

KARLSTADS UNIVERSITET
INSTITUTIONEN FÖR MATEMATIK OCH DATAVETENSKAP

TEORETISK DATALOGI
DVGA17

Automater och Språk

Skriven av:

Alexander FLOREAN
florean.alexander@gmail.com
Emanuel SVENSSON
emansven100@student.kau.se

Handledare:

Kerstin ANDERSSON

30 november 2019



Innehåll

1	Översikt	2
2	Antaganden	2
2.1	Lexikalanalys	2
3	Resultat	2
3.1	bash	2
3.2	grep	3
3.3	expr	5
4	Lexikal analys	6
5	Sammanfattning	8
6	Referenser	9
7	Bilagor	10

1 Översikt

Laborationen gick ut på att utforska unix-verktygen `bash`, `grep` och `expr` och hur dem använder sig av reguljära uttryck, samt att skapa en förenklad C-skanner med lexikalanalys och verktyget `flex`. Som källa till unixverktygen användes dokumentationen som unixprogrammet `man` förfogade.

2 Antaganden

2.1 Lexikalanalys

I denna analys centreras språket kring själva koden *fak.c*. Ett antagande gjordes att komplexiteten på det språk som *fak.l* skulle vara i samma grad av komplexitet som exemplet som gavs för *p1.p*

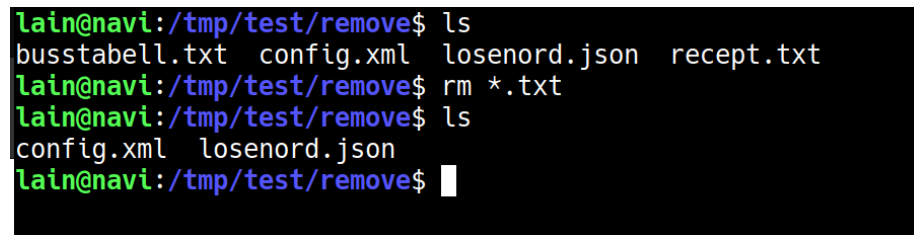
3 Resultat

3.1 `bash`

`bash` (Bourne-again shell) är en kommandotolk som är vanligt på unix-liknande system som Ubuntu och OSX.

Funktionalitet En stark funktionalitet hos `bash` är dess kraftfulla förmåga att använda shellglobar för att matcha filnamn. Shellglobbing är motsvarigheten av reguljära uttryck för filer och kataloger. Karaktären `*` används för att matcha alla karaktärer i en sträng.

Exempel: matchning av filtyper

A terminal window with a black background and green text. The prompt is 'lain@navi:/tmp/test/remove\$'. The first command is 'ls', which lists 'busstabell.txt', 'config.xml', 'losenord.json', and 'recept.txt'. The second command is 'rm *.txt', which removes the two .txt files. The third command is 'ls', which now only lists 'config.xml' and 'losenord.json'.

```
lain@navi:/tmp/test/remove$ ls
busstabell.txt  config.xml  losenord.json  recept.txt
lain@navi:/tmp/test/remove$ rm *.txt
lain@navi:/tmp/test/remove$ ls
config.xml  losenord.json
lain@navi:/tmp/test/remove$
```

Figur 1: Filer raderas efter filtyper

I exemplet används `*` för att matcha alla filnamn som slutar med `.txt` och därefter effektivt raderar kommandot alla textfiler.

Exempel: matchning av alla filer

```
lain@navi:/tmp/test$ ls
move remove remove_cont
lain@navi:/tmp/test$ ls remove/
config.xml losenord.json
lain@navi:/tmp/test$ mv remove/* move/
lain@navi:/tmp/test$ cd move/
lain@navi:/tmp/test/move$ ls
config.xml losenord.json
lain@navi:/tmp/test/move$
```

Figur 2: Filer flyttas från en katalog till en annan katalog

Kommandot i figuren matchar alla filer i en katalog och flyttar över dem till en annan katalog.

Exempel: matchning av innehåll i filnamn

```
lain@navi:/tmp/test/remove_cont$ ls
auto_sent_mail.kmn config.xml losenord.json skapad_auto_config.txt verification_auto.js
lain@navi:/tmp/test/remove_cont$ rm *auto*
lain@navi:/tmp/test/remove_cont$ ls
config.xml losenord.json
lain@navi:/tmp/test/remove_cont$
```

Figur 3: Filer raderas efter innehåll i deras filnamn

I figuren visas hur man i bash kan använda globbing för att matcha filer som innehåller ett visst nyckelord och ta bort dem.

3.2 grep

grep är ett kommandotolkverktyg som används för att hitta textlinjer som matchar reguljära uttryck.

