正则表达式及其基本实现

Cig01

Dec 14, 2014

Contents

1 基本知识

1.1 Chomsky 文法体系

根据 Chomsky 文法体系，把文法分为四种类型。

文法类别	生成的语言	对应的识别器
0 型文法	无限制语言	图灵机
1 型文法	上下文有关语言	线性有界自动机
2 型文法	上下文无关语言	下推自动机
3 型文法	正则语言	有限状态自动机

1.2 基本正则结构及扩展

1.2.1 基本结构

Kleene 在 20 世纪 50 年代提出了最基本的正则表达式，只支持三种基本结构：

1. 并 (union)：$a|b$
2. 连接 (concatenation)：ab
3. Kleene 闭包 (closure)：a^*

本文将采用这样的优先级约定：closure 优先级最高，concatenation 的优先级次之，union 的优先级最低。

1.2.2 简单的扩展

下面这些简单的扩展，很容易通过前面三种基本结构来表达。

1. +：一个或多个实例
2. ?：零个或一个实例
3. []：字符类

1.2.3 高级的扩展

现在主流的正则引擎都支持反向引用 (backreference)。

As far as the theoretical term is concerned, regular expressions with backreferences are not regular expressions. The power that backreferences add comes at great cost: in the worst case, the best known implementations require exponential search algorithms, like the one Perl uses.[1]

含有反向引用的正则表达式已经超出了通常的正则表达式的概念范畴。增加反向引用带来了很大的花费：在最坏情况下，现有已知的最好实现也需要指数级搜索算法，如 Perl 使用的那个。

2 实现正则——递归下降法

2.1 Pike 实现

Rob Pike 用递归下降法写了一个简单的正则匹配器[2]来实现 grep 工具，它支持下面特性（注：不支持 union 操作 $a|b$）：

```
c    matches any literal character c
.    matches any single character
^    matches the beginning of the input string
$    matches the end of the input string
*    matches zero or more occurrences of the previous character
```

[1] 摘自：http://swtch.com/~rsc/regexp/regexp1.html

[2] 参考：http://www.cs.princeton.edu/courses/archive/spr09/cos333/beautiful.html

Rob Pike 的实现非常精巧[3]，这两本书都引用了它：The Practice of Programming(Chapter 9), Beautiful Code(Chapter 1).

Listing 1: Rob Pike regex

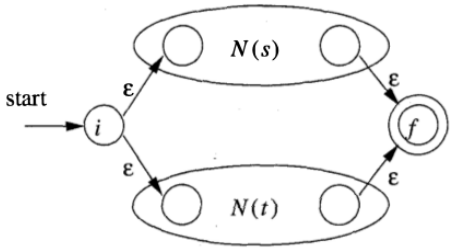
```c
#include <assert.h>

/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}

/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
    do {    /* a * matches zero or more instances */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}

int main(int argc, char **argv)
{
    assert(1 == match("a.c","xxxabcxxx"));
    assert(1 == match("^a.c$","abc"));
    assert(1 == match("^b*cde$","cde"));
    assert(1 == match("^b*cde$","bbbbbcde"));
    return 0;
}
```

[3]可从这里下载：http://cm.bell-labs.com/cm/cs/tpop/code.html

3 实现正则——有限自动机

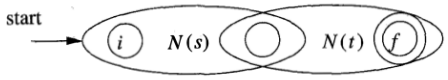
3.1 正则表达式转换为 NFA

Thompsonth 算法[4]。
正则表达式的三种基本结构：并，连接，Kleene 闭包分别对应 Figure 1,Figure 2,Figure 3所示 NFA。



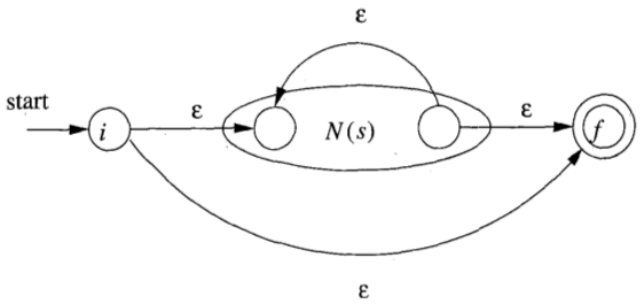
NFA for the union of two regular expressions

