Oblig 2 – INF122, H-2018

Hvordan komme igang / hvordan levere

- Oppgaven skal leveres på Mitt UiB under Oppgaver, samme sted som du fant oppgaveteksten. Løsningen skal bestå av én zip-fil med navn på formen FORNAVN.ETTERNAVN.zip.
- zip-filen skal inneholde tre filer: Oblig2.hs, Main.hs (fra VisAST/app/) og Del3.pdf. Alle tre filene skal ha ditt Mitt UiB-brukernavn i en kommentar i toppen.
- Du skal klone (eller laste ned) dette git-repoet: github.com/aaalvik/Oblig2-INF122
- Innleveringsfristen er **Tirsdag 27. november, kl. 23.59**
- Når oppgaven ber deg om å lage en funksjon med gitt navn og type, så er det **viktig** at du ikke endrer på navnet/typen. Du kan selvfølgelig lage hjelpefunksjoner med andre navn og typer

Oversikt

OBS: Før du begynner skal du klone git-repoet nevnt over, følg instruksene i README-en på github.
All kode skal skrives inn i de gitte filene og mappene i din kopi av dette repoet.

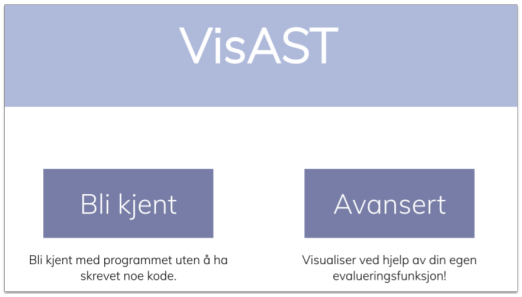
Vi skal jobbe med et lite språk som består av mattestykker i prefiksnotasjon (dvs operatoren først) og if-then-else uttrykk. Det er ingen parenteser i språket. Grammatikken til språket ser slik ut:

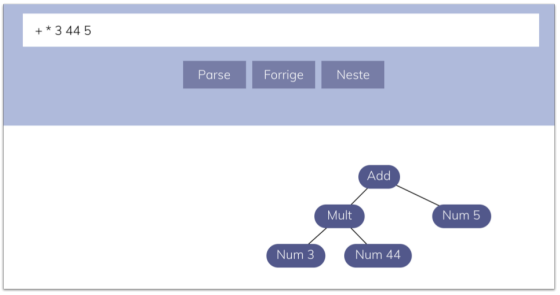
```
S ::= * S S
    | + S S
    | - S                       unær minus
    | if S then S else S
    | number_or_digit           ett eller flere siffer
```

Ifølge grammatikken kan eksempler på uttrykk i språket være:

```
+ 3 4                       dvs 3 + 4 i vanlig infiksnotasjon

* 6 7                       dvs 6 * 7 i vanlig infiksnotasjon

- 3                         unær minus, altså minus 3.

if 2 then + 3 4             sjekker om if-condition (her tallet 2) er ulik 0. Er den
      else 6                ulik 0 går man til then-branchen, ellers else-branchen.

* + 3 4 5                   dvs (3 + 4) * 5 i vanlig infiksnotasjon. Man trenger
                            altså ikke parenteser i språket.

+ if 2 then + 3 4           dvs (if 2 then (3+4) else 6) + 8. Her blir if-condition
        else 6 8            true, returnerer 7, og resultatet av hele uttrykket blir
                            7 + 8 = 15.
```

Om visAST





I løpet av oppgaven vil vi bruke programmet visAST, som er et verktøy for å visualisere abstrakte syntakstrær. visAST er under utvikling, og dere vil bli de første til å teste dette i undervisningssammenheng. Vi ønsker derfor tilbakemelding på hvordan dette var å bruke, så tenk gjennom hvordan du opplever det underveis.

Sammendrag av oppgaver:
(Du finner nøyere forklaring og beskrivelse av hver oppgave lenger ned.)

