Projekt i kompilatorkonstruktion

Originalförfattare: Thomas Johnsson
Omarbetning: Lennart Augustsson, Niklas Röjemo
Korrekturläsning och uppdatering: Thomas Hallgren

1 Allmänt

I detta PM återfinns en beskrivning av ett litet programmeringsspråk. Utgående från denna beskrivning skall ni konstruera en kompilator för detta språk.

För att kunna få högsta betyg på kursen krävs att man gör hela projektet, inklusive en kodgenerator för en 3-operandsmaskin. *Alla* skall ha gjort klart fram till och med typkontroll senast den **16/2**. Då skall det gå att kontrollera om program är riktiga eller inte med er kompilator. Den behöver dock inte producera någon körbar kod.

Nedan följer några hållpunkter för genomförandet av projektet.

Studera språket. Utgående från språkbeskrivningen, bestående av exempel på varje konstruktion i språket som återfinns i detta PM, ta reda på den lexikala och syntaktiska strukturen på språket, samt förvissa er om att ni förstår vad varje konstruktion betyder. *Om ni tycker att beskrivningen är vag, så använd egen fantasi och eget (goda!) omdöme!* Lista de olika lexikala symbolerna i språket, samt skriv en grammatik för språket.

Abstrakt syntax. Som ett led i att förstå strukturen på det språk som ska kompileras, ange den *abstrakta* syntaxen m.h.a. en eller flera datatyper i Haskell/ML.

Parser. Konstruera en lexikal- och syntaxanalys med hjälp av något tillgängligt verktyg, dvs någon av ALEX/HAPPY (Haskell), ML-LEX/MLYACC (ML), JLex/CUP (Java), LEX/YACC (C), FLEX/BISON (också C). Om ni vill får ni gärna leta reda på andra dylika verktyg, t.ex. om ni inte skriver labben i något av ovanstående språk. Det finns stor chans att hitta sådant via WWW!

Vad gäller syntaxen för uttryck, kan metoden med flertydig grammatik och upplösning av konflikter användas, om verktyget stödjer det.[1] Denna metod ger i regel en mer lättläst grammatik, som är lättare att ändra, och ger ofta en mindre parser. Om ni använder denna metod, skriv då först syntaxen utan att ange prioriteter på operatorer, och studera vilka konflikter detta ger upphov till. Lägg sedan till prioritetsdeklarationer.

Om verktyget inte stödjer denna metod, måste metoden med term/faktor, etc, användas.

Syntaxstyrd översättning. Sätt in aktioner i parsern, som bygger ett träd enligt den abstrakta syntaxen. Den fortsatta kompileringsprocessen kommer att ske genom genomlöpning av syntaxträdet ett antal gånger.

Typkontroll. Kontrollera att programmet är typmässigt korrekt, d.v.s. att t.ex. inte ett boolskt värde tilldelas en numerisk variabel, att antal och typer på aktuella parametrar stämmer med de formella parametrarna, etc. etc.

[1] Även HAPPY gör det, fr o m version 1.8.

Minnes-allokering. Bestäm variablers och parametrars plats på aktiveringsposten.

Kodgenerering. Utgående från ett korrekt typat syntaxträd skall maskinkod genereras, först för en stackmaskin och sedan för en 3-operandsmaskin. Den senare är endast nödvändig för de som vill kunna få högsta betyg på kursen. (Notera att man inte får högsta betyg bara för att man gjort en kodgenerator för en 3-operandsmaskin. Man måste även skriva ett bra resultat på tentan!)

Två datatyper med tillhörande utskriftsfunktioner ges, en för stackmaskinkod och en för 3-operandskod. Utdata från stackmaskinen kan köras med hjälp av en interpretator, för 3-operandskoden gäller att den måste assembleras innan den kan köras.

Mer om datatyperna för instruktionerna kommer senare i detta PM.

Det är *ej* lämpligt att typkontrollera och generera kod före det att hela programmet lästs in! Typkontrollen måste ha hela programmet innan den kan börja köras, för att kunna hantera ömsesidig rekursion. Och typkontrollen måste vara gjord före kodgenereringen.

2 "Vilket språk ska jag programmera i?"

Ni får i princip skriva labben i vilket språk ni vill! Uppenbara valalternativ är Haskell/Hugs (som jag själv helst använder), ML, Java, eller C. Men en begränsande faktor är tillgängligheten på verktyg för generering av lexikal- och syntax-analysatorer.

3 Inlämning

För att få godkänt kräver vi följande i dokumentationsväg.

