingsmåte og forslagene kan diskuteres med hverandre, men hver enkel må levere inn en uavhengig løsning som ikke må ha noen identiske deler med andres. Løsninger som har slike identiske deler blir vurdert som stryk/F. Hva som teller som 'identisk' bestemmes av lærer.
- Les nøye *hele* oppgavesettet før du begynner å løse enkelte deloppgaver.

  Oppgaven har flere deler, hvor de senere bygger på de foregående. Hvis du besvarer hele oppgaven er resultatet et program, og dette kan leveres inn uten å ha egne filer for de første deloppgavene. Men dette programmet må virke! Hvis du har jobbet med en deloppgave uten å ha fått den helt til, så legg den ved i en separat fil slik at vi kan evaluere det du har fått til. Skriv i en innledende kommentar, hvilke deloppgaver du har besvart.
- Spørsmål om oppgaven kan stilles på gruppen: **Mandag/Torsdag/Fredag, 10/13/14 oktober**.
- Det er ingen forelesning Torsdag/Fredag, 13/14 Oktober.
- Gjennomgang av løsning på gruppen: Mandag/Torsdag, 17/20/21 Oktober.

1 Tester

Som et tillegg til obligen kommer det en fil kalt `Oblig12016Tests.hs`. Denne inneholder noen tester, skrevet i rammeverket HUnit. *Å bruke testene er helt valgfritt, de er utelukkende laget for å gjøre utviklingen enklere for deg.* For å bruke testene må du

- kalle besvarelsen for `Oblig12016.hs`,
- ha `Oblig12016Tests.hs` i samme mappe som `Oblig12016.hs`,
- skrive `module Oblig12016 where` helt øverst i besvarelsen, og
- åpne `Oblig12016Tests.hs` i haskell-tolkeren.

Etter å ha gjort dette, bruker du funksjonen runTestTT med testene som argument, f.eks `runTestTT testBasics` for å teste grunnleggende funksjonalitet (tilsvarende for `runTestTT testC`, `runTestTT testB` og `runTestTT testA`). **NB!** At alle testene i `testA` passerer garanterer ikke at du får A, men de må passere for å få A.

Det er en god idé å legge inn noen flere tester etterhvert som du løser deloppgaver. Det gjør det enklere å unngå regresjon (at noe som virket slutter å virke uten at du merker det).

2 Oversikt

Du skal implementere parser og evaluator for et enkelt funskjonelt programmeringsspråk i flere steg. Syntaksen til språket består av entydige blokker (`Block`) av uttrykk (`Expr`) og er gitt ved den følgende grammatikken Gr i EBNF, der `<Block>` er startsymbolet:

```
      <Char>  ::=  a | ...| z | A | ...| Z
     <Digit>  ::=  0 | ...| 9
    <Number>  ::=  <Digit> | <Digit><Number>
      <Func>  ::=  + | - | * | /
       <App>  ::=  (<Expr>, <Expr>) <Func> | <Var> (<Expr>) | <Lambda> (<Expr>)
      <Bool>  ::=  (<Expr>, <Expr>) == | <Expr>, <Expr>) != | (<Expr>, <Expr>) < |
                   (<Expr>, <Expr>) >
   <Default>  ::=  otherwise
      <Case>  ::=  case <Bool> -> <Expr>, <Case> | case <Default> -> <Expr>.
       <Var>  ::=  <Char> | <Char><Var>
    <Lambda>  ::=  lambda <Var> ( <Expr> )
       <Set>  ::=  set <Var> <Expr>
      <Expr>  ::=  <Number> | <App> | <Case> | <Var> | <Set> | <Lambda>
     <Block>  ::=  <Expr>; | <Expr>; <Block>
```

Følgende skal gjelde for implementasjonen din gjennom **hele** oppgaven:

- Mellomrom (whitespace) brukes til å skille mellom strenger av bokstaver, men kan også brukes mellom andre symboler (tokens). Det er uvesentlig hvor mange eller få mellomrom som er med så lenge bokstav-strenger er skilt.
- **Grammatikken skal ikke endres - selv om du misliker den.**

Tre enkle eksempler på `<Block>` er
`(2, 4)+;`,
`((10, 3)-, (9, 1)+)-;` og
`((2, 3)-, 1)+; 5;`
Vi vil starte med en enkel parser og evaluator som bare kan tolke blokker av ett enkelt uttrykk, og ende opp med et språk som har alle ingrediensene for et fullverdig programmeringsspråk.

Evaluatoren er essensielt en transformator fra `<Block>` til `<Block>`. (Men første del er bare en ett enkelt postfiksbasert kalkulatorspråk.) Evaluatoren skal evaluere i følge disse reglene:

- Tall evalueres til seg selv.
- Var dvs. variabelnavn, evalueres til verdien de er tilordnet i minnet eller konteksten.[1] Denne verdien må være et tall eller (i siste seksjon 6) en funksjon.
- Andre typer utrykk i oppgaven slik som f.eks. Lambda, App, Set og If vil bli spesifisert gradvis i oppgaven. Implementasjon av parsingen og tolkningen av disse uttrykkene utgjør hoveddelen av oppgaven.

Ingen andre former er gyldige programmer, og evaluatoren bør i så fall gi en passende melding. (Dette presiseres nærmere i seksjoner **Feilhåndtering**.)