1.1 Implementer en parser for språket beskrevet over.

1.2 Print abstrakte syntakstrær – Expr – på en pen måte.

1.3 Implementer en small-step evaluator for språket.

1.4 Lag et program som leser inn kommandoer fra bruker og utfører kommandoen.

2.1 Utvid 1.4 til å kunne visualisere ved hjelp av visAST.

2.2 Svar på spørsmål om visAST.

3.1 Finn typen til en funksjon ved hjelp av reglene for typeinferens og unifikasjon.

Del 1 – Parse og prettyprint expressions

I hele del 1 (oppgave 1.1-1.4) skal du jobbe i filen Oblig2.hs

Oppgave 1.1 – parse :: String → Expr

Du skal implementere en parser som oversetter strenger i språket til et abstrakt syntakstre av typen Expr:

```
data Expr
   = Num Int              et tall, feks “2” eller “222”
   | Add Expr Expr        et pluss-uttrykk, feks “+ 4 5”
   | Mult Expr Expr       et gange-uttrykk, feks “* 4 5”
   | Neg Expr             et negert uttrykk, feks “- 6” eller “- + 4 5”
   | If Expr Expr Expr    et if-uttrykk, feks “if 2 then 3 else 4”
   deriving (Show, Read)
```

Du bør begynne med en tokenizer (tokenize :: String -> [String]), som kan splitte inputstrengen i meningsfulle tokens. Tokens i språket ser man av grammatikken at er en av operatorene “+-*”, nøkkelordene “if”, “then” og “else”, i tillegg til tall, som består av ett eller flere siffer.

Videre anbefales det å lage en hjelpefunksjon parseExpr :: [String] → (Expr, [String]), som kan kalles fra parse. Parse bør da først tokenize inputstrengen til en liste med tokens (Strings) som kan sendes til parseExpr. Følg reglene i grammatikken (på side 1) slik at parseren oppfører seg som under.
NB: Denne parseren er mye enklere enn den fra Oblig 1, da alle operatorer/keywords kommer prefiks i stedet for infiks.

Eksempler på parsing av uttrykk:

```
parse “+ 4 5”                    == Add (Num 4) (Num 5)
parse “* + 5 6 70”               == Mult (Add (Num 5) (Num 6))  (Num 70)
parse “*+5 6 70”                 == Mult (Add (Num 5) (Num 6))  (Num 70)
parse “- 4001”                   == Neg (Num 4001)
parse “* 8 + 4 - 52”                    == Mult (Num 8) (Add (Num 4) (Neg (Num
52)))
parse “if 0 then 4 else + 3 3”   == If (Num 0) (Num 4) (Add (Num 3) (Num 3))
parse “3 + 4”                    == error “Illegal expression”
```

Legg merke til i tredje eksempel at operatorene kan stå inntil hverandre (også operator og tall) uten at det gjør noen forskjell. Dette må håndteres av tokenizeren.

Oppgave 1.2 – prettyPrint :: Expr → IO ( )

Du skal implementere funksjonen prettyPrint, som tar inn et expression (abstrakt syntakstre av typen Expr) og printer dette til terminal på en “pen” måte. For hver node i treet Expr skal du skrive ut navnet på noden, og så barna på hver sin linje under, indentert én tab (evt fire spaces) lenger inn. Du må altså holde styr på antall tabs man skal indentere på hvert nivå.
Tips: Lag en hjelpefunksjon `prettyPrint' :: Expr -> Int -> IO ()` som har med som parameter antall tabs man skal indentere så langt.

Eksempel 1 – prettyPrint (parse “+ 4 5”) skal printe:

```
Add
  Num 4
  Num 5
```

Eksempel 2 – prettyPrint (parse “* 8 * - 9 1”) skal printe:

```
Mult
  Num 8
  Mult
    Neg
      Num 9
    Num 1
```

Eksempel 3 – prettyPrint (parse “if * 2 3 then 10 else 20”) skal printe:

```
If
  Mult
    Num 2
    Num 3
  Num 10
  Num 20
```

Frivillig oppgave – Test parseren din med visAST

Du så kanskje at prettyPrint ikke gjorde trærne så veldig pretty likevel, derfor kan du nå teste parseren ved hjelp av visAST, som tegner trærne slik man ofte gjør på tavla. Les beskrivelsen om visAST under Del 2 lenger ned før du begynner. Fremgangsmåte (også delvis forklart i README):

1. Lim din implementasjonen av *parse* inn i filen VisAST/app/TestParser.hs.

2. Åpne terminal (den i VS Code e.l. vil ikke fungere, men vanlig terminal/cmd)
3. Naviger til mappen VisAST/
4. Skriv inn **stack build** i terminalen (dette må gjøres på nytt hver gang du gjør endringer i parseren)
5. Skriv inn **stack exec TestParser-exe** i terminalen. Du vil nå bli bedt om å skrive inn mitt.uib brukernavnet ditt og et uttrykk i språket, deretter åpner visAST seg automatisk i en browser. I browseren: klikk "Avansert", fyll inn brukernavn og du får opp treet ditt.

Oppgave 1.3 – takeOneStep :: Expr → Expr

Implementer funksjonen takeOneStep, som tar inn et abstrakt syntakstre (Expr) og utfører ett evalueringssteg før den returnerer. Det vil si, funksjonen skal ta det minste steget den kan på et Expr men fortsatt komme nærmere resultatet (en verdi). **Når et uttrykk har blitt til et Num num, så er det en verdi**, og man kan ikke ta flere evalueringssteg.

Regler for evaluering:

1. Num x evaluerer til seg selv, feks Num 5 er en verdi og kan ikke evalueres mer.
   Feks: `takeOneStep (Num 5) == Num 5`

2. Add x y skal først evaluere ferdig x til en verdi, ett steg om gangen.
   Dersom x er en verdi skal man på samme vis evaluere y til en verdi, ett steg om gangen.
   Dersom både x og y er verdier, skal man plusse dem sammen til et Num og returnere dette.

3. Mult x y skal som i Add først evaluere x til en verdi, deretter y, deretter gange x og y sammen.

4. Neg x skal først evaluere x til en verdi – Num n – deretter negere n (dvs gange n med minus 1)

5. If eCond eThen eElse skal som over først evaluere eCond (altså conditionen til if-setningen) til en verdi, ett steg om gangen.
   Dersom eCond er en verdi skal man sjekke om den er ulik 0 (fordi vi kun jobber med Ints og ikke boolske verdier, så definerer vi alt ulik 0 som True). Er eCond ulik 0 returnerer man eThen, ellers returneres eElse, uten å evaluere videre.

*Et uttrykk som + 4 * 5 6 evaluerer til slutt til 34, men dette kan deles opp i to steg:*

- *Gange sammen 5 og 6 til 30, slik at man får + 4 30*
- *Plusse sammen 4 og 30 til 34*

Eksempler:

```
// x er ikke en verdi
takeOneStep (Add (Add (Num 4) (Num 4)) (Num 2)) == Add (Num 8) (Num 2)

// x er en verdi, y er ikke
takeOneStep (Add (Num 4) (Mult (Num 5) (Num 6))) == Add (Num 4) (Num 30)

// både x og y er verdier
takeOneStep (Add (Num 4) (Num 2)) == Num 6

// eCond er ikke en verdi:
takeOneStep (If (Add (Num 1) (Num 1))  (Mult (Num 2) (Num 3))  (Num 5)) ==
       If (Num 2)  (Mult (Num 2) (Num 3))  (Num 5))

// eCond er en verdi, og ulik 0:
takeOneStep (If (Num 3)  (Num 4)  (Mult (Num 2) (Num 7))) == Num 4

// eCond er en verdi, og lik 0:
takeOneStep (If (Num 0)  (Num 4)  (Mult (Num 2) (Num 7))) == Mult (Num 2) (Num 7)

// når til slutt en verdi (et Num) og man kan ikke evaluere lenger
takeOneStep (Mult (Num 2) (Num 7))      == Num 14
takeOneStep (Num 14)                          == Num 14
```

Oppgave 1.4 – mainStep :: Expr → IO ( )

