Laboration: Framsida till Kompilator

Arman Shamsgovara
ens12asa@cs.umu.se
Oskar Ottander
c12oor@cs.umu.se

Kurs: Kompilatorns Första Faser: Automater och Grammatik
Institutionen för Datavetenskap
Umeå Universitet

Umeå
14 augusti 2014

Innehåll

1 Introduktion

I denna laboration ska vi skriva framsidan till en kompilator som ska kompilera ett programspråk som delar likheter med Pascal. Framsidan ska konstruera allting fram till och med genereringen av mellankod. Se nästa underrubrik för den grammatik som genererar språket.

För att kunna få programmet att fungera ordentligt ska vi även ändra lite i språket eller lägga till lämpliga actions för att slippa pinsamma felmeddelanden.

1.1 Språkets Grammatik

Det relevanta programspråket genereras av följande grammatik [1]:

```
prog        ->  program id ; dekllist fndekllist compstat .
dekllist    ->  dekllist dekl
            |   dekl
dekl        ->  type idlist ;
type        ->  integer
idlist      ->  idlist , id
            |   id
fndekllist  ->  fndekllist fndekl
            |
fndekl      ->  fnhead dekllist fndekllist compstat ;
fnhead      ->  function fname ( parlist ) : type ;
parlist     ->  dekllist
            |
fname       ->  id
compstat    ->  begin statlist end
statlist    ->  statlist stat
            |   stat
stat        ->  compstat
            |   if expr then stat else stat
            |   while expr do stat
            |   write ( exprlist ) ;
            |   read ( idlist ) ;
            |   id assop expr ;
            |   fname assop expr ;
exprlist    ->  exprlist , expr
            |   expr
expr        ->  aexp relop aexp
            |   aexp
aexp        ->  aexp addop aexp
            |   aexp mulop aexp
            |   fname ( arglist )
            |   ( expr )
            |   id
            |   num
arglist     ->  exprlist
            |
```

2 Användarhandledning

Uppgiftens lösning kan återfinnas på följande plats under cs servrar:

```
/home/c12/c12oor/edu/dv3/lab/
```

De intressanta filerna är de följande;

```
lab2*          // Den körbara kompilatorn.
lab.y          // Huvuddelen av programmet.
lab.h          // Konstanter och typer använda.
lab.l          // Lexfil använd.
Makefile       // Kompilerar och bygger allt i ordning.
stack.c (.h)   // Datatyp använd
table.c        // Hantering av symboler och kommandon med mera.
```

För att bygga den körbara kompilatorn kan *make* användas i en mapp med alla nödvändiga filer. Denna kommer att utföra alla mellansteg i ordning.

Då filen har byggds korrekt kan man sedan köra den. Den tar ingen indata vid uppstart utan väntar på att text matas på standard-input. Den läser sedan tills texten tar slut och matar sedan ut tolkningen av symboler och den genererade mellankoden.

Om någonting inte stämmer med den kod man matat in kommer fel rapporteras, och programmet antingen avslutas direkt, eller fortsätta om endast mindre fel (som en odeklarerad variabel) upptäckts.

3 Ändringar och Justeringar

För att få mellankodsgenereraren att fungera som önskat har ett antal element ur de olika tillhandahållna resurserna ändrats. Både i grammatiken och den grundläggande koden, men utöver detta har ett antal förändringar av hur resultatet ser ut också gjorts.

3.1 Grammatik

Dels har ett par regler lagts till i grammatiken motsvarande utökningar på den grundläggande funktionaliteten. Utöver detta har även ett antal regler modifierats för att undvika konflikter.

3.1.1 Markörer

För att kunna hålla reda på de kodpositioner till vilka man behöver hoppa eller förflytta sig användes ett par markörer.

Ntok Markerar ett goto-kommando och ger en lista vilken kan användas för att peka en serie andra delar till kommandot.

Mark1 Håller koll på platsen i koden som efterföljer den nuvarande.

Dessa markörer läggs sedan till i kontrollstrukturer och loopar för att generera korrekt mellankod, med hopp till rätt platser. Till exempel ser en if-else-sats ut som följer med markörer:

```
Stat :IFSY expr THENSY Mark1 stat Ntok ELSESY Mark1 stat
```

Med backpatching utfört genererar detta mellankod som ser ut som i tabell 1.

Här har markeringar och efterföljarlistor använts för att ge goto-kommandona sina adresser, då dessa inte var kända medans de lästes, utan endast kunnat utrönas då kommandona i de båda stat laggts till.