I starten vil du implementere støtte for de innebygde aritmetiske funksjonene $+$, $-$, osv. (med 2 argumenter), og evaluere uttrykk som `(2, 3)+;` og `(2, (20, 4)-)+;`. Vi vil så legge til støtte for en enkel kondisjonal, som lar oss evaluere uttrykk som `(2, case (0, 0)== -> 20, case otherwise -> 4.)+;`. Så legger vi til mulighet for å jobbe med enkle variabler og blokker av kode bestående av mer enn ett utrykk. F.eks. `set var 2; (2, var)+;` er en kodeblokk bestående av to uttrykk, hvor det første setter variablen var til 2 og den andre regner ut summen av 2 og var. I den siste biten skal du implementere funksjoner og funksjonskall v.h.a. lambdaer. Det lar oss skrive uttrykk som `lambda x ( (x, (x, 1)+)+);`, som definerer en anonym funksjon, og `lambda x ((x, (x, 1)+)+) (2);`, som anvender funksjonen på argumentet 2 og returnerer 5. Videre skal du også definere lambdaer som førstesklasses data, som betyr at de kan bindes til variabler via set.

Oppgave I: tokenize

Som en forberedelse til parsing i alle følgende oppgaver, begynn med å programmere funksjonen `tokenize::String -> [String]`, som returnerer en liste av GR-tokene fra inputstrengen. F.eks.,
`> tokenize "(1, 12)+;"` kan gi `["(", "1", ",", "12", ")", "+", ";"]`
Tokenize trenger kun å fungere for de delene av gramatikken du har implementert, så langt du er kommet i oppgaven. Se f.eks. delgramatikken for første del av implementasjonen på starten av neste side.

3 Et enkelt kalkulatorspråk (for bestått)

I denne oppgaven betrakter vi kun følgende delgrammatikken GR_1 av GR:

[1] Vi skal skille presist mellom minne og kontekst senere i oppgaven.

```
 <Digit>  ::=  0 | ...| 9
<Number>  ::=  <Digit> | <Digit><Number>
  <Func>  ::=  + | - | * | /
   <App>  ::=  (<Expr>, <Expr>) <Func>
  <Expr>  ::=  <Number> | <App>
 <Block>  ::=  <Expr>;
```

Denne deloppgaven går ut på implementere parsing og evaluering av uttrykk i dette språket. Selv om vi setter opp `<App>` til å ha vilkårlig antall argumenter, i denne deloppgaven bruker vi bare aritmetiske funksjoner som tar nøyaktig to argumenter. Gradvis i løpet av oppgaven vil grammatikken bli utvidet og du vil bli nødt til å utvide parseren og evaluatoren til å ta hensyn til utvidelsene.

Noen eksempler på uttrykk som skal kunne parses og evalueres (i denne oppgaven) er:

```
> (5,    5)+;     > (5, 5)+;
```

skal begge evaluere til `10`, mens

```
> ( (10, 2)-, 4)+;
```

skal evaluere til `12` (den tilsvarer $(10 - 2) + 4$ i infiks notasjon).

Grammatikken støtter ikke unær "-", så eneste måte å få negative tall på er ved å subtrahere noe fra 0. Følgende, f.eks., skal evaluere til -4:

```
> (0, 4)-;
```

3.1 Parsing

Parser funksjonen skal ha typen `parse::String -> Ast` og ta som input et program (en streng) og returnere et abstrakt syntakstre (AST) for programmet **når det er syntaktisk korrekt, som du kan anta at det er**. Denne funksjonen er bare en wrapper rund mer spesifikke funksjoner for parsing av enkelte typer uttrykk. Disse har en type `[String] -> (Ast,[String])`, og skal gradvis programmeres, muligens også endres, i de resterende oppgavene etterhvert som grammatikken utvides. Følgende datatypen **skal** brukes (i denne deloppgaven, bare de første fire konstruktorene):

```
data Ast =  Number Integer | Name String | App Ast [Ast] | Block [Ast] |
            Case Ast [Ast] | Bool Ast Ast Ast | Default | Set String Ast |
            | Lambda String Ast | Lambda String Ast
                                                    deriving (Eq,Show,Ord)
```

Noen tips for løsning:

- `isDigit::Char -> Bool` fra modulen `Data.Char` tester om et tegn er et siffer. [2]
- Husk at funksjon `read::String -> Integer` fra Haskell Prelude ikke leser noenting, men kun foretar typekonvertering. For eksempel, for å lese inn et tall, kan du bruke denne kun etter å ha først hentet relevant delstreng fra inputstrengen som faktisk representerer et tall.
- For å sjekke om en liste av strenger starter med et bestemt prefiks (f.eks., `(`), kan man bruke mønsteret `...("(":s)= ...`.

Oppgave II: parseBlock og parse .....

Skriv parser funksjonen `parseBlock::[String] -> (Ast,[String])`, som gitt en liste av tokens returnerer et par av det første full-parsede uttrykket, og det som eventuelt er igjen av strengen:

[2] Du vil altså trenge `import Data.Char` øverst i filen for å bruke `isDigit`.

```
> parseBlock (tokenize "(1, 12) +;") gir
(Block [App (Name "+") [Number 1, Number 12]], [])
> parseBlock (tokenize "((1, 12)*, 234)+;") gir
(Block [App (Name "+") [App (Name "*") [Number 1, Number 12], Number 234]],[])
> parseBlock (tokenize "(1, 12)+; skrot") gir
(Block [App (Name "+") [Number 1, Number 12]], ["skrot"])
```

Del gjerne parsingen av uttrykkene inn i flere funksjoner slik som vist på forelesningene om parsing. I denne deloppgaven lag i alle fall funksjonene (med nødvendige hjelpefunksjoner) som parser Expr og App uttrykk, nemlig:

`parseExpr::[String] -> (Ast, [String])` og
`parseApp::[String] -> (Ast, [String])`.

