Udvidelser af oversætter for 100

Karakteropgave på kurset Oversættere

Vinter 2011

1 Introduktion

Dette er den karaktergivende eksamensopgave på Oversættere, vinter 2011. Opgaven skal løses individuelt. Opgaven bliver stillet mandag d. 16/1 2011 og skal afleveres via Absalon senest fredag d. 20/1 2011 kl. 23.55.

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

Der er stillet fem alternative opgaver. Det er et krav, at studerende, der har været i samme gruppe til G-opgaven, skal vælge forskellige opgaver 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 det gruppemedlem, hvis fødselsdato 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 (Torben), som vil rulle en stor terning.

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

Hvis I ikke kan få kontakt til alle jeres gruppemedlemmer inden mandag d. 16/1 kl. 15.00, så kontakt den kursusansvarlige (Torben), som så vil finde en løsning.

Hvis du har opnået eksamensberettigelse et tidligere år, og derfor ikke har en gruppe i år, kan du frit vælge din opgave uden lodtrækning, også selv om nogle af dine G-opgave gruppemedlemmer fra tidligere år også deltager i denne eksamen.

2 Om opgaven

Opgaven går ud på at udvide den oversætter for sproget 100, som er beskrevet i G-opgaven.

Som hjælp hertil er der givet en implementering af G-opgaven, som er afprøvet til at virke for de til G-opgaven udleverede testprogrammer. Du kan også vælge at bruge din egen eller en anden gruppes implementering af G-opgaven eller

kombinere flere. Under alle omstændigheder, skal rapporten klart angve, hvilken implementering af G-opgaven, der bruges som udgangspunkt.

På eksamensopgavesiden i Absalon findes en zip-fil med opgaveteksten (som du læser nu), filer til implementering af G-opgaven samt testprogrammer til udvidelserne i denne eksamensopgave (se afsnit 6).

Det er nødvendigt at modificere/udvide følgende filer:

`S100.sml` Datatypeerklæringer for den abstrakte syntaks for 100.

`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.

I nogle tilfælde kan det være nødvendigt eller fordelagtigt at modificere andre moduler. Hvis du gør dette, så begrund det i rapporten.

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 MARS.

Dine programmer bliver evalueret ved kørsel på DIKUs systemer. Da der kan være små forskelle i opførslen af Moscow ML og MARS på forskellige platforme, vil det være en god ide, at verificere, at din løsning kan køre korrekt på DIKUs systemer, inden du afleverer den. Ellers risikerer du, at din løsning bliver bedømt som ikke-virkende, selv om du kunne køre den på din egen maskine.

Krav til besvarelsen

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

Rapportens forside skal tydeligt angive opgaveløserens fulde navn, KU indentitet (svensk nummerplade) og CPR-nummer samt navn og nummer på den valgte opgave.

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 5.2 og 6.3 i *Introduction to Compiler Design*.

Du behøver ikke inkludere hele programteksterne i rapportteksten, men du skal 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. Bilag i form af kode tæller ikke med til sidebegrænsningen, men det anbefales alligevel kun at vedlægge de ændrede dele af programteksterne.

Rapporten skal beskrive hvorvidt oversættelse og kørsel af testprogrammer (jvf. afsnit 4) 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, skal laves og køres.

Kendte mangler i typechecker og oversætter skal beskrives, og i det omfang det er muligt, skal det skitseres, hvordan disse kan udbedres.

Rapporten skal ikke alene *beskrive*, hvordan du har løst opgaven, den skal også *evaluere*, hvor godt denne løsning fungerer.

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 ikke fylde mere end 10 sider (ekslusive forside), 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 har 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.

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

Filen `C100.sml` indeholder kildeteksten til et program, der kan indlæse, typechecke og oversætte et 100-program. Dette program 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 typechecket, 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 fib.asm`" vil køre programmet og læse inddata fra standard input og skrive uddata til standard output. Mars kan også køres interaktivt, men så skal man selv taste input ind i i/o-vinduet. Brug ikke copy/paste med musen, da det giver forkerte resultater.

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.sh` på kommandolinien (mens man er i et katalog med alle de relevante filer, inclusive `compile.sh`).

Hvis du kører i et Windows-miljø kan du bruge `compile.bat`, som adskiller sig fra `compile.sh` ved at sætte `.exe` på den eksekverbare `C100`-fil.

4 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 ligesom før.

Hvert eksempelprogram *program*`.100` (som ikke skal give typefejl) skal kunne 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` (undtaget forskelle i versionsbeskeden fra MARS). 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 ikke der er en `.in` fil til et `error` program, skal typecheckeren melde fejl. Hvis der er en `.in` til programmet, skal programmet kunne oversættes uden typefejl, men skal give en fejlmeddelelse som output. Der er i dette tilfælde ikke givet nogen `.out` fil, da formulering og positionsangivelser kan variere.