- Den lexikala delen av språkets grammatik. Ange de lexikala symbolerna. Tala om vilka attributvärden de lexikala symbolerna eventuellt har associerade till sig.
- Den syntaktiska delen av språkets grammatik. Använd en mer 'informell notation', utan att för den skull ge avkall på precisionen, d.v.s.

  $$\langle \textit{id-sekvens} \rangle ::= \langle \textit{id} \rangle \ \{ \ , \langle \textit{id} \rangle \ \}$$

  (där {...} anger upprepning noll eller flera gånger, [...] anger noll eller en gång), *eller* (hellre)

  $$\langle \textit{id-sekvens} \rangle ::= \langle \textit{id}_1 \rangle \ , ..., \langle \textit{id}_n \rangle \qquad (n \geq 1)$$

  snarare än

  $$\langle \textit{id-sekvens} \rangle ::= \langle \textit{id} \rangle \mid \langle \textit{id} \rangle \ , \langle \textit{id-sekvens} \rangle$$

  Reserverade ord och skiljetecken kan skrivas som sig själva. (Den strikt formulerade kontextfria grammatiken framgår ju av /HAPPY/ML-YACC/YACC/BISON-filen.)
- Om du ovan har gett en flertydig grammatik skall du ange hur flertydigheten skall lösas upp.

- Körningsexempel, med källprogram (i Javalette) och producerad stack-maskin-assembler-kod. Glöm inte att provköra dem genom att först översätta till körbar kod och sedan köra resultatet. Testa att ömsesidig rekursion fungerar, samt tomma parameterlistor, m.m., m.m. Testa även några felaktiga program, speciellt typfelaktiga, och sådana där variabler/funktioner saknar deklaration/definition.

  Som ett led i labrättningen kommer er laboration att provköras. Se därför till att en aktuell version av er kompilator (med all källkod) finns på ert konto när ni rapporterar. Om ni använder make skall man kunna köra 'make clean', och sedan 'make' för att kompilera om er kompilator.

  Kompilatorn skall köras med kommandot `lab1` vid första inlämningen och kommandot `lab2` vid andra inlämningen. I båda fallen skall kompilatorn ta ett argument som är den Javalette-fil som skall kompileras.

  Kommandot `lab1` skall ha texten "`ERROR`" i utskriften ifall indatafilen har någon form av fel. En beskrivning av felet bör även ingå. Om det inte finns några fel i indatafilen skall "`OK`" skrivas ut.

  Kommandot `lab2` skall förutom de utskrifterna som krävs för `lab1` producera en fil med antingen stackmaskinsinstruktioner eller 3-operandsinstruktioner. Namnet på den filen skall vara namnet på indatafilen, men med suffixet bytt till `.sm` respektive `.3m`. Eventuell path i argumentet skall inte tas med. Exempel:

  ```
  lab1 ../../testexempel/korrekt_program.v
  ```

  Ovanstående kommando skall skriva ut en text där "`OK`" ingår, och producerar en ny fil `korrekt_program.sm` (eller `korrekt_program.3m` ifall 3-operandsinstruktioner genereras) i det bibliotek där kommandot körs.

- Programkoden för kompilatorn, d.v.s. /ALEX/ML-LEX/LEX/FLEX, HAPPY/ML-YACC/YACC/BISON- och övrig kod i Haskell/ML/C etc. (Den kod som verktygen producerar är dock både volymiös och ointressant) Programkoden skall naturligtvis vara snyggt skriven, bra kommenterad och lätt att följa. Retur ges för program som är svåra att förstå tex pga brist på kommentarer.

Inlämning sker elektroniskt med kommandot `rapportera`. Se WWW för detaljer.

Lycka till!

4 Beskrivning av det lilla språket JAVALETTE

De flesta konstruktionerna i Javalette är helt ordinära, och har sina motsvarigheter i språk såsom C, Java, Ada, Pascal, Algol, m.m. Nedan följer ett exempel på ett program i Javalette.

```
int fak(int x)
{
    if(x <= 0)
        return 1;
    else
        return x * fak(x-1);
}

int main()
{
    int i;
    int x;
    i = readInt();
    while(i > 0) {
        x = readInt();
        writeInt(fak(x));
        i = i-1;
    }
    return 0;
}
```

Ett program består av ett antal (1 eller flera) funktionsdefinitioner, varav en funktion heter `main`. När programmet körs anropas funktionen `main` utan några argument. Funktionen `main` skall alltid returnera ett tal, programmets returkod. I UNIX är normalt returkoden 0 ifall allt gått bra och skilt från 0 ifall det blivit fel.

Alla satser avslutas med semikolon.

I Javalette finns tre typer av variabler och parametrar: `int`, `double`, och `boolean`. `int` är typen för heltal, `double` är typen för flyttal med dubbel precision, och `boolean` är typen för sanningsvärden: `true` eller `false`.

Funktioner i Javalette har typer som anger vilka typer parametrarna och resultatet har. En funktion kan också ha returtypen `void`, vilket betyder att den inte returnerar något värde. En sådan funktion kallas vanligtvis procedur och kan användas i satser men inte uttryck.

Nedan ger vi exempel på varje konstruktion i Javalette.

