Udvidelser af Janus-oversætter

Karakteropgave på kurset Oversættere

Vinter 2009

1 Introduktion

Dette er den anden del af rapportopgaven på Oversættere, vinter 2009. Opgaven skal løses individuelt. Opgaven bliver stillet fredag d. 22/1 2009 og skal afleveres via Absalon senest fredag d. 29/1 2009 kl. 23.55.

En forudsætning for at lave denne K-opgaver er, at man dette eller tidligere år har fået godkendt G-opgaven samt opnået enten godkendelse af fire ud af de fem ugeopgaver eller en bestået skriftlig eksamen fra tidligere år.

Der er stillet en (lille) fælles opgave og fire alternative tillægsopgaver. Det er et krav, at studerende, der har været i samme gruppe til G-opgaven, skal vælge forskellige tillægsopgaver i K-opgaven. Der trækkes lod om førstevælgerretten på følgende måde: Herunder er angivet en permutation af tallene fra 1 til 31. Alle gruppemedlemmer finder deres fødselsdato (dag i måneden) i listen, og den, hvis dato står først i listen, vælger først og så fremdeles. Hvis to gruppemedlemmer har samme fødselsdato gentages processen med månedsnummeret. Hvis dette også er det samme, så kontakt den kursusansvarlige, som vil rulle en stor terning.

14 12 17 9 16 23 18 31 7 5 10 20 27 30 26
21 15 11 3 24 6 1 29 19 22 25 13 28 2 4 8

Hvis I ikke kan få kontakt til alle jeres gruppemedlemmer inden fredag d. 22/1 kl. 15.00, så kontakt den kursusansvarlige (Torben), som så vil finde en løsning.

Hvis du har bestået G-opgaven et tidligere år, og derfor ikke har lavet G-opgaven om Janus, kan du frit vælge din tillægsopgave uden lodtrækning.

2 Om opgaven

Opgaven går ud på at udvide den oversætter for sproget Janus, som er beskrevet i G-opgaven.

Som hjælp hertil gives en implementering af G-opgaven, som er afprøvet til at virke for de til G-opgaven udleverede testprogrammer (og lidt til).

På kursushjemmesiden findes en zip-fil med opgaveteksten (som du læser nu), implementeringen af G-opgaven samt testprogrammerne.

Det er nødvendigt at modificere følgende filer:

`Janus.sml` Datatypeerklæringer for den abstrakte syntaks for Janus.

`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 (inklusive de 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 skal være en zip-fil bestående af en en rapport i PDF-format samt alle kildetekster, inklusive ekstra testprogrammer og deres inddata.

Rapportens forside skal tydeligt angive opgaveløserens eksamensnummer, navn og CPR-nummer samt navn og nummer på den valgte tillægsopgave.

Rapporten skal indeholde en kort beskrivelse af de ændringer, der laves i ovenstående komponenter. Beskrivelsen skal begrunde alle væsentlige valg omkring design og implementering af løsningen.

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. Suppler evt. beskrivelserne med figurer i stil med figur 6.2 og 7.3 i *Basics of Compiler Design*.

Du skal ikke inkludere hele programteksterne i rapportteksten, men du kan inkludere de væsentligt ændrede eller tilføjede dele af programmerne i rapportteksten som figurer, bilag e.lign. Hvis der henvises til kildetekst i en vedlagt fil, skal filens navn og linjenumrene på den omtalte del angives.

Rapporten skal beskrive hvorvidt oversættelse og kørsel af testprogrammer (jvf. afsnit 6) giver den forventede opførsel, samt beskrivelse af afvigelser derfra. Testens resultat skal vurderes, og evt. mangler i testen skal beskrives. Ekstra testprogrammer, der afhjælper de vigtigste mangler kan evt. laves og køres.