Kombiner så `parseBlock` og `tokenize` for å lage `parse::String -> Ast`. Du kan anta her at inputstrengen inneholder et syntaktisk korrekt `<Block>`, med et enkelt `<Expr>` avsluttet med ";" (og muligens noen "skrot" deretter), og da returnerer den Ast for dette uttrykket.

3.2 Programevaluering

I oppgave III skal du lage en evaluator for uttrykk i grammatikken GR_1. Alle slike uttrykk evaluerer til tall og reglene er at

- tall evaluerer til seg selv, mens
- `(b,c)f` evaluerer til resultatet av å anvende `f` (altså, +, *, - eller /) på resultatet av evaluering av `b` og av `c`;

Du står fritt til å løse denne oppgaven som du ønsker, så lenge du lager `run` funksjonen som spesifisert under. All testing av programmet vil gå via den. Vi vil dog anbefale at du bruker en hjelpefunksjon

```
eval ::  Ast -> Context -> Memory -> (Ast,Context,Memory).
```

`Context` og `Memory` er strukturener som skal brukes senere i oppgaven. Foreløpig, ville det klare seg med en enkel funksjon `eval::Ast -> Ast`, som tar inn ett parset `Ast` og gir ut `Ast` tilsvarende resultatet av å evaluere programmet. Men det er lurt å begynne med en gang med den generelle versjonen. Vi vil senere beskrive hvordan denne funksjonen kan utvides til å støtte minne og variabler (avsnitt 5.2). Inntil da, kan du bare kalle den med tomme lister og la den returnere disse.

Eksempler av kjøring med `eval` (fordi vi modellerer minne som funksjon kann ikke brukes `show/print`):

```
> fst3 $ eval (parse "(5, 5)+;") emptyContext emptyMemory
Number 10
> fst3 $ eval (parse "((5, 5)+, (2, 1) -)+;") emptyContext emptyMemory
Number 11
> fst3 $ eval (parse "4;") emptyContext emptyMemory
Number 4
```

3.3 Feilhåndtering

Generelt: dersom du oppdager noen tvetydigheter, spesielt i grammatikken, skal du kommentere de i besvarelsen og forklare kort hvordan de håndteres.

Parseren bør returnere en feil eller exception hvis det forsøkes å parse ett utrykk som ikke er lovlig i G_{R_1}. Dette er ikke et krav for å bestå, men en "bør" som kan hjelpe på karakteren. I denne delen, sikrer syntaktisk korrekthet muligheten til å evaluere uttrykket. Senere i oppgaven vil vi møte situasjoner der det ikke lenger er tilfelle og da "bør" din `eval`uator gi passende feilmeldinger.

Gjennom hele oppgaven skal vi gjøre det på enklest mulig måte, nemlig, ved et kall til Haskell `error`-funksjon med passende tekst. Det viktige er at du kaller denne funksjonen på riktige steder med en informativ feilmelding. Lag en kort oversikt over feil som håndteres av din implementasjon.

Oppgave III: eval og run .....

Programmer funksjonen `eval` som beskrevet over.

Programmer så en funksjon `run::String -> Ast` som for en gitt (syntaktisk korrekt) streng returnerer `Ast`'et resulterende fra parsing og evaluering av programmet, f.eks.:

```
> run " ((5, 5) +, (2, 1) -)+;"
Number 11
```

Når vi senere i oppgaven sier at et uttrykk `E` evaluerer til `x` eller returnerer `x`, så mener vi at `run "E"` returnerer `x`. Eval trenger som sagt ikke å ha signaturen brukt over, så lenge `run` er tilpasset slik at den kan ta inn en streng og returnerer et `Ast`.

4 Kondisjonaler

Grammatikken G_{R_1} utvides til G_{R_2} ved å legge til støtte for "case-setninger":

```
      ...        ...
   <Bool>  ::=  (<Expr>, <Expr>) == | (<Expr>, <Expr>) != | (<Expr>, <Expr>) < |
                (<Expr>, <Expr>) >
<Default>  ::=  otherwise
   <Case>  ::=  case <Bool> -> <Expr>, <Case> | case <Default> -> <Expr>.
   <Expr>  ::=  ...  | <Case>
```

Semantikken til et gyldig uttrykk på formen `case a == b -> c, case otherwise -> d.` er å først evaluere a og b. Dersom a = b, evaluer c og returner den verdien, ellers gå til neste case i listen. Dersom case otherwise nås så evaluer og returner d .

```
> case ((1, 2)+, 4) != -> 1, case otherwise -> 2.;
```

returnerer `Number 1`, mens

```
> case (2, (2, 0)+)< -> 1, case (1, 3)== -> 2, case (1, 1)> -> 3, case otherwise
-> 4.;
```

returnerer `Number 4`.

Oppgave IV: utvid parse og eval til Gr_2 .....

Utvid parseren din til å parse også `case`-utrykk. Eventuelle endringer til de tidligere implementerte funksjonene bør være minimale. Utvid så eval slik at den kan evaluere også slike uttrykk.

