5DV204/VT18: OU2 – Reguljära språk

Buster Hultgren Wärn & Victor Liljeholm
dv17bhn@cs.umu.se & dv17vlm@cs.umu.se
2018-03-14

Buster Hultgren Wärn dv17bhn@cs.umu.se & Victor Liljeholm dv13vlm@cs.umu.se

Innehåll

Buster Hultgren Wärn dv17bhn@cs.umu.se & Victor Liljeholm dv13vlm@cs.umu.se

1 Kompilering och körning

Det finns tre olika skrivna program. Dem alla kompileras med make filen, och det är endast att skriva in make i terminalen så är programmen körbara. Kompilation kommer dock att gås igenom.

cleancomments.c

Kompilering: gcc -std=c99 -Wall -g -o cleancomments cleancomments.c

Exekvering: ./cleancomments [file0] [file1]
./cleancomments – Kommando för att exekvera programmet.
file0 – Namn på fil att läsa av. Filen som innehåller kommentarer.
File1 – Namn på fil att skriva till. Kommer innehålla file0 fast utan kommentarer.

wordcount.c

Kompilering: gcc -std=c99 -Wall -g -o wordcount wordcount.c

Exekvering: ./wordcount [file0]
./wordcount – Kommando för att exekvera programmet.
file0 – Namn på fil att läsa av.

rundfa.c

Kompilering: gcc -std=c99 -Wall -g -o rundfa rundfa.c dfa.c

Exekvering: ./rundfa [file0]
./rundfa – Kommando för att exekvera programmet.
file0 – Namn på fil som innehåller DFA specifikation att läsa av.

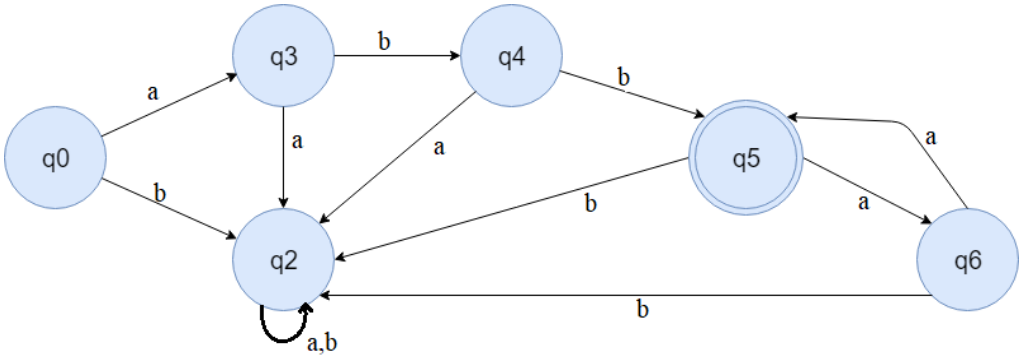
2 Reguljära utryck ifrån 1.1

Utryck 1

Uttrycket innehåller först abb och sedan ett jämnt antal a. Antal a får upprepas godtyckligt, bara dem är jämna. Därmed får vi uttrycket:

$$(abb)(aa)^*$$

Detta ger en DFA liknande figur 1.



Figur 1: DFA för utryck 1

Utryck 2

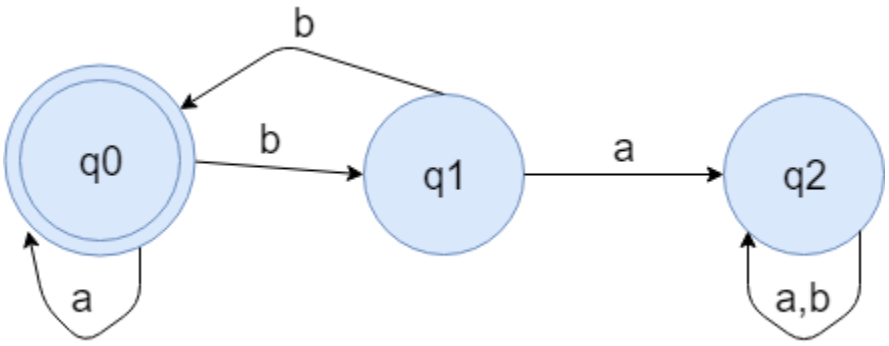
Uttrycket innehåller jämnt många b:n och en godtycklig mängd a:n. Uttrycket behöver även kunna upprepa båda dealarna med mellan a och b godtyckligt många gånger, i alla följder efter varandra. Uttrycket:

(a*(bb)*)*

Uppfyller detta. Det säger att:

1. Godtyckligt många a:n får användas.
2. Två b:n i följd av varandra får användas godtyckligt många gånger.
3. Allt detta får upprepas, godtyckligt många gånger.

