En oversætter for Cat

Godkendelsesopgave på kurset Oversættere

Efterår 2010

1 Introduktion

Dette er den første del af rapportopgaven på Oversættere, efterår 2010. Opgaven skal løses i grupper på op til 3 personer. Opgaven bliver stillet mandag d. 15/11 2010 og skal afleveres senest onsdag d. 17/12 2010. 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.

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 Cat, som er beskrevet i afsnit 3.

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

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

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

`Type.sml` Typechecker for Cat.

`Compiler.sml` Oversætter fra Cat til MIPS assembler. Oversættelsen sker direkte fra Cat 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.

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 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*.

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, men for mange irrelevante detaljer og udenomssnak vil trække ned.

2.1 Afgrænsninger af oversætteren

Det er helt i orden, at lexer, parser, typechecker og oversætter 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 Cat

Cat er et simpelt funktionelt programmeringssprog.

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

Prog → *TyDecs FunDecs Exp*

TyDecs → `type` **id** `=` `(` *Types* `)` *TyDecs*
TyDecs →

FunDecs → `fun` **id** `:` *Type* `->` *Type Match* `end` *FunDecs*
FunDecs →

Type → `int`
Type → `bool`
Type → **id**

Types → *Type*
Types → *Types* `,` *Types*

Match → *Pat* `=>` *Exp*
Match → *Match* `|` *Match*

Pat → **num**
Pat → `true`
Pat → `false`
Pat → `@`
Pat → **id**
Pat → `(` *Pats* `)`

Pats → *Pat*
Pats → *Pats* `,` *Pats*

Exp → **num**
Exp → `true`
Exp → `false`
Exp → `@` `:` **id**
Exp → **id**
Exp → `(` *Exps* `)` `:` **id**
Exp → *Exp* `+` *Exp*
Exp → *Exp* `-` *Exp*
Exp → *Exp* `=` *Exp*
Exp → *Exp* `<` *Exp*
Exp → `not` *Exp*
Exp → *Exp* `and` *Exp*
Exp → *Exp* `or` *Exp*
Exp → `if` *Exp* `then` *Exp* `else` *Exp*
Exp → `let` *Dec* `in` *Exp*
Exp → `case` *Exp* `of` *Match* `end`
Exp → **id** *Exp*
Exp → `read`
Exp → `write` *Exp*
Exp → `(` *Exp* `)`

Exps → *Exp*
Exps → *Exps* `,` *Exps*

Dec → *Pat* `=` *Exp*
Dec → *Dec* `;` *Dec*

Figur 1: Syntaks for Cat

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 begge venstreassociative.
- Operatorerne < og = har samme præcedens og er begge ikkeassociative.
- Operatoren `not` binder stærkere end `and`, som binder stærkere end `or`. Både `and` og `or` er højreassociative. Alle binder svagere end < og =.
- `else` og `in` har samme præcedens og binder svagere end de logiske operatorer.
- => binder svagere end `else` og `in`.
- Funktionsanvendelse og `write` binder stærkere end + og -. Funktionsanvendelse er højreassociative, så `f g x` grupperes som `f (g x)`. Bemærk, at dette er modsat konventionen i SML.
- Der er separate navnerum for variabler, funktioner og typer.
- 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 SML. Dog er tal 32-bit tokomplementtal og aritmetik (+ og -) er uden detektion af *overflow*.

Et Cat program består af erklæringer af typer efterfulgt af erklæringer af funktioner og til sidst et udtryk. Kørsel af et program sker ved beregning af dette udtryk. Alle typer og funktioner har virkefelt i hele programmet, så de er gensidigt rekursive. Typenavne og funktionsnavne bruger forskellige navnerum, så det er tilladt med en funktion med samme navn som en type, men det er ikke tilladt at have to typer eller to funktioner med samme navn.

Udover heltal (typen `int`) og boolske værdier (typen `bool`) har Cat også tupelværdier, dvs. par, tripler, kvadrupler osv. En tupeltype erklæres i en typeerklæring, hvor den beskrives som en ikke-tom liste af typer adskilt af kommaer inde i et parentespar. En værdi af en tupeltype tt, hvor tt er erklæret som $(t_1, \ldots, t_n)$ kan enten være en nulreference eller en reference til n hoballokerede maskinord, der

indeholder værdier af typerne $t_1, \ldots, t_n$. En nulreference skrives som `@`. Når `@` bruges som udtryk, skal dets type angives efter et `:`, f.eks. `@ : list`. Man bygger en tupel (der ikke er en nulreference) ved at angive elementerne adskilt af kommaer inde i et sæt parenteser og derefter angive tuplets type efter et kolon.

Man kan f.eks. definere typen `list` af lister af heltal med erklæringen

```
type list = (int,list)
```

En liste af heltallene 1, 2 og 3 kan da konstrueres med udtrykket

```
(1, (2, (3, @:list):list):list):list
```

Bemærk, at typen angives både ved nulreferencer og ved konstruktion af ikketomme lister.

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 `and` (konjunktion) og `or` (disjunktion) er sekventielle operatorer, så andet argument udregnes ikke, hvis det første argument er tilstrækkeligt til at afgøre resultatet. Nøgleordene `true` og `false` er boolske konstanter, der kan bruges både i mønstre og udtryk.

Udtrykket `if` e_1 `then` e_2 `else` e_3 virker ligesom det tilsvarende udtryk i SML. Det skal verificeres på oversættelsestid, at e_1 er af typen `bool` og at e_2 og e_3 har samme type.

Cat bruger mønstergenkendelse (*pattern matching*) i stil med SML. En funktionsdefinition består af erklæring af funktionens navn, parametertype og resultattype samt en *Match*, der er en række regler adskilt af `|`. En regel består af et mønster (*Pat*), en dobbeltpil (`=>`) og et udtryk(*Exp*). Semantikken er, at reglerne i et *Match* afprøves en af gangen, indtil man finder et, hvor mønstret matcher argumentværdien. Når dette sker, beregnes det tilhørende udtryk. Variable i et mønster bindes til de tilsvarende dele af argumentet og kan bruges i udtrykket. Hvis ingen regler matcher argumentværdien, udskrives en fejlmeddelelse. Et mønster matcher en værdi efter følgende regler:

- Et mønster, der er en talkonstant k, matcher heltalsværdien k.
- Et mønster, der er en boolsk konstant (`true` eller `false`) matcher henholdsvis den sande og den falske boolske værdi.
- Mønsteret `@` matcher en nulreference af en vilkårlig tupeltype.
- Mønsteret `(`p_1`,` $\ldots$ `,`p_n`)` matcher en tupelværdi af formen `(`v_1`,` $\ldots$ `,`v_n`)`, hvis p_i matcher v_i for alle i fra 1 til n.
- Et mønster, der er en variabel x, matcher enhver værdi, og definerer x til at have denne værdi i det udtryk, der hører til mønstret.

Det skal på oversættelsestidspunktet verificeres, at mønsteret er konsistent med den erklærede parametertype. Der bruges følgende regler:

- Et mønster, der er en talkonstant k, er konsistent med typen `int`.

- Et mønster, der er en boolsk konstant (`true` eller `false`) er konsistent med typen `bool`.
- Mønsteret `@` er konsistent med alle tupeltyper.
- Mønsteret `(`p_1`,...,`p_n`)` er konsistent med tupeltypen tt, hvis tt er defineret til `(`t_1`,...,`t_n`)` og p_i er konsistent med t_i for alle i fra 1 til n.
- Et mønster, der er en variabel x, er konsistent med enhver type t. x vil have typen t i det udtryk, der hører til mønstret.

Alle udtrykkene i reglerne i samme *Match* skal have samme type.

Bemærk, at en funktion altid har præcis en parameter, der dog kan være en tupel. Det skal verificere, at funktionens erklærede resultattype stemmer overens med typen, der returneres af den *Match*, der udgør funktionens krop.

Man kan også bruge en *Match* i `case`-udtryk. Et sådant har formen `case` e `of` M `end`, hvor e er et udtryk og M er en *Match*. Udtrykket e beregnes og matches mod reglerne i M, ligesom i funktionsdefinitioner.

Mønstre kan også bruges i `let`-udtryk. Et sådant har formen `let` p_1`=`e_1`;...;`p_n`=`e_n `in` e_0. Semantisk har dette udtryk samme betydning som udtrykket `case` e_1 `of` p_1 `=>` `...` `case` e_n `of` p_n `=>` e_0 `end` `...` `end`

En funktionsanvendelse er af formen $f\ e$, hvor f er funktionens navn og e er et udtryk, der beregner argumentet til funktionen. Det skal verificeres på oversættelsestidspunktet, at argumentet har samme type som den erklærede parametertype til f. Typen af kaldet er funktionens erklærede resultattype.

Udtrykket `read` indlæser et heltal og returnerer dens værdi. Udtrykket `write` e beregner e til et heltal, udskriver dette og returnerer det. Det skal verificeres, at e har typen `int`.

6 En delmængde af Cat

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

- Tupler, boolske værdier og erklæringer, udtryk og mønstre, der definerer, bruger eller returnerer disse, er ikke implementeret.
- `case`-udtryk og `let`-udtryk er ikke implementeret.

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

7 Abstrakt syntaks og oversætter

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

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

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`).

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 Cat

`ackermann.cat` indlæser to tal m og n og udskriver $ackermann(m,n)$, hvor *ackermann* er Ackermann's funktion. Afprøver `case` og boolske mønstre.

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

`logic.cat` afprøver logiske funktioner. Der indlæses ikke noget inddata.

`pair.cat` indlæser to tal, bygger et par af dem og skriver dem ud i omvendt rækkefølge.

`qsort.cat` indlæser tal indtil et 0 indlæses, bygger en liste af tallene (fraregnet nullet), sorterer listen og skriver den sorterede liste ud.

`reverse.cat` indlæser tal, indtil et 0 indlæses. Derefter skrives tallene ud i omvendt rækkefølge.

`rwlist.cat` indlæser tal indtil et 0 indlæses, bygger en liste af tallene (fraregnet nullet) og skriver listeelementerne ud.

`treesort.cat` Har samme funktion som `qsort.cat`, men bruger en anden sorteringsalgoritme.

`option.cat` Definerer og bruger en tupeltype med et element.

Hvert eksempelprogram *program*`.cat` 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.cat, ..., error19.cat`), der indeholder diverse fejl eller inkonsistenser. (`error01.cat, ..., error16.cat`) indeholder fejl, der skal fanges i checkeren. Der er ikke input- eller outputfiler til disse programmer. (`error17.cat, ..., error19.cat`) fejler en `pattern match` ved en funktionsdefinition, en `case` eller en `let`. De skal derfor kunne oversættes uden fejl, men skal ved kørsel (nogle med tilhørende inputfiler) give fejlmeddelelser, der angiver køretidsfejlenes omtrentlige position i programmerne.

Kun `fib.cat` og `error17.cat` kan oversættes med den udleverede oversætter. De andre programmer bruger de manglende sprogelementer, og vil derfor give syntaksfejl.

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.

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

Uge 47 Parseren kan genereres og oversættes. Rapportafsnit om lexer og parser færdigt.

Uge 48 Checkeren er implementeret. Rapportafsnit om checker skrives.

Uge 49 Oversætteren er implementeret, rapportafsnit om denne skrives.

Uge 50 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 `CC.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

- For at undgå en shift/reduce konflikt kan det være en ide at opdele produktionen udtryk til tupelkonstruktion i to separate produktioner: En til tupler med ét element og en til tupler med flere elementer. Produktionerne for *Exps* kan dermed ændres, så der er mindst to udtryk adskilt af komma.

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