5 Variabler og sekvensering (for C eller bedre)

Vi skal nå innføre tilstandshåndtering i programmer. Grammatikken GR_2 utvides til GR_3:

```
    ...       ...
 <Char>  ::=  a | ...| z | A | ...| Z
  <Var>  ::=  <Char> | <Char><Var>
  <Set>  ::=  set <Var> <Expr>
 <Expr>  ::=  ...  | <Var> | <Set>
<Block>  ::=  <Expr>; | <Expr>; <Block>
```

Du skal utvide parseren og evaluatoren din til å håndtere alle utrykk i dette språket, nemlig kodeblokker bestående av én eller flere `<Expr>;`. Som før, antar vi at input er syntaktisk korrekt.[3]

Parser skal returnere nå ett `Ast` der `Block` har en liste med et `Ast` for hvert `Expr`. `Var`-iabel med navn `var` kan naturlig representers ved `Ast`'et `Name ‘‘var’’`.

`eval` skal utvides til å støtte variabler, samt kodeblokker med flere uttrykk som evalueres sekvensielt inntil det returneres verdien av evaluering av det siste uttrykket. F.eks.,

```
2; (3, 3)+;
```

evaluerer først 2, men deretter evaluerer (3,3)+ og returnerer resultatet av det siste, dvs., 6.

Kodeblokker med mer en ett `<Expr>` er kun nyttig sammen med `set` – denne lar oss tilordne verdier til variabler, som kan hentes og endres (muteres) senere i blokken. Et velformet `set`-uttrykk er på formen `set a b` – `eval` evaluerer da først `b`, og så setter `a` til denne verdien, som returneres. F.eks., `set var 3;` returnerer `Number 3`, mens

```
set var 3; set a (var, 2)+;
```

returnerer `Number 5`. `set` brukes både til å definere nye variabler og redefinere eksisterende, så

```
set var 3; set var (var, 4) + ;
```

er lovlig og returnerer `Number 7`. Det er et krav at idet et uttrykk evalueres, må man kunne evaluere alle nødvendige deluttrykk. Dermed er, f.eks. følgende ulovlig

```
set a a;
```

siden "den andre" `a` ikke har noen verdi når den forsøkes evaluert (og lagres i "den første" `a`).

Blokk og `set` begrenser ikke rekkevidden og danner ikke noen synlighetsbegrensinger. Blokken

```
 set a (1, set b 2)+; (b,4)+;
```

er helt i orden og bør evalueres til `Number 6`.

5.1 Tilstandsflyt

Det er viktig hvordan tilstand "beveger seg gjennom" alle språkkonstruksjonene. En generell regel er at alt som skal evalueres, evalueres fra venstre til høyre, og eventuelt endret tilstand blir sendt videre. Så `(set a 1, a)+;` returnerer `Number 2`, mens `(a, set a 1)+;` gir en `error`. For `case` er det viktig at bare testen og det riktige argumentet evalueres:

```
set a 0; case (set a (a, 1)+, 0)== -> set a (a, 1)+, case otherwise -> set a (a, 1)+.;
```

skal returnere `Number 2` og ikke `Number 3` (som ville resultert ved evaluering av alle argumentene).

5.2 Implementasjon

Du får nå bruk for de ekstra argumentene `Memory/Context` i funksjonen

[3] Spesielt, trenger du ikke å teste om reserverte nøkkelord, som `set` eller `lambda`, brukes som variabelnavn. I prinsippet, bør blokker som `set set 4; +(set, 2);` utelukkes, men det er ikke et krav i oppgaven.

```
eval ::  Ast -> Context -> Memory -> (Ast,Context,Memory).
```

Minne er en avbildning fra minneadresser til evaluerbare Ast, og representerer minnet i datamaskinen. Kontekst er en avbildning fra navn (variabler) til minneadresser. For å aksessere minnet definerer vi ett enkelt grensesnitt:

```
type Memory = (Integer, Integer -> Maybe Ast)
```

Det første komponentet i tupplet gir den neste frie addresen (vi starter med addressen 0) og vi definerer avbildningen i det andre komponentet som en funksjon som returnerer Nothing for manglende addresser og et AST med den lagrede verdien ellers. Vi kan bruke hjelpefunksjonen

```
emptyMem ::  Memory
emptyMem = (0, \ _ -> Nothing)
```

for manglende addreser, og implementere funksjoner med følgende signaturer

```
lookupMem ::  Memory -> Integer -> Maybe Ast
addToMem ::  Memory -> Ast -> (Integer, Memory)
```

for å hente ut og sette inn variabler i minnet. Dere skal ikke forandre signatur eller type synonym for minnet til funksjonene. Returverdien til funksjonen addToMem skal returnere addressen som ble oppdatert med den nye variabelinformasjonen. Hvis vi har lagt inn to variabler i minnet, så skal vi kunne hente ut verdiene fra addressene på følgende måte:

```
let (ptr1, mem) = addToMem emptyMem (Number 10) /* ptr1 = 0 */
    (ptr2,mem') = addToMem mem (Number 20) /* ptr2 = 1 */
 in lookupMem mem' ptr1
```

som skal gi resultatet

```
Just (Number 10)
```

Videre dersom vi prøver å hente ut en ikke eksisterende addresse,

```
lookupMem mem' 25
```

så skal vi få

```
Nothing.
```

Kontekst er en avbildning fra navn (variabler) til minneadresser. Grunnen til at vi har splittet de opp slik, og ikke har en direkte avbildning fra variabelnavn til Ast er at vi ønsker å gi programmer mulighet til å endre verdier til variabler, og denne oppsplittingen gjør det enklere.[4] Vi bruker samme konstruksjonen for konteksten som for minnet, men vi må deklarere det som en newtype slik at vi kan definere den som typeclass `Show` (fordi vi har `data Ast ...deriving Show`). Basert på dette kan vi lage en kontekst-type

```
newtype Context = Context (String -> Maybe Integer)
instance Show Context where
  show _ = ""
```

og hjelpefunsjonen

```
emptyCtx ::  Context
emptyCtx = Context (const Nothing)
```