Nå skal du implementere funksjonen mainStep som skal ta inn kommandoer fra bruker, printe ting til skjerm samt skrive til/lese fra fil. mainStep skal begynne med å prettyPrinte expr gitt som parameter. Deretter skal man lese inn input fra bruker, og gjøre ulike kommandoer basert på hva bruker skrev inn. Funksjonen skal kjøre videre helt til bruker velger å avslutte med kommandoen q.

new <expression>	Tar inn et nytt expression og parser det. Du skal parse alt som kom etter nøkkelordet *new*, og kalle mainStep videre med dette nye expr. Feks "new + 4 5" skal kalle parse på "+ 4 5".
ENTER-KLIKK	Dette skal skje dersom bruker trykker på enter-knappen uten å ha skrevet noe mer. Her skal du ta ett evalueringssteg (med takeOneStep fra 1.3) på expr som ble gitt som parameter til mainStep, og så fortsette programmet (kalle mainStep på nytt med det evaluerte expr).
w <filnavn>	Skriv expr til en fil med navn <filnavn>.txt, og fortsett programmet. For å gjøre uttrykket om til en streng kan du kalle **show** på det før du skriver til filen.
r <filnavn>	Les expr som er lagret i filen <filnavn>.txt, og kall mainStep videre med det nye uttrykket. Hvis du kalte show på uttrykket før du skrev til filen så kan du nå kalle **read str :: Expr** på str lest fra filen, og få ut et Expr.
q	Avslutt programmet.

Eksempel på kjøring:

```
> mainStep (Add (Add (Num 4) (Num 4)) (Num 2))
Add
   Add
      Num 4
      Num 4
   Num 2
* Hva vil du gjøre? (new expr, ENTER, w filnavn, r filnavn, q)
[Bruker trykket ENTER-knappen]
Add
   Num 8
   Num 2
* Hva vil du gjøre? (new expr, ENTER, w filnavn, r filnavn, q)
w minFil.txt
Skrev uttrykket til fil.
Add
   Num 8
   Num 2
* Hva vil du gjøre? (new expr, ENTER, w filnavn, r filnavn, q)
[Bruker trykket ENTER-knappen]
Num 10
* Hva vil du gjøre? (new expr, ENTER, w filnavn, r filnavn, q)
new * 1 2
Mult
   Num 1
   Num 2
* Hva vil du gjøre? (new expr, ENTER, w filnavn, r filnavn, q)
q
```

Del 2 – Visualiser abstrakte syntakstrær

Du skal nå jobbe i mappen VisAST/app, i filen Main.hs. Før du begynner på oppgaven skal du lime inn de fire funksjonene du implementerte i Del 1:

- parse
- prettyPrint
- takeOneStep
- mainStep
- + eventuelle hjelpefunksjoner

I denne oppgaven må du bruke **stack** for å kompilere og kjøre programmet, som beskrevet i README. Du vil ikke kunne teste ting i ghci, og terminalen i feks VS Code vil mest sannsynlig heller ikke fungere (bruk vanlig terminal/iterm/command prompt etc)

Oppgave 2.1 – utvid mainStep-metoden i Main.hs til å kunne starte visAST
Nå skal du bruke visAST til å få visualisert trærne på en penere måte enn med prettyPrint. Før du begynner å kode bør du gjøre deg kjent med visAST under "Bli kjent" her, der du kan få visualisert uttrykk du skriver inn uten å ha skrevet noe kode.

Etter å ha gjort deg kjent med visAST skal du utvide mainStep (som du har limt inn i VisAST/app/Main.hs) med en kommando som starter visAST. visAST vil da visualisere trær **ved hjelp av din egen takeOneStep fra 1.3.** Utvid mainStep med følgende kommando:

vis <brukernavn>	<brukernavn> skal være mitt.uib brukernavnet ditt – på formen abc123. Du skal så evaluere expr slik at du får en liste med expressions – alle "tilstandene"/evalueringsstegene til expr frem til det blir en verdi. Det gjør du ved å kalle eval-funksjonen (som bruker din egen takeOneStep). Til slutt skal du bruke den importerte funksjonen **visualise :: String -> [Expr] -> IO ()** til å starte visAST. visualise tar inn brukernavn og listen med expressions og vil åpne en browser for deg på nettsiden til visAST. På nettsiden velger du "Avansert" -> Skriver inn brukernavnet du brukte, og så vil du få visualisert listen med evalueringssteg som du sendte til visualise. Det første treet du ser er startuttrykket, og du kan steppe frem og tilbake med knappene og se alle evalueringsstegene frem til du står igjen med en verdi (et Num).