4.1 Uttryck

Ett uttryck i Javalette sätts ihop av

- boolska konstanter `true` och `false`,
- variabler och parametrar som består av en sekvens bokstäver och siffror som inleds och avslutas med en bokstav,
- positiva heltalskonstanter, såsom `0`, `6`, `3`, `42`,

- positiva flyttalskonstanter, såsom `0.0`, `6.30002`, `3.141592e100`, `42.718281828e-53`,
- funktionsanrop, t.ex. `theta(4.5)`,
- operatorer: `*`, `/`, `%`, `+`, `-` (både negation och subtraktion, % är modulooperatorn), `<`, `>`, `<=` ($\leq$), `>=` ($\geq$), `==`, `!=` ($\neq$), `!` ($\neg$), `&&` ($\wedge$) och `||` ($\vee$),
- `({` *satser* `})` se vidare under 'satser i uttryck'.

Operatorerna `&&` och `||` är villkorliga; t.ex. i uttrycket `a && b` beräknas först `a`, om detta har värdet `false` så beräknas inte `b`, eftersom hela uttrycket ändå måste ha värdet `false`. (Nåja, finessen är egentligen att beräkningen av uttrycket `b` kanske inte terminerar, eller kanske ger upphov till ett fel, som därmed kan undvikas.) Följande likheter gäller:

a `&&` *b* $\equiv$ `({ if(` *a* `) return` *b* `else return false;})`
a `||` *b* $\equiv$ `({ if(` *a* `) return true else return` *b* `;})`
`!` *a* $\equiv$ `({ if(` *a* `) return false; else return true; })`

(Tips: det kan t.o.m. vara lämpligt att översätta `&&`, `||` och `!` till `if`-satser internt i kompilatorn.)

4.2 Funktionsdefinitioner

En funktionsdefinition består av ett funktionshuvud, och en funktionskropp. I funktionshuvudet anges namnen och typerna på de formella parametrarna, samt typen för resultatet.

```
int f(boolean b, boolean c, int d)
{
    if (b && c)
         return d*2;
    else
         return d+3;
}
```

Denna funktion `f` har typen (`boolean`, `boolean`, `int`) $\rightarrow$ `int`. Denna typ kan dock inte skrivas i ett Javalette-program.

4.3 while-satsen

Ett exempel på while-satsen finns i det inledande exemplet.

4.4 if-satsen

If-satsen i Javalette behöver inte ha en else-gren,

Exempel:

```
if (a < b) {
   x = a;
   y = b;
}
```

4.5 tilldelnings-satsen

Ett exempel på tilldelnings-satsen finns i det inledande exemplet. Det är bara tillåtet att tilldela variabler, det är alltså inte tillåtet att tilldela till parametrar. (Att implementera denna kontroll är frivilligt.)

4.6 block-satsen

Ett block kan innehålla ett antal variabel-deklarationer och övriga satser i godtycklig ordning. En variabel får dock inte användas innan den är deklarerad. Blocket omges av { } . De deklarerade variablerna i blocket är bara tillgängliga för satserna i blocket (som befinner sig efter deklarationen). Ett block är en sats, och kan således förekomma inuti ett annat block.

```
{
   int x;
   x = my_theta_function(2);
   int y;
   y = my_funny_valentine(4);
   {
      int z;
      z = x;
      x = y;
      y = z;
   }
}
```

4.7 in och utmatning

Javalette har inte någon speciell syntax för in och utmatning, istället används anrop till fördefinierade funktioner. De funktioner som finns tillgängliga är:

- `void printInt(int i)`
- `void printDouble(double d)`
- `int readInt()`
- `double readDouble()`
- `void error()`

De två print-funktionerna användas i satser, och de två read-funktionerna i uttryck. Funktionen `error` skriver ut ett felmeddelande och stoppar exekveringen.

4.8 satser i uttryck

Denna konstruktion kan liknas vid en "in-line"-funktion. Det är ett sätt att kunna beräkna värdet av ett uttryck procedurellt. Följande uttryck

```
1 + ({ int x;
       x = 3;
       return x*x;
    })
```

har värdet 10.

Satser i uttryck begränsas av `({` och `})` , och måste ha minst en `return`-sats. Värdet av `({-})`-uttrycket blir det värde som returneras av `return`-satsen. Det går att ha mer än en `return`-sats, men de måste alla returnera värden med samma typ. Det är ett fel ifall exekveringen kan lämna satserna utan att göra `return`.

Till sist följer här ett annat exempel på användningen av satser i uttryck, nämligen iterativ beräkning av fakultets-funktionen.

```
f = ({
  int m,n;
  m = 1;
  n = i;
  while (n > 0) {
      m = m*n;
      n = n-1;
  }
  return m;
});
```

Konstruktionen `int m, n;` där flera variabler (parametrar) deklareras 'samtidigt' är frivillig.

5 Kodutläggning för Javalette

5.1 Allmänt om stackmaskiner

I en stackmaskin används en stack för beräkning av uttryck där delresultaten sparas på stacken. I en "renodlad" stackmaskin används stacken även för att lagra värden av variabler och parametrar. Här ges en kort sammanfattning av idén bakom stackmaskinerna.