Uthenting og innlegging/oppdatering av variabler må dere løse selv, men kan gjøres direkte i `eval` funksjonen. Eller med å definere funksjonen `addToCtx` og `lookupCtx` for sette inn/oppdatere og hente ut variabler i en kontekst.

Følgende gir ett eksempel på hvordan minne- og kontekts-systemet vårt vil fungere. Vi bygger på det tidligere eksemplet innsetting og uthenting fra minnet. De to minneposisjonene som

[4] Det kan hende at du klarer deg, fram til oppgave 6, med en direkte avbildning fra navn til Ast, uten Context. Det kan allikevel være lurt, med tanke på fortsettelsen, å beholde Context som et trivielt mellomledd i implementasjon av en slik direkte mapping.

ble satt defineres i språket vårt i variabler, som f.eks. med `set var 20; set b 10;`. Her allokeres en ny minneadresse for hver av `var` og `b` som fylles med henholdsvis verdiene `20` og `10`. Variabelnavnet linkes så til denne minneadressen i konteksten den ble definert i, før evalueringen av programmet fortsetter i den utvidede konteksten/minnetilstanden. Evalueringen returnerer en verdi, samt oppdatert kontekst og minnetilstand. Vi kan utvide minne-delen av `set var 20; set b 10;`-eksemplet med konteksten på følgende måte

```
let ctx = addToCtx emptyCtx "var" 0
let ctx' = addToCtx ctx "b" 1
```

da vil følgende gjelde for `ctx'`:

- minne inneholder dataen til `var`-variablen i addresse 0 og dataen til `b`-variablen i addresse 1,
- `ctx'` er slik at `lookupCtx ctx' "var"` gir `Just 0`, og
- til en lookup av `var` med `lookupMem mem (lookupCtx ctx' "var")`, gir `Number 10` som verdi.

5.3 Feilhåndtering

Når en variabel som ikke har fått tilordnet noen verdi blir prøvd evaluert får vi en feil, som f.eks. i uttrykkene `set a b;` eller `set a a;`. Din `eval` bør i det minste kunne oppdage slike feil.

Oppgave V: utvidelse til Gr_3

Utvid `parse` og `eval` funksjoner for å håndtere språket til Gr_3.

`eval` må utvides slik at den kan evaluere variabler ved å slå opp i kontekst/minnet, for å kunne evaluere verdien til variablene du har laget. Du må selv finne ut hvilken verdi ev kontekst/minnetilstand skal videresendes. Muligens, må du tilpasse litt din tidligere implementasjon.

6 Lambda (for B eller bedre, gjør etter 7. Oktober)

Denne delen bruker lambda uttrykk som dere skal lære om 7. Oktober. Vi anbefaler derfor at dere gjør den følgende delen av oppgaven etter denne forelesningen.

Implementasjon skal til slutt utvides til å støtte unære funksjoner v.h.a. `lambda`. Grammatikken utvides til den fulle Gr fra seksjon 2:

```
      ...       ...
    <App>  ::=  ...  | <Var> ( <Expr> ) | <Lambda> ( <Expr> )
 <Lambda>  ::=  lambda <Var> ( <Expr> )
   <Expr>  ::=  ...  | <Lambda>
```

Parseren skal utvides slik at alle uttrykk i Gr kan parses og `eval` skal utvides som beskrevet under for å støtte `lambda`-funksjoner. Et lambda-uttrykk, som definerer en funksjon, er gitt ved `<Lambda>` regelen, mens et kall er, i tillegg til det vi har fra før, et uttrykk på formen `<Lambda> ( Expr )`. Funksjoner definert ved `lambda` er førsteklasses objekter, så de kan også bindes som verdier til variabler. Dermed kan et funksjonskall ha en variabel – bundet til en funksjon – i første posisjon, slik utvidelse av `<Func>` over viser.

6.1 Evaluering av funksjons*definisjon*, `lambda x (E)`,

returnerer en funksjon med et argument `x` som kan brukes i kroppen `E`, dvs. et `Ast` bestående av `Lambda ‘‘x’’ E-Ast`, der det siste er Ast for funksjonskroppen `E`. Funksjonen lagret i et slikt Ast kan tenkes på som en anonym funksjon `\x -> E` i Haskell, og det er slik vi skal betegne resultatet av evaluering av fuksjonsdefinisjon.

De eneste variablene som kan brukes i `E` er de som eksisterer når funksjonen defineres (dette kalles leksikalsk rekkevidde, “lexical scoping”), og `x` introdusert av `lambda` er bland disse.[5] F.eks.,

- `lambda x ((x, x) +)` er korrekt og evaluerer til en funksjon tilsvarende `\x -> x+x`.
- `lambda y (lambda x ((x, y) +))` er korrekt (`x` og `y` er introdusert med hver sin `lambda`) og tilsvarer funksjonen `\y -> \x -> x+y`.