Position	Kommando	Efterföljande Position	Kommentar
i:	<expr>	i+2	Jämförelsen görs på rad i. (Gå till nästa rad om falsk, annars följs addressen.)
i+1:	GOTO	i+j+3	Hoppa över första och gör andra.
i+2:	<stat1>	?	Sant utför första stat (längd j).
...			
i+j+2:	GOTO	i+j+k+3	Hoppa till fortsättning då stat är klar.
i+j+3:	<stat2>	?	Utför andra stat (längd k).
i+j+k+3:	<forts>	?	Fortsätt program efter.

Tabell 1: Mellankod för if-sats efter backpatching av addresser till gotosatser och så vidare.

3.1.2 Repeat-Until

Som en del av uppgiften skulle en ny regel för ”upprepa tills uttrycket uppfylls” läggas till. Denna regel ser ut som följer:

```
stat  ->  repeat stat until expr ;
```

Med markörer blir detta:

```
stat  : REPEATSY Mark1 stat UNTILSY Mark1 expr SEMI  ;
```

Då backpatching och så vidare hade tagits hand om för resten av koden krävdes sedan endast tilläggning av följande rader för att ta hand om denna regel.

```
backpatch( $6.falselist, $2 );  // Backpatch expr false to goto the first mark
backpatch( $3, $5 );            // Backpatch stat nextlist to mark before expr


$$ = $6.truelist;               // Return nextlist that is the truelist of expr
```

Detta utför korrekt funktionaliteten som upprepar uttrycket *stat* genom att hoppa till den första markeringen så länge som *expr* är falskt, samt fortsätta jämföringen och loopningen ända tills den blir sann.

3.1.3 Kollisioner

Då grammatiken testades med de symboler och regler som givits upptäcktes kollisioner. Programmet kunde inte se skillnad mellan två olika regler utefter vad som den läst. Dessa regler var de följande:

```
stat    -> ...
        | id assop expr ;
        | fname assop expr ;   // Removed bc collision, handled as id special case
```

Genom att ta bort en av dem, och istället utföra bestämmelsen av hurvida det som tilldelades var en variabel eller ett funktionsnamn med hjälp av en if-sats på identifierarnamnets typ kunde denna kollision undvikas.

3.2 Flyttal

För att undersöka funktionaliteten hos typhantering i grammatiken lades ytterligare en typ till. Det existerar alltså numera utöver *integer* även en typ *float*, vilken kan identifieras med nyckelordet ”float” i deklarationslistor, eller som <siffror.siffror> vid definition av konstanter.

Då ingen regel, symbol, tolkning, eller hantering av denna typ fanns har ändringar alltså gjorts i såväl token-, och regel-definitionen i y-filen, samt tolkning av nyckelord och konstantuttryck i lex-filen. Även table.c och lab.h har ändrats för att kunna identifiera och skriva ut denna nya typ.

Genom att lägga till den nya typen visas styrkan i kompilatorns typhantering. Det blir enkelt att utöka med fler grundläggande typer, utan att nya regler och hanteringar behöver läggas till, förutom de som tar hand om identifiering.

3.3 Symboltabell och Åtkomst

En del ändringar har gjorts angående hur de returnerade symbol och kodtabellerna ser ut, samt även direkta förändingar på hur språket fungerar. Dessa kan i vissa fall få tidigare korrekta program att inte längre fungera som de ska.

3.3.1 Globala Variabler

En ändring som utförts var på hurvida globala variabler, eller variabler deklarerade utanför det nuvarande namnrummet, skulle få användas. Då detta leder till läsbarhetsproblem utan att ge mer användbarhet än några möjligheter till enklare sidoeffekter i funktioner, beslutades det att detta fall skulle tas bort.

Det är alltså inte längre lagligt att använda variabler deklarerade i yttre funktioner, utan allt som används måste nu mera antingen passeras som argument till den anropade funktionen, eller deklareras på nytt.

Ett vanligt problem att kolla för problem mellan de olika versionerna blir alltså *identifier not declared*, vilket dyker upp då man inte deklarerat en variabel korrekt.

3.3.2 Rekursion

För att öka styrkan hos programspråket, och göra det enkelt och elegant att utföra ett flertal uppgifter har rekursion tillåtits. Detta innebär att funktioner kan kallas även från funktionen själv.

Denna typ av upprepning är ofta använd och ganska enkel att överskoda. Förutom denna fördel är det även enkelt att lägga till och ta bort, då allt som krävs är att symboltabellen för funktionen får en post till med funktionen själv tillagd vid deklarationen.

3.3.3 Main Scope

Vid utskrift av symboltabell har vi valt att även mata ut det sista namnrummets symboler under ”Main Scope:”. Denna innehåller i de tillhandahållna programmen endast programmet på nivå (*level*) 0.