5.1.1 *Beräkning av uttryck*

Idén med att använda en stack för beräkning av uttryck, är att värdet av uttrycket hamnar överst på stacken, och att alla operationer tar sina operander från stacken.

Uttryck med binäroperator, t.ex. $e_1 + e_2$, beräknas på följande sätt: Först beräknas värdet av e_1, som då hamnar överst på stacken. Därefter beräknas värdet av e_2 på samma sätt. Stacken ser nu ut på följande vis:

v_2	v_1	⋯	⋯	⋯	⋯

Där v_1 och v_2 är värdena av e_1 resp e_2, respektive. (I alla våra bilder av stackar ligger stacktoppen längst till vänster.) Därefter utförs additionsoperationen, som tar de två översta värdena från stacken och ersätter dem med deras summa:

v_1+v_2	⋯	⋯	⋯	⋯

Att beräkna värdet av en enkel konstant, t.ex. 3, är lätt: lägg konstanten på stacken.

Att generera kod enligt ovan består alltså i att lägga ut en följd av instruktioner som lägger konstanter på stacken, och som adderar de två översta elementen, enligt följande två regler:

$$\mathcal{E}[\![e_1\texttt{+}e_2]\!] = \mathcal{E}[\![e_1]\!];\ \mathcal{E}[\![e_2]\!];\ \texttt{ADD}$$
$$\mathcal{E}[\![c]\!] = \texttt{NUMERAL}\ c$$

Här står $\mathcal{E}[\![e]\!]$ för den instruktionsföljd som beräknar värdet av uttrycket e och lägger värdet överst på stacken. I vår notation ovan sätts semikolon mellan instruktionerna[2].

Exempel:

$$\begin{aligned}\mathcal{E}[\![\texttt{1+(2+3)}]\!] &= \mathcal{E}[\![\texttt{1}]\!];\ \mathcal{E}[\![\texttt{2+3}]\!];\ \texttt{ADD}\\ &= \texttt{NUMERAL 1};\ \mathcal{E}[\![\texttt{2}]\!];\ \mathcal{E}[\![\texttt{3}]\!];\ \texttt{ADD};\ \texttt{ADD}\\ &= \texttt{NUMERAL 1};\ \texttt{NUMERAL 2};\ \texttt{NUMERAL 3};\ \texttt{ADD};\ \texttt{ADD}\end{aligned}$$

I en kompilator skriven i ett funktionellt språk, tex Haskell eller ML, motsvaras $\mathcal{E}$-schemat tämligen direkt av en funktion (semikolon blir konkatenering). I en kompilator skriven i ett imperativt språk, tex C, mosvarar lämpligen istället vårt $\mathcal{E}$-schema av en rekursiv procedur, som traverserar ett uttrycksträd och lägger ut (skriver ut) kod.

5.1.2 Åtkomst av variabler

I vårt exempel ovan fanns inga variabler inblandade. Som vi tidigare nämnt, ligger även variablerna på stacken. Frågan är alltså: Hur lägger vi värdet av en variabel överst på stacken? Vi måste då bestämma oss för hur en specifik variabel på stacken adresseras, och här finns det i princip två olika alternativ.

1. Variabler adresseras relativt en pekare, som pekar på början, basen, av aktiveringsposten. Fördelen med denna metod är att oavsett hur stacktoppen ser ut och hur långt vi har kommit i en beräkning, så är adressen till variabeln via bas-pekaren alltid densamma. Nackdelen är att vi i vår interpretator eller genererade kod också verkligen måste reservera ett register för detta ändamål, samt administrera bas-pekaren vid proceduranrop.

2. Variabler adresseras via toppen på stacken, som man då förutsätts ha en pekare till, och att instruktionerna (`ADD` o.s.v.) ser till att denna pekare pekar rätt. Man slipper då att administrera bas-pekaren, men i gengäld får en viss lokal variabel på stacken olika offset relativt stackpekaren beroende på hur mycket som har lagts på stacken, och man måste i alla lägen hålla reda på precis hur djup stacken är.

Ni väljer själva vilken metod ni vill använda, men metod 1 rekommenderas då det gör det något enklare att leta fel i koden.

5.2 Kod för Javalette

Vi ska nu bli mer specifika, och diskutera några Javalette-konstruktioner, och vilka instruktioner det finns som passar till dessa. En fullständig lista av tillgängliga instruktioner finns sist i detta PM. (Observera att `NUMERAL` och `ADD` ovan *inte* tillhör de instruktioner som används i denna laborationen.)

[2]. I praktiken skall varje instruktion vara på egen rad i utdata.

