```c
/*
 * Regular expression implementation.
 * Supports only ( | ) * + ?.  No escapes.
 * Compiles to NFA and then simulates NFA
 * using Thompson's algorithm.
 * Caches steps of Thompson's algorithm to
 * build DFA on the fly, as in Aho's egrep.
 *
 * 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.
 *
 * Copyright (c) 2007 Russ Cox.
 * Can be distributed under the MIT license, see bottom of file.
 */
#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*
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 '+':
                case '?':
                        if(natom == 0)
                                return NULL;
                        *dst++ = *re;
                        break;
                default:
                        if(natom > 1){
                                --natom;
                                *dst++ = '.';
                        }
                        *dst++ = *re;
                        natom++;
                        break;
                }
        }
        if(p != paren)
                return NULL;
        while(--natom > 0)
                *dst++ = '.';
        for(; nalt > 0; nalt--)
                *dst++ = '|';
        *dst = 0;
        return buf;
}

/*
 * Represents an NFA state plus zero or one or two arrows exiting.
 * if c == Match, no arrows out; matching state.
 * If c == Split, unlabeled arrows to out and out1 (if != NULL).
 * If c < 256, labeled arrow with character c to out.
 */
enum
{
        Match = 256,
        Split = 257
};
typedef struct State State;
struct State
{
        int c;
        State *out;
        State *out1;
        int lastlist;
};
State matchstate = { Match };   /* matching state */
int nstate;

/* Allocate and initialize State */
State*
```

```
state(int c, State *out, State *out1)
{
	State *s;

	nstate++;
	s = malloc(sizeof *s);
	s->lastlist = 0;
	s->c = c;
	s->out = out;
	s->out1 = out1;
	return s;
}

/*
 * A partially built NFA without the matching state filled in.
 * Frag.start points at the start state.
 * Frag.out is a list of places that need to be set to the
 * next state for this fragment.
 */
typedef struct Frag Frag;
typedef union Ptrlist Ptrlist;
struct Frag
{
	State *start;
	Ptrlist *out;
};

/* Initialize Frag struct. */
Frag
frag(State *start, Ptrlist *out)
{
	Frag n = { start, out };
	return n;
}

/*
 * Since the out pointers in the list are always
 * uninitialized, we use the pointers themselves
 * as storage for the Ptrlists.
 */
union Ptrlist
{
	Ptrlist *next;
	State *s;
};

/* Create singleton list containing just outp. */
Ptrlist*
list1(State **outp)
{
	Ptrlist *l;

	l = (Ptrlist*)outp;
	l->next = NULL;
	return l;
}

/* Patch the list of states at out to point to start. */
void
patch(Ptrlist *l, State *s)
{
	Ptrlist *next;

	for(; l; l=next){
		next = l->next;
		l->s = s;
	}
}

/* Join the two lists l1 and l2, returning the combination. */
Ptrlist*
append(Ptrlist *l1, Ptrlist *l2)
{
	Ptrlist *oldl1;

	oldl1 = l1;
	while(l1->next)
		l1 = l1->next;
	l1->next = l2;
	return oldl1;
}

/*
 * Convert postfix regular expression to NFA.
 * Return start state.
 */
State*
post2nfa(char *postfix)
{
	char *p;
	Frag stack[1000], *stackp, e1, e2, e;
	State *s;

	// fprintf(stderr, "postfix: %s\n", postfix);

	if(postfix == NULL)
		return NULL;

	#define push(s) *stackp++ = s
	#define pop() *--stackp

	stackp = stack;
	for(p=postfix; *p; p++){
		switch(*p){
		default:
			s = state(*p, NULL, NULL);
			push(frag(s, list1(&s->out)));
			break;
		case '.':	/* catenate */
			e2 = pop();
			e1 = pop();
			patch(e1.out, e2.start);
			push(frag(e1.start, e2.out));
			break;
		case '|':	/* alternate */
			e2 = pop();
			e1 = pop();
			s = state(Split, e1.start, e2.start);
			push(frag(s, append(e1.out, e2.out)));
			break;
		case '?':	/* zero or one */
			e = pop();
			s = state(Split, e.start, NULL);
			push(frag(s, append(e.out, list1(&s->out1))));
			break;
		case '*':	/* zero or more */
			e = pop();
			s = state(Split, e.start, NULL);
			patch(e.out, s);
			push(frag(s, list1(&s->out1)));
```

```
				break;
			case '+':	/* one or more */
				e = pop();
				s = state(Split, e.start, NULL);
				patch(e.out, s);
				push(frag(e.start, list1(&s->out1)));
				break;
			}
	}

	e = pop();
	if(stackp != stack)
		return NULL;

	patch(e.out, &matchstate);
	return e.start;
#undef pop
#undef push
}

typedef struct List List;
struct List
{
	State **s;
	int n;
};
List l1, l2;
static int listid;

void addstate(List*, State*);
void step(List*, int, List*);

/* Compute initial state list */
List*
startlist(State *start, List *l)
{
	l->n = 0;
	listid++;
	addstate(l, start);
	return l;
}

/* Check whether state list contains a match. */
int
ismatch(List *l)
{
	int i;

	for(i=0; i<l->n; i++)
		if(l->s[i] == &matchstate)
			return 1;
	return 0;
}

/* Add s to l, following unlabeled arrows. */
void
addstate(List *l, State *s)
{
	if(s == NULL || s->lastlist == listid)
		return;
	s->lastlist = listid;
	if(s->c == Split){
		/* follow unlabeled arrows */
		addstate(l, s->out);
		addstate(l, s->out1);
		return;
	}
	l->s[l->n++] = s;
}

/*
 * Step the NFA from the states in clist
 * past the character c,
 * to create next NFA state set nlist.
 */
void
step(List *clist, int c, List *nlist)
{
	int i;
	State *s;

	listid++;
	nlist->n = 0;
	for(i=0; i<clist->n; i++){
		s = clist->s[i];
		if(s->c == c)
			addstate(nlist, s->out);
	}
}

/*
 * Represents a DFA state: a cached NFA state list.
 */
typedef struct DState DState;
struct DState
{
	List l;
	DState *next[256];
	DState *left;
	DState *right;
};

/* Compare lists: first by length, then by members. */
static int
listcmp(List *l1, List *l2)
{
	int i;

	if(l1->n < l2->n)
		return -1;
	if(l1->n > l2->n)
		return 1;
	for(i=0; i<l1->n; i++)
		if(l1->s[i] < l2->s[i])
			return -1;
		else if(l1->s[i] > l2->s[i])
			return 1;
	return 0;
}

/* Compare pointers by address. */
static int
ptrcmp(const void *a, const void *b)
{
	if(a < b)
		return -1;
	if(a > b)
		return 1;
	return 0;
}
```

```
/*
 * Return the cached DState for list l,
 * creating a new one if needed.
 */
DState *alldstates;
DState*
dstate(List *l)
{
	int i;
	DState **dp, *d;

	qsort(l->s, l->n, sizeof l->s[0], ptrcmp);
	dp = &alldstates;
	while((d = *dp) != NULL){
		i = listcmp(l, &d->l);
		if(i < 0)
			dp = &d->left;
		else if(i > 0)
			dp = &d->right;
		else
			return d;
	}

	d = malloc(sizeof *d + l->n*sizeof l->s[0]);
	memset(d, 0, sizeof *d);
	d->l.s = (State**)(d+1);
	memmove(d->l.s, l->s, l->n*sizeof l->s[0]);
	d->l.n = l->n;
	*dp = d;
	return d;
}

void
startnfa(State *start, List *l)
{
	l->n = 0;
	listid++;
	addstate(l, start);
}

DState*
startdstate(State *start)
{
	return dstate(startlist(start, &l1));
}

DState*
nextstate(DState *d, int c)
{
	step(&d->l, c, &l1);
	return d->next[c] = dstate(&l1);
}

/* