En oversætter for Janus

Godkendelsesopgave på kurset Oversættere

Efterår 2009

1 Introduktion

Dette er den første del af rapportopgaven på Oversættere, efterår 2009. Opgaven skal løses i grupper på op til 3 personer. Opgaven bliver stillet mandag d. 9/11 2009 og skal afleveres senest onsdag d. 16/12 2009. Opgaven afleveres via kursushjemmesiden på Absalon. Brug gruppeafleveringsfunktionen i Absalon. Alle medlemmer af gruppen skal på rapportforsiden angives med navn og fødselsdato. Der er ikke lavet en standardforside, så lav en selv.

Denne del af rapportopgaven bedømmes som godkendt / ikke godkendt. Godkendelse af denne opgave er (sammen med godkendelse af fire ud af fem ugeopgaver) en forudsætning for deltagelse i den andel del af rapporteksamenen, der er en karaktergivende opgave, der løses individuelt. En ikke-godkendt godkendelsesopgave kan *ikke* genafleveres.

2 Om opgaven

Opgaven går ud på at implementere en oversætter for sproget Janus, som er beskrevet i afsnit 3.

Som hjælp hertil gives en fungerende implementering af en delmængde af Janus. I afsnit 6 er denne delmængde beskrevet.

Der findes på kursussiden en zip-fil kaldet "G.zip", der indeholder opgaveteksten, implementeringen af delmængden af Janus samt et antal testprogrammer med input og forventet output. Der kan blive lagt flere testprogrammer ud i løbet af de første uger af opgaveperioden.

Det er nødvendigt at modificere følgende filer:

`Parser.grm` Grammatikken for Janus med parseraktioner, der opbygger den abstrakte syntaks.

`Lexer.lex` Leksikalske definitioner for *tokens* i Janus.

`Type.sml` Typechecker for Janus.

`Compiler.sml` Oversætter fra Janus til MIPS assembler. Oversættelsen sker direkte fra Janus til MIPS uden brug af mellemkode.

Andre moduler indgår i oversætteren, men det er ikke nødvendigt at ændre disse.

Til oversættelse af ovennævnte moduler (og andre moduler, der ikke skal ændres) bruges Moscow-ML oversætteren inklusive værktøjerne MosML-lex og MosML-yacc. `Compiler.sml` bruger datastruktur og registerallokator for en delmængde af MIPS instruktionssættet. Filen `compile` indeholder kommandoer for oversættelse af de nødvendige moduler. Der vil optræde nogle *warnings* fra compileren. Disse kan ignoreres, men vær opmærksom på evt. nye fejlmeddelelser eller advarsler, når I retter i filerne.

Til afvikling af de oversatte MIPS programmer bruges simulatoren SPIM.

Krav til besvarelsen

Besvarelsen afleveres som en zip-fil, der indeholder rapportteksten (som PDF fil) og alle relevante program- og datafiler, sådan at man ved at pakke zip-filen ud i et ellers tomt katalog kan oversætte og køre oversætteren på testprogrammerne. Dette kan f.eks. gøres ved, at I zipper hele jeres arbejdskatalog (og evt. underkataloger).

Zip-filen afleveres via kursushjemmesiden.

Rapporten skal angive alle medlemmer af gruppen med navn og fødselsdato.

Rapporten skal indeholde en kort beskrivelse af de ændringer, der laves i ovenstående komponenter.

For `Parser.grm` skal der kort forklares hvordan grammatikken er gjort entydig (ved omskrivning eller brug af operatorpræcedenserklæringer) samt beskrivelse af eventuelle ikke-åbenlyse løsninger, f.eks. i forbindelse med opbygning af abstrakt syntaks. Det skal bemærkes, at alle konflikter skal fjernes v.h.a. præcedenserklæringer eller omskrivning af syntaks. Med andre ord må MosML-yacc *ikke* rapportere konflikter i tabellen.

For `Type.sml` og `Compiler.sml` skal kort beskrives, hvordan typerne checkes og kode genereres for de nye konstruktioner. Brug evt. en form, der ligner figur 6.2 og 7.3 i *Basics of Compiler Design*.