Ett värde eller en variabel av typen `boolean` upptar ett ord (ett stackelement) liksom även `int`, medan en `double` upptar två ord. Det är lämpligt att representera booleaner som heltal med värdet antingen 0 eller 1. Därför kommer lämpligen de bool-värda konstruktionerna i Javalette att ge upphov till `INT`-instruktioner. Vårt $\mathcal{E}$-schema för Javalette blir då istället:

$$
\begin{array}{lll}
\mathcal{E}[\![e_1 \texttt{ + } e_2]\!] & = \mathcal{E}[\![e_1]\!] \text{ ; } \mathcal{E}[\![e_2]\!] \text{ ; } \texttt{DOUBLE\_ADD} & \textit{om } e_1 \textit{ och } e_2 \textit{ har typen double} \\
 & \quad \mathcal{E}[\![e_1]\!] \text{ ; } \mathcal{E}[\![e_2]\!] \text{ ; } \texttt{INT\_ADD} & \textit{om } e_1 \textit{ och } e_2 \textit{ har typen int} \\
\mathcal{E}[\![c_{double}]\!] & = \texttt{DOUBLE\_CONST } c_{double} & \\
\mathcal{E}[\![c_{int}]\!] & = \texttt{INT\_CONST } c_{int} & \\
\mathcal{E}[\![c_{boolean}]\!] & = \texttt{INT\_CONST } boolToInt(c_{boolean}) & \\
\mathcal{E}[\![x_{local}]\!] & = \texttt{REF\_FP\_VAR } (loc(x_{local})) \text{ ; } \texttt{DOUBLE\_DEREF} & \textit{om } x \textit{ har typen double} \\
 & \quad \texttt{REF\_FP\_VAR } (loc(x_{local})) \text{ ; } \texttt{INT\_DEREF} & \textit{om } x \textit{ har typen int eller boolean} \\
\mathcal{E}[\![x_{global}]\!] & = \texttt{REF\_GLOBAL\_VAR } x_{global} \text{ ; } \texttt{DOUBLE\_DEREF} & \textit{om } x \textit{ har typen double} \\
 & \quad \texttt{REF\_GLOBAL\_VAR } x_{global} \text{ ; } \texttt{INT\_DEREF} & \textit{om } x \textit{ har typen int eller boolean} \\
 & \quad \vdots &
\end{array}
$$

5.2.1 If-satser

Först beräknas villkorsuttrycket, ett boolskt värde, som hamnar överst på stacken. Därefter utföres en `IF_FALSE_GOTO`-instruktion, som poppar det boolska värdet, och hoppar till else-grenen om det är falskt. Om det var sant utförs inget hopp, och man hamnar då i koden för then-grenen.

Exempel:

```
if(x + 3 < y)
    ...
else
    ...
```

kod:

```
            REF_FP_VAR (-8)           @@@ lägg värdet av x på stacken
            DOUBLE_DEREF
            DOUBLE_CONST 3            @@@ konstanten 3
            DOUBLE_ADD                @@@ addera
            REF_FP_VAR 44             @@@ lägg värdet av y på stacken
            DOUBLE_DEREF
            DOUBLE_LT                 @@@ 'less-than', ger boolskt värde
            IF_FALSE_GOTO 111         @@@ ta bort boolskt värde, hoppa ev. till else-grenen

            ...                       @@@ 'then'-grenen

            GOTO 222                  @@@ hoppa till slutet av if-satsen
 LABEL 111                            @@@ början av else-grenen

            ...                       @@@ 'else'-grenen

 LABEL 222                            @@@ slutet av if-satsen
```

Ifall else-grenen saknas så bortfaller `GOTO`-instruktionen i then-grenen, samt tillhörande läge (`LABEL 222` i exemplet).

Alla lägen i hela koden för programmet måste vara unika.

5.2.2 Kod för funktionsanrop

Vi ska nu gå in på kod för funktionsanrop, d.v.s.

$$\mathcal{E}[\![f(e_1, ... e_m)]\!] \ .$$

Beräkna först värdet av funktionsargumenten e_1 ... e_m, vars värden alltså hamnar på stacken. I vilken ordning man gör det spelar roll, men det viktigaste är att man har samma ordning i funktionsdefinitionen som vid anropet. Då dessa instruktioner exekverats, ser stacken ut på följande vis:

e_m	...	...	...	e_1	...	...	...	...

Lägg därefter ut instruktionen

`CALL` f

Då man kommer tillbaka från anropet, ligger värdet av funktionsanropet överst på stacken, med argumenten bort-poppade:

result	...	...	...	...

Exempelvis bör anropet `f(5, true)` producera följande kod:

```
DOUBLE_CONST 5
INT_CONST 1
CALL f
```

Här beräknas argumenten från vänster till höger.

5.2.3 Kod för funktionsdefinitioner

Vi ska nu studera kod för funktionsdefinitioner, d.v.s.

$$\mathcal{F}[\![T\ f(T\ x_1, ..., T\ x_m)\ \{\ \mathit{satser}\ \}\ ]\!]$$

där $\mathcal{F}$ är ett kompileringsschema som ger kod för en funktionsdefinition. Som nämnts ovan, så beräknas värdena av argumenten och dessa värden läggs på stacken.