Som led i bedømmelse af opgaven vil udvalgte testkørsler blive lavet på DIKUs maskiner. Jeres egen test bør derfor afvikles med SPIM på DIKUs maskiner (f.eks. tuxray maskinerne) for at sikre mod evt. afvigelser i opførslen af SPIM på forskellige maskiner.

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 dig selv at bestemme, hvad du mener er væsentligt at medtage i rapporten, sålænge de eksplicitte krav i dette afsnit er opfyldt.

Rapporten bør maksimalt fylde 10 sider, 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.

Oversætteren kan antage, at programmerne par passeret typecheckeren, så det er ikke nødvendigt igen at checke disse ting.

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 stak eller hob. 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 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 kildeteksten til et program, der kan indlæse, typechecke og oversætte et Janus-program. Dette program kaldes ved at angive filnavnet for programmet (uden extension) på kommandolinien, f.eks. `JC reverse`.

Extension for Janus-programmer er `.jan`, f.eks. `reverse.jan`. Når Janus-programmet er indlæst og typechecket, skrives den oversatte kode ud på en fil med samme navn som programmet men med extension `.as`. Kommandoen "`JC reverse`" vil altså tage en kildetekst fra filen `reverse.jan` og skrive kode ud i filen `reverse.as`.

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

Typecheckeren 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`).

Hvis du kører i et Windows-miljø kan du kopiere filen `compile` over i en fil `compile.bat` og ændre den sidste linje til

```
mosmlc -o JC.exe JC.sml
```

Vedlæg dog den oprindelige `compile` fil i din besvarelse, så det er nemmere for opgaveretterne at genoversætte din løsning.

4 Fælles opgave: input/output variable

Udover en af de nummererede tillægsopgaver, skal *alle* implementere følgende udvidelse:

Janus udvides, så det er tilladt, at en variabel forekommer både i listen af input variable of i listen af output variable. En variabel, der gør dette, skal indlæses fra inddata og udskrives ved programmets afslutning, men skal ikke (som andre output variable) initialiseres til 0 eller (som andre input variable) verificeres som 0 ved programmets afslutning.

Med denne udvidelse kan programmet `identity.jan` omskrives til:

```
// io-identity: Copies input to output

x -> x;
skip
```

Bemærk, at dette program er identisk med `error01.jan` fra G-opgaven, men efter udvidelsen med input/output variable er det nu et lovligt program.

Hvis en variabel forekommer både i inputlisten og outputlisten, skal den have samme type: En heltalsvariabel i inputlisten må ikke have samme navn om en tabelvariabel i outputlisten (eller omvendt), og hvis en tabelvariabel forekommer i både inputlisten og outputlisten, skal de to forekomster angive det samme antal elementer. Dette skal checkes af typecheckeren.

4.1 Testprogrammer

Testprogrammer til den fælles opgave starter med `io`, f.eks. `io-reverse.jan` og `io-error01.jan`. Se endvidere afsnit 6.

Endvidere vil de nummererede opgaver have testprogrammer, der bruger input/output variable.

5 De individuelle tillægsopgaver

Udover den fælles opgave, skal hver eksaminand vælge én af følgende fire tillægsopgaver. Se afsnit 1 for proceduren omkring valg af opgave.

5.1 Opgave 1: Stak

Syntaksen af Janus udvides med følgende produktioner:

$$
\begin{array}{lcl}
\textit{Stat} & \rightarrow & \texttt{push}\ \textit{Lval} \\
\textit{Stat} & \rightarrow & \texttt{pop}\ \textit{Lval} \\
\textit{Stat} & \rightarrow & \texttt{swap} \\
\textit{Cond} & \rightarrow & \texttt{empty}
\end{array}
$$

5.1.1 Semantik

Janusprogrammer kan nu bruge en stak til midlertidigt at gemme indholdet af variable. Stakken er tom ved programmets begyndelse, og skal være tom ved programmets afslutning. Dette skal verificeres ved kørslen af programmet.