Funksjonen skal “huske” hvilke variabler som eksisterte når den ble lagd. I det siste eksemplet, er det indre lambda-uttrykket `lambda x ((x,y)+)` korrekt, fordi “det husker” `y`’en introdusert av det ytterste lambda-uttrykket. Tilsvarende ville skjedd i blokken `set y 2; lambda x ((x,y)+);`, som evaluerer til funksjonen `\x -> +(x,2)`. [6]

6.2 Evaluering av funksjons*kall*

`lambda`-funskjoner kan kalles på to måter. Man kan kalle funksjonen direkte i det man har definert den ved å etterfølge definisjonen med et aktuelt argument (i paranteser): `lambda x (E) (arg)`. Eller man kan tilordne den anonyme funksjonen som en verdi til en variabel og deretter kalle variablen med et aktuelt argument: `set minfun lambda x (E); minfun (arg);`. Begge disse kallene har formen `fun (arg)` og for å evaluere de skal `eval`:

1. sjekke at `fun` er en funksjon og gi feilmelding dersom den ikke er det; ellers
2. evaluerer argumentet `arg` før
3. den evaluerer funksjonskroppen `E` av `fun` anvendt på verdien av `arg`.

Applikasjon er altså “ivrig” (“eager”, og ikke “lazy”). F.eks., uttrykket `lambda x ((x,2)+) (1)` evalueres ved, først, å evaluere `lambda x ((x,2)+)` til en funksjon, nemlig `\x->x+2`, så evalueres `1` til `Number 1`, og til slutt kalles funksjonen `\x->x+2` med `Number 1` som argument. Dette skjer ved at kroppen til funksjonen evalueres i sitt originale minnemiljø, utvidet med `x` bundet til `Number 1`.

Funksjoner er første-klasses data, som betyr at de kan bindes til variabler, sendes som argumenter til andre funksjoner og returneres som verdi fra funksjoner. Et eksempel på det første er `set f lambda x (expr)`, mens på bruk av en funksjon som argument er følgende:

[5] Vi bruker Haskell notasjon for anonyme funksjoner kun for å *skrive* resultat av evaluering av `lambda`-uttrykk som representerer funksjoner. Det betyr *ikke* at du må implementere evaluering ved bruk av slike anonyme Haskell funksjoner eller at de faktisk er det som alle våre `lambda`-uttrykk betegner.

`lambda x (E)` kan oppfattes som en deklarasjon av variabel `x` med `E` som rekkevidden (scope) til denne deklarasjonen. Den kan overskygges av indre redeklarasjon, f.eks., i `lambda y (lambda y y)`, som tilsvarer `(\y -> \y -> y)` og, når den blir evaluert f.eks. i `lambda y (lambda y (y)) (2) (4)`, returnerer 4.

[6] Funksjonen sammen med miljøet den hadde når den ble lagd kalles en “tillukking” (closure). Du skal altså implementere funksjoner på en måte som respekterer leksikalsk scope, `http://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scope_vs._dynamic_scope`, og beholder tillukking til funksjonene, `http://en.wikipedia.org/wiki/Closure_(computer_programming)`. Disse termene bør kunne forstås utfra oppgaveteksten og eksempler lenger nede. Linkene er bare for ekstra referanse. Om funksjoner som første-klasses objekter kan leses mer under `http://en.wikipedia.org/wiki/First-class_function`.

```
lambda x (x (3)) (lambda y ((y, y) +)).
```

Denne skal returnere `Number 6`. Vi har her funksjonen `lambda x (x (3))` som tar inn et argument, og anvender det på 3. Argumentet det får inn her er funksjonen `lambda y ((y,y)+)`, som tar inn et tall og dobler det. Evalueringen av dette uttrykket kan representeres som følger:

```
lambda x (x (3)) (lambda y ((y,y) +)) ⇝
     (\x -> (x (3))) (\y -> ((y,y)+)) ⇝
     \y -> ((y,y)+) (3) ⇝ (Number 3, Number 3)+ ⇝ Number 6
```

Evaluering av uttrykket

```
lambda x (lambda y (x)) (5)
```

returnerer en funksjon som har samme effekten som `lambda y (5)`, dvs., som tar inn et argument og returnerer `Number 5` uansett hva den får inn. Dette fordi den indre funksjonen `lambda y (x)` "husker" hvilket minnemiljø den ble skapt i, og da hadde `x` verdien 5.

Følgende illustrerer evalueringsrekkefølge og mulige mellomresultater for et mer komplekst uttrykk:

```
lambda x (x (3)) (lambda x (lambda y (x)) (5))  ⇝
     \x -> (x (3)) (lambda x (lambda y (x)) (5)) ⇝
     \x -> (x (3)) (\x -> (\y -> (x)) (5)) ⇝
     \x -> (x (3)) (\x -> (\y -> (x)) (Number 5)) ⇝
     \x -> (x (3)) (\y -> Number 5) ⇝
     \y -> (Number 5) (3) ⇝
     Number 5
```

Siden funksjoner er førstesklasses data kan de også tilordnes til variabler, for så kalles senere:

```
set  dobbel lambda x ((x,x)+);
```

returnerer verdien av evaluering av `lambda x (+(x,x))` (som vi har betegnet `\x -> x+x`, og som er et passende `Ast` med `Lambda` i roten). Denne verdien tilordnes så til variabelen `dobbel`.

```
set dobbel lambda x ((x,x)+); dobbel (5);
```

tilordner først funksjonen til `dobbel`, som så kalles med argumentet 5, slik at `eval` returnerer her `Number 10`.