Detta ger en DFA likande figur 2.



Figur 2: DFA för utryck 2

Uttryck 3

Uttryck 3 är en kombination av alla möjliga fall i specifikationen och ser ut som följande:

```
\\b([a-z][aeiouy]{2}ing|
    [aeiouy][aeiouy][a-z]ing|
    [a-z]{2}[aeiouy]{2}ly|
    [a-z][aeiouy]{2}[a-z]ly|
    [aeiouy]{2}[a-z]{2}ly)\\b
```

Bara ord med sex bokstäver, två på varandra följande vokaler och slutar på alltingen *ly* eller *ing* matchas med detta uttryck.

3 Resultat av körning i 1.2

När programmet wordcount körs (vilket använder uttrycket i 1.1.2) mot den givna textfilen så ges följande resultat som visas i figur 3. Varje rad i figur 3 är ett ord ifrån texten som det reguljära uttrycket har hittat och hur många gånger det förekommer i texten.

```
really 11
freely 7
nearly 9
easily 4
saying 2
seeing 3
cooing 2
deeply 1
eating 2
laying 1
fairly 4
weakly 1
keenly 1
loudly 1
deadly 2
paying 2
Number of total words found: 53
```

Figur 3: Resultat av att köra wordcount.

Totalt så hittar programmet 53 ord som matchar uttrycket och matchar de tre första raderna i specifikationerna.

4 Redogörelse för programstrukturen i 1.2

I wordcount.c så används en struct (wordCount), som syns i figur 4. WordCount agerar som en datatyp, fast den har ingen fast gränsyta eller satta operationer.

```
typedef struct wordCount{

    int inUse;
    int capacity;
    word *words;
} wordCount;
```

Figur 4: WordCount ”datatype”.

Denna struct håller tre variabler. *inUse* håller koll på hur många olika ord det finns för tillfället i *wordCount*. *capacity* håller koll på hur många olika ord som för tillfället får plats i *wordCount*, detta utrymme kan ökas dynamiskt. Den sista variabeln är en egen struct *word*, syns i figur 5

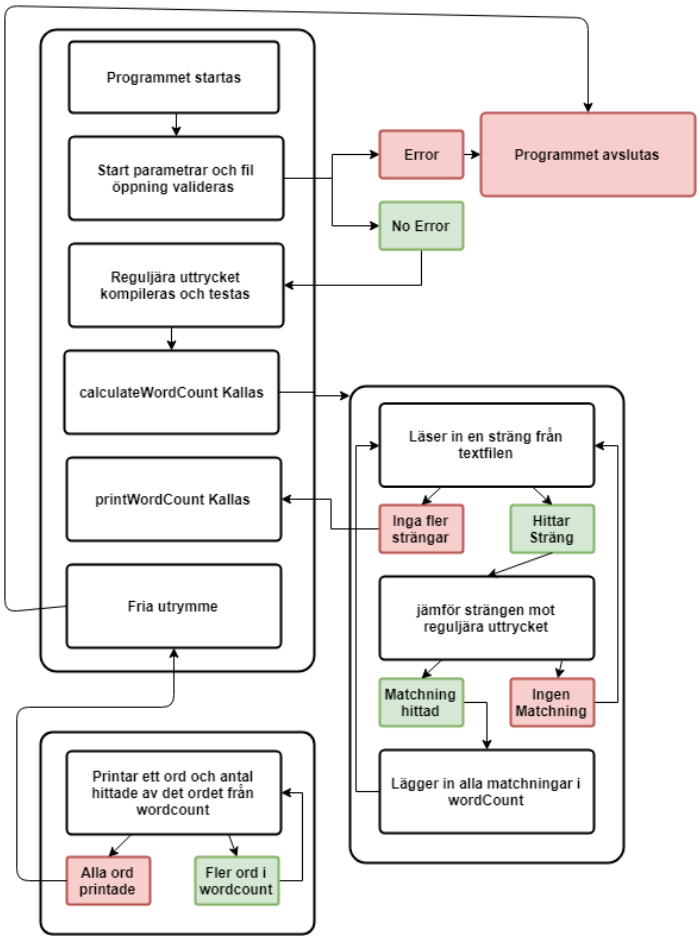
```
typedef struct word{

    char word[7];
    int count;
} word;
```

Figur 5: word ”datatype”.

wordCount skapar ett fält av denna struct igenom att allokera minne för samma antal som talet i variabeln *capacity*. structen *word* själv har två variabler, en char som har koll på ett ord som matchat uttrycket i texten och en integer som håller koll på hur många gånger det ordet matchats i texten.

Programmet själv skapar en *wordCoun*t variabel (med till en början *capacity* 100), den gör sedan en fil och start parameter validering. Efter att sen ha kompilerat det reguljära uttrycket så läser den in en sträng i taget från angiven fil och checkar strängen mot uttrycket, Inspiration av hur detta ska kodas har hittats online[1].

Hittar den ett matchade ord så lägger den till ordet i *wordCount*. När den gått igenom hela textfilen så skriver den ut alla matchande ord och antal gånger ordet hittats i texten. Följande figur(figur 6) visar programmets flödesschema.

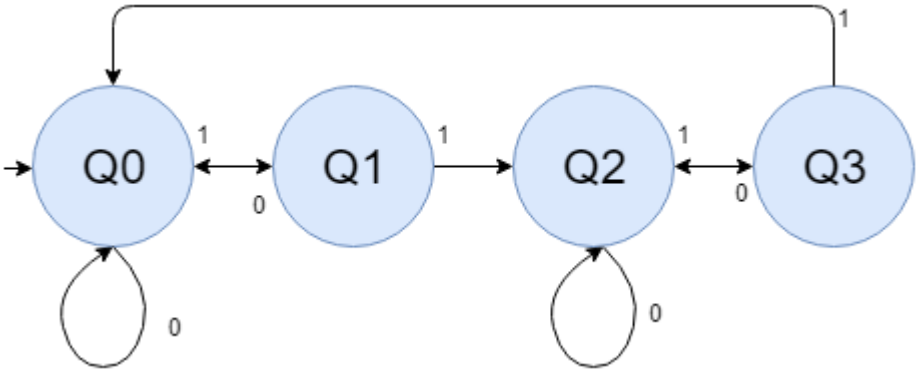


Figur 6: flödesschema för wordcount

5 DFA ifrån 2.1

Uppgiften i 2.1 är att läsa en fil, tag bort alla kommentarer ifrån den och sedan skriva ut samma fil fast nu utan kommentarerna. En kommentar består av två huvudsakliga delar – en början och ett slut. Början på kommentaren består av två enskilda tecken – /* – och slutet är samma tecken fast i annan ordning– /*. Alltså kan ett enskilt ”/” läsas utan att detta tolkas som en kommentar.

DFA:n för att rensa kommentarer är inte alltför tydlig. Den har inget accepterande sluttillstånd då den skall kunna läsa godtyckligt långa texter, och de fyra stegen som först uppenbarar säger är inte tillräckliga. Denna DFA visas i figur 7.



Figur 7: DFA för cleancomments

Varje tillstånd representerar då en karaktär texten som har lästs. Tillstånd Q0 är starttillståndet som representerar att ingen kommentar har lästs. Alltså då ett tecken läses här skall de skrivas, ty det är ingen kommentar. Då ett tecken läses så tar Q0 vägen 0, förutom då ett ”/” har lästs, och Q0 tar vägen 1.

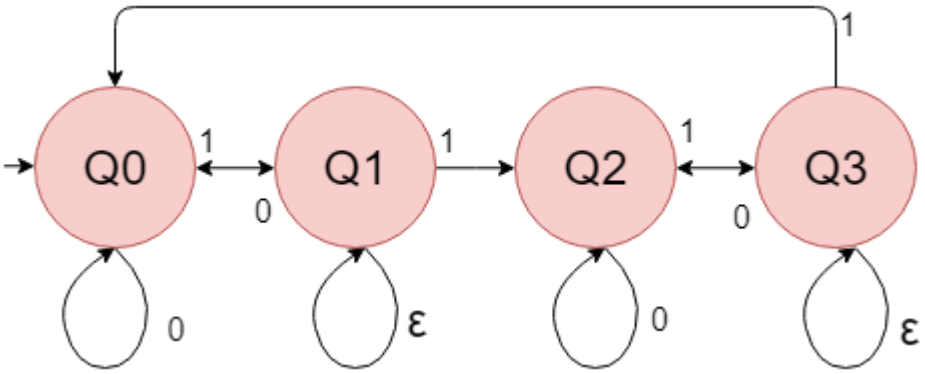
Då är det nuvarande tillståndet Q1. Detta tillstånd representerar att början på en kommentar har lästs. Dock är det inte säkert att detta är en hel kommentar, så läses en ”*” tar Q1 vägen 1 och hamnar i Q0. Om ett annat tecken läses så tas vägen 0, som leder tillbaka till tillstånd Q0.

