MÄLARDALENS HÖGSKOLA

CDT301 Lab3 Rapport

Vårterminen 2009, lp4

Trac42 Kompilator

Niklas Pettersson	npn06002@student.mdh.se
Lars Cederholm	lcm06001@student.mdh.se

Lärare: Christer Sandberg

Sammanfattning

This report constains the brilliant knowledge locked away for centuries: This document details the solutions and problems in creating a trac42 compiler using the YACC, FLEX and Bison compiler creation tools.

Innehåll

1 Inledning

Uppgiften som rapporten behandlar gäller att implementera en kompilator för programmeringsspråket trac42. Till vår hjälp hade vi ett framework innehållande grammatik för trac42 och en fungerande parser.

Vår uppgift därför ut på att generera ett Abstrakt syntaxträd för att sedan kunna göra en semantisk analys innehållande namn- och typanalys. Vi har sedan tagit det trädet och genererat kod till trac42i miljön.

2 Översikt av kompilatorn

2.1 Parsern

Följer grammatiken för att kunna bygga upp en träd struktur som används i de senare stegen. Här hittas alla syntax fel som programmeraren har gjort.

2.2 Namnanalys

Går igenom hela syntax trädet och undersöker om alla namn som använts är deklarerade och lägger då till dessa namn i symboltabeller beroende på om namnen är lokala i funktionen eller globala över hela programmet.

2.3 Typanalys

Går igenom alla noder från trädet och undersöker användningen av typerna i programmet för att hitta fel där typerna inte är kompatibla med den nodtyp som undersöks.

2.4 Offsetberäkningen

Här räknar vi ut var på stacken de olika variablerna och argumenten kommer finnas och hur stora dessa är.

2.5 Kodgenerering

Nu stegar vi igenom trädet och för varje uttryck genererar vi den motsvarande koden till trac42is stackspråk.

3 Parsern

Med hjälp av grammatiken som beskriver trac42 som Bison följer, för varje grammatisk regel har vi skapat en nod till det abstrakta syntaxträdet som sedan skickas uppåt med noder som underliggande regler genererat för att skapa ett träd.

4 Abstrakt syntaxträd

I de olika passen som görs mot syntaxträdet så används rekursiva funktioner som innehåller switchsatser för att avgöra vilken typ av nod som hanteras just

nu. Beroende på pass och typ av nod så görs rekursionen i olika ordning, detta är för att vissa saker måste göras före andra.

5 Semantisk Analys

I den semantiska analysen har vi gjort två pass över trädet för att se till att inga semantiska fel finns i programmet.

5.1 Namnanalys

I det här passet bygger vi upp en symboltabell för varje funktion i programmet samt en symboltabell för globala namn. Det första vi gör är att skapa den globala symboltabellen för funktionsnamnen för att tillåta användning av funktionerna i funktionskropparna. De lokala symboltabellerna byggs upp genom att först gå igenom alla deklarationer samt parametrar till funktionen och lägga dessa namn i den lokala tabellen, om dessa redan är deklarerade så genereras ett fel. Sedan gås funktionen igenom och alla namn testas mot symboltabellen för att se till att inga okända namn används.

5.2 Typanalys

I detta pass stegas trädet igenom och för varje nod som stöts på kontrolleras att de typer som finns i undernoderna stämmer överens med de förväntade typerna.

5.2.1 Program

En Programnod består av Function som förväntas returnera void.

5.2.2 Function

En Function innehåller Variable, Stmnt och nästa funktion alla ska returnera void för ett komplett program.

5.2.3 Variable

I variablenoden så läggs typen på variabeln in i symboltabellen och om allt gick som det ska så returneras void.

5.2.4 Stmnt

Samlingsnamn för noderna Assign, If, While, Read, Write, Return, FuncCall-Stmnt som alla ska returnera void om de är korrekta.

5.2.5 Assign

Jämför vilken typ som nodens identifierare har med hjälp av symboltabellen mot typen som Expr returnerar, efter detta kontrolleras att nästa Stmnt returnerar void.

5.2.6 If

Består av en Exprnod som ska ha typen bool, det finns även then, else och nextnod som alla ska returnera void om de är korrekta.