6.3 Call by Reference

Man har to strategier man kan bruke for å sende eksisterende variabler som argumenter til en funksjon. Den ene er å lese verdien fra variabelen og så sette variabelen til funksjons argumentet til verdien man har lest ut. Dette kalles for Call by Value. Den andre strategien er å hente ut plasseringen til variabelen i minnet (referansen) og så erstatte selve variabelen i argumentet med plasseringen man har hentet. Det vil si at når forandringer da blir gjort på argumentet inne i lambdaen så forandres også den opprinnelige variabelen. Dette kalles Call by Reference. Følgende eksempel demonstrerer forskjellen mellom de to strategiene,

```
set v 10; lambda x (set x (x, 1)+) (v); v;
```

returnerer `Number 10` med Call by Value og `Number 11` med Call by Reference.

Når variabler sendes som ett argument skal dere bruke Call by Reference. Mens dere naturligvis må bruke Call by Value i tilfeller hvor dere har frie verdier, som f.eks.:

- `lambda x (x) (2);`
- `lambda x (x) ((2,1)+);`
- `set v 1; lambda x (x) (lambda x (x) (v));`

Å bruke Call by Reference gjør det mulig å skrive generell kode som aksesserer minne utenfor scopet til selve funksjonen. F.eks. kan man skrive følgende funksjon som inkrementerer en gitt verdi med en

```
set inc lambda x (set x (x, 1)+);
```

I språk som C og C++ har man mer direkte tilgang til å manipulere refereanser/pekere enn det vi har i vårt språk og i Haskell. Man kan for eksempel dereferensere slik at man henter ut verdien av en variabel direkte. Det kan være at vi faktisk ønsker å sende inn en variabel som Call by Value. Triksett i vårt språk blir da å ta en variabel og transformere den til en fri verdi. Vi kan bruke den siste lambdaen i eksempellisten vår over frie verdier for å lage våres egen "dereferensiator,"

```
set dereference lambda x (x);
```

Og vi kan bruke det i praksis på det første eksempelet vårt,

```
set dereference lambda x (x); set v 10; lambda x (set x (x, 1)+) (dereference (v));
v;
```

Siden vi henter ut verdien til `x` med dereference blir ikke `x`-en i `lambda x (set x (x, 1)+)` satt til å være samme variabel som `v`. Siden returverdien til en lambda er en fri verdi så blir i praksis `v` sendt som Call by Value istedenfor Call by Reference. Dermed blir returverdien til dette eksempelet `Number 10` ikke `Number 11` .

6.4 Tillukking og mutering

Husk at en funksjon bare skal ha tilgang til de variablene som var i rekkevidden når den ble skapt. Dette har viktige konsekvenser når det kombineres med mutering og det er kun for håndtering av dette aspektet at det anbefalles å skille mellom `Context` og `Memory`. I det følgende eksemplet definers det først en funksjon `konto`:

```
set konto lambda balanse (
                    lambda diff (set balanse (balanse, diff)+ ));
       set a konto (100);
       a (13);
       a (2);
```

`konto (100)` returnerer en funksjon som blir satt til variabelen `a`, dvs. `a` skapes i en kontekst hvor `balanse` har verdi 100. Når man så kaller denne funksjonen (med nye verdier for `diff`) så endres verdien til `balanse`. Hele kodesnutten over skal returnere `Number 115`.

Den følgende kodesnutten skal returnere `Number 52`, og hvis vi hadde lagt til `a (1);` på slutten skulle den returnert `Number 114`. De forskjellige `balanse` variablene som eksisterer i tillukking til henholdsvis `a` og `b` peker altså til forskjellige plasser i minnet.

```
set konto lambda balanse (
                    lambda diff (set balanse (balanse, diff)+ ));
       set a konto (100);
       set b konto (50);
       a (13);
       b (2);
```

At en funksjon har tilgang til variablene som fantes når den ble skapt, og ikke de som finnes når den blir kalt, betyr at

```
set fun lambda x ((y,x)+); set y 4; fun (1);
```

ikke er et lovlig program, siden `fun` prøver å aksessere variabelen `y` som ikke finnes når den ble lagd (men som finnes når den kjøres). På den andre siden, er

```
set y 4; set fun lambda x ((y, x)+); fun (1);
```

et lovlig uttrykk, siden nå er `fun` essensielt funksjonen `\x -> 4+x`, da `y` har denne verdien når `fun` lages. Hele uttrykket returnerer `Number 5`.

Merk at funksjoner har tilgang til variabler tilgjengelig ved deres definisjon, men disse kan få endret verdier før funksjonen kalles. F.eks. skal

```
set a 1; set fun lambda x (a); set a 2; fun (3);
```

returnere `Number 2` og ikke `Number 1`.

6.5 Feilhåndtering

(1) Din `eval` bør oppdage feil med ubundne variabler, som i `lambda x ( (x,y)+ )`, når `y` ikke er deklarert i den aktuelle konteksten. Slike oppdages neppe under parsing, men de bør oppdages i det verdien til en ubunden variable `y` forsøkes å aksessere. Følgende gir et mer involvert eksempel. Uttrykket `lambda x (x (3)) (lambda y (z))` er ikke et gyldig funksjonskall, fordi argumentet `lambda y (z)` ikke evaluerer til en kjørbar funksjon, siden `z` ikke finnes på det tidspunktet den blir evaluert. Evaluator kan gå fram som følger

```
lambda x (x (3)) (lambda y (z)) ⇝ (\x -> (x (3))) (\y -> (z))
                                                  ⇝ \y -> (z) (3) ⇝ z
```

og bør oppdage dette. Men det samme uttrykket evaluert i en blokk etter, f.eks., `set z 5;`, blir til et gyldig kall som returnerer `Number 5`.

