En oversætter for 100

Godkendelsesopgave på kurset Oversættere

Vinter 2011/12

1 Introduktion

Dette er projektopgaven på Oversættere, vinter 2011 – 2012. Opgaven skal løses i grupper på op til 3 personer. Opgaven bliver stillet mandag d. 21/11 2011 og skal afleveres senest fredag d. 22/12 2011. Opgaven afleveres via kursushjemmesiden på Absalon. Brug gruppeafleveringsfunktionen i Absalon. Alle medlemmer af gruppen skal på rapportforsiden angives med navn. Der er ikke lavet en standardforside, så lav en selv.

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

2 Om opgaven

Opgaven går ud på at implementere en oversætter for sproget 100, som er beskrevet i afsnit 3.

Som hjælp hertil gives en fungerende implementering af en delmængde af 100. I afsnit 6 er denne delmængde beskrevet.

Der findes på kursussiden en zip-fil kaldet "G.zip", der indeholder opgaveteksten, implementeringen af delmængden af 100 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 100 med parseraktioner, der opbygger den abstrakte syntaks.

`Lexer.lex` Leksikalske definitioner for *tokens* i 100.

`Type.sml` Typechecker for 100.

`Compiler.sml` Oversætter fra 100 til MIPS assembler. Oversættelsen sker direkte fra 100 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. Filerne `compile.sh` og `compile.bat` indeholder kommandoer for hhv. Linux og Windows til 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 MARS. Der er link til denne fra kursets Absalonside.

Krav til besvarelsen

Besvarelsen afleveres som en PDF fil med rapporten samt en zip-fil, der indeholder 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).

Filerne afleveres via kursushjemmesiden. Brug gruppeaflevering – der skal *ikke* afleveres en kopi pr. gruppemedlem.

Rapportforsiden skal angive alle medlemmer af gruppen med navn.

Rapporten skal indeholde en begrundet 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*.

Der vil primært lægges vægt på, at sproget implementeres korrekt, men effektivitet af den genererede kode kan også inddrages. Optimeringer af særtilfælde vægtes ikke særligt højt, men omtanke omkring den almindelige kodegenerering gør. Hvis der er åbenlyse ineffektiviteter ved generering af almindelig kode, vil forsøg på optimeringer af særtilfælde ligefrem kunne trække ned, da det vidner om forkert prioritering eller manglende forståelse.

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. Endvidere skal det vurderes, i hvilket omfang de udleverede testprogrammer

er dækkende og der skal laves nye testprogrammer, der dækker de største mangler ved testen.

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 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 bør holdes under 16 sider. Al væsentlig information om løsningen bør medtages i rapporten, så man i det væsentlige ikke behøver at læse jeres kildekode. Men for mange irrelevante detaljer og udenomssnak vil trække ned. Alle væsentlige designbeslutninger skal dog beskrives og begrundes, og alle fejl og mangler skal beskrives.

2.1 Afgrænsninger af oversætteren

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

Hovedprogrammet `CC.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 og at tupler er så små, at størrelse og *offsets* kan ligge i konstantfeltet i en instruktion.

Det ikke nødvendigt at frigøre lager i 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.

3 100

100 er et simpelt imperativt programmeringssprog inspireret af C.

Herunder beskrives syntaks og uformel semantik for sproget 100 og en kort beskrivelse af de filer, der implementerer sproget.