Detta görs då det annars är oklart vad nivåer representerar, samt även för att tillåta ett flertal program och så vidare i samma textfil.

3.3.4 Konsekvenser

De flesta program, även korrekta sådana, genererar efter dessa ändringar annan mellankod. Främst ser symboltabeller annorlunda ut från tidigare, då funktionsnamn följer med i funktionen också, samt utskrifter av ytterligare ett namnrum.

Att notera är dock att vissa program som tidigare varit otillåtna nu är tillåtna, och vice versa. Rekursiva funktioner räknas till exempel numera som acceptabla, medans variabler använda från yttre namnrum ger felmeddelanden.

Specifikt är testprogrammet **ok4.prg** inte längre korrekt då variabler *f1v2*, *v2*, och *f2v2* på vissa ställen inte deklareras i de funktioner som de används.

3.4 Ändringar av Kod

Även delar av den tilhandahållna hjälpkoden har modifierats, tagits bort, eller utökats. Bland annat funktioner för att rensa nuvarande symboltabell laggts till, och den är även tillgänglig för andra filer så att den kan sparas undan för att plockas fram senare. Även nya egna datatyper har laggts till, till exempel listor för identifierare, och stackar för namnrum. Mer om dessa ändringar och utökningar under systembeskrivningen.

4 Systembeskrivning

Systemet består av ett antal filer, listade under sektion 2, Användarhandledning. Dessa krävs alla för att kunna bygga programmet, och ansvarar vardera för någon deluppgift under kompileringen. I denna sektion beskrivs systemets uppbyggnad samt hur de olika delarna tas omhand.

Mer information om specifika delar av programmet kan återfinnas under de kommentarer som lämnats i koden.

4.1 Lexning

För att kunna läsa en textfil och identifiera olika ord, specialtecken, och numeriska värden krävs vad som kallas en *lexikalanalysator*. Denna tolkar i vårt fall text med hjälp utav reguljära uttryck, och skickar sedan tolkningen vidare i form av *tokens*, samt värden tillhörande dessa. Den läser alltså tecken för tecken och skickar vidare de tokens den hittar till kompilatorn.

Ett exempel är tolkningen av ett heltal, vilket återfinns med ett reguljärt uttryck som accepterar en eller flera siffror, och sedan skickar vidare tokenet ”INTCONST”, samt värdet i form av siffrorna i textformat.

För att kunna utföra sitt arbete använder lexikalanalysatorn, återfunnen i *lab.l*, även en tabell över nyckelord, söfunktioner, och lite felhantering.

4.2 Parsning

Då token skickats korrekt ska de sedan tolkas enligt de grammatiska regler som språket följer. Detta görs utav filen *lab.y*, i vilken grammatiken matats in, tillsammans med mellankodsgenerering.

Tolkningen efter reglerna görs troligtvis via en algoritm lik någon LR-parsning [1]. För att detta ska fungera krävs att reglerna är tillräckligt olika, varför ett par regler har ändrats lite från grundgrammatiken. Även tokens vilka inte motsvarar någon text har lagts till som markörer för hjälp med mellankodsgenereringen.

Korrekt mellankod genereras genom tilläggande av c-kod på strategiska punkter under reglernas, och denna körs på den punkt den återfinns i regeln då den tolkas. Mer om mellankodsgenereringen kan återfinnas i följande sektion (4.3).

4.3 Mellankod

Mellankoden som genereras består dels utav tabeller med symboler, med all information dessa behöver. Samt mellankod med korrekta addresser, symboler och kommandon.

Genereringen av denna sker i kod tillagd i regler och specialsektioner i *lab.y*, men ett flertal andra filer för datastrukturer, utskrift, och så vidare, används också. Att notera är *table.c*, samt *stack.c*.

4.3.1 Argumentlistor

I till exempel anrop till ”read” tas ett flertal argument in. För att kunna hantera alla dessa används listor som byggs upp argument för argument. Till detta ändamål har en simpel, enkel-länkad lista implementerats, kallad ”idlist”. Denna används då ett flertal identifierare behöver läsas in, som i read-funktione, eller i deklarationer.

Även en implementation av datatypen stack har gjorts (även den enkel-länkad). Denna används bland annat vid uppsamlandet av ett flertal uttryck, som vid anrop till write.

Det krävs insamling av värden i dessa fall, då reglerna för uttrycks- eller id-listor inte känner till mycket nog om vad värdena ska användas till, samt att de används till ett flertal olika saker.

[1] Se referenser till DV3 slides.