`push` x lægger værdien af x på stakken og sætter den nye værdi af x til at være 0. x kan være en heltalsvariabel eller et tabelelement.

`pop` x verificerer, at x har værdien 0 og at stakken er ikke-tom (og stopper programmet med en fejlangivelse, hvis dette ikke er tilfældet). Derefter sættes den nye værdi af x til at være den værdi, der ligger øverst på stakken, og denne værdi afstakkes.

Eksempelvis kan man bytte om på indholdet af x og y med denne sekvens: `push` x; `push` y; `pop` x; `pop` y.

`swap` verificerer, at der er mindst to elementer på stakken (og stopper programmet med en fejlangivelse, hvis dette ikke er tilfældet). Derefter byttes om på de to øverste elementer på stakken.

Betingelsen `empty` er sand, hvis stakken er tom, og falsk, hvis stakken ikke er tom.

`push` og `pop` inverterer til hinanden og `swap` til sig selv:

$$
\begin{array}{lcl}
R(\texttt{push}\ lv) & = & \texttt{pop}\ lv \\
R(\texttt{pop}\ lv) & = & \texttt{push}\ lv \\
R(\texttt{swap}) & = & \texttt{swap}
\end{array}
$$

For at sikre invertibilitet skal følgende krav være opfyldt:

- i `push` x`[`e`]` må x ikke forekomme i e.
- i `pop` x`[`e`]` må x ikke forekomme i e.

Dette skal verificeres i typecheckeren.

Bemærk, at man ikke kan bruge samme stak, som bruges til returadresser, da man kan stakke en værdi inde i en procedure og afstakke den, efter man har forladt proceduren. Der skal afsættes stakplads nok til, at alle testprogrammerne kan afvikles, men det er ikke et krav, at der checkes for stakoverløb.

5.1.2 Opgavens art

Den overvejende del af opgaven ligger i kodegenerering.

5.1.3 Testprogrammer

Testprogrammer til opgave 1 starter med `stack`, f.eks. `stack-fib.jan` og `stack-error01.jan`. Se endvidere afsnit 6.

5.2 Opgave 2: Ombytning

Syntaksen af Janus udvides med følgende produktioner:

$$\begin{array}{lcl} \textit{Stat} & \rightarrow & \texttt{rotate}\ \textit{LvalList} \\ \textit{LvalList} & \rightarrow & \textit{Lval} \\ \textit{LvalList} & \rightarrow & \textit{Lval}\ \textit{LvalList} \end{array}$$

5.2.1 Semantik

Sætningen `rotate` $x_1 \ldots x_n$, hvor x_i enten er en variabel eller et tabelelement, bytter om på indholdet af $x_1 \ldots x_n$ sådan at hvis $x_1 \ldots x_n$ inden ombytningen har værdierne $v_1,\ldots, v_n$ så har de efter ombytningen værdierne $v_2,\ldots, v_n, v_1$. Der er altså tale om en rotation mod venstre. Eksempel:

Hvis `x` har værdien 7, `a[13]` har værdien 9 og `y` har værdien 13, så vil udførsel af sætningen `rotate x a[13] y` give `x` værdien 9, `a[13]` værdien 13 og `y` værdien 7.

Bemærk, at $n = 1$ er tilladt, dvs. at der kun er et element i listen. I givet fald har sætningen `rotate` x_1 ingen effekt.

Man kan invertere en `rotate`-sætning ved at invertere rækkefølgen af variablerne og tabelelementerne:

$$R(\texttt{rotate}\, x_1 \ldots x_n) \quad = \quad \texttt{rotate}\, x_n \ldots x_1$$

For at sikre invertibilitet, er det et krav, at en variabel ikke må forekomme mere end en gang listen. Mere præcist:

- Hvis x_i er et tabelopslag af formen $a[e]$, så må a ikke forekomme i e.
- Hvis x_i og x_j (hvor $i \neq j$) begge er variable, skal x_i og x_j være forskelllige.
- Hvis x_i er en variabel og x_j er et tabelopslag af formen $a[e]$, så må x_i ikke forekomme i e.
- Hvis x_i er et tabelopslag af formen $a[e_1]$, og x_j (hvor $i \neq j$) er et tabelopslag af formen $b[e_2]$, så skal a og b være forskellige og a må ikke forekomme i e_2 og b må ikke forekomme i e_1.

Dette skal verificeres af typecheckeren.

5.2.2 Opgavens art

Der er nogenlunde lige dele typecheck og kodegenerering.

5.2.3 Testprogrammer

Testprogrammer til opgave 2 starter med `rotate`, f.eks. `rotate-fib.jan` og `rotate-error01.jan`. Se endvidere afsnit 6.

5.3 Opgave 3: For-løkker

Syntaksen af Janus udvides med følgende produktioner:

$$\begin{array}{lcl} \mathit{Stat} & \rightarrow & \texttt{for}\ \textbf{id}\ \texttt{from}\ \textbf{id}\ \texttt{upto}\ \textbf{id}\ \texttt{do}\ \mathit{Stat}\ \texttt{od} \\ \mathit{Stat} & \rightarrow & \texttt{for}\ \textbf{id}\ \texttt{from}\ \textbf{id}\ \texttt{downto}\ \textbf{id}\ \texttt{do}\ \mathit{Stat}\ \texttt{od} \end{array}$$

5.3.1 Semantik

Sætningen `for` x `from` y `upto` z `do` s `od` skal først checke at værdien af x er 0. Hvis ikke, laves en køretidsfejlmeddelelse.

Ellers checkes om værdien v_1 af y er mindre end eller lig med værdien v_2 af z. Hvis dette ikke er tilfældet, gøres ikke yderligere.

Hvis $v_1 \leq v_2$, udføres s $(v_2 - v_1 + 1)$ gange, først med x sat til v_1, dernæst med x sat til $v_1 + 1$ op til og med x sat til v_2. Derefter sættes x til at have værdien 0.

Varianten med `downto` gør ingenting, hvis $v_1 < v_2$. Men hvis $v_1 \geq v_2$, udføres s $(v_1 - v_2 + 1)$ gange, først med x sat til v_1, dernæst med x sat til $v_1 - 1$ ned til og med x sat til v_2. Ligesom før, skal der først checkes om x er 0 og tilsidst skal x sættes til 0.

Man kan invertere for-løkker på følgende måde:

$$\begin{array}{lcl} R(\texttt{for}\ x\ \texttt{from}\ y\ \texttt{upto}\ z\ \texttt{do}\ s\ \texttt{od}) & = & \texttt{for}\ x\ \texttt{from}\ z\ \texttt{downto}\ y\ \texttt{do}\ R(s)\ \texttt{od} \\ R(\texttt{for}\ x\ \texttt{from}\ y\ \texttt{downto}\ z\ \texttt{do}\ s\ \texttt{od}) & = & \texttt{for}\ x\ \texttt{from}\ z\ \texttt{upto}\ y\ \texttt{do}\ R(s)\ \texttt{od} \end{array}$$

For at sikre invertibilitet, skal følgende krav være opfyldt:

- x og y skal være forskellige variable.
- x og z skal være forskellige variable.
- Ingen af variablene x, y eller z må modificeres inde i s.
- s må ikke indeholde procedurekald.[1]

Dette skal verificeres af typecheckeren.

5.3.2 Opgavens art

Der er lidt overvægt af typecheck i forhold til kodegenerering.

5.3.3 Testprogrammer

Testprogrammer til opgave 3 starter med `for`, f.eks. `for-fib.jan` og `for-error01.jan`. Se endvidere afsnit 6.

[1] Fordi en procedure potentielt kan modificere x, y og z.