I skal ikke inkludere hele programteksterne i rapportteksten, men I skal inkludere de væsentligt ændrede eller tilføjede dele af programmerne i rapportteksten som figurer, bilag e.lign. Hvis I henviser til dele af programteksten, skal disse dele inkluderes i rapporten.

Rapporten skal beskrive hvorvidt oversættelse og kørsel af eksempelprogrammer (jvf. afsnit 8) giver den forventede opførsel, samt beskrivelse af afvigelser derfra.

Kendte mangler i typechecker og oversætter skal beskrives, og i det omfang det er muligt, skal der laves forslag til hvordan disse evt. kan udbedres.

Det er vigtigere at kunne køre alle eksempelprogrammerne end at generere effektiv kode kun for nogle af disse, så det anbefales at man ikke laver avancerede optimeringer før man har en fuldt fungerende oversætter. Da en fungerende oversætter og en god rapport herom er rigeligt for godkendelse af G-opgaven, er eventuelle optimeringer af koden primært for jeres egen dannelses skyld. Det skal

bemærkes, at SPIM ikke angiver køretid eller antal udførte instruktioner, så det kan være svært at vurdere effekten af optimeringer.

Det er i stort omfang op til jer selv at bestemme, hvad I mener er væsentligt at medtage i rapporten, sålænge de eksplicitte krav i dette afsnit er opfyldt.

Rapporten må maksimalt fylde 16 sider, helst mindre, dog uden at udelade de eksplicitte krav såsom beskrivelser af mangler i programmet og væsentlige designvalg.

2.1 Afgrænsninger af oversætteren

Det er helt i orden, at lexer, parser, typechecker og oversætter stopper ved den første fundne fejl.

Hovedprogrammet `JC.sml` kører typecheck på programmerne inden oversætteren kaldes, så oversætteren kan antage, at programmerne er uden typefejl m.m.

Det kan antages, at de oversatte programmer er små nok til, at alle hopadresser kan ligge i konstantfelterne i branch- og hopordrer.

Hvis der bruges en hob, er det ikke nødvendigt at frigøre lager på hoben mens programmet kører. Der skal ikke laves test for overløb på stakken eller hoben. Den faktiske opførsel ved overløb er udefineret, så om der sker fejl under afvikling eller oversættelse, eller om der bare beregnes mærkelige værdier, er underordnet.

2.2 MosML-Lex og MosML-yacc

Beskrivelser af disse værktøjer findes i Moscow ML's Owners Manual, som kan hentes via kursets hjemmeside. Yderligere information samt installationer af systemet til Windows og Linux findes på Moscow ML's hjemmeside (følg link fra kursets hjemmeside, i afsnittet om programmel). Desuden er et eksempel på brug af disse værktøjer beskrevet i en note, der kan findes i `Lex+Parse.zip`, som er tilgængelig via kursets hjemmeside samt i kataloget
`/usr/local/del1/dat-oversaet/Lex+Parse/`.

3 Janus

Janus er et eksempel på et *reversibelt programmeringssprog*, dvs. et programmeringssprog, hvor alle sætninger kan køres både forlæns og baglæns. Det betyder, at kun bijektive funktioner kan programmeres i sproget. Sproget er designet på Caltech og er nærmere beskrevet i [1]. Bemærk dog, at der er små forskelle mellem det dér beskrevne sprog og det, der bruges i denne opgave.