Prog → *FunDecs*

FunDecs → *Type Sid* (*Decs*) *Stat FunDecs*
FunDecs →

Type → `int`
Type → `char`

Decs →
Decs → *Decs*1 *Dec*

*Decs*1 →
*Decs*1 → *Decs*1 *Dec* ;

Dec → *Type Sids*

Sids → *Sid*
Sids → *Sid* , *Sids*

Sid → **id**
Sid → * **id**

Stats →
Stats → *StatStats*

Stat → *Exp* ;
Stat → `if` (*Exp*) *Stat*
Stat → `if` (*Exp*) *Stat* `else` *Stat*
Stat → `while` (*Exp*) *Stat*
Stat → `return` *Exp* ;
Stat → { *Decs*1 *Stats* }

Exp → **numConst**
Exp → **charConst**
Exp → **stringConst**
Exp → *Lval*
Exp → *Lval* = *Exp*
Exp → *Exp* + *Exp*
Exp → *Exp* - *Exp*
Exp → *Exp* < *Exp*
Exp → *Exp* == *Exp*
Exp → (*Exp*)
Exp → **id** (*Exps*)
Exp → (*Exp*)

Exps →
Exps → *Exps*1

*Exps*1 → *Exp*
*Exps*1 → *Exp* , *Exps*1

Lval → **id**
Lval → **id** *
Lval → **id** [*Exp*]

Figur 1: Syntaks for 100

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 (**numConst**) er ikke-tomme følger af cifrene 0-9. Talkonstanter er begrænset til tal, der kan repræsenteres som positive heltal i Moscow ML.
- En tegnkonstant (**charConst**) består af en tegnspecifikator omgivet af enkelte anførselstegn (`'`). En tegnspecifikator kan være en af følgende:

  1. Et tegn med ASCII kode mellem 32 og 126 *undtagen* tegnene `'`, `"` og `\`.
  2. Tegnet `\` efterfulgt af et tegn med ASCII kode mellem 32 og 126.

  Kun tegnspecifikationer, der er lovlige i C, er tilladt.
- En stringkonstant (**stringConst**) består af en sekvens af tegnspecifikatorer omgivet af dobbelte anførselstegn (`"`).
- Operatorerne + og - har samme præcedens og er begge venstreassociative.
- Operatorerne < og == har samme præcedens og er begge venstreassociative. De binder begge svagere end +.
- Tildelingsoperatoren = binder svagere end < og er højreassociativ.
- `else` binder til nærmeste `if` som beskrevet i lærebogen.
- Der er separate navnerum for variabler og funktioner.
- Kommentarer starter med `/*` og slutter med `*/`. Mellem disse kan være vilkårlige tegn som ikke indeholder sekvensen `*/`.

5 Semantik

Hvor intet andet er angivet, er semantikken for de forskellige konstruktioner i sproget identisk med semantikken for tilsvarende konstruktioner i C.

Et 100 program består af funktionserklæringer. Der skal være en funktion med navn `main` som ikke har nogen parametre og hvor returtypen er `int`. Kørsel af et program sker ved kald til denne funktion. Alle funktioner har virkefelt i hele programmet, så de er gensidigt rekursive. Der må ikke erklæres to funktioner med

samme navn. Bemærk, at 100 her adskiller sig fra C, hvor kun funktioner, der er erklæret tidligere kan ses, og hvor det er muligt at omdefinere funktioner.

Udover heltal (typen `int`) og tegn (typen `char`) har 100 også referencer. En variabel `x` erklæret som `int *x` er f.eks. en reference til et heltal. Den bagvedstillede operator `*` følger en reference til dens indhold. Referencer kan bruges som tabeller, så hvis f.eks. `x` er erklæret som `int *x` vil `x[7]` referere til det heltal, der ligger 7 maskinord efter det maskinord, som `x` peger på, og hvis `s` er erklæret som `char *s` vil `s[7]` referere til det tegn, der ligger 7 bytes efter den byte, som `s` peger på. Bemærk, at kodegenerering af et udtryk af formen `x[`e`]` kræver kendskab til typen af `x`. Det er udefineret, hvad der sker, hvis man bruger offsets udenfor det område, der er allokeret til referencen/tabellen. Endvidere er det udefineret, hvad der sker, hvis man følger referencen i en uinitialiseret referencevariabel.