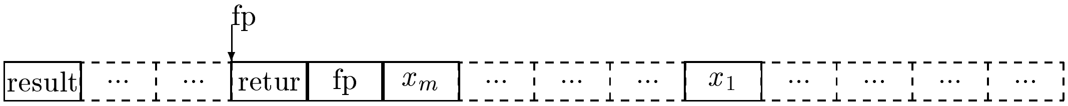
Därefter skall anropet ske, detta göres med antingen `CALL` Efter `CALL`-instruktionen exekverats, men före det att funktionskoden börjat exekveras, ser stacken ut på följande sätt:

fp		fp+2								
retur	fp'	x_m	...	...	...	x_1	...	...	...	...

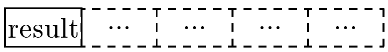
Här är 'retur' returadressen, och upptar 1 ord, och `fp'` är det gamla värdet på frame-pekaren. Funktionens sista argument finns på `fp+2`.

Kompilatorn måste nu lägga in i symboltabellen information om var på stacken de olika parametrarna finns; tänk på att `double` inte har samma storlek som `int` och boolska värden.

Efter det läggs kod ut för de satser som ingår i funktionskroppen. Oavsett vilken väg exekveringen tar genom satserna skall den sista instruktionen bli en returinstruktion. Returinstruktionen återställer det gamla värdet på frame-pekaren och stacken, dvs alla lokala variabler, den sparade returadressen, och den sparade frame-pekaren tas bort från stacken. Efter att det är gjort tas så många ord som anges i returinstruktionen bort från stacken. Ifall det var en `INT_RETURN` eller `DOUBLE_RETURN` läggs det värde som var överst på stacken när returinstruktionen startade tillbaka på stacken. Dvs om stacken ser ut som följer före return-instruktionen,



ser den efter return-instruktionen ut som.



Det finns tre olika returinstruktioner beroende på om funktionen returnerar en int, en double eller ingenting alls. Om inget värde skall returneras finns inte något "result"-värde på stacken efter returinstruktionen, men i övrigt skiljer sig det inte från ovanstående beskrivning.

Exempel:

```
int f(int x, boolean b) {return x;}
```

kod:

```
FUNCTION_START f          @@@ "Header" för funktionen
    REF_FP_VAR 3          @@@ värdet av x
    INT_DEREF
    INT_RETURN 2          @@@ returnera och ta bort argumentet
  FUNCTION_END            @@@ avslutning av funktionen
```

5.2.4 satser i uttryck

En `return`-sats i ett uttryck skall inte returnera till den anropande funktionen. Istället skall eventuella lokala variabler tas bort från stacken så att endast det översta elementet är kvar. Därefter skall alla `return`-satser inom `({-})`-hoppa till ett gemensamt läge efter satserna så att evalueringen av uttrycket kan fortsätta.

Exempel:

```
({
     int x;
     int b;
     ....
     return x;
})
```

kod:

```
ALLOC 2              @@@ plats för x och b, b överst
....
REF_SP_VAR 1         @@@ värdet av x
INT_DEREF
REF_SP_VAR 2         @@@ adressen där resultatet skall ligga
INT_STORE            @@@ skriv resultatet där det skall vara
POP 1                @@@ ta bort variablerna i blocket (2 ord)
                     @@@ men spara resultatet (1 ord)
```

Ett alternativ till ovanstående kod att allokera alla variabler i början av funktionen, även de i `({-})`-uttryck. Om man gör så slipper man hålla reda på det nuvarande stackdjupet när man lägger ut kod.

5.3 Instruktioner för stackmaskinen

`DOUBLE_CONST` x "Pushar" flyttalsliteralen x på stacken, d.v.s. 2 ord. Talet x kan få se ut precis som det gjorde i det ursprungliga programmet. Det är alltså inte nödvändigt att omvandla strängen för x till ett flyttal i kompilatorn, det går lika bra att spara strängen för x och skriva ut den.

`INT_CONST` x 'Pushar' heltalskonstanten x på stacken, 1 ord. I Javalette används även denna till boolska konstanter `false`, dvs 0, och `true`, dvs 1.

`REF_FP_VAR` n 'Pushar' en pekare till en variabel på stacken, 1 ord. n är avståndet från det ord frame-pekaren pekar på, dvs avståndet från den sparade returadressen. Positiva avstånd används för argument och negativa för lokala variabler.

`REF_SP_VAR` n 'Pushar' en pekare till en variabel på stacken, 1 ord. n är avståndet från toppen av stacken till början på variabeln före det att pekaren "pushats", alltså 0 för den variabeln som ligger överst på stacken.

`REF_GLOBAL_VAR` g 'Pushar' en pekare till den global variabel g på stacken, g är variabelnamnet.