Herunder beskrives syntaks og uformel semantik for sproget Janus og en kort beskrivelse af de filer, der implementerer sproget.

$$
\begin{array}{lcl}
\mathit{Prog} & \rightarrow & \mathit{Defs}\ \texttt{->}\ \mathit{Defs}\ \texttt{with}\ \mathit{Defs}\ \texttt{;}\ \mathit{Stat}\ \mathit{Procs} \\
\mathit{Prog} & \rightarrow & \mathit{Defs}\ \texttt{->}\ \mathit{Defs}\ \texttt{;}\ \mathit{Stat}\ \mathit{Procs} \\
\mathit{Defs} & \rightarrow & \textbf{id}\ \mathit{Defs} \\
\mathit{Defs} & \rightarrow & \textbf{id}\ \texttt{[}\ \textbf{num}\ \texttt{]}\ \mathit{Defs} \\
\mathit{Defs} & \rightarrow & \\
\mathit{Procs} & \rightarrow & \texttt{procedure}\ \textbf{id}\ \mathit{Stat}\ \mathit{Procs} \\
\mathit{Procs} & \rightarrow & \\
\mathit{Stat} & \rightarrow & \mathit{Lval}\ \texttt{+=}\ \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \mathit{Lval}\ \texttt{-=}\ \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \mathit{Stat}\ \texttt{;}\ \mathit{Stat} \\
\mathit{Stat} & \rightarrow & \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \mathit{Stat}\ \texttt{else}\ \mathit{Stat}\ \texttt{fi}\ \mathit{Cond} \\
\mathit{Stat} & \rightarrow & \texttt{from}\ \mathit{Cond}\ \texttt{do}\ \mathit{Stat}\ \texttt{loop}\ \mathit{Stat}\ \texttt{until}\ \mathit{Cond} \\
\mathit{Stat} & \rightarrow & \texttt{call}\ \textbf{id} \\
\mathit{Stat} & \rightarrow & \texttt{uncall}\ \textbf{id} \\
\mathit{Stat} & \rightarrow & \texttt{skip} \\
\mathit{Lval} & \rightarrow & \textbf{id} \\
\mathit{Lval} & \rightarrow & \textbf{id}\ \texttt{[}\ \mathit{Exp}\ \texttt{]} \\
\mathit{Exp} & \rightarrow & \textbf{num} \\
\mathit{Exp} & \rightarrow & \mathit{Lval} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \texttt{+}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \texttt{-}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp}\ \texttt{/2} \\
\mathit{Exp} & \rightarrow & \texttt{(}\ \mathit{Exp}\ \texttt{)} \\
\mathit{Cond} & \rightarrow & \mathit{Exp}\ \texttt{<}\ \mathit{Exp} \\
\mathit{Cond} & \rightarrow & \mathit{Exp}\ \texttt{==}\ \mathit{Exp} \\
\mathit{Cond} & \rightarrow & \texttt{!}\ \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \mathit{Cond}\ \texttt{\&\&}\ \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \mathit{Cond}\ \texttt{||}\ \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \texttt{(}\ \mathit{Cond}\ \texttt{)}
\end{array}
$$

Figur 1: Syntaks for Janus

4 Syntaks

4.1 Leksikalske og syntaktiske detaljer

- Et navn (**id**) består af bogstaver (både store og små), cifre og understreger og skal starte med et bogstav. Bogstaver er engelske bogstaver, dvs. fra A til Z og a til z. Nøgleord som f.eks. `if` er *ikke* legale navne.
- Talkonstanter (**num**) er ikke-tomme følger af cifrene 0-9. Talkonstanter er begrænset til tal, der kan repræsenteres som positive heltal i Moscow ML.
- Operatorerne `+` og `-` har samme præcedens og er alle venstreassociative.
- Operatoren`/2` er et enkelt symbol, så de to tegn kan ikke adskilles af blanktegn og lignende. Den binder stærkere end + og -.
- Operatoren `!` binder stærkere end `&&`, som binder stærkere end `||`. Både `&&` og `||` er højreassociative.
- Semikolon er højreassosiativ.
- Parenteser bruges blot til at gruppere med, så de forekommer ikke i den abstrakte syntaks.
- Der er separate navnerum for variabler og procedurer, så en procedure kan have samme navn som en variabel. Tabelvariabler og heltalsvariabler deler samme navnerum, så en tabelvariabel og en heltalsvariabel kan ikke have samme navn.
- Kommentarer starter med `//` og slutter ved det efterfølgende linjeskift.