Figure 1: union 对应的 NFA



NFA for the concatenation of two regular expressions

Figure 2: concatenation 对应的 NFA



NFA for the closure of a regular expression

Figure 3: closure 对应的 NFA

注：Thompson 算法得到的 NFA 的有下面特点：每个节点最多两个出边，当某节点有两个出边时，这两个出边都是 ϵ 转移。
除了 Thompson 算法外，还有其它方法可以由正则表达式构造出 NFA，如 Glushkov 算法[5]。

3.1.1 Thompson 构造实例

正则表达式 $(a|b)*a$，按 Thompson 算法可求得到对应的 NFA 如 Figure 4所示。

[4]参考：《编译原理（第 2 版）》3.7 节，算法 3.23。
[5]参考："Flexible Pattern Matching in Strings - Practical On-line Search Algorithms for Texts and Biological Sequences" 5.1 section.

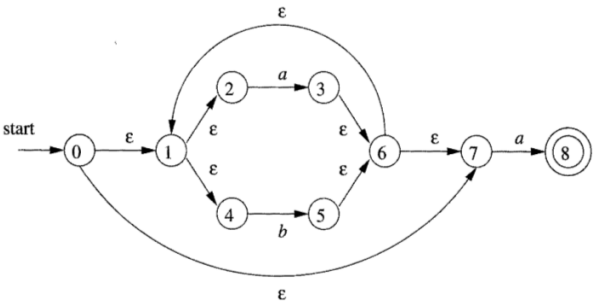


Figure 4: $(a|b)*a$ 对应 NFA

3.1.2 程序模拟 NFA

模拟 NFA 算法如 Figure 5所示[6]。

```
1)  S = ε-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)          S = ε-closure(move(S, c));
5)          c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "yes";
8)  else return "no";
```

模拟一个 NFA

Figure 5: 模拟 NFA 算法

由于 NFA 是不确定的，遇到无法确定的分支时有两种方法可以选择：一是尝试一个分支，失败后回溯到分支点再尝试另一个分支。二是并行尝试所有分支，允许状态机一次处于多个状态。Figure 5采用的是并行尝试所有分支，$\epsilon - closure$ 就是计算所有可达状态。

3.2 NFA 转换为 DFA

子集构造算法[7]。

3.2.1 最小化 DFA

分割法[8]。

3.2.2 程序模拟 DFA

比程序模拟 NFA 更简单，算法如 Figure 6所示[9]。

[6]参考：《编译原理（第 2 版）》3.7 节，算法 3.22。
[7]参考：《编译原理（第 2 版）》3.7 节，算法 3.20。
[8]参考：《编译原理（第 2 版）》3.9.6 节，算法 3.39。
[9]参考：《编译原理（第 2 版）》3.6 节，算法 3.18。

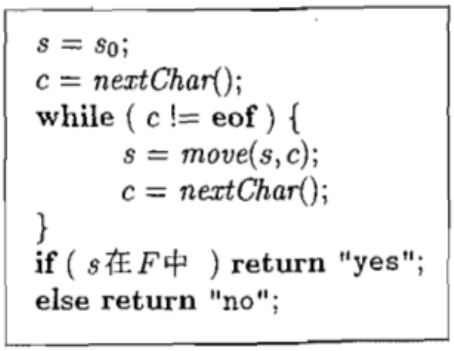


Figure 6: 模拟 DFA 算法

3.3 选择 NFA 还是 DFA

模拟 DFA 的效率比模拟 NFA 高，但构造 DFA 的也需要不少代价。那到底是选择 NFA 还是 DFA 呢？

如果字符串处理器被频繁使用，比如词法分析器，那么转换到 DFA 时付出的任何代价都是值得的。然而在另一些字符串处理应用中，例如 grep，用户指定一个正则表达式，并在一个或多个文件中搜索这个表达式所描述的模式，那么跳过构造的 DFA 步骤直接模拟 NFA 可能更加高效[10]。

3.4 Russ Cox 实现

Russ Cox 的几篇关于正则表达式的文章深入浅出[11]，且给出了 NFA 和 DFA 两种实现的完整代码，本节仅引用其 NFA 的实现代码，这份代码和后文的 Jan Burgy 字节码实现的代码都是对 Thompson 论文的重新实现。

Listing 2: Russ Cox regex(NFA)

```
/*
 * Regular expression implementation.
 * Supports only ( | ) * + ?.  No escapes.
 * Compiles to NFA and then simulates NFA
 * using Thompson's algorithm.
 *
 * See also http://swtch.com/~rsc/regexp/ and
 * Thompson, Ken.  Regular Expression Search Algorithm,
 * Communications of the ACM 11(6) (June 1968), pp. 419-422.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/*
 * Convert infix regexp re to postfix notation.
 * Insert . as explicit concatenation operator.
 * Cheesy parser, return static buffer.
 */
char*
```