4.3.2 Typhantering

Typhanteringen byggdes genom att alltid kolla på värdet som *type*-tokenet tilldelats, även fast det i ursprungsversionen inte fanns fler typer än en. Det krävdes även lite smart strukturering av koden i reglerna så att typen hade lästs in innan den användes.

För att kunna hantera de många olika reglerna krävdes det att ett antal olika typer (listor, stackar, heltal, och strängar med mera) kunde skickas som värden med tokens. Detta styrdes med .y-filens %union.

4.3.3 Namnrumshantering

Hanteringen av symboltabeller gjordes via table.c's SYMBOL hanteringsfunktioner. Den innehåller funktioner för att kunna lägga till, kolla upp och skriva ut sådanna SYMBOLer. För att kunna använda dem enkelt utfördes dock ett par ändringer. Dels så att variablen symbtab kan nås även från parsaren, och därigenom användas för att spara ett namnrum för senare användning, och dels tilläggandet av en funktion för att ta bort en symboltabell.

Även stackdatatypen används vid hanteringen av symboler. Då man stegar in i en ny funktion vill man ju ha kvar den yttre funktionens symboler så att de kan användas när den inre funktionen är klar. Detta plus det faktum att inget ur de yttre funktionerna får användas i de inre tillät oss att hålla de äldre namnrummen på stacken och sedan plocka av dem då de innre är klara. Med ett par extrafunktioner som kallas varje gång man går in och ut ur nya funktioner kan symboltabeller hållas i en stack för varje nivå.

4.3.4 Backpatching

Då koden parsas krävs det ibland att kommandon för hopp till andra rader i koden behöver läggas till. En sådan situation är till exempel i if-satser, där koden ska fortsätta från olika platser beroende på om ett uttryck är sant eller falskt. Detta utförs med ett GOTO kommando i mellankoden, men ett problem uppstår då dessa hopp ska göras frammåt. Addressen

för den position till vilket hoppet ska göras har då inte lästs in ännu när GOTO-kommandot matas ut. För att fixa detta används så kallad backpatching[2].

Markörer läggs till i de grammatiska reglerna på strategiska platser. Dels markörer som håller reda på vilken plats i mellankoden som återfinns på ett visst ställe i regeln, och dels markörer som håller reda på en viss GOTO-sats och kan användas för att i efterhand gå igenom dessa och ändra måladressen till de korrekta platserna. Typen som används för att kunna gå tillbaka och ändra måladresser för nästa kodrad att köra efter kommandon kallas nextlist, och de kan innehålla ett antal poster med kodrader, vilka alla kommer att pekas mot det nya målet vid backpatchning.

De flesta av reglerna som kallas stat(ments) har hand om sådanna nextlistor, och backpatchar, mergar och skapar nya sådanna då det behövs. Till exempel för att ta hand om alla de loopar och hopp som behöver göras i kontrollstrukturerna.

För att ta hand om jämförelseuttryck används även typen expression, vilken har hand om två nextlistor (och en symbol). En av dessa nextlistor används för fallet då uttrycket var sant, och pekar till jämförelseradens måladdress. Det andra, för då uttrycket var falskt, pekar till en gotosats som hoppar över den kod som körs om det hade varit sant.

[2]Se referens till Drakboken, *Compilers: Principles, Techniques, & Tools.*

5 Resultat

Då koden byggs och körs verkar resultatet stämma med det (ändrade) målet. Med de testfiler som använts har det inte stötts på något oförutsett problem, och även tester med flyttal fungerar.

5.1 Outputformat

Resultatet av en körning skrivs ut på det format som beskrivs i denna sektion.

Symboltabeller skrivs ut namnrum för namnrum, med titeln innan. Titeln är funktionen eller programmets namn, eller ”Main Scope” för det yttersta.

Varje symbol innehåller sedan följande poster, i ordning:

```
<namn>    type <typnamn> class <klassnamn> offset  <offset> level  <level>
```

Namnet är här variabelnamn, eller värdet hos en konstant.

Typnamn är i vårt fall antingen INTEGER eller FLOAT, men skulle kunna utökas med fler grundläggande typer.

Klassen anger vilken typ av symbol det är, och kan ange något av värdena:

VAR En variabel.

CONST En konstant.

PAR Verkade inte användas, men skulle kunna användas för att markera parametrar.

TEMP Temporära värden som till exempel kan innehålla delresultat i beräkningar. Upp till 6 kan existera samtidigt.

UNDEF Ges till variabler utan typ, till exempel sådana som skapas vid något fel.

Offset anger positionen bland de deklarerade variablerna. Skulle till exempel kunna användas för att lagra dem i någon slags array. Funktioner och konstanter lagras vanligtvis på andra sätt, så dessa ökar alltså inte offset.