Tillstånd Q2 representerar att en kommentar har lästs. Detta tillstånd är motsatsen till Q0 då inga tecken skrivs till fil och alla tecken förutom ”*” leder tillbaka till sig själv. Annars tas väg 1 som leder till Q3.

Q3 är ett tillstånd som är likt Q1. Början av slutet på en kommentar har lästs, men det är osäkert på ifall detta är slutet på en kommentar eller endast ett enskilt ”*” tecken. Ifall ett ”/” tecken läses så tas väg 1, som leder tillbaka till Q0. Annars så tas väg 0 som leder till Q2.

Det finns dock ett problem med denna DFA. Ifall början (eller likaså slutet) på en kommentar läses med dubbla / tecken så kommer DFA:n inte att uppfatta detta. T.ex. läses //* så kommer DFA:n först att gå ifrån Q0 till Q1 eftersom ett / har lästs. Nästa tecken är inte ett ”*”, Q1 går tillbaka till Q0. När sedan ”*” tecken läses tar Q0 väg 0, som går till sig själv. På detta sätt finns en kommentar, men som inte hanteras av DFA:n.

En mer komplicerad DFA behövs alltså för att skrivas för att hantera detta. Figur 8 är exempel på denna DFA:s motsvarande RFA.



Figur 8: RFA för wordcount

Den inkluderar två epsilon steg, som hanterar just detta problem. Epsilon steget läser om samma tecken i Q1 och Q3 ifall tecken inte leder till väg 1. Det går att göra om RFA:n till en DFA, men ett beslut fattades om att koden skulle bli mer svårläst i sådana fall, och RFA:n implementerades istället för DFA:n.

För att testa ifall programmet tog bort kommentarer som förväntat så kördes det med denna text:

Inte ////////////////* kommentar*/ /**/ k/om/me//n*/tar /*KOM/ME /N /// * ** //
/*TAR*****/INTE KOMMEN/*TAR

I detta fall testas att flera stycken ”/” eller ”*” kan läsas utan att Q1 går till Q2 eller Q3 går till Q0. Då alla faktiska kommentarer tas bort, ska denna text skrivas ut:

Inte /////////////// k/om/me//n*/tar INTE KOMMEN

Programmet skriver ut exakt denna text, och därmed fungerar det.

6 Programstruktur ifrån 2.2

Lösningen till 2.2 krävde två olika .c filer. Den ena, dfa.c, är implementationen på en datatyp som representerar en DFA. Den andra, rundfa.c, är programfilen som skapar, och använder DFA:n.

Datatypen DFA

DFA är en statisk datatyp implementerad med hjälp av ett endimensionellt fält. Varje element i fältet är ett tillstånd i DFA:n. Varje tillstånd i DFA:n har två andra tillstånd kopplade till sig, som nås genom valet a eller b. DFA:n har ett nuvarande tillstånd, vilket börjar som starttillståndet men går att byta från starttillståndets olika vägar.

Detta ger, ifrån användarens perspektiv, rätt stora nackdelar. Först och främst är den inte dynamisk. För att skapa en DFA måste alltså storleken kännas till på förhand. Sen så kan DFA:n inte hantera olika namn på varje tillstånd. Tillstånden måste vara numrerade, med början ifrån 0 och sedan framåt. Kommer ett tillstånd ha ett högre nummer en antalet tillstånd -1 så klarar inte DFA av detta. Ett tredje problem är att starttillståndet som finns på rad 1 måste finnas på antingen rad 2 eller 3.

Gränsyta till DFA

Empty() – Returnerar en tom DFA.
IsEmpty(dfa) – Ifall DFA:n ”dfa” är tom returneras 1, annars 0.
SetStates(dfa, size) – Ger storleken ”size” till DFA:n ”dfa” och allokerar minne till varje tillstånd.