[10]摘自：《编译原理（第 2 版）》3.7.5 节。

[11]参考：http://swtch.com/~rsc/regexp/

```
re2post(char *re)
{
    int nalt, natom;
    static char buf[8000];
    char *dst;
    struct {
        int nalt;
        int natom;
    } paren[100], *p;

    p = paren;
    dst = buf;
    nalt = 0;
    natom = 0;
    if(strlen(re) >= sizeof buf/2)
        return NULL;
    for(; *re; re++){
        switch(*re){
        case '(':
            if(natom > 1){
                --natom;
                *dst++ = '.';
            }
            if(p >= paren+100)
                return NULL;
            p->nalt = nalt;
            p->natom = natom;
            p++;
            nalt = 0;
            natom = 0;
            break;
        case '|':
            if(natom == 0)
                return NULL;
            while(--natom > 0)
                *dst++ = '.';
            nalt++;
            break;
        case ')':
            if(p == paren)
                return NULL;
            if(natom == 0)
                return NULL;
            while(--natom > 0)
                *dst++ = '.';
            for(; nalt > 0; nalt--)
                *dst++ = '|';
            --p;
            nalt = p->nalt;
            natom = p->natom;
            natom++;
            break;
        case '*':
        case '+':
```

```
76          case '?':
77              if(natom == 0)
78                  return NULL;
79              *dst++ = *re;
80              break;
81          default:
82              if(natom > 1){
83                  --natom;
84                  *dst++ = '.';
85              }
86              *dst++ = *re;
87              natom++;
88              break;
89          }
90      }
91      if(p != paren)
92          return NULL;
93      while(--natom > 0)
94          *dst++ = '.';
95      for(; nalt > 0; nalt--)
96          *dst++ = '|';
97      *dst = 0;
98      return buf;
99  }
100
101 /*
102  * Represents an NFA state plus zero or one or two arrows exiting.
103  * if c == Match, no arrows out; matching state.
104  * If c == Split, unlabeled arrows to out and out1 (if != NULL).
105  * If c < 256, labeled arrow with character c to out.
106  */
107 enum
108 {
109     Match = 256,
110     Split = 257
111 };
112 typedef struct State State;
113 struct State
114 {
115     int c;
116     State *out;
117     State *out1;
118     int lastlist;
119 };
120 State matchstate = { Match };   /* matching state */
121 int nstate;
122
123 /* Allocate and initialize State */
124 State*
125 state(int c, State *out, State *out1)
126 {
127     State *s;
128
129     nstate++;
```

```
130     s = malloc(sizeof *s);
131     s->lastlist = 0;
132     s->c = c;
133     s->out = out;
134     s->out1 = out1;
135     return s;
136 }
137
138 /*
139  * A partially built NFA without the matching state filled in.
140  * Frag.start points at the start state.
141  * Frag.out is a list of places that need to be set to the
142  * next state for this fragment.
143  */
144 typedef struct Frag Frag;
145 typedef union Ptrlist Ptrlist;
146 struct Frag
147 {
148     State *start;
149     Ptrlist *out;
150 };
151
152 /* Initialize Frag struct. */
153 Frag
154 frag(State *start, Ptrlist *out)
155 {
156     Frag n = { start, out };
157     return n;
158 }
159
160 /*
161  * Since the out pointers in the list are always
162  * uninitialized, we use the pointers themselves
163  * as storage for the Ptrlists.
164  */
165 union Ptrlist
166 {
167     Ptrlist *next;
168     State *s;
169 };
170
171 /* Create singleton list containing just outp. */
172 Ptrlist*
173 list1(State **outp)
174 {
175     Ptrlist *l;
176
177     l = (Ptrlist*)outp;
178     l->next = NULL;
179     return l;
180 }
181
182 /* Patch the list of states at out to point to start. */
183 void
```

```
184 patch(Ptrlist *l, State *s)
185 {
186     Ptrlist *next;
187 
188     for(; l; l=next){
189         next = l->next;
190         l->s = s;
191     }
192 }
193 
194 /* Join the two lists l1 and l2, returning the combination. */
195 Ptrlist*
196 append(Ptrlist *l1, Ptrlist *l2)
197 {
198     Ptrlist *oldl1;
199 
200     oldl1 = l1;
201     while(l1->next)
202         l1 = l1->next;
203     l1->next = l2;
204     return oldl1;
205 }
206 
207 /*
208  * Convert postfix regular expression to NFA.
209  * Return start state.
210  */
211 State*
212 post2nfa(char *postfix)
213 {
214     char *p;
215     Frag stack[1000], *stackp, e1, e2, e;
216     State *s;
217 
218     // fprintf(stderr, "postfix: %s\n", postfix);
219 
220     if(postfix == NULL)
221         return NULL;
222 
223     #define push(s) *stackp++ = s
224     #define pop() *--stackp
225 
226     stackp = stack;
227     for(p=postfix; *p; p++){
228         switch(*p){
229         default:
230             s = state(*p, NULL, NULL);
231             push(frag(s, list1(&s->out)));
232             break;
233         case '.':   /* catenate */
234             e2 = pop();
235             e1 = pop();
236             patch(e1.out, e2.start);
237             push(frag(e1.start, e2.out));
```

```
238             break;
239         case '|':   /* alternate */
240             e2 = pop();
241             e1 = pop();
242             s = state(Split, e1.start, e2.start);
243             push(frag(s, append(e1.out, e2.out)));
244             break;
245         case '?':   /* zero or one */
246             e = pop();
247             s = state(Split, e.start, NULL);
248             push(frag(s, append(e.out, list1(&s->out1))));
249             break;
250         case '*':   /* zero or more */
251             e = pop();
252             s = state(Split, e.start, NULL);
253             patch(e.out, s);
254             push(frag(s, list1(&s->out1)));
255             break;
256         case '+':   /* one or more */
257             e = pop();
258             s = state(Split, e.start, NULL);
259             patch(e.out, s);
260             push(frag(e.start, list1(&s->out1)));
261             break;
262         }
263     }
264
265     e = pop();
266     if(stackp != stack)
267         return NULL;
268
269     patch(e.out, &matchstate);
270     return e.start;
271 #undef pop
272 #undef push
273 }
274
275 typedef struct List List;
276 struct List
277 {
278     State **s;
279     int n;
280 };
281 List l1, l2;
282 static int listid;
283
284 void addstate(List*, State*);
285 void step(List*, int, List*);
286
287 /* Compute initial state list */
288 List*
289 startlist(State *start, List *l)
290 {
291     l->n = 0;
```

```
292     listid++;
293     addstate(l, start);
294     return l;
295 }
296
297 /* Check whether state list contains a match. */
298 int
299 ismatch(List *l)
300 {
301     int i;
302
303     for(i=0; i<l->n; i++)
304         if(l->s[i] == &matchstate)
305             return 1;
306     return 0;
307 }
308
309 /* Add s to l, following unlabeled arrows. */
310 void
311 addstate(List *l, State *s)
312 {
313     if(s == NULL || s->lastlist == listid)
314         return;
315     s->lastlist = listid;
316     if(s->c == Split){
317         /* follow unlabeled arrows */
318         addstate(l, s->out);
319         addstate(l, s->out1);
320         return;
321     }
322     l->s[l->n++] = s;
323 }
324
325 /*
326  * Step the NFA from the states in clist
327  * past the character c,
328  * to create next NFA state set nlist.
329  */
330 void
331 step(List *clist, int c, List *nlist)
332 {
333     int i;
334     State *s;
335
336     listid++;
337     nlist->n = 0;
338     for(i=0; i<clist->n; i++){
339         s = clist->s[i];
340         if(s->c == c)
341             addstate(nlist, s->out);
342     }
343 }
344
345 /* Run NFA to determine whether it matches s. */
```

```
346 int
347 match(State *start, char *s)
348 {
349     int i, c;
350     List *clist, *nlist, *t;
351 
352     clist = startlist(start, &l1);
353     nlist = &l2;
354     for(; *s; s++){
355         c = *s & 0xFF;
356         step(clist, c, nlist);
357         t = clist; clist = nlist; nlist = t;    /* swap clist, nlist */
358     }
359     return ismatch(clist);
360 }
361 
362 int
363 main(int argc, char **argv)
364 {
365     int i;
366     char *post;
367     State *start;
368 
369     if(argc < 3){
370         fprintf(stderr, "usage: nfa regexp string...\n");
371         return 1;
372     }
373 
374     post = re2post(argv[1]);
375     if(post == NULL){
376         fprintf(stderr, "bad regexp %s\n", argv[1]);
377         return 1;
378     }
379 
380     start = post2nfa(post);
381     if(start == NULL){
382         fprintf(stderr, "error in post2nfa %s\n", post);
383         return 1;
384     }
385 
386     l1.s = malloc(nstate*sizeof l1.s[0]);
387     l2.s = malloc(nstate*sizeof l2.s[0]);
388     for(i=2; i<argc; i++)
389         if(match(start, argv[i]))
390             printf("%s\n", argv[i]);
391     return 0;
392 }
```