Level representerar djupet som de nuvarande variablerna är hittade på. Main Scope har alltså level 0, och varje program eller funktion som nästlas har en level högre.

Mellankod skrivs ut rad för rad, numrerade från noll och uppåt. Varje rad innehåller följande:

```
<radnr>: <kommando>   <första symbol>   <andra symbol>   <måladress>
```

Det finns en 21 kommandon i mellankoden, vilka opererar på olika antal variabler. Symbolerna är från symboltabellen och innehåller de värden som operationen ska utföras på.

Måladresser innehåller det radnummer från vilket exekveringen ska fortsätta efter kommandot utförts. Att speciellt notera är dock att de i jämförelseoperationer ska tolkas som att de ska följas endast då jämförelsen var sann. Vid en falsk jämförelse ska följande rad utföras. Att fortsätta med nästa rad är också vad som händer efter de allra flesta kommandon, dessa har måladress 0. Endast GOTO och jämförelser har måladresser som används.

5.2 Testkörningar

Vid körning på de flesta av de tilhandahållna exempelprogrammen blir resultatet som väntat. Det stämmer då med facit i allt utom ett par detaljer på grund av de ändringar som gjorts (Se sektion 3.3.4).

Nedan följer en körning av ett testprogram, vilket innehåller flyttal, och kördes som:

```
~c12oor/edu/dv3/lab> lab2 < tstf.prg
```

Programmet ser ut som följer:

```
program ok1;
float b, c;
integer x;

begin
   read(b, c);
   x := (b + c) - (b - c);
   x := (1 + 2.4) - x;
   while (x > 0) do begin
      if (b = c) then
          write(b);
      else
          write(c);
      x := x - 1;
   end
end.
```

Detta är den symbol-tabell som genereras:

```
ok1:
0        type INTEGER class CONST offset  3 level  1
2.4      type FLOAT   class CONST offset  3 level  1
1        type INTEGER class CONST offset  3 level  1
x        type INTEGER class VAR   offset  2 level  1
c        type FLOAT   class VAR   offset  1 level  1
b        type FLOAT   class VAR   offset  0 level  1
ok1      type INTEGER class FUNC  offset  0 level  1

Main Scope:
ok1      type INTEGER class FUNC  offset  0 level  0
```

Och följande är mellankoden:

```
 0: FSTART  ok1               0
 1: READ    b                 0
 2: READ    c                 0
 3: PLUS    b         c       0
 4: MINUS   b         c       0
 5: MINUS   temp0     temp1   0
 6: ASS     x         temp0   0
 7: PLUS    1         2.4     0
 8: MINUS   temp0     x       0
 9: ASS     x         temp0   0
10: GT      x         0       12
11: GOTO                      20
12: EQ      b         c       14
13: GOTO                      16
14: WRITE   b                 0
15: GOTO                      17
16: WRITE   c                 0
17: MINUS   x         1       0
18: ASS     x         temp0   0
19: GOTO                      10
20: HALT                      0
21: FEND    ok1               0
```

Som vi kan se stämmer symboltabellerna, alla variabler finns med i ordning och med rätt typ. Även konstanter har plockats upp där de används. Operationerna stämmer också. Att notera är att de temporära variablerna skapas i ordning vid operationer som MINUS, och tas bort så fort de används. Om man följer hoppen som anges vid målen vid jämförelser och goto's ser man att den representerar samma flöde som while-, och if-else-satserna.

6 Diskussion

Det var från början väldigt mycket olika program att sätta sig in i, så det tog lång tid innan man förstod hur allt satt ihop. Sedan var det en hel del att läsa och förstå av den tillhandahållna koden. Hur den skulle användas, och vad som var syftet med dess olika delar. En del av den var även lite svårt skriven, och illa kommenterad, vilket gjorde att det tog en stund att förstå vad allt var till för. Några i klassen kommer säkert ha haft det lite jobbigt med uppgiften på grund av detta.

När man förstått hur saker hängde ihop och lärde sig hålla isär de olika värden man hade att göra gick det dock bra. Det känns som att man fick en väldigt bra förståelse för de olika programmen och i framtiden skulle kunna skriva i princip hela denna delen av en parsning själv.

7 Referenser

1. `http://www8.cs.umu.se/kurser/5DV156/ST14/moment2/`

2. Aho, Lam, Sethi, Ullman, *Compilers: Principles, Techniques, & Tools*, second edition, Addison-Wesley, 2007

3. `http://www8.cs.umu.se/kurser/5DV156/ST14/slides/DV3-3.pdf`

4. `http://www8.cs.umu.se/kurser/5DV156/ST14/slides/DV3-4.pdf`