5.4 Opgave 4: Switch

Syntaksen af Janus udvides med følgende produktioner:

$$
\begin{array}{lcl}
\textit{Stat} & \rightarrow & \texttt{switch}\ \textbf{id}\ \texttt{in}\ \textit{Cases}\ \texttt{ni} \\
\textit{Cases} & \rightarrow & \textbf{num}\ \texttt{=>}\ \textit{Stat} \\
\textit{Cases} & \rightarrow & \textit{Cases}\ \texttt{|}\ \textit{Cases}
\end{array}
$$

Associativiteten af | har ikke betydning for semantikken, så du kan selv vælge, om du foretrækker venstre- eller højreassociativitet (men begrund valget).

5.4.1 Semantik

Sætningen `switch` x `in` cs `ni` vælger en case i cs, som på venstresiden angiver et tal, der er identisk med værdien af x, og udfører derefter højresiden. Hvis ingen venstreside matcher værdien af x, gøres ingenting.

Man kan invertere en `switch`-sætning på følgende måde:

$$
\begin{array}{lcl}
R(\texttt{switch}\ x\ \texttt{in}\ cs\ \texttt{ni}) & = & \texttt{switch}\ x\ \texttt{in}\ R_C(cs)\ \texttt{ni} \\
R_C(n\ \texttt{=>}\ s) & = & n\ \texttt{=>}\ R(s) \\
R_C(cs_1\ \texttt{|}\ cs_2) & = & R_C(cs_1)\ \texttt{|}\ R_C(cs_2)
\end{array}
$$

hvor funktionen R_C inverterer en liste af cases.

For at sikre invertibilitet, skal følgende krav være opfyldt:

- Ingen konstant må forekomme på venstresiden af mere end en case i samme `switch`-sætning.
- Switch-variablen x må ikke modificeres af nogen af højresiderne og højresiderne må ikke indeholde procedurekald.[2]

Dette skal verificeres af typecheckeren.

5.4.2 Opgavens art

Der er nogenlunde lige dele typecheck og kodegenerering.

5.4.3 Testprogrammer

Testprogrammer til opgave 4 starter med `switch`, f.eks. `switch-prime.jan` og `switch-error01.jan`. Se endvidere afsnit 6.

[2] Fordi en procedure potentielt kan modificere x.

6 Testprogrammer

Udover de i hver opgave angivne testprogrammer, skal alle testprogrammer fra G-opgaven kunne oversættes og køres med den nye oversætter eller give (type)fejlmeddelelser som før, med mindre den tidligere fejl ikke er en fejl i det udvidede sprog. For eksempel skal den nye oversætter *ikke* give typefejl for `error01.jan`.

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.

Der er endvidere givet et antal testprogrammer (med tegnfølgen `error` i filnavnet), der indeholder diverse typefejl eller andre inkonsistenser. For hvert program skal der meldes en relevant fejlmeddelelse og angives omtrentlig position i programmet for fejlen. Hvis der er en `.in` til programmet, skal programmet kunne oversættes uden typefejl, men skal give en fejlmeddelelse på køretid. Hvis ikke der er en `.in` fil, skal typecheckeren melde fejl.

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.

7 Vink

- Hvis du har spørgsmål til opgaven, så brug debatforummet. Der er flere, der kan hjælpe dig, og andre kan få glæde af de svar, du får. Du må ikke bede om hjælp til egentlig løsning af opgaverne, men du kan stille opklarende spørgsmål om opgaven og du kan spørge om SML, MosML-lex, MosML-yac, MIPS osv.

- Hvis du er i tvivl om den eksakte betydning af den udvidelse, du skal implementere, kan også du bruge testprogrammerne som guide: Hvis din fortolkning af semantikken ikke giver det samme uddata, som testprogrammernes `.out` filer, så er din tolkning (eller implementering) nok forkert. Bemærk dog, at den copyrighttekst, som SPIM altid udskriver, kan variere fra version til version, så tag højde for dette ved sammenligning.

Se endvidere vinkene i G-opgaven.