4 实现正则——用虚拟机模拟自动机

模拟自动机为什么要引入虚拟机？Russ Cox 认为用虚拟机实现更方便，能轻松地增加新功能。Viewing regular expression matching as executing a special machine makes it possible to add new features just by

adding (and implementing!) new machine instructions.[12]

1968 年 Ken Thompson 在其论文中[13]把正则表达式编译为 IBM 7094 的机器代码，然后执行机器代码完成正则匹配。

4.1 Thompson 论文的字节码实现及其分析

Jan Burgy 分别用字节码和 x86 汇编[14]对 Thompson 的论文进行了重新实现。

4.1.1 Jan Burgy 字节码实现

下面引用 Jan Burgy 的字节码实现代码[15]，这份源码有个 bug：正则表达式 $ab*$ 能匹配字符串 ab 但不能匹配字符串 a 和 abb，问题出在模拟执行虚拟机指令的函数 execute 中，下面引用的源码对这个函数进行了修改以解决这个 bug。

Listing 3: Transliteration of Thompson's code for bytecode(Jan Burgy)

```
/*
 * Bytecode machine implementation of Thompson's
 * on-the-fly regular expression compiler.
 *
 * See also Thompson, Ken.  Regular Expression Search Algorithm,
 * Communications of the ACM 11(6) (June 1968), pp. 419-422.
 *
 * Copyright (c) 2004 Jan Burgy.
 * Can be distributed under the MIT license, see bottom of file.
 */

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <assert.h>
enum    {
    STOP,
    JUMP,
    MATCH,
    BRANCH,
    LPAREN = CHAR_MAX + 1,
    RPAREN,         /* This should  */
    ALTERN,         /* reflect the  */
    CONCAT,         /* precedence   */
    KLEENE          /* rules!   */
};

unsigned char *prepare (const char *src)
{
    unsigned char   escape[CHAR_MAX + 1] = "";
    unsigned char   *dest = malloc (2 * (strlen (src) + 1));
    int     c, i, j = 0, concat = 0, nparen = 0;
```