I udtryk gøres ikke forskel på tegn (`char`) og heltal (`int`): De repræsenteres begge som heltal. Det er kun, når værdier gemmes og hentes i variabler eller tabeller, at der gøres forskel: Når der hentes og gemmes i tegnvariabler eller tegntabeller overføres 8 bit, mens der overføres 32 bit, når der hentes og gemmes i heltalsvariabler eller heltalstabeller. Referencevariabler fylder også 32 bit, men konverteres ikke automatisk til eller fra heltal.

Bemærk, at det betyder, at et udtryk af type `int` er et lovligt argument til en funktion, der har en parameter af type `char`. Alle parametre overføres som hele maskinord i registre eller lager, men kun de mindst betydende 8 bit af `char` parametre bruges.

En tegnreference kan også ses som en string bestående af alle tegn fra det, referencen peger på, frem til den nærmeste nulbyte. En stringkonstant er en reference til en string, der indeholder de angivne tegn og afsluttes med en nulbyte. Tegnet `\` bruges i tegn- og stringkonstanter som escape-tegn, der indkoder tegn, der ellers ikke er lovlige i en tegnkonstant. De tegn, der kan komme efter et `\` er de samme som i C. Det er udefineret, hvad der sker, hvis indholdet af en stringkonstant modificeres på køretid.

Operatoren `+` kan tage både heltal og referencer som argumenter. Reglerne er:

- Hvis begge argumenter er heltal, er resultatet et heltal.
- Hvis det ene argument er et heltal n og det andet er en heltalsreference r, er resultatet en reference til et heltal, der er n maskinord efter det, som r peger på.
- Hvis det ene argument er et heltal n og det andet er en tegnreference r, er resultatet en reference til et tegn, der er n byte efter det, som r peger på.
- Hvis begge argumenter er referencer, er det en typefejl.

Operatoren `-` kan tage både heltal og referencer som argumenter. Reglerne er:

- Hvis begge argumenter er heltal, er resultatet et heltal.

- Hvis det første argument er et heltal og det andet er en reference, er det en typefejl.
- Hvis det første argument er en heltalsreference *r* og det andet argument er et heltal *n*, er resultatet en reference til et heltal, der er *n* maskinord før det, som *r* peger på.
- Hvis det første argument er en tegnreference *r* og det andet argument er et heltal *n*, er resultatet en reference til et tegn, der er *n* bytes før det, som *r* peger på.
- Hvis begge argumenter er heltalsreferencer, er resultatet et heltal, der angiver antallet af maskinord mellem de to referencer.
- Hvis begge argumenter er tegnreferencer, er resultatet et heltal, der angiver antallet af bytes mellem de to referencer.
- Hvis de to argumenter er referencer af forskellig type, er det en typefejl.

Bemærk, at den genererede kode for + og - afhænger af typen af argumenterne, så man skal i oversætteren holde styr på typen af udtryk, selv om dette allerede er gjort i typecheckeren.

Operatorerne < og == kan tage både heltal og referencer som argumenter. Begge argumenter skal være af samme type. Hvis sammenligningen er sand, returneres tallet 1, ellers tallet 0.

Tildelingen $l = e$ beregner l til enten en variabel eller en adresse og beregner e til en værdi, som lægges i variablen eller adressen. Værdien af tildelingen er den gemte værdi. Hvis l er en heltalsvariabel, et element i en heltalstabel eller indholdet af en heltalsreference, skal e være et heltalsudtryk. Hvis l er en tegnvariabel, et element i en tegntabel eller tegnreference, skal e være et heltalsudtryk, men kun de sidste 8 bit af værdien af e gemmes i l. Hvis l er en referencevariabel skal e beregne en reference af den samme type. Alle disse begrænsninger skal checkes i typecheckeren.

Betingelser i `if` og `while` sætninger er udtryk af heltalstype og betragtes som sande, hvis deres værdi er forskellig fra 0. Både `if` og `while` sætninger fungerer på samme måde som i C.