`DOUBLE_DEREF` Använder översta värdet på stacken som en pekare till en variabel av typen double. Pekaren tas bort från stacken och värdet av variabeln läggs dit. Notera att pekaren är 1 ord, men det nya värdet är 2 ord.

`INT_DEREF` Använder översta värdet på stacken som en pekare till en variabel av typen `int`. Pekaren tas bort från stacken och värdet av variabeln läggs dit.

`DOUBLE_STORE` 'Poppar' de två översta värdena på stacken, det ord som låg överst används som pekare till en variabel av typen `double`, och de två orden därunder som det värde som skall flyttas till variabeln. Totalt tre ord poppas från stacken.

`INT_STORE` 'Poppar' de två översta värdena på stacken, det ord som låg överst används som pekare till en variabel av typen `int`, och ordet därunder som det värde som skall flyttas till variabeln. Totalt två ord poppas från stacken.

`DOUBLE_NEG` Flyttalsnegering av det översta dubbelordet.

DOUBLE_ADD, DOUBLE_SUB, DOUBLE_MUL, DOUBLE_DIV Binära flyttalsoperationer, 'poppar' två dubbelord, och 'pushar' ett dubbelordsresultat. Högeroperanden ska ligga överst på stacken.

DOUBLE_EQ, DOUBLE_NE, DOUBLE_LT, DOUBLE_LE, DOUBLE_GT, DOUBLE_GE Flyttalsjämförelser `=`, `<>`, `<`, `<=`, `>`, `>=` respektive. 'Poppar' två dubbelord, och 'pushar' ett boolskt värde, 1 om sant, 0 om falskt. Högeroperanden ska ligga överst på stacken.

INT_NEG Heltalsnegering av det översta ordet.

INT_ADD, INT_SUB, INT_MUL, INT_DIV, INT_MOD Binära heltalsoperationer, 'poppar' två ord, och 'pushar' ett ord. Högeroperanden ska ligga överst på stacken.

INT_EQ, INT_NE, INT_LT, INT_LE, INT_GT, INT_GE Heltalsjämförelser `=`, `<>`, `<`, `<=`, `>`, `>=` respektive. 'Poppar' två ord, och 'pushar' ett boolskt värde, 1 om sant, 0 om falskt. Högeroperanden ska ligga överst på stacken.

DOUBLE_TO_INT och INT_TO_DOUBLE Konverterar mellan flyttal och heltal. `DOUBLE_TO_INT` 'poppar' ett dubbelord, konverterar till heltal, och 'pushar' ett ord. `INT_TO_DOUBLE` 'poppar' ett ord, konverterar till flyttal, och 'pushar' ett dubbelord.

POP n Poppar bort n ord från stacken.

ALLOC n Flyttar stackpekaren n ord, för att reservera plats för variabler.

LABEL q Ett läge i koden, att hoppa till. Tänk på att alla lägen i koden måste vara unika.

GOTO q Hoppar ovillkorligt till läge q.

IF_FALSE_GOTO q **och IF_TRUE_GOTO** q Poppar ett boolskt värde från stacken, om detta är `false` (`true`) , d.v.s. 0 ($\neq$0), så hoppar man till läge q.

CALL f Anropar funktionen f, f är funktionsnamnet. Pushar det gamla värdet på frame-pekaren samt returadressen, och sätter frame-pekaren till att peka på den sparade returadressen. Den sparade frame-pekaren och returadressen upptar ett ord vardera.

INT_RETURN n **och DOUBLE_RETURN** n Returnerar från en funktion, med ett värde. Överst på stacken ligger returvärdet (1 eller 2 ord långt). Det ord som utpekas av frame-pekaren används som återhoppsadress, ordet ovanför används för att återställa frame-pekaren. Den int/double som låg överst på stacken läggs tillbaka på stacken efter att den sparade returadressen och frame-pekaren avlägsnats och n ord tagits bort från stacken.

VOID_RETURN n Returnerar från en funktion, utan något värde. Det ord som utpekas av frame-pekaren används som återhoppsadress, ordet ovanför används för att återställa frame-pekaren. Stackpekaren sätts till att peka på ordet över den sparade frame-pekaren och sedan tas n ord bort från stacken.

FUNCTION_START f Start på en funktion.

FUNCTION_END Slut på en funktion.

INT_GLOBAL s Reserverar plats för en global variabel av typen `int` med namnet s.

DOUBLE_GLOBAL s Reserverar plats för en global variabel av typen `double` med namnet s.

5.3.1 *Form på koden från er kompilator*

En datastruktur i Haskell med konstruerare för ovanstående instruktioner ges, tillsammans med en utskriftfunktion. Om ni vill göra en egen datastruktur gäller att det skall vara en instruktion per rad, och eventuella kommentarer skall starta med '`#`'.

En text-fil med stackmaskinsinstruktioner kan sedan köras i en interpretator (`stackmachine`). Interpretatorn körs på följande sätt:

```
stackmachine filnamn.sm
```

Observera alltså att namnet på filen med stackinstruktioner skall sluta på `.sm`.