[12]摘自：http://swtch.com/~rsc/regexp/regexp2.html

[13]参考：http://doi.acm.org/10.1145/363347.363387

[14]源码：http://swtch.com/~rsc/regexp/regexp-x86.c.txt

[15]源码：http://swtch.com/~rsc/regexp/regexp-bytecode.c.txt，注：这份源码用 gcc 4.9 编译后运行有问题，原因是在函数 compile 中类似这样的代码code[pc++] = assemble(JUMP, pc + 1);的行为在 gcc 中是没有定义的，用选项-Wsequence-point 编译时会提示 warning(参见：https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#Warning-Options)。修改为code[pc] = assemble(JUMP, pc + 1);pc++后可正常运行。

```
34
35      escape['a'] = '\a';
36      escape['b'] = '\b';
37      escape['f'] = '\f';
38      escape['n'] = '\n';
39      escape['r'] = '\r';
40      escape['t'] = '\t';
41      escape['v'] = '\v';
42      for (i = 0; (c = "\"()*\\|"[i]); i++)
43          escape[c] = c;
44
45      for (i = 0; (c = src[i]); i++) {
46
47          switch (c) {
48
49              case '(':
50                  dest[j++] = LPAREN;
51                  concat = 0;
52                  nparen++;
53                  continue;
54              case ')':
55                  dest[j++] = RPAREN;
56                  nparen--;
57                  break;
58              case '*':
59                  dest[j++] = KLEENE;
60                  break;
61              case '|':
62                  dest[j++] = ALTERN;
63                  concat = 0;
64                  continue;
65              case '\\':
66                  c = escape[(int)src[i + 1]];
67                  c ? i++ : (c = '\\');
68              default:
69                  if (concat)
70                      dest[j++] = CONCAT;
71                  dest[j++] = c;
72
73          }
74          concat = 1;
75          if (nparen < 0)
76              printf ("unbalanced␣parentheses\n");
77
78      }
79      dest[j++] = RPAREN;
80      dest[j++] = '\0';
81
82      return  dest;
83  }
84
85  unsigned char *convert (const char *src)
86  {
87      unsigned char   stack[BUFSIZ] = "";
```

```
88      unsigned char   *dest = prepare (src);
89      int     c, i, j = 0, top = 0;
90
91      stack[top++] = LPAREN;
92      for (i = 0; (c = dest[i]); i++) {
93
94          switch (c) {
95
96              case LPAREN:
97                  stack[top++] = c;
98                  break;
99
100             case RPAREN:
101                 while (c <= stack[top - 1])
102                     dest[j++] = stack[--top];
103                 --top;  /* discard LPAREN */
104                 break;
105
106             case ALTERN:
107             case CONCAT:
108             case KLEENE:
109                 while (c <= stack[top - 1])
110                     dest[j++] = stack[--top];
111                 stack[top++] = c;
112                 break;
113
114             default:
115                 dest[j++] = c;
116                 break;
117
118         }
119
120     }
121     dest[j++] = '\0';
122
123     return  dest;
124 }
125
126 struct  instr   {
127     short   operand;
128     short   address;
129 };
130
131 struct instr assemble (short operand, short address)
132 {
133     struct  instr   this;
134
135     this.operand = operand;
136     this.address = address;
137     return  this;
138 }
139
140 size_t memlen (const unsigned char *s)
141 {
```

```
142     const unsigned char     *p = s;
143
144     while (*p)
145         p++;
146     return  p - s;
147 }
148
149 struct instr *compile (const unsigned char *src)
150 {
151     int     i, c, pc = 0, top = 0;
152     int     stack[BUFSIZ];
153     struct  instr   *code = malloc (5 * memlen (src) * sizeof *code / 2);
154
155     for (i = 0; (c = src[i]); i++) {
156
157         switch (c) {
158
159             default:
160                 stack[top++] = pc;
161                 code[pc] = assemble (JUMP, pc + 1); pc++;
162                 code[pc++] = assemble (MATCH, c);
163                 break;
164
165             case CONCAT:
166                 --top;
167                 break;
168
169             case KLEENE:
170                 code[pc++] = assemble (BRANCH, '*');
171                 code[pc++] = code[stack[top - 1]];
172                 code[stack[top - 1]] = assemble (JUMP, pc - 2);
173                 break;
174
175             case ALTERN:
176                 code[pc] = assemble (JUMP, pc + 4); pc++;
177                 code[pc++] = assemble (BRANCH, '|');
178                 code[pc++] = code[stack[top - 1]];
179                 code[pc++] = code[stack[top - 2]];
180                 code[stack[top - 2]] = assemble (JUMP, pc - 3);
181                 code[stack[top - 1]] = assemble (JUMP, pc);
182                 --top;
183                 break;
184
185         }
186
187     }
188
189     code[pc] = assemble (STOP, pc);
190
191     return  code;
192 }
193
194 struct instr *study (const char *re)
195 {
```

```
196     unsigned char   *p = convert (re);
197     struct  instr   *q = compile (p);
198 
199     if (p) free (p), p = NULL;
200     return  q;
201 }
202 
203 void dump_code (struct instr *code)
204 {
205     int     i, op;
206     char    *str[] = {"STOP", "JUMP", "MATCH", "BRANCH"};
207 
208     for (i = 0; (op = code[i].operand); i++)
209         printf (op == JUMP ? "%2d:␣%s\t%3d\n" : "%2d:␣%s\t'%c'\n",
210                 i, str[op], code[i].address);
211 }
212 
213 int execute (struct instr *code, const char *src)
214 {
215     short   i = 0, c = src[i++], pc = 0;
216     short   clist[BUFSIZ], cnode = 0, shift = 0;
217     short   nlist[BUFSIZ], nnode = 0;
218     /* clist is the current set of states that the NFA is in,
219      * nlist is the next set of states that the NFA will be in. */
220 
221     while (c) {
222 
223         switch (code[pc].operand) {
224 
225             case STOP:
226                 break;
227 
228             case JUMP:
229                 pc = code[pc].address;
230                 continue;
231 
232             case MATCH:
233                 if (c == code[pc].address)
234                     nlist[nnode++] = pc + 1;
235                 break;
236 
237             case BRANCH:
238                 clist[cnode++] = pc + 1;
239                 pc = pc + 2;
240                 continue;
241 
242         }
243 
244         if (shift == cnode) {
245             if (!nnode) return 0;
246             shift = cnode = 0;  /* clear clist */
247             while (nnode > 0)  /* move all states from nlist into clist */
248                 clist[cnode++] = nlist[--nnode];
249             c = src[i++];
```

```
250         }
251         pc = clist[shift++];
252 
253     }
254 
255     /* is any of the current states final? */
256     while (shift <= cnode) {
257         switch (code[pc].operand) {
258             case STOP:
259                 return 1;
260             case JUMP:
261                 pc = code[pc].address;
262                 continue;
263             case MATCH:
264                 break;
265             case BRANCH:
266                 clist[cnode++] = pc + 1;
267                 pc = pc + 2;
268                 continue;
269         }
270         pc = clist[shift++];
271     }
272 
273     return  0;
274 }
275 
276 int main (void)
277 {
278     short   i;
279     struct  {
280         char    *re;
281         char    *s;
282         int     match;
283     } test[] = {
284         { "abcdefg",    "abcdefg",    1 },
285         { "ab*",        "a",          1 },
286         { "ab*",        "ab",         1 },
287         { "ab*",        "abb",        1 },
288         { "ab*",        "abc",        0 },
289         { "(a|b)*a",    "a",          1 },
290         { "(a|b)*a",    "abababababab", 0 },
291         { "(a|b)*a",    "aaaaaaaaba", 1 },
292         { "(a|b)*a",    "aaaaaabac",  0 },
293         { "a(b|c)*d",   "abccbcccd",  1 },
294         { "a(b|c)*d",   "abccbcccde", 0 },
295         { NULL,         NULL,         1 }
296     };
297 
298     for (i = 0; test[i].re; i++) {
299 
300         struct  instr   *this = study (test[i].re);
301 
302         printf ("%s␣%s␣/%s/\n",
303                 test[i].s,
```

```
304                 execute (this, test[i].s) ? "~" : "!~",
305                 test[i].re);
306         assert(execute(this, test[i].s) == test[i].match);
307
308         if (this) free (this), this = NULL;
309     }
310
311     return  EXIT_SUCCESS;
312 }
```