5.2.7 While

Består av en Exprnod som ska ha typen bool och en Stmnt och next som båda ska returnera void om de är korrekta.

5.2.8 Read

Kollar id mot symboltabellen och kollar nextnoden så att den returnerar void.

5.2.9 Write

Kollar så att next returnerar void och att exprnoden inte är error.

5.2.10 Return

Kollar så att next returnerar void och att exprnoden inte är error.

5.2.11 FuncCallStmnt

Hämtar ut från symboltabellen med hjälp av funktionsnamnet typen för funktionen och jämför med Actualslistan, sen kollas next noden mot void.

5.2.12 Expr

Samlingsnamn för FuncCallExpr, Unary, Binary, IntConst, BoolConst, StringConst och RValue som alla ska returnera typen som associeras med innehållet i noden.

5.2.13 FuncCallExpr

Hämtar ut från symboltabellen med hjälp av funktionsnamnet typen för funktionen och jämför med Actualslistan, sen returneras retur typen för funktionen.

5.2.14 Actual

Kollar så att next returnerar void och att exprnoden inte är error.

5.2.15 Unary

Returnerar typen av Exprnoden.

5.2.16 Binary

Jämför typerna för leftoperandnoden med rightoperandnoden och returnera den typen.

5.2.17 IntConst

Returnera int som typ.

5.2.18 BoolConst

Returnera bool som typ.

5.2.19 StringConst

Returnera string som typ.

5.2.20 RValue

Hämtar ut typen ur symboltabellen och returnerar den.

6 Kodgenerering

Då det är stackkod som genereras måste vi först räkna ut offset för alla variabler och parametrar relativt till funktionens frame pointer (FP). När alla offsets är beräknade kan vi påbörja kodgenereringen.

6.1 Offset beräkning

6.1.1 Parameter offset

Offset för parametrar börjar om på 2 för varje funktion därför att aktiveringsposten innehåller returaddressen precis ovanför FP. När man pushar parametrar på stacken blir offseten olika om man pushar från vänster till höger eller vice versa. Detta i sig spelar ingen roll så länge man är konsekvent och håller sig till det ena eller andra sättet. Detta är viktigt när vi sedan vill komma åt en parameter vi tidigare pushat då vi måste kunna relatera vart på stacken parametern finns jämfört med vilken position parametern har i funktionens parameterlista. Vi har därför valt att pusha parametrar från höger till vänster, vilket betyder att den första parametern i funktionens parameterlista hamnar längst upp på stacken jämfört med de övriga parametrarna. Vi behöver veta vad nästa offset blir om det är fler parametrar så därför skrivs värdet över med den nya offseten.

Algoritm:

```
for each Function func in Program do {
  formaloffset := 2
  for each Formal f in func do {
    AddOffset(f, formaloffset)
    formaloffset := formaloffset + f.size
  }
}
```

6.1.2 Local offset

För att beräkna offset för lokala variabler så behöver vi först räkna ut offseten som är föregående offset minus variabelns storlek.

Algoritm:

```
for each Function func in Program do {
  localoffset := 0
  for each Variable v in func do {
```

```
    localoffset := localoffset - v.size
    AddOffset(v, localoffset)
  }
}
```

6.2 Stackkods generering

Kod genereras genom att traversera trädet och kalla befintliga funktioner beroende på typ av nod. Genom att använda de befintliga kodgenererings verktygen har ett minimalt antal problem uppstått. Det enda egentliga problemet som varit var att veta vart t.ex. en if-sats skall hoppa om den utvärderas till sant. Vi löste detta problem genom att spara undan de noder som skapas när instruktionerna för 'BRF' och 'BRA' läggs till i kod-trädet.

7 Testning

Vi testade alla medföljande testfiler och jämförde våran kompilators resultat med goodt42 och fann att den ända skillnanden är att vi hanterar argument i en annan ordning mot den kompilatorn men får ändå samma slutresultat på programmen.

8 Slutsats

Att skriva en kompilator med verktygen vi använde och med den grammatik samt stödfunktioner från labbskelettet har gett oss en god inblick i de utmaningar och till viss del problem man kan stöta på.