Testprogrammerne kan de på ingen måde siges at være en udtømmende test. Man skal vurdere, om der er ting i oversætteren, der ikke er testet, og lave yderligere testprogrammer efter behov.

5 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, MARS osv. Hvis der er et specifikt problem, der betyder, at du ikke kan komme videre, så bed Torben om hjælp. Husk at skrive i rapporten, at du fik denne hjælp.

- 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. Hvis dette ikke er nok til at afklare spørgsmålet, så beskriv din tolkning af betydningen. Se også første vink herover.

Se endvidere vinkene i G-opgaven.

6 Opgaverne

Hver eksaminand skal vælge én af følgende fem opgaver jvf. proceduren for valg af opgave, som er beskrevet i afsnit 1.

Besvarelsen afleveres i Absalon. Rapporten bør uploades som separat fil, og kodefiler m.m. uploades som en zip-fil.

Afleveringsfristen bør overholdes. Overskridelser vil som minimum gå ud over karakteren, og signifikante overskridelser vil medføre resultatet “Udeblevet”.

6.1 Opgave 1: Switch og break

Syntaksen af 100 udvides med følgende produktioner:

Stat	→	`switch (` *Exp* `) {` *Cases* `}`
Stat	→	`break ;`
Cases	→	`default :` *Stats*
Cases	→	`case` *ConstExp* `:` *Stats Cases*
ConstExp	→	**numConst**
ConstExp	→	**charConst**

6.1.1 Semantik

En sætning af formen `switch (`e`) {` cs`}` udregner e til et heltal n. Hvis der i cs er en gren af formen `case` k`:`ss cs', hvor værdien af k er lig med n udføres ss efterfulgt af alle sætninger i cs' (eller indtil et `break`, se herunder). Hvis ikke, udføres sætningen efter `default:`. Bemærk, at k kan være enten en heltalskonstant eller en tegnkonstant. e skal have typen `int`. Der må ikke være to grene (`case...`) med samme værdi (hvor en tegnkonstant har sin ASCII-værdi). Begge dele skal checkes af typecheckeren.

Semantikken af en sætning af formen `break;` afhænger af, hvor den står:

1. Hvis `break;` står inde i kroppen af en løkke (dvs. i s i en sætning af formen `while (`e`)` s), forlades løkken og koden lige efter løkken udføres.

2. Hvis `break;` står inde i en `switch`-sætning (dvs. i cs i en sætning af formen `switch (`e`) {`cs`}`), forlades `switch`-sætningen og koden lige efter denne udføres.

3. Hvis `break;` ikke står i hverken eller, er det en fejl, som skal rapporteres. Se endvidere vinket længere nede.

4. Hvis `break;` står inde i flere indlejrede sætninger (`while` eller `switch`), er det kun den inderste af disse, der forlades.

Semantikken for begge sætningstyper er ligesom i C (hvis C's `switch`-sætning indskrænkes til den herover viste form).

Bemærk, at testen for, om en funktion altid returnerer med en `return`-sætning, skal tage højde for `switch` og `break`. **Vink**: En sætning kan returnere på tre måder: Normalt (ved at nå slutningen af sætningen), med `return` eller med `break`. Find for hver sætning mængden af mulige returmåder, og brug denne information om delsætninger til at finde informationen om hele sætningen. For eksempel kan en while-løkke altid returnere normalt, aldrig med `break` og med `return` kun hvis kroppen kan.

6.1.2 Opgavens art

Der er lidt mere typecheck end kodegenerering.

6.1.3 Testprogrammer

Testprogrammer til opgave 1 starter med `switch`, f.eks. `switch-sort.100` og `switch-error01.100`. Se endvidere afsnit 4.

6.2 Opgave 2: Funktionsreferencer

Syntaksen af 100 udvides med følgende produktion:

$$\mathit{Sid} \rightarrow \textbf{id}\ (\ )$$

Denne udvidelse gør ikke grammatikken tvetydig (udover, hvad der er håndteret med præcedenserklæringer i G-opgaveløsningen), men den introducerer en *shift/reduce* konflikt, der skyldes manglende *lookahead* i parseren.

Denne konflikt skal løses med omskrivning af grammatikken – den kan ikke løses med præcedenserklæringer. Vink: Opdel produktionerne for $FunDecs$ i flere specialtilfælde.