(2) Funksjonskall kan også føre til en feil når yttrykket som burde evaluere til en funksjonen ikke gjør det. F.eks., blokken `set v 10; v(5);` er syntaktisk korrekt, men dens evaluering feiler fordi variabelen `v` ikke er bundet til en funksjon men til `Number 10`. Også evaluering av uttrykket

```
lambda x (x (3)) (5) ⇝ (\x -> (x (3))) (5) ⇝ 5 (3) ⇝ Number 5 (3)
```

gir ikke noen gyldig resultat/funksjonskall. Evaluator bør gi passende `error`-melding.

(3) Husk at mens variabler som defineres i toppnivået til `<Block>` er tilgjengelig i hele blokken ("gloabalt"), så gjelder ikke det samme for `lambda`-uttrykk. Disse generer et nytt rekkeviddenivå som slutter med funksjonen.

```
lambda x (x) (5); x;
```

er ikke gyldig, siden (den siste) `x` ikke er i rekkevidden når den evalueres. Tilsvarende

```
lambda x (set a x) (0); a;
```

er ikke gyldig, siden variabelen `a` ikke er definert når den blir prøvd evaluert (rekkevidden til dens deklarasjon `set a...` avsluttes i enden av `lambda`-uttrykket.) Slike rekkeviddefeil trenger man ikke å oppdage ved parsing (selv om det ville være mulig), men de burde oppdages ved evaluering.

(4) At en funksjon har tilgang til variablene som fantes når den ble skapt, og ikke de som finnes når den blir kalt, betyr at, f.eks.,

```
set fun lambda x ( (y,x)+ ) ; set y 4 ; fun (1);
```

ikke er et lovlig program, siden fun prøver å aksessere variabelen y som ikke finnes når den ble lagd (men som finnes når den kjøres). På den andre siden, er

```
set y 4 ; set fun lambda x ( (y,x)+ ) ; fun (1);
```

et lovlig uttrykk, siden nå er fun essensielt funksjonen `\x -> 4+x`, da y har verdien 4 når fun lages. Hele uttrykket returnerer Number 5.

6.6 Implementasjon

Det kan være nyttig å utvide `Ast` med en ny konstruktor

```
data Ast ::= ...  | Function String Ast Context,
```

som benyttes hver gang man *evaluerer* et `lambda`-uttrykk (definisjon eller kall).

Parsing av lambda*definisjon* skal resultere i et `Lambda Ast` med parameternavnet og kroppen. Når et `lambda`-uttrykk evalueres, har man dette `Ast`'et tilgjengelig og så sjekker man om det representerer en funksjon ved å konvertere det til `Function Ast`. Dette `Ast`'et representerer funksjonen med parameternavnet, funksjonskroppen *og hele konteksten*, dvs., konteksten tilgjengelig ved definisjonspunktet utvidet med de lokale variablene deklarert inne i denne funksjonen.

Dersom `lambda`-uttrykket inngår i en funksjons*applikasjon*, evaluerer man deretter argumentet. Så må det allokeres en ny minneadresse som fylles med denne verdien. Parameternavnet må så linkes til denne minneadressen i den konteksten funksjonen ble definert i, før kroppen evalueres i den utvidede konteksten/minnetilstanden. Denne evalueringen returnerer en verdi (muligens en ny funksjon), en kontekst og minnetilstand. Du må selv finne ut hvilken kontekst/minnetilstand som skal returneres sammen med verdien. Bruk gjerne hjelpfunksjonen fra seksjon 5.2.

Du står du fritt til å løse oppgaven på en annen måte – vi kommer til å teste funksjonaliteten via `run` funksjonen som beskrevet tidligere.

Oppgave VI: utvidelse til hele Gr .....

Utvid `parse` og `eval` til å håndtere hele språket. Dette er en betydelig utvidelse, så det er best å gjøre den gradvis, men du må i så fall selv lage en plan hvordan du vil gå fram.

Dersom du ikke blir helt ferdig, skriv gjerne i kommentarer hvilke biter av språket evt. hvilke tester som ikke blir håndtert tilfredstillende av din implementasjon.

Hele besvarelsen bør ikke ha mer enn ca. 200 kodelinjer.

7 Epilog

Denne delen inneholder ingen oppgaver og er ikke pensum, men ekstra informasjon for interesserte.

7.1 Haskell, monader og slikt

Hvis du leser om Haskell på internett tar det ikke lang tid før monader blir nevnt. Det er ikke uten grunn, for det finnes mange monader som gjør mye enklere, blant annet håndtering og endring av tilstand. Vi har ikke tid til å gå gjennom dette, men hvis du senere vil lære mer Haskell så kan du prøve å omskrive programmet ditt til å bruke enten Control.Monad.ST eller Data.IORef for å håndtere tilstand, istedenfor å "tråde" den gjennom `eval` som vi har gjort her. Skal ikke se bort ifra at det gjør alt vakrere og, til slutt, kanskje også enklere.