SetStart(dfa, startState) – Ger starttillstånd ”startState”
InsertState(dfa, acceptable, stateNr) – Lägger in tillståndet ”stateNr” i DFA:n ”dfa” där ”acceptable” bestämmer ifall tillståndet är accepterande.
ModifyState(dfa, state, path, toState) – Sätter vägen ”path” i tillståndet ”state” från DFA:n ”dfa” till tillståndet ”toState”. ”toState” kan vara tillståndet som modifieras.
ChangeState(dfa, path) – Byter nuvarande position ifrån DFA:n ”dfa” till vägen ”path” ifrån det nuvarande tillsåndet.
GetState(dfa) – Returnerar det nuvarande tillståndet från DFA:n ”dfa”.
IsAcceptable(dfa) – Returnerar 1 ifall det nuvarande tillståndet i DFA:n ”dfa” är acceptabelt, annars 0.

Kill(dfa) – Avallokerar allt minne till DFA:n ”dfa”.

Programstruktur

Programstrukturen för rundfa.c är inte komplicerad, fast källkoden blir ändå väldigt rörig. Detta beror mycket på att datatypen DFA är implementerad statisk. Figur 9 hjälper till med att beskriva flödesschemat i programmet.

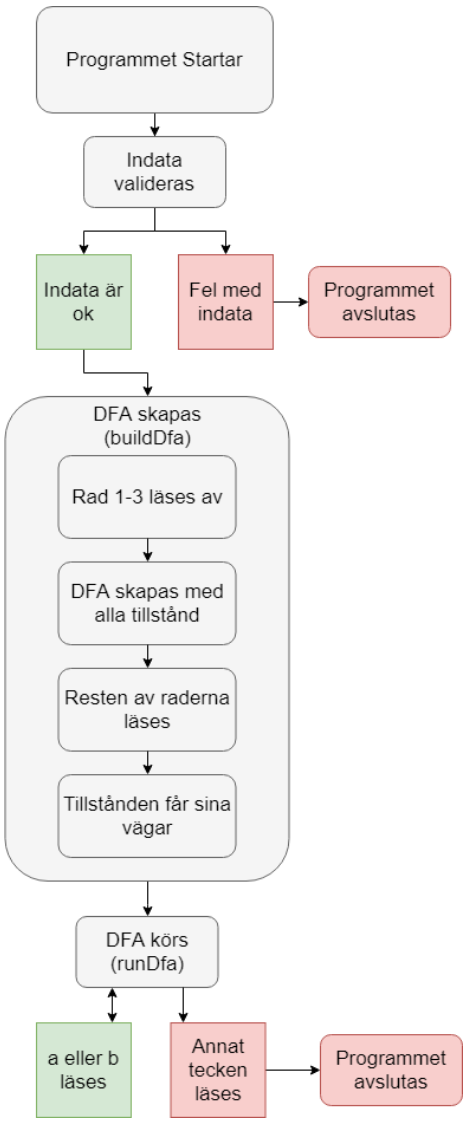
Det första som sker är parameter och filvalidering. Först avgörs det om det finns rätt antal argument (detta är två), och sedan ifall infilen går att öppna för läsning.

Hela DFA:n byggs i funktionen buildDfa. Funktionen börjar med att öppna och läsa av filen som innehåller informationen kring den DFA som skall byggas. De tre första raderna läses av, de är dessa som innehåller informationen kring de olika tillstånden i DFA:n. Antal tillstånd räknas, och en DFA skapas med dessa tillstånd. När detta är klart, har rad två och tre snurrats igenom minst tre gånger var. Alltså, inte optimalt implementerat.

Efter detta läses resten av filen av, som innehåller varje väg som tillstånden har. Ett efter ett läggs dem till i DFA:n. Detta görs i funktionen setPaths.

Med detta är DFA:n färdigbyggd. Då körs funktionen runDfa, som kör DFA:n via input ifrån användaren. DFA:n börjar i start steget, och därifrån kan användaren välja varje väg från det nuvarande tillståndet genom att skriva in ”a, ”b” eller en kombinatorisk sträng av dessa två tecken i terminalen. Användern får information om strängen är accepterande eller inte.

Då annat tecken skrivs in så avslutas funktionen. Allt minne frigörs därefter, och programmet avslutas.



Figur 9: flödesschema rundfa

7 Resultat av testkörningar i 2.3

för att testa dem reguljära uttrycken i 1.1.1 och 1.1.2 så matas dem in i rundfa igenom textfiler som ser ut som följande, figur 10:

```
1   q0
2   q5
3   q0 q1 q2 q3 q4 q6
4   q0 0 q3
5   q0 1 q2
6   q2 0 q2
7   q2 1 q2
8   q3 0 q2
9   q3 1 q4
10  q4 0 q2
11  q4 1 q5
12  q5 0 q6
13  q5 1 q2
14  q6 0 q5
15  q6 1 q2
```

```
1   q0
2   q0
3   q1 q2 q3
4   q0 0 q0
5   q0 1 q2
6   q2 0 q3
7   q2 1 q0
8   q3 0 q3
9   q3 1 q3
10
```

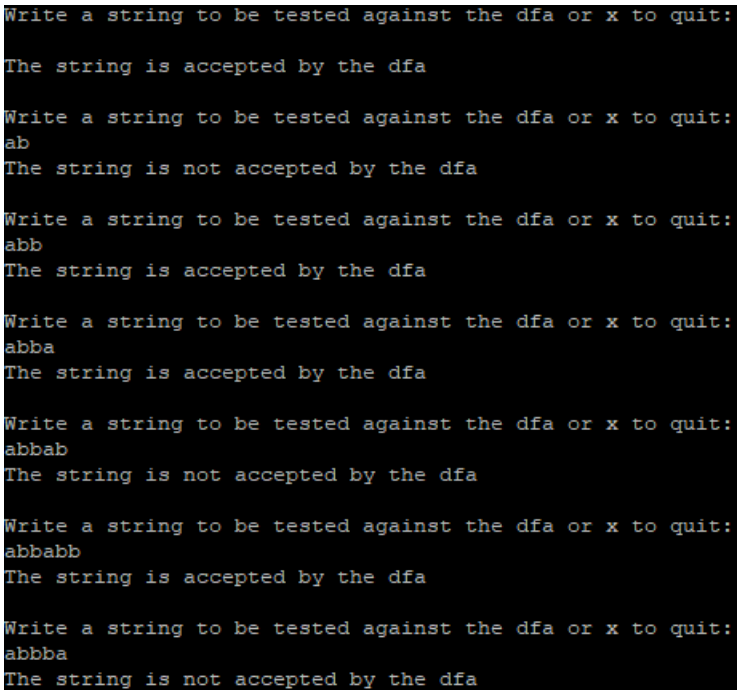
Figur 10: DFA – indata för 1.1.1 till vänster, 1.1.2 till höger

Vid test av uttrycket i 1.1.1 som matats in i rundfa programmet så fås följande resultat, se figur 11

```
Write a string to be tested against the dfa or x to quit:

The string is not accepted by the dfa

Write a string to be tested against the dfa or x to quit:
abb
The string is accepted by the dfa

Write a string to be tested against the dfa or x to quit:
abba
The string is not accepted by the dfa

Write a string to be tested against the dfa or x to quit:
abbaa
The string is accepted by the dfa

Write a string to be tested against the dfa or x to quit:
abbaaaaaa
The string is accepted by the dfa

Write a string to be tested against the dfa or x to quit:
abbaaaaaab
The string is not accepted by the dfa
```

Figur 11: Resultat av testning av det reguljära uttrycket i 1.1.1.

Här testas fler olika strängar mot DFA:n, alla testade uttryck får rätt utkomst. Även reguljära uttrycket i 1.1.2 testas genom att matas in i rundfa, där fås följande resultat, se figur 12.



Figur 12: Resultat av testning för det reguljära uttrycket i 1.1.2.

Även här testas flera olika strängar mot DFA:n, där igen alla strängar får rätt utkomst.

8 Diskussion

Vi (Buster och Victor) har arbetat tillsammans genom att först och främst tänka igenom de olika delarna till varje program och sedan implementerat dem. Ibland har vi suttit och kodat på samma dator, medan ibland sitter vi på olika datorer tätt inpå varandra. På detta sätt har vi redan löst problemen, och kan skriva koden mer effektivt. Då vi båda har suttit bredvid varandra har vi även kunnat diskutera då någon av oss dykt på ett problem.

Rapporten har skrivits på liknande vis. Vi diskuterade först igenom den, och sedan började vi individuellt skriva på var sin del. Då delen var färdigskriven så läste vi båda igenom den och diskuterade, för att sedan ändra de delar som var ofullständiga.

I vårt arbetssätt har vi kunnat få ut de bästa delarna av att jobba i par. Vi har tillsammans löst alla problem, samtidigt som ingen av oss har begränsats av den andre. Starka sidor som finns hos oss båda har lyckats komma fram.

9 Referenser

[1] G. Ellis, "Match and Capture - Regular Expressions - C example", Wellho.net. [Online]. Available: http://www.wellho.net/resources/ex.php4?item=c206/reg2.c. [Accessed: 12- Mar- 2018]