4.1.2 虚拟机指令

用一句话概括上面的实现：先把中缀形式的正则表达式转换为其后缀形式，再把后缀形式的正则表达式编译为虚拟机指令，最后模拟虚拟机的执行来实现正则匹配。

上面实现中定义了一个虚拟机，仅包含下面 4 条指令：

```
JUMP X    跳到地址X处执行
MATCH C    匹配字符C
BRANCH    同时执行接下来的两条指令
STOP    结束执行
```

4.1.3 编译 **concatenation** 为虚拟机指令

正则表达式 *abc* 会编译为下面指令：

```
0: JUMP      1      ;由源码第161行产生
1: MATCH   'a'      ;由源码第162行产生
2: JUMP      3      ;由源码第161行产生
3: MATCH   'b'      ;由源码第162行产生
4: JUMP      5      ;由源码第161行产生
5: MATCH   'c'      ;由源码第162行产生
6: STOP      6
```

首先，*abc* 会在 convert 函数中转换为后缀形式 $ab[C]c[C]$，其中 $[C]$ 表示 concatenation，然后在 compile 函数中转换为上面的虚拟机指令。

Listing 4: Code snippnet(concatenation) in function compile(Jan Burgy)

```
159             default:
160                 stack[top++] = pc;
161                 code[pc] = assemble (JUMP, pc + 1); pc++;
162                 code[pc++] = assemble (MATCH, c);
163                 break;
164
165             case CONCAT:
166                 --top;
167                 break;
```

说明：

- JUMP 指令在仅有 concatenation 的正则表达式中是多余的，它们对于编译 closure 和 union 带来了方便。
- 函数 compile 中用栈 stack 保存着已经参加过转换的 “正则片断”，这些 “正则片断” 在编译 closure 和 union 时会用到。

4.1.4 编译 **closure** 为虚拟机指令

正则表达式 $ab*$ 会编译为下面指令：

```
0: JUMP      1
1: MATCH   'a'
2: JUMP      4       ;在源码172行中更新了此行
3: MATCH   'b'
4: BRANCH  '*'       ;由源码第170行产生
5: JUMP      3       ;由源码第171行产生
6: STOP      6
```

首先，$ab*$ 会在 convert 函数中转换为后缀形式 $ab[K][C]$，其中 $[K]$ 表示 Kleene 闭包，然后在 compile 函数中把后缀形式的正则表达式转换为上面的虚拟机指令。

Listing 5: Code snippnet(closure) in function compile(Jan Burgy)

```
169             case KLEENE:
170                 code[pc++] = assemble (BRANCH, '*');
171                 code[pc++] = code[stack[top - 1]];
172                 code[stack[top - 1]] = assemble (JUMP, pc - 2);
173                 break;
```

4.1.5 编译 **union** 为虚拟机指令

正则表达式 $abc|de$ 会编译为下面指令：

```
 0: JUMP     11       ;在源码180行中更新了此行
 1: MATCH   'a'
 2: JUMP      3
 3: MATCH   'b'
 4: JUMP      5
 5: MATCH   'c'
 6: JUMP     14       ;在源码181行中更新了此行
 7: MATCH   'd'
 8: JUMP      9
 9: MATCH   'e'
10: JUMP     14       ;由源码第176行产生
11: BRANCH  '|'       ;由源码第177行产生
12: JUMP      7       ;由源码第178行产生
13: JUMP      1       ;由源码第179行产生
14: STOP     14
```

首先，$abc|de$ 会在 convert 函数中转换为后缀形式 $ab[C]c[C]de[C][U]$，其中 $[U]$ 表示 Union，然后在 compile 函数中把后缀形式的正则表达式转换为上面的虚拟机指令。

Listing 6: Code snippnet(union) in function compile(Jan Burgy)

```
175             case ALTERN:
176                 code[pc] = assemble (JUMP, pc + 4); pc++;
177                 code[pc++] = assemble (BRANCH, '|');
178                 code[pc++] = code[stack[top - 1]];
179                 code[pc++] = code[stack[top - 2]];
180                 code[stack[top - 2]] = assemble (JUMP, pc - 3);
181                 code[stack[top - 1]] = assemble (JUMP, pc);
182                 --top;
183                 break;
```

4.1.6 模拟执行虚拟机指令（递归）

前面生成的虚拟机指令可看作是 NFA 的表达形式，函数 execute 对 NFA 进行了模拟执行，其中使用了两个数组：clist 中保存着当前 NFA 所在的状态集合，nlist 保存着下一步将要到达的状态集合，算法描述可参见 Figure 5。

除上面的算法外，用递归函数也能方便地对其进行模拟执行，程序简单易懂。

Listing 7: A recursive backtracking implementation for Jan Burgy bytecode

```
int recursive (struct instr *base, struct instr *pc, const char *src)
{
    switch (pc->operand) {
    case STOP:
        if (*src == '\0') return 1;
        else return 0;
    case JUMP:
        return recursive(base, base + (pc->address), src);
    case MATCH:
        if (pc->address != *src) return 0;
        else return recursive(base, pc + 1, src + 1);
    case BRANCH:
        return recursive(base, pc + 1, src) || recursive(base, pc + 2, src);
    }

    assert(0);  /* not reached */
    return -1;
}

int execute (struct instr *code, const char *src)
{
    return recursive(code, code, src);
}
```

但这种递归的实现可能多次重复扫描相同的子字符串，Thompson 论文中给出的算法更高效。

4.1.7 支持更多虚拟机指令

重复零次或一次 $a?$，重复一次或多次 $a+$，等正则语法通过扩展虚拟机指令容易实现。Pike 定义的虚拟机实现了跟踪子匹配（如跟踪正则表达式 $a(c|d)$ 中 $(c|d)$ 匹配的是哪个字符串）[16]。

5 如何实现反向引用

With backreferences, (.*)\1 matches catcat or dogdog but not dogcat. Matching with backreferences takes exponential time in all current implementations. Matching with backreferences is NP-complete, a fact you will probably prove![17]

基于有限自动机技术的正则实现很高效，但这种技术却很难实现正则表达式反向引用的高级特性。Perl, PCRE, Python, Ruby, Java 以及其他很多语言的正则表达式实现是基于递归下降的（可能要回溯），实现反向引用相对比较容易。

[16] 参考：http://swtch.com/~rsc/regexp/regexp2.html

[17] 摘自：http://swtch.com/~rsc/talks/regexp.pdf