Funktionalitet *grep* tar emot ett reguljärt uttryck och söker sedan efter en linje som matchar och returnerar linjer med matchande delar upplyst. *grep* kan konfigureras till att söka i olika medier som i en fil, input från användaren som unix pipes. Den kan även söka igenom kataloger efter filer vars namn matchar ett reguljärt uttryck och sedan kolla ifall den finner en matchning i dess innehåll.

Exempel: sökning i fil

```
lain@navi:~/Nextcloud/univ/datastrukturer_och_algoritmer/labbar/lab1/lab1-perf$ grep -e 'int \*[b-k]' analyze.c
static int *create_sorted_list(int n){
static int *create_reverse_list(int n){
static int *create_random_list(int n){
static void create_quickbest_switch(const int *src, int *dest, int start, int stop, int *pos){
static int *create_quickbest_list(int n){
lain@navi:~/Nextcloud/univ/datastrukturer_och_algoritmer/labbar/lab1/lab1-perf$
```

Figur 4: Sökning i fil

Sökning i en c-fil efter linjer som innehåller deklARATIONER med int-pekare vars namn börjar med bokstäver i alfabetet från b till k.

Exempel: sökning i input

```
lain@navi:~$ ps -e | grep mullvad
1407 ?          00:00:07 mullvad-daemon
1757 ?          00:00:07 mullvad-gui
1809 ?          00:00:00 mullvad-gui
1878 ?          00:00:01 mullvad-gui
1889 ?          00:00:23 mullvad-gui
lain@navi:~$ sudo kill -9 1407
```

Figur 5: Sökning i input från annat program

I exemplet tar *grep* emot input från ett annat program för att hitta PID:en till en process med ett namn innehållande mullvad. Kommandot *ps -e* har som output alla processer som körs på systemet som hade varit för mycket att söka igenom för hand.

Exempel: sökning efter fil med uttryck

```
lain@navi:~$ grep -e [a-c].random -r .
./.thunderbird/ow7yadqn.default/ImapMail/imap.gmail-1.com/INBOX:
  |
```

Figur 6: Sökning efter filer som har linjer som matchar uttryck

grep kan söka igenom en katalog efter filer vars innehåll matchar ett uttryck

3.3 expr

Funktionalitet *expr* är ett kommandotolkverktyg som utvärderar uttryck med tal, strängar och jämförelser. För att undvika att kommandotolken utvärderar uttrycken måste speciella karaktärer utflysa med \.

Exempel: matematiskt uttryck

```
lain@navi:~$ expr \( \( 20 + 5 \) \% 23 \) \* 24
48
lain@navi:~$
```

Figur 7: Utvärdering av matematiskt uttryck

I figuren demonstreras hur aritmetik och uträkningar kan utföras.

Exempel: blandat uttryck

```
lain@navi:~$ expr \( index storbritannien brit \< 5 \) \* 10
10
lain@navi:~$
```

Figur 8: Utvärdering av blandat uttryck

I exemplet utvärderas uttryck med både matematiska tal och karaktärer. Uttrycket mäter först vart i strängen storbritanniensom britförekommer och jämför sedan om indexet är mindre än 5. Resultatet blir sant, en 1:a, vilket sedan multipliceras med 10.

Exempel: reguljära uttryck

```
lain@navi:~$ expr abbbbbbbba : '[a-z]b\+'
9
lain@navi:~$
```

Figur 9: Utvärdering av reguljärt uttryck

I figuren visar ett reguljärt uttryck som räknar hur många gånger det förekommer bokstaven b en eller flera gånger med bokstäver från a-z framför.

4 Lexikal analys

Eftersom det skulle skapas ett språk för att skanna *fak.c*, valdes att enbart det alfabet som *fak.c* använder [2]. Språket togs fram genom att först analysera *fak.c*.

pas.l användes som en mall för att utveckla språket för specifikt *fak.c*, därav de tokens som fanns men var ej nödvändiga för *fak.c* togs bort, och de som behövdes lades till.

```
#include <stdio.h>
#include <stdlib.h>

int fak(int n) {
    if (n == 0)
        return 1;
    else
        return n * fak(n - 1);
}

int main(int argc, char *argv[]) {
    int n;
    if (argc != 2) {
        fprintf(stderr, "usage %s <n> \n\n", argv[0]);
        exit(1);
    }
    n = atoi(argv[1]);
    printf("\tn = %3d n! = %3d\n", n, fak(n));
    return 0;
}
```

Figur 10: fak.c koden

För att bygga språket till fak.l användes dessa uttryck för att känna igen tokens:

#include I fak.l är *#include* ett token för import av bibliotek.

ID*.h Med det fördefinierade *ID* på rad 16 i fak.l, som säger att alla ord som är konstruerade med *ID* och slutar med *.h*

DIGIT+ inkluderar alla heltal.

if/else/return/exit Alla nyckel ord som uppstår i fak.c som existerar i programmerings språket c.

int/char Typ-deklarationerna som uppstår i fak.c.

\t|\n Token för att känna igen speciella karaktärer i en sträng sekvens.

rad 26-31 Tokens som beskriver start och slut på grupp sekvenser för dem operationer som använder sig av klamrar, måsvingar och parenteser.

rad 34-49 Beskriver tokens för diverse speciella karaktärer, deklarationer av tillstånd som separerar, operatorer för olika operationer samt deklarationer för strängar och karaktärer.

ID+ För att skapa tokens åt identifierare används ett reguljärt uttryck för alla strängar som består av enbart *ID*.

rad 41-42 För resterande text finns det uttryck som känner igen tomt utrymme och tecken som är inte inkluderat i språket.

Tillsammans beskrivs språket i filen fak.c

```
$ flex -o aut fak.l
$ flex -o aut.c fak.l
$ gcc -o aut aut.c
$ ./aut<fak.c
```

Figur 11: Bash commando för flex


```
Start of range definition: [  
An integer: 0 (0)  
End of range definition: ]  
End of domain: )  
End of statement: ;  
A keyword: exit  
Start of domain: (  
An integer: 1 (1)  
End of domain: )  
End of statement: ;  
End of statement body: }  
Identifier:  n  
An operator: =  
Identifier:  atoi  
Start of domain: (  
Identifier:  argv  
Start of range definition: [  
An integer: 1 (1)
```

Figur 12: Scanner för fak.c nr:1

5 Sammanfattning

Laborationen gick väl. Alla krav enligt specifikationerna uppfyllades. Vi har fått en större förståelse för unixverktygen som användes och hur reguljära uttryck används i vår vardag. Vi har lärt oss använda diverse verktyg som *grep* för att matcha textlinjer med reguljära uttryck, *expr* för att utvärdera uttryck för både strängar och tal och *bash* för att utnyttja shellglobbing på kommandolinjen. Vi har även lärt oss använda *flex* för att skapa språkautomater. Totalt tog det ca 15 timmar att utföra laborationen och skriva rapporten.

6 Referenser

Referenser

- [1] unix-program 'man', *man*, Linux man, 30 november 2019,
- [2] *Lexical Analysis with Flex* , Vern Paxson, Will Estes, John Millaway, 2012-07-22

7 Bilagor

```
1  /* scanner for a toy C-like language */
2  %option noyywrap
3  %{
4  /* need this for the call to atof() below */
5  #include <math.h>
6  %}
7
8  DIGIT    [0-9]
9  ID       [a-z][a-z0-9]*
10
11  %%
12
13  #include      {printf("A import keyword: %s\n", yytext );}
14  {ID}*.h      { printf( "A library: %s\n", yytext); }
15
16  {DIGIT}+     { printf( "An integer: %s (%d)\n", yytext, atoi( yytext ) ); }
17
18
19
20  if|else|return|exit { printf( "A keyword: %s\n", yytext ); }
21
22  int|char      { printf( "A type: %s\n", yytext ); }
23
24
25
26  "]"          printf( "End of range definition: %s\n", yytext );
27  "["          printf( "Start of range definition: %s\n", yytext );
28  "{"          printf( "Start of statement body: %s\n", yytext );
29  "}"          printf( "End of statement body: %s\n", yytext );
30  "("          printf( "Start of domain: %s\n", yytext );
31  ")"          printf( "End of domain: %s\n", yytext );
32
33
34  ";"          printf( "End of statement: %s\n", yytext );
35  ","          printf( "Seperator: %s\n", yytext );
36  "="|"+"|"-"|"*"|"/"|"%"|"&"|">"|"<"|"?"|"!"|"\" printf( "An operator: %s\n", yytext );
37  "\"          printf( "String indicator: %s\n", yytext );
38  "'"          printf( "Char indicator: %s\n", yytext );
39  "\\t|\\n     printf( "Special string characters: %s\n", yytext );
40  {ID}+        { printf( "Identifier: %s\n", yytext ); }
41  [ \\t\\n]+   /* eat up whitespace */
42  .            printf( "Unrecognized character: %s\n", yytext );
43
44  %%
45
46  int main(int argc, char **argv) {
47  ++argv, --argc;          /* skip over program name */
48  if (argc > 0)
49  yyin = fopen(argv[0], "r");
50  else
51  yyin = stdin;
52  yylex();
53  return 0;
54 }
```

Figur 13: fak.l språket