5 Semantik

Hvor intet andet er angivet, er semantikken for de forskellige konstruktioner i sproget identisk med semantikken for tilsvarende konstruktioner i C. For eksempel er aritmetik på 32-bit tokomplementtal uden detektion af *overflow*.

Et Janus program består af erklæringer af input variable, output variable, temporære variable, en sætning og derefter evt. procedureerklæringer. Mellem input variablerne og output variablerne er der et `->`, og de temporære variabler er angivet efter nøgleordet `with`. Hvis der ikke er nogen temporære variabler, kan nøgleordet `with` udelades. Erklæringerne afsluttes med et semikolon.

Et program køres ved at indlæse værdier for input variablerne (i den rækkefølge, de er erklæret), initialisere alle andre variabler og tabelelementer til 0, udføre sætningen og udskrive værdierne af output variablerne (i den rækkefølge, de er erklæret). Ved programmets afslutning må kun outputvariablerne have værdier forskellig fra 0. Det skal verificeres, at alle andre variabler og tabeller er nul og en

fejlmeddelelse skal udskrives hvis ikke. Indlæsning eller udskrift af tabelvariabler sker ved at indlæse eller udskrive alle elementerne i rækkefølge.

En variabelerklæring er enten en heltalsvariabel (angivet ved et navn) eller en tabelvariabel (angivet ved et navn efterfulgt af en konstant størrelse i firkantede parenteser). Der må ikke erklæres to variabler med samme navn. Dette skal checkes i typecheckeren.

Tabeller er nul-indicerede, dvs. at en tabel erklæret som `x[10]` har elementerne `x[0]`...`x[9]`. Det er udefineret, hvad der sker, hvis man bruger indices uden for grænserne, men der skal ikke i denne opgave implementeres køretidscheck for dette.

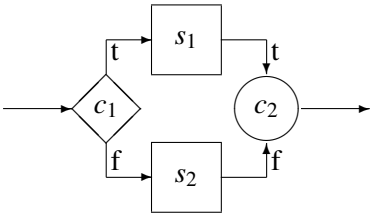
En procedure består af nøgleordet `procedure`, et navn på proceduren og en sætning, der udgør kroppen af proceduren. Der må ikke erklæres to procedurer med samme navn. Dette skal checkes i typecheckeren.

En sætning kan antage følgende former:

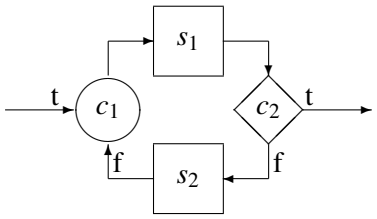
Opdatering. En opdatering består af en *Lval* ("left value"), som enten er en variabel eller et tabelelement, efterfulgt af en opdateringsoperator og et udtryk. Opdateringsoperatoren er enten `+=` eller `-=` og har samme semantik som de tilsvarende operatorer i C, dvs. opdatering af venstresiden med summen eller differencen af ventresiden og højresiden. Det er et krav (der skal checkes i typecheckeren), at den variabel eller tabelvariabel, der opdateres, ikke må forekomme i udtrykket på højresiden.

Sekvens. En sætning af formen s_1; s_2 udføres ved først at udføre s_1 og derefter udføre s_2.

Betinget sætning. En sætning af formen `if` c_1 `then` s_1 `else` s_2 `fi` c_2 udføres ved at beregne betingelsen c_1. Hvis denne er sand, udføres sætningen s_1, og det verificeres, at betingelsen c_2 er sand. Hvis dette ikke er tilfældet, skal programmet stoppes med en fejlmeddelelse. Hvis c_1 er falsk, udføres sætningen s_2 og det verificeres, at betingelsen c_2 er falsk. Hvis dette ikke er tilfældet, skal programmet stoppes med en fejlmeddelelse. c_1 er altså en forgreningsbetingelse, mens c_2 er en *assertion*, der skal checkes. Konstruktionen kan illustreres med følgende *flowchart*, hvor en ruderformet kasse er en forgrening med udgange t og f (sand og falsk) og en cirkel er en *assertion* med indgange t og f. En *assertion* kræver, at betingelsen matcher indgangspilen, ellers meldes en køretidsfejl.



