G-opgave
Oversættere
Vinter 2011/2012

Naja Wulff Mottelson (vsj465)
Søren Egede Pilgård (vpb984)
Mark Schor (vcl329)

22. december 2011

Indhold

1 Indledning

Nærværende rapport tjener som dokumentation af vores arbejde med G-opgaven i kurset Oversættere. Vi har her implementeret lexer, parser, typetjekker og oversætter for sproget 100. På nuværende tidspunkt er vi i stand til at køre samtlige udleverede fejlbehæftede programmer med de forventede fejl. Vi kan derudover oversætte et program der tester for tilstedeværelse af en main-funktion (se Appendix), samt oversætte og køre de udleverede korrekte testprogrammer uden at modtage fejl.

2 Lexer og parser

I vores arbejde med lexeren har vi indledningsvist tilføjet de manglende nøgleord (`while` og `char`) samt fjernet nøgleordet then, eftersom dette ikke er til stede i den udleverede grammatik for 100. Vi har herefter oprettet regulære udtryk i reglen Token til at matche tilføjelserne (som ligeledes tæller ==-operatoren, return-sætninger,referencer, krølleparenteser samt firkantede parenteser).

I arbejdet med parseren har vi genemgået den udleverede grammatik og indføjet de manglende termer som tokens og types i `Parser.grm` (hvor vi i samme omgang har tilføjet præcedens og associativitet - hovedsageligt ved at tilføje venstreassociativitet til ==-operatoren).

Ved kørsel af `compile.sh` modtager vi ingen fejlbeskeder eller notifikationer om shift/reduce konflikter. Eftersom vi antager, at parsergeneratoren er korrekt implementeret, konkluderer vi derfor at den grammatik, vi parser, er entydig.

3 Typetjekker

Hoveddelen af det arbejde vi har lagt i typetjekkeren har været i form af udvidelser i funktionen `Type.checkExp` og `Type.checkStat`. Udover dette har vi dog indledningsvist tilføjet 100s indbyggede funktioner (`balloc`, `walloc`, `getstring` og `putstring`) i `Type.checkProg`, samt tilføjet typerne for 100s abstrak- te syntaks (`Char, CharRef, IntRef`).

3.1 Hjælpefunktioner i typetjekkeren

I typetjekkeren har vi udvidet en række hjælpefunktioner, som beskrives nedenfor:

3.1.1 ignoreChar

For at undgå unødig mønstergenkendelse (både i typetjekkeren og i `Compiler.sml`) har vi introduceret hjælpefunktionen `Type.ignoreChar`, som konverterer tegn til heltal:

```
fun ignoreChar (Char) = Int
  | ignoreChar ty = ty
```

3.1.2 getType

For at kunne implementere referencer har vi udvidet `Type.getType` fra udelukkende at kalde `Type.convertType` på sit input til at angive

3.1.3 mismatch og typeToString

Begge disse funktioner benyttes udelukkende i forbindelse med aflusning af koden, hvor de sørger for at brugeren kan få nogenlunde sigende fejlbeskeder ved fejlagtig sammensætning af typer.

```
fun typeToString Int = "Int"
  | typeToString Char = "Char"
  | typeToString IntRef = "IntRef"
  | typeToString CharRef = "CharRef"

fun mismatch t1 t2 = (typeToString t1) ^ " != " ^ (typeToString t2)
```

3.2 Tjek af udtryk

I `Type.checkExp` har vi tilføjet mønstre for de manglende 100-typer (`CharConst, StringConst, Equal`) samt tilføjet overlæsning af plus- og minusoperatorerne, så at de er i stand til at tage både heltal, tegreferencer og heltalsreferencer som operander.

Eftersom der kan blive erkæret nye variable i starten af en blok har vi brug for at kunne opdatere symboltabellen for at kunne typetjekke blokke. Vi har derfor udvidet `Type.checkStat` til at tage den eksisterende `vtable` som input og generere en ny symboltabel som konkateneres med den gamle i tilfælde af blokke.