5.4 Instruktioner för 3-operandsmaskinen

`DOUBLE_MOVI` *dst* *x* Kopierar flyttalsliteralen x till flyttalsregister *dst*.

`DOUBLE_MOV` *dst* *src* Kopierar innehållet i flyttalsregister *src* till flyttalsregister *dst*.

`DOUBLE_LOAD` *dst* *address* Kopierar flyttalet på positionerna *address* och $address + 1$ i minnet till flyttalsregister *dst*.

`DOUBLE_STORE` *address* *src* Kopierar flyttalet i flyttalsregister *src* till positionerna *address* och $address + 1$ i minnet.

`DOUBLE_NEG` *dst* *src* Flyttalsregistret *dst* tilldelas negationen av innehållet i flyttalsregistret *src*.

`DOUBLE_(ADD/SUB/MUL/DIV)` *dst* *src*1 *src*2 Binär flyttalsoperation av formen $dst := src1$ +-*/ $src2$.

`DOUBLE_CMP` *src*1 *src*2 Jämför två flyttalsregister. Resultatet kan användas i villkorliga flyttalshopp.

`DOUBLE_GOTO_(GT/GE/EQ/NE/LT/LE)` *q* Hoppar till läge *q* ifall villkoret var sant i den senaste `DOUBLE_CMP`.

`DOUBLE_TO_INT` *dst* *src* Konverterar flyttalet i flyttalsregister *src* till ett heltal i heltalsregister *dst*.

`INT_TO_DOUBLE` *dst* *src* Konverterar heltalet i heltalsregister *src* till ett flyttal i flyttalsregister *dst*.

`INT_MOVI` *dst* *x* Sparar heltalsliteralen x i heltalsregister *dst*.

`INT_MOVA` *dst* *q* Sparar adressen till läget *q* i heltalsregister *dst*.

`INT_MOV` *dst* *src* Kopierar innehållet i heltalsregister *src* till heltalsregister *dst*.

`INT_LOAD` *dst* *address* Kopierar heltalet på position *address* i minnet till heltalsregister *dst*.

`INT_STORE` *address* *src* Kopierar heltalet i heltalsregister *src* till position *address* i minnet.

`INT_NEG` *dst* *src* Heltalsregistret *dst* tilldelas negationen av innehållet i heltalsregistret *src*.

`INT_(ADD/SUB/MUL/DIV/MOD)` *dst src1 src2* Binär heltalsoperation av formen *dst* := *src1* +-*/% *src2*.

`INT_CMP` *src1 src2* Jämför två heltalsregister. Resultatet kan användas i villkorliga heltalshopp.

`INT_GOTO_(GT/GE/EQ/NE/LT/LE)` *q* Hoppar till läge *q* ifall villkoret var sant i den senaste `INT_CMP`.

`LABEL` *q* Ett läge i koden, att hoppa till. Tänk på att alla lägen i koden måste vara unika.

`GOTO` *address* Hoppar ovillkorligt till *address*.

`GOTO_LINK` *address* Hoppar ovillkorligt till *address*, och sparar samtidigt adressen till efterföljande instruktion i heltalsregister 15.

`FUNCTION_START` *f* Start på en funktion.

`FUNCTION_END` Slut på en funktion.

`INT_GLOBAL` *s* Reserverar plats för en global variabel av typen int med namnet *s*.

`DOUBLE_GLOBAL` *s* Reserverar plats för en global variabel av typen double med namnet *s*.

5.4.1 Register och adresser

Register är heltal i intervallet 0 till 15. Det finns en uppsättning för helltal och en för flyttal, dvs heltalsregister 1 är inte samma register som flyttalsregister 1.

Följande register är speciella:

Heltalsregister 0 Programmets returvärde är det värde som ligger i heltalsregister 0 när programmet terminerar.

Heltalsregister 13 pekar på en stack när programmet startar. Stacken växer nedåt.

Heltalsregister 14 är frame-pointer.

Heltalsregister 15 används av *GOTO_LINK* för att spara returadressen.

Adresser kan vara antingen en symbol, t ex namnet på en funktion, eller ett heltalsregister plus ett heltalsoffset.

5.4.2 Form på koden från er kompilator

En datastruktur i Haskell med konstruerare för ovanstående instruktioner ges, tillsammans med en utskriftfunktion. Om ni vill göra en egen datastruktur gäller att det skall vara en instruktion per rad, och eventuella kommentarer skall starta med '`#`'.

Den resulterande filen från ert program ges som indata till `3op2sparc` vilket ger en Sparc-assemblerfil som kan länkas till et körbart program med `3opln`.

Kommandona körs på följande sätt:

```
3op2sparc filnamn.3m
3opln filnamn.s
```

Observera alltså att namnet på filen med 3-operandinstruktioner skall sluta på `.3m`.