Løkke. En sætning af formen `from` c_1 `do` s_1 `loop` s_2 `until` c_2 udføres ved at beregne betingelsen c_1. Hvis denne er falsk, stoppes programmet med en fejlmeddelelse. Ellers udføres s_1 og c_2 beregnes. Hvis c_2 er sand, afsluttes løkken, ellers udføres s_2 og c_1 beregnes. Hvis c_1 er sand, stoppes programmet med en fejlmeddelelse, ellers gentages løkken ved at udføre s_1 osv. Modsat i betingede sætninger, er c_1 en *assertion*, mens c_2 er en forgreningsbetingelse. Konstruktionen kan illustreres med følgende *flowchart*.



Procedurekald. Et procedurekald har enten formen `call` p eller `uncall` p, hvor p er navnet på en procedure. `call` p udfører p's krop og returnerer derefter til lige efter kaldet. `uncall` p udfører p's krop *baglæns* og returnerer derefter til lige efter kaldet. Procedurekald kan være gensidigt rekursive.

Skip. En sætning af formen `skip` har ingen effekt.

Udtryk er konstanter, variabler, tabelopslag eller en operator andvendt på en eller to udtryk. Operatorer er `+` (addition), `-` (subtraktion) og `/2` (halvering). Beregning sker som af tilsvarende udtryk i C.

Betingelser er sammenligning af udtryk med sammenligningsoperatorerne `<` (mindre end) eller `==` (lig med) eller en logisk operator anvendt på et eller flere logiske udtryk. De logiske operatorer `!` (negation), `&&` (konjunktion) og `||` (disjunktion) har samme betydning som i C.

5.1 Baglæns udførsel af sætninger

Idet procedurer kan udføres baglæns med `uncall`, skal alle sætninger kunne udføres baglæns. Derfor skal der for hver procedure laves to oversættelser: En, der udfører procedurens krop forlæns og en, der udfører procedurens krop baglæns. Oversættelse til baglæns udførsel af en sætning kan ske ved først at invertere sætningen og derefter oversætte denne normalt. Inverting af sætninger sker ved funktionen R, som er beskrevet ved følgende regler:

$$
\begin{array}{lcl}
R(lv \texttt{ += } e) & = & lv \texttt{ -= } e \\
R(lv \texttt{ -= } e) & = & lv \texttt{ += } e \\
R(s_1\texttt{; } s_2) & = & R(s_2)\texttt{; } R(s_1) \\
R(\texttt{if } c_1 \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi } c_2) & = & \texttt{if } c_2 \texttt{ then } R(s_1) \texttt{ else } R(s_2) \texttt{ fi } c_1 \\
R(\texttt{from } c_1 \texttt{ do } s_1 \texttt{ loop } s_2 \texttt{ until } c_2) & = & \texttt{from } c_2 \texttt{ do } R(s_1) \texttt{ loop } R(s_2) \texttt{ until } c_1 \\
R(\texttt{call } p) & = & \texttt{uncall } p \\
R(\texttt{uncall } p) & = & \texttt{call } p \\
R(\texttt{skip}) & = & \texttt{skip}
\end{array}
$$

6 En delmængde af Janus

Den udleverede oversætter håndterer kun en delmængde af Janus. Begrænsningerne er som følger:

- Betingelser, `if-then-else-fi` og `from-do-loop-until` er ikke implementeret.
- Tabeller er ikke implementeret.
- `uncall` er ikke implementeret.

Bemærk, at filen `Janus.sml` har abstrakt syntaks for hele sproget.

7 Abstrakt syntaks og oversætter

Filen `Janus.sml` angiver datastrukturer for den abstrakte syntaks for programmer i Janus. Hele programmet har type `Janus.Prog`.

Filen `JC.sml` indeholder et program, der kan indlæse, typechecke og oversætte et Janus-program. Det kaldes ved at angive filnavnet for programmet (uden extension) på kommandolinien, f.eks. `JC fib2`. Extension for Janus-programmer er `.jan`, f.eks. `fib2.jan`. Når Janus-programmet er indlæst og checket, skrives den oversatte kode ud på en fil med samme navn som programmet men med extension `.as`. Kommandoen "`JC fib2`" vil altså tage en kildetekst fra filen `fib2.jan` og skrive kode ud i filen `fib2.as`.

Den symbolske oversatte kode kan indlæses og køres af SPIM. Kommandoen "`spim fib2.as`" vil køre programmet og læse inddata fra standard input og skrive uddata til standard output.

Checkeren er implementeret i filerne `Type.sig` og `Type.sml`. Oversætteren er implementeret i filerne `Compiler.sig` og `Compiler.sml`.

Hele oversætteren kan genoversættes (inklusive generering af lexer og parser) ved at skrive `source compile` på kommandolinien (mens man er i et katalog med alle de relevante filer, inclusive `compile`).

8 Eksempelprogrammer

Der er givet en række eksempelprogrammer skrevet i Janus

`identity.jan` indlæser et tal og skriver samme tal ud.

`sumdif.jan` indlæser to tal og skriver deres sum og differens ud.

`fib1.jan` indlæser et positivt tal n og udskriver tallene $fib(n)$ og $fib(n+1)$, hvor fib er Fibonacci's funktion.

`fib2.jan` indlæser et positivt tal n og udskriver tallene $fib(n+1)$, $fib(n)$ og n.

`fib3.jan` indlæser $fib(n+1)$, $fib(n)$ og n og udskriver n. `fib3.jan` bruger samme rekursive procedure som `fib2.jan`, men kalder den med `uncall` for at udføre den baglæns.

`fall.jan` indlæser en tid t og udskriver hastighed og højde af en genstand, der falder i t sekunder startende med hastighed 0 og højde 0.

`encrypt.jan` indlæser en nøgle og et tal og udskriver samme nøgle og et „krypteret" tal.

`decrypt.jan` beregner den inverse funktion til `encrypt.jan` ved at bruge `uncall` på samme procedure.

`logic.jan` tester de logiske operatorer.

`reverse.jan` indlæser 10 tal og skriver dem ud i omvendt rækkefølge.

`sum.jan` indlæser 10 tal og skriver summerne af de 10 ikke-tomme præfikser ud.

`stack.jan` simulerer en tre-element stak med en tabel.

Hvert eksempelprogram *program*`.jan` skal oversættes og køres på inddata, der er givet i filen *program*`.in`. Uddata fra kørslen af et program skal stemme overens med det, der er givet i filen *program*`.out`. Hvis der ikke er nogen *program*`.in` fil, køres programmet uden inddata.

Kun `identity.jan` og `sumdif.jan` kan oversættes med den udleverede oversætter; de andre programmer bruger de manglende sprogelementer.

Der er endvidere givet et antal nummererede testprogrammer (`error00.jan, ..., error15.jan`), der indeholder diverse fejl eller inkonsistenser. (`error00.jan, ..., error09.jan`) indeholder fejl, der skal fanges i checkeren. Der er ikke input- eller outputfiler til disse programmer. (`error10.jan, ..., error15.jan`) fejler *assertions* under kørslen eller har ikke-nulstillede variabler ved afslutning. De skal derfor kunne oversættes uden fejl, men skal ved kørsel med de tilhørende inputfiler give fejlmeddelelser, der angiver fejlenes omtrentlige position i programmerne.

Selv om testprogrammerne kommer godt rundt i sproget, kan de på ingen måde siges at være en udtømmende test. Man bør vurdere, om der er ting i oversætteren, der ikke er testet, og lave yderligere testprogrammer efter behov.

Selv om registerallokatoren ikke laver spill, er der rigeligt med registre til at eksempelprogrammerne kan oversættes uden spill. Derfor betragtes det som en fejl, hvis registerallokatoren rejser undtagelsen `not_colourable` for et af eksempelprogrammerne.