Oppgave 2.2 – Svar på spørsmål om visAST
Svar på spørsmålene her: goo.gl/forms/nELqqXVZ5lVnedCp1.
Du skal ikke levere inn noen fil i denne oppgaven, kun fylle ut skjemaet i lenken.

Del 3 - Finn typen til en funksjon

Oppgi svaret i en fil del3.pdf som du oppretter selv. Denne skal inneholde hele utledningen.

Oppgave 3.1 - Finn typen til \x -> \y -> \z -> z (x y)
Bruk reglene for typeinferens og unifikasjon oppgitt på siste side til å finne typen til funksjonen.
Du kan løse oppgaven for hånd og ta bilde/scanne utledningen, eller skrive direkte inn på pc. Husk at du må vise hele utledningen.

VEDLEGG – Regler for typeinferens og unifikasjon

Typeinferens – Hindley-Milners algoritme

(t1, konstant)	$E(\Gamma \mid c :: \tau)$	$= \{ \tau = \theta \}$	der θ er typen til konstanten (feks Int / Bool etc, en kjent type)
(t2 - variabel)	$E(\Gamma \mid x :: \tau)$	$= \{ \tau = \theta \}$	der θ = lookup(x, Γ) for variabel x
(t3 - funksjonskall)	$E(\Gamma \mid f\ x :: \tau)$	$= E(\Gamma \mid x :: a) \cup E(\Gamma \mid f :: a \rightarrow \tau)$	
(t4 - abstraksjon)	$E(\Gamma \mid \backslash x \rightarrow y :: \tau)$	$= \{ \tau = a \rightarrow b \} \cup E(\Gamma, x :: a \mid y :: b)$	

(t5 - tuppel)	$E(\Gamma \mid (ex1, ex2) :: t)$	$= \{ t = (a, b) \} \cup E(\Gamma \mid ex1 :: a) \cup E(\Gamma \mid ex2 :: b)$
(t6 - ikke-tom liste)	$E(\Gamma \mid x{:}xs :: t)$	$= \{ t = [a] \} \cup E(\Gamma \mid x :: a) \cup E(\Gamma \mid xs :: [a])$
(t7 - tom liste)	$E(\Gamma \mid [] :: t)$	$= \{ t = [a] \}$

Forklaringer:

1. ***Γ*** *står for environmentet, altså typene vi kjenner fra før*
2. ***E(Γ | expr :: myType)*** *kan leses slik:*
   *"Et uttrykk **E** som inneholder et environment **Γ** hvor **expr** har typen **myType**"*
3. *Eksempel på t1-uttrykk:* **$E(\Gamma \mid 13 :: \tau) \mid\mid E(\Gamma \mid$ "Hei" $:: \tau)$**
   Vi kjenner typen til konstanten 13, og kan sette den rett inn: **$= \{ \tau =$ Int $\}$**

Unifikasjonsregler

(u1)	E,t = t	⇒ E		
(u2)	E, f (t1…tn) = f (s1…sn)	⇒ E, t1 = s1, …, tn = sn		
(u3)	E, f (t1…tn) = g(s1…sm)	⇒ ⊥ (occurs check)	der	f /= g ∨ n /= m
(u4)	E, f (t1 …tn) = x	⇒ E, x = f (t1…tn)		
(u5)	E, x = t	⇒ E[x/t], x = t	der	x not∈ Var(t)
(u6)	E, x = t	⇒ ⊥	der	x ∈ Var(t)