3.3 Tjek af sætninger

Idet 100 indeholder en del kontrolstrukturer, som kan skabe usikkerhed, om hvorvidt en funktion vil returnere, er dette ikke som sådan et trivielt problem. Under typetjekket kan man f. eks. møde en lignende funktion:

```
int foo(int bar)
    if 1
        return bar
```

Funktionen foo er syntaktisk korrekt men indeholder en typefejl, eftersom det ikke er sikkert, at return-sætningen vil blive nået. Der findes adskillige måder at tjekke for denne slags typefejl - en metode kunne f.eks. være at undersøge en funktions sidste linje, og kontrollere at denne altid vil returnere. Denne løsning vil dog træffe en problematik ifbm. `if-else`-konstruktionen, da denne kan have return-sætninger i begge sine tilfælde. Således vil enhver funktion indeholdende

en sådan `if-else` altid være sikker på at returnere, ligegyldigt hvad der står i dens sidste linje kode.

Vi implementerer tjekket ved at tilføje et yderligere output i `Type.checkStat` - en sandhedsværdi som angiver, hvorvidt den matchede sætning indeholder en return-sætning. I tilfældene `if` og `while` returneres falsk, eftersom begge konstruktioner vil kræve en return-sætning i en følgende blok. I tilfældet `if-else` kontrollerer vi, at begge tilfælde returnerer og sender denne sandhedsværdi videre. `Type.checkStats` output kontrolleres senere i `Type.checkFunDec`, hvor en fejl kastes i tilfælde af manglende returns.

Som nævnt skal det dog også kontrolleres, at return-sætningens og funktionens type er kongruente, til hvilket vi har overvejet forskellige løsninger. Den mest funktionelle løsning forekommer at være at lade `Type.checkFunDec` returnere funktionens returtype. `Type.checkStat` vil herfter kunne udføre sit vanlige tjek af tilstedeværelsen af return-sætninger og herefter returnere enten `NONE` i tilfælde af manglende returns eller `SOME` af sætningens returtype, samt sætningens position. Sammenligningen af funktionens og return-sætningens typer (samt tjekket for manglende returns) vil herefter kunne foretages i `Type.checkFunDec`. Denne implementering bliver dog problematisk, så snart forgreninger i koden medfører flere return-sætninger i samme `Type.checkStat` ville i dette tilfælde være nødt til at returnere en liste indeholdende tupler af positioner og returtyper, som ville skulle gennemsøges i `Type.checkFunDec` for at udføre typesammenligningen.

Den måde, vi har valgt at implementere tjekket på, er (ligesom i algoritmen nævnt ovenfor) ved at sende en funktions returtype samt position med som output fra `Type.checkFunDec`, hvorefter kontrollen af typekongruensen foretages i `Type.checkStat`:

```
  fun checkStat s vtable ftable returntype =
    case s of
      S100.EX e => (checkExp e vtable ftable; false)
    | S100.If (e,s1,p) =>
        if checkExp e vtable ftable = Int
then (checkStat s1 vtable ftable returntype; false)
else raise Error ("Condition should be integer",p)
    | S100.IfElse (e,s1,s2,p) =>
      let
        val r1 = checkStat s1 vtable ftable returntype
        val r2 = checkStat s2 vtable ftable returntype
      in
        if checkExp e vtable ftable = Int
        then r1 andalso r2
else raise Error ("Condition should be integer",p)
      end
    | S100.While (e,s,p) =>
      if checkExp e vtable ftable = Int
      then (checkStat s vtable ftable returntype; false)
```

```
      else raise Error ("Condition should be integer",p)
    | S100.Return (e,p) => if checkExp e vtable ftable = returntype
                           then true
                           else raise Error ("Returned value not of "^
                                             "functions returntype: "^mismatch
                                              returntype (checkExp e vtable
                                              ftable) ,p)

    | S100.Block ([],[],p) => false
    | S100.Block ([], s::ss,p) =>
      let
        val r1 = checkStat s vtable ftable returntype
        val r2 = checkStat (S100.Block ([], ss, p)) vtable ftable returntype
      in
        r1 orelse r2
      end
    | S100.Block (ds, ss,p) =>
      let
        val vtable' = checkDecs ds
      in
        checkStat (S100.Block([], ss, p)) (vtable' @ vtable) ftable returntype
      end

  fun checkFunDec (t,sf,decs,body,p) ftable =
        if checkStat body (checkDecs decs) ftable (getType t sf)
        then ()
        else raise Error ("Function needs valid return statements",p)
```