I en variabelerklæring (*Decs* eller *Decs*1) må det samme navn ikke forekomme flere gange (dette skal checkes), men forskellige erklæringer kan godt erklære variable med samme navn. Variable, der er erklæret i en erklæring lige efter et `{` har virkefelt (*scope*) indtil det matchende `}`. Parametre til en funktion har virkefelt i kroppen af funktionen.

Det betragtes som en fejl, hvis en funktion kan afslutte uden at udføre en `return` sætning. Typecheckeren skal angive fejl, hvis dette er muligt, eller hvis typen af et udtryk, der er angivet efter `return`, ikke har samme type som funktionens erklærede resultattype. Man kan antage, at alle betingelser i `if`- og `while`-sætninger kan være både sande og falske, så hvis der f.eks. ikke er en `return`-

sætning efter en `if-else`-sætning, skal begge grene af `if-else`-sætningen indeholde en `return`-sætning.

Der er et antal foruddefinerede funktioner, der kan kaldes fra programmet:

- `walloc()` tager et heltal n og returnerer en *word-aligned* reference til mindst n maskinord, der er allokeret på hoben. Returtypen er `int *`.
- `balloc()` tager et heltal n og returnerer en reference til mindst n bytes, der er allokeret på hoben. Returtypen er `char *`.
- `getint()` tager ingen argumenter men returnerer et heltal, der er læst fra standard input.
- `getstring()` tager et heltal n som argument og returnerer en hob-allokeret string, der indeholder op til $n-1$ tegn læst fra standard input efterfulgt af en nul-byte. Der læses kun frem til og med nærmeste linjeskift. Hvis linjen slutter inden, der er læst $n-1$ tegn, læses frem til og med linjeskiftet og der tilføjes derefter et nul-byte. Uanset dette allokeres mindst n byte på hoben til indholdet.
- `putint()` tager et heltal som argument og skriver det ud på standard output. Resultatet er det samme som argumentet.
- `putstring()` tager en string som argument og skriver det ud på standard output. Resultatet er det samme som argumentet.

Kode for disse funktioner skal tilføjes koden for det oversatte program. Brug de systemkald for i/o, der er beskrevet i dokumentationen til SPIM i bogen *Computer Organisation & Design*, som blev brugt på arkitekturkurset. Bemærk, at denne dokumentation kan downloades fra kursussiden.

Navne på foruddefinerede funktioner er ikke er nøgleord, så variable kan godt hedde det samme. Der kan dog ikke defineres nye funktioner med disse navne.

6 En delmængde af 100

Den udleverede oversætter håndterer kun en delmængde af 100. Begrænsningerne er som følger:

- Referencer, referencetyper og alle operationer på referencer er ikke implementeret.
- Typen `char` og alle operationer på tegn er ikke implementeret.
- Sammenligningsoperatoren == er ikke implementeret.
- Løkker og blokke ({ ... }) er ikke implementeret.

- Der checkes ikke om `return`-sætninger returnerer samme type som funktionen eller om en funktion kan afslutte uden at komme til en `return`-sætning.

Bemærk, at filen `S100.sml` har abstrakt syntaks for resten af sproget i kommentarer.

7 Abstrakt syntaks og oversætter

Filen `S100.sml` angiver datastrukturer for den abstrakte syntaks for programmer i 100. Hele programmet har type `S100.Prog`.

Filen `C100.sml` indeholder et program, der kan indlæse, typechecke og oversætte et 100-program. Det kaldes ved at angive filnavnet for programmet (uden extension) på kommandolinien, f.eks. `C100 fib`. Extension for 100-programmer er `.100`, f.eks. `fib.100`. Når 100-programmet er indlæst og checket, skrives den oversatte kode ud på en fil med samme navn som programmet men med extension `.asm`. Kommandoen "`C100 fib`" vil altså tage en kildetekst fra filen `fib.100` og skrive kode ud i filen `fib.asm`.

Den symbolske oversatte kode kan indlæses og køres af MARS. Kommandoen "`java -jar Mars.jar fib2.asm`" vil køre programmet og læse inddata fra standard input og skrive uddata til standard output. Lav evt. en makro til denne kommando.

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

Hele oversætteren kan i Linux og MacOS genoversættes (inklusive generering af lexer og parser) ved at skrive `source compile.sh` på kommandolinien (mens man er i et katalog med alle de relevante filer, inklusive `compile.sh`). I Windows bruges `compile.bat` i stedet.

Som hjælp til debugging af parser kan man bruge programmet `SeeSyntax.sml`. Hvis man kører dette program i det interaktive system (`mosml SeeSyntax.sml`) kan man bruge funktionen `showsyntax` med et filnavn som argument, og se ML datastrukturen for den abstrakte syntaks.

8 Eksempelprogrammer

Der er givet en række eksempelprogrammer skrevet i 100:

`fib.100` indlæser et ikke-negativt tal n og udskriver $fib(n)$, hvor fib er Fibonacci's funktion.

`copy.100` kopierer 14 tegn fra input til output.

`charref.100` læser et tal i og en string s ind og udskriver ASCII-koden for det $(i-1)$'te tegn i s. Er medtaget fordi en funktion returnerer en tegnreference.

`sort.100` indlæser et tal *n* og derefter *n* tal, som sorteres og skrives ud.

`sort2.100` gør det samme som `sort.100`, men bruger referencearitmetik.

`ssort.100` indlæser en string *s*, sorterer tegnene i denne og skriver resultatet ud.

`ssort2.100` gør det samme som `ssort.100`, men bruger referencearitmetik.

textttdfa.100 læser et binært tal ind og skriver dets divisionsrest modulo 3 ud.

Hvert eksempelprogram *program*`.100` 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.

Der er endvidere givet et antal nummererede testprogrammer (`error01.100, ..., error15.100`), der indeholder diverse fejl eller inkonsistenser, der skal fanges i checkeren. Der er ikke input- eller outputfiler til disse programmer.

Kun `fib.100` kan oversættes med den udleverede oversætter. De andre programmer bruger de manglende sprogelementer, og vil derfor give fejl allerede under lexing eller parsing.

Selv om testprogrammerne kommer godt rundt i sproget, kan de på ingen måde siges at være en udtømmende test hverken af normal kørsel eller fejlsituationer. Vurder, om der er ting i oversætteren, der ikke er testet, og lave yderligere testprogrammer efter behov. Hvis bedømmerne finder en fejl i jeres kode, som ville være afsløret af et simpelt ekstra testprogram, vil det trække ned. Derimod vil det give bonus, hvis ekstra testprogrammer afslører fejl, som sidenhen rettes. Beskrive gerne dette i jeres testafsnit.

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 skrives.

Uge 49 Typecheckeren er implementeret. Rapportafsnit om typechecker skrives.

Uge 50 Oversætteren er implementeret, rapportafsnit om denne skrives.

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

Bemærk, at typechecker og kodegenerering er væsentligt større opgaver end lexer og parser. Lexeren kan udvides og genereres på en times tid og parseren på 2–3 timer. Typecheckeren vil nok kræve 5–8 timers fuldtidsarbejde og kodegeneratoren 10–20 timer. Her er ikke medregnet tid til at læse op på stoffet i bogen – tiderne forudsætter, at man har nogenlunde styr på de relevante dele af pensum.

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 `C100.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

- **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.sh`). 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. Det er tilladt at bruge dobbelte anførslestegn (`"`) også om enkelttegn.

- I kan bruge ML-funktionerne `String.toCString` og `String.fromCString` til at indsætte og afkode escapesekvenser i stringkonstanter og `Char.toCString`

og `Char.fromCString` til ditto for tegnkonstanter. De kan også bruges til at checke, om escapesekvenser i tegn- og stringkonstanter er lovlige. MARS bruger C's escapesekvenser i f.eks. `.asciiz` direktivet.