6.2.1 Semantik

Funktioner og variable deler i denne udvidelse navnerum, så en variabel med navn f kan "skygge for" en funktion med samme navn. En ny type værdi er en funktionsreference. Hvis et funktionsnavn i et udtryk bruges uden en efterfølgende parameterliste, er det et udtryk, der returnerer en funktionsreference (dvs. adressen på funktionens kode). En *Sid* af formen $f()$ erklærer f til at være en funktionsreference. Erklæringen

```
int x, f(), *y;
```

erklærer for eksempel tre variable: `x`, der er et heltal, `f`, der er en reference til en funktion, der returnerer et heltal og `y`, der er en heltalsreference. Bemærk, at parametrenes typer ikke specificeres ved erklæringer af funktionsreferencer.

En variabel, der er en funktionsreference, kan tildeles en værdi, som er en funktionsreference med samme type (igen uden hensyn til parameterenes typer). Hvis der f.eks. er en funktion erklæret som

```
int g(int p, *q) {...}
```

vil tildelingen `f = g` give en værdi, der er en reference til `g`. Oversætteren skal checke, at returtyperne af `f` og `g` stemmer overens, men da erklæringen af `f` ikke specificerer argumenternes typer, skal der ikke checkes for overenstemmelse med disse. Man kan ikke lave en tildeling, der overskriver en globalt erklæret funktion, så `g = f` vil i ovenstående tilfælde være ulovligt. Dette skal checkes af oversætteren.

Et funktionskald har stadig formen $f(\cdots)$, men f kan nu være enten en global funktion eller en variabel, der har funktionsreferencetype. Hvis f er en global funktion, skal der (ligesom før) checkes, at argumenternes antal og type stemmer overens med erklæringen, men hvis f er en funktionsreferencevariabel (som jo ikke angiver parametertyper), antages det, at argumenternes antal og typer passer. Det er udefineret, hvad der sker på køretid, hvis f ikke er initialiseret eller hvis f peger

på en funktion med et andet antal eller typer af parametre end kaldet. Resultattypen af kaldet er som angivet i f's type.

Funktionsreferencer kan frit overføres som parametre eller returneres som funktionsresultater.

6.2.2 Opgavens art

Der skal elimineres en ikke-triviel *shift/reduce* konflikt i parseren. Derudover er der omtrent lige dele typecheck og kodegenerering.

6.2.3 Testprogrammer

Testprogrammer til opgave 2 starter med `fun`, f.eks. `fun-map.100` og `fun-error01.100`. Se endvidere afsnit 4.

6.3 Opgave 3: Globale variable og generelle referencer

Syntaksen af 100 udvides med følgende produktion:

$$FunDecs \quad \rightarrow \quad Dec \ ; \ FunDecs$$

og produktionen

$$Sid \quad \rightarrow \quad * \ \textbf{id}$$

erstattes med

$$Sid \quad \rightarrow \quad * \ Sid$$

Endvidere tilføjes nogle prædefinerede allokerings funktioner, der alle virker ligesom `walloc` (de kan altså bruge samme kode), men hvor resultattyperne er anderledes: `cralloc` returnerer en reference til en tegnreference og `iralloc` returnerer en reference til en heltalsreference. `cralloc` kan f.eks. bruges til at allokere en tabel af strings.

6.3.1 Semantik

Der kan nu erklæres globale variable på samme måde som lokale variable (selv om nonterminalen hedder *FunDecs* erklæres nu både funktioner og variable). Alle globale variable er tilgængelige i alle funktioner, men kan "skygges" af lokale variable og parametre på samme måde som lokale variable kan skygge for parametre og andre lokale variable. Der må ikke erklæres flere globale variable med samme navn. Dette skal checkes.

Semantisk set fungerer en global variabel ligesom en lokal variabel, bortset fra, at den netop er global og bevarer sin værdi hen over funktionskald og -retur. Det anbefales, at globale variable lægges i lageret frem for i registre jvf. *Introduction to Compiler Design* afsnit 9.9.1.

Endvidere kan der nu være referencer til referencer. Erklæringen

```
int ***x;
```

erklærer f.eks. `x` til at være en reference til en reference til en reference til et heltal. Referencer fylder 32-bit, så operationer på referencer til referencer fungerer ligesom operationer på referencer til heltal (dog skal typerne matche på den oplagte måde, så man ikke kan trække to forskellige referencetyper fra hinanden osv.).

6.3.2 Opgavens art

Der er lidt mindre typecheck end kodegenerering.

6.3.3 Testprogrammer

Testprogrammer til opgave 3 starter med `global`, f.eks. `global-sort.100` og `global-error01.100`. Se endvidere afsnit 4.

6.4 Opgave 4: Void og &

Syntaksen af 100 udvides med følgende produktioner

$$
\begin{array}{lcl}
\textit{Type} & \rightarrow & \texttt{void} \\
\textit{Stat} & \rightarrow & \texttt{return ;} \\
\textit{Exp} & \rightarrow & \texttt{\&}\ \textit{Lval}
\end{array}
$$

6.4.1 Semantik

Typen `void` kan kun bruges som resultattype af en funktion – hverken udtryk, parametre eller lokale variable kan have denne type (hvilket typecheckeren skal sikre). Typen `void` indikerer, at funktionen *ikke* returnerer nogen værdi. Det betyder, at funktionen i stedet for at returnere med en sætning af formen `return` *e*`;` skal returnere med en sætning af formen `return;` *eller* ved at nå slutningen af funktionskroppen. Der må altså ikke i kroppen være en sætning af formen `return` *e*`;`. Dette skal checkes.

Operatoren `&` returnerer adressen af den *Lval*, den står foran. Eksempelvis vil udtrykket `&a[3]` returnere adressen på det fjerde element i tabellen `a` og `&x` vil returnere adressen på variablen `x`. Bemærk, at dette tvinger `x` til at ligge i lageret (på kaldstakken), da registre ikke har adresser. Se endvidere Afsnit 9.9.2 i *Introduction to Compiler Design*.

I udtrykket `&`*m* skal *m* have typen `int` eller `char` og udtrykket vil følgelig være en reference til hhv. et heltal eller et tegn. F.eks. vil følgende kodebid være lovlig:

```
int x, *y;
y = &x;
```

mens udtrykket `&y` i denne kontekst ikke vil være lovlig, da resultatet vil være en reference til en reference til et heltal, hvilket ikke er en lovlig type i 100.

6.4.2 Opgavens art

Der er overvægt af kodegenerering. Specielt skal der findes en mekanisme til at sørge for at variable, hvis adresser bliver taget med `&`-operatoren, ligger i lageret. Vink: Brug evt. registerallokatoren. Det kan kræve modifikation af `RegAlloc.sig` at få adgang til de relevante dele af registerallokatoren.

6.4.3 Testprogrammer

Testprogrammer til opgave 4 starter med `void`, f.eks. `void-sort.100` og `void-error01.100`. Se endvidere afsnit 4.

6.5 Opgave 5: Undtagelser

Syntaksen af 100 udvides med følgende produktioner:

$$\begin{array}{lcl} \mathit{Stat} & \rightarrow & \texttt{throw}\ \mathit{Exp}\ \texttt{;} \\ \mathit{Stat} & \rightarrow & \texttt{try}\ \mathit{Stats}\ \texttt{catch}\ \textbf{id}\ \texttt{:}\ \mathit{Stat} \end{array}$$

`catch` har samme præcedens og associativitet som `else`.

6.5.1 Semantik

Der tilføjes undtagelser (*exceptions*) med heltallige værdier. Sætningen `throw` *e*`;` beregner *e* (som skal have typen `int`) til en værdi *n* og kaster en undtagelse med denne værdi. Denne kan fanges af en `catch`-sætning, som beskrevet herunder.

En sætning af formen

`try` *ss* `catch` *x* `:` *s*

udfører sætningerne i *ss*. Hvis *ss* bliver færdig uden at kaste undtagelser (som ikke fanges allerede i *ss*) sker ikke yderligere. Men hvis *ss* kaster en undtagelse, der ikke fanges i *ss* selv, bindes variablen *x* til værdien af den kastede undtagelse og *s* udføres. *x* har typen `int` og har virkefelt i *s*. Bemærk, at undtagelsen kan kastes af en funktion, der er kaldt inde i *ss*.

Hvis en undtagelse med værdi *n* kastes, men ikke fanges af nogen `try-catch`-sætning, skal programmet udskrive meddelelsen "`Uncaught exception:` *n*" og terminere.

Testen for, om en funktion returnerer med `return` skal ændres til en test, om funktionen returnerer med `return` eller `throw`, da begge dele er gyldige måder at returnere på. I den forbindelse kan det antages, at kald til andre funktioner kan returnere både normalt og med en exception. Se iøvrigt vinket til opgave 1.

6.5.2 Opgavens art

Der er overvægt af kodegenerering i forhold til typecheck. Specielt skal mekanismen for at fange undtagelser kastet fra andre funktioner overvejes. Vink: Husk at bruge denne mekanisme også for prædefinerede funktioner (`getint` osv).

6.5.3 Testprogrammer

Testprogrammer til opgave 5 starter med `except`, f.eks. `except-find.100` og `except-error01.100`. Se endvidere afsnit 4.