3.4 Tjek af venstreværdier

Vi har udvidet `Type.checkLval` med tilfældene `S100.Deref` samt `S100.Lookup`, så at det bliver muligt at typetjekke forsøg på at indeksere arrays (eller rettere repræsentationen af arrays i form af en reference). Essentielt kan to fejl opstå i denne situation: det indekserede array kan være udefineret, eller indekset kan være af anden type end heltal. I tilfældet af opslag (`Deref` eller `Lookup`)har vi endvidere ændret `Type.checkLvals` returtype, til at returnere den type referencen peger på, i stedet for hhv. `Int-` eller `CharRef`. Implementeringen er følgende (bemærk at `Type.checkLval` er gensidigt rekursiv med `Type.checkExp` og derfor erklæres med `and`):

```
  and checkLval lv vtable ftable =
    case lv of
      S100.Var (x,p) =>
        (case lookup x vtable of
   SOME t => t
 | NONE => raise Error ("Unknown variable: "^x,p))
    | S100.Deref (x,p) =>
      (case lookup x vtable of
         SOME IntRef => Int
       | SOME CharRef => Char
```

```
      | SOME _ => raise Error ("You can not use a non-reference type as an address!", p)
      | NONE   => raise Error ("Unknown variable: "^x,p))
  | S100.Lookup (x,e,p) =>
    if ignoreChar (checkExp e vtable ftable) = Int
    then
      (case lookup x vtable of
         SOME IntRef => Int
       | SOME CharRef => Char
       | SOME _ => raise Error ("You can not use a non-reference type as an address!", p
       | NONE   => raise Error ("Unknown variable: "^x,p))
    else raise Error ("Index not a number",p)
```

4 Kodegenerering og oversætter

Eftersom det er muligt at have deklarationer i starten af blokke, er det nødvendigt at kunne udvide symboltabellen for at implementere dem i oversætteren. Vi har derfor introduceret funktionerne `Compile.rcompileDecs` og `Compiler.extend`, som begge er hentet fra `Type.sm` - `Compiler.extend` er dog en let modificeret version af `Type.extend`, som også opretter en ny lokation.

4.1 Håndtering af tildelinger

Et pudsigt tilfælde ved tildelingsoperatoren er dens opførsel ved simultane tildelinger af tegn og heltal. Et eksempel på dette kunne være følgende blok:

```
{
int i; char *c;
i = c = 256;
i == 0;
c == 0;
}
```

Eftersom antallet af allokerede bits varierer for heltal og tegn, og det kun er den niende bit der sat i den binære repræsentation af tallet 256, vil `c` skulle sættes til 00000000 i tildelingen `i = c = 256;`. I den oprindelige version af `Compiler.compileExp` bliver variablen `place` gemt ved tildeling af tegnreferencer - her vil `c` således blive sat korrekt, men `i` vil blive sat til 256 grundet de ekstra bits i `place` den kan registrere. Vi har løst dette problem ved at afkorte `place` (reelt: `t`, eftersom `place` tidligere bliver flyttet over i `t`), således at det kun er de otte mindst betydende bits der bliver brugt til at gemme referencen. Se nedenfor for implementeringen (i `Compiler.checkExp`):

```
    | S100.Assign (lval,e,p) =>
        let
          val t = "_assign_"^newName()
  val (code0,ty,loc,_) = compileLval lval vtable ftable
  val (_,code1) = compileExp e vtable ftable t
in
```

```
  case (ty,loc) of

            (tty, Reg x) =>
              (tty,
               code0 @ code1 @ [Mips.MOVE (x,t), Mips.MOVE (place,t)])
          | (Type.Int, Mem x) =>
            (Type.Int,
             code0 @ code1 @
             [Mips.SW (t,x,"0"), Mips.MOVE (place,t)])
          | (Type.Char, Mem x) =>
            (Type.Char,
               code0 @ code1 @
               [Mips.ANDI(t,t,"255"), Mips.SB (t,x,"0"), Mips.MOVE (place,t)])
          | (Type.IntRef, Mem x) => raise Error ("Type error, "^
                                                 "asignment of"^
                                                 "pointer-pointers "^
                                                 "not implemented!", p)
          | (Type.CharRef, Mem x) => raise Error ("Type error, "^
                                                  "asignment of"^
                                                  "pointer-pointers "^
                                                  "not implemented!", p)

end
```