9 Milepæle

Da opgaven først skal afleveres efter fem uger, kan man fristes til at udskyde arbejdet på opgaven til sidst i perioden. Dette er en meget dårlig ide. Herunder er angivet retningslinier for hvornår de forskellige komponenter af oversætteren bør være færdige, inklusive de dele af rapporten, der beskriver disse.

Uge 47 Lexeren kan genereres og oversættes (husk at erklære de nye tokens i parseren). Rapportafsnit om lexer skrives.

Uge 48 Parseren kan genereres og oversættes. Rapportafsnit om lexer og parser færdigt.

Uge 49 Checkeren er implementeret. Rapportafsnit om checker skrives.

Uge 50 Oversætteren er implementeret, rapportafsnit om denne skrives.

Uge 51 Afsluttende afprøvning og rapportskrivning, rapporten afleveres om onsdagen.

Bemærk, at typechecker og kodegenerering er væsentligt større opgaver end lexer og parser.

Efter hvert af de ovenstående skridt bør man genoversætte hele oversætteren og prøvekøre den for testprogrammerne. De endnu ikke udvidede moduler kan ved oversættelse rapportere om ikke-udtømmende pattern-matching, og ved køretid kan de rejse undtagelsen "Match". Man kan i `JC.sml` udkommentere kald til de senere faser for at afprøve sprogudvidelserne for de moduler (faser), der allerede er implementerede.

Jeres instruktor vil gerne løbende læse og komme med feedback til afsnit af rapporten. I skal dog regne med, at der kan gå noget tid, inden I får svar (så bed ikke om feedback lige før afleveringsfristen), og I skal ikke forvente, at et afsnit bliver læst igennem flere gange.

10 Vink

- *Assertions* har to indgange: En, hvor betingelsen skal være sand, og en, hvor den skal være falsk. Overvej nøje, hvordan dette implementeres.

- `uncall` kan implementeres ved at oversætte hver procedure i to udgaver: En forlæns og en baglæns. Den baglæns udgave kan laves ved at invertere kroppen (som beskrevet i afsnit 5.1) og derefter oversætte den inverterede krop normalt.
- Tabeller har konstant størrelse, så de kan allokeres i `data` arealet med assemblerdirektivet `.space`. Se dokumentationen af SPIM. Bemærk, at tabeller skal initialiseres til 0, hvilket ikke gøres af `.space`.
- Bemærk, at der ikke bruges symboltabel til variabler idet alle variabler er globale, og ikke to har samme navn. Derfor bruges variablenavne uændret i den genererede kode. Det betyder, at navne på labels og temporære variabler i den genererede kode skal adskilles fra variabelnavne, f.eks. ved at lade dem starte med *underscore*.
- **KISS**: *Keep It Simple, Stupid*. Lav ikke avancerede løsninger, før I har en fungerende simpel løsning, inklusive udkast til et rapportafsnit, der beskriver denne. Udvidelser og forbedringer kan derefter tilføjes og beskrives som sådan i rapporten.
- I kan antage, at læseren af rapporten er bekendt med pensum til kurset, og I kan frit henvise til kursusbøger, noter og opgavetekster.
- Hver gang I har ændret i et modul af oversætteren, så genoversæt hele oversætteren (med `source compile`). Dog kan advarsler om "pattern matching is not exhaustive" i reglen ignoreres indtil alle moduler er udvidede.
- Når man oversætter signaturen til den genererede parser, vil `mosmlc` give en "Compliance Warning". Denne er uden betydning, og kan ignoreres.
- Når I udvider lexeren, skal I erklære de nye tokens i parseren med `%token` erklæringer og derefter generere parseren og oversætte den *inden* i oversætter lexeren, ellers vil I få typefejl.
- I lexerdefinitionen skal enkelttegn stå i *backquotes* (`` ` ``), *ikke* almindelige anførselstegn (`'`), som i C eller Java.

Litteratur

[1] Tetsuo Yokoyama, Robert Glück: A Reversible Programming Language and its Reversible Interpreter, PEPM 2007.