4.2 Implementering af referencer

Til brug i vores implementering af referencer har vi valgt at udvide datatypen `Location` med typen `Mem` til at repræsentere referencer, som ligger i hukommelsen istedet for registerbanken.
Idet hovedformålet ved referencerne i 100 er at de kan indekseres, bliver denne udvidelse af `Location` hovedsageligt udnyttet i `Compiler.compileLval` og `Compiler.compileExp` : Ved `Compiler.compileLval` vil funktionen, i tilfældet `S100.Lookup`, udregne det udtryk der benyttes til at indeksere referencen, og returnerer den kode der udregner forskydningen ift. referencens adresse samt referencens type og lokation. `Compiler.compileLval` bliver kaldt i `Compiler.compileExp`, hvor funtionen skelner imellem hvorvidt lokationen angiver Reg eller Mem. I tilfælde af registre vil `Compiler.compileExp` ændre i adressen, i tilfælde af hukommelse vil den ændre i selve dataet på bemeldte adresse. Det samme gælder for `Compiler.compileExp`s opførsel ved tildeling.

5 Afprøvning

Vi har udvidet antallet af afprøvninger med en test for manglende main-funktioner ourerror01. Alle vores testprogrammer kan oversættes og afleverer de korrekte resultater ifølge vores afprøvninger. Vores afprøvningsdata er følgende:

- Charref er blevet testet med inputdataen: (0) (1) (2) og (4) (5) og gav

outputtet (97) (98) (114) (99) (97), hvilket svarer overens med de rigtige bogstaver i ASCII-tegnsættet.

- Copy er blevet testet med inputdataen: (Husk at teste jeres compiler ordentligt!) og fik outputtet (Husk at teste)
- DFA er blevet testet med inputdataen: (1) (10) (11) (100) (1000) og fik outputtet (1) (2) (0) (1) (2)
- Fib er blevet testet med inputdataen: (5) (6) (7) (20) og fik outputtet (5) (8) (13) (6765)
- Sort er blevet testet med inputdataen: (9,6,8,2,4,7,5,4,3,9) og fik outputtet (2,3,4,4,5,6,7,8,9)
- Sort2 er blevet testet med inputdataen: (9,6,8,2,4,7,5,4,3,9) og fik outputtet (2,3,4,4,5,6,7,8,9)
- Ssort er blevet testet med inputdataen: (Husk at teste jeres compiler ordentligt!) og fik outputtet (!Hacdeeeeeegiijkllmnooprrrsssttttttu)
- Ssort2 er blevet testet med inputdataen: (Husk at teste jeres compiler ordentligt!) og fik outputtet (!Hacdeeeeeegiijkllmnooprrrsssttttttu)
- error1 gav fejlbeskeden: Redeclaration of main
- error2 gav fejlbeskeden: Double declaration of x
- error3 gav fejlbeskeden: Unknown function: getchar
- error4 gav fejlbeskeden: Redeclaration of putint
- error5 gav fejlbeskeden: Type mismatch in addition at line
- error6 gav fejlbeskeden: Type mismatch in substraction
- error7 gav fejlbeskeden: Type mismatch in substraction
- error8 gav fejlbeskeden: Can't compare different types
- error9 gav fejlbeskeden: Can't compare different types
- error10 gav fejlbeskeden: Arguments don't match declaration of f
- error11 gav fejlbeskeden: Arguments don't match declaration of f
- error12 gav fejlbeskeden: Bad string!
- error13 gav fejlbeskeden: Function needs valid return statements
- error14 gav fejlbeskeden: Function needs valid return statements
- error15 gav fejlbeskeden: Returned value not of functions returntype: Int != CharRef

6 Appendix

6.1 Test af tilstedeværelse af main-funktioner

Error-programmet som vi har benyttet til at afprøve for tilstedeværelse af main-funktioner er implementeret således:

```
/* ourerror01.100 */ /* Missing main function */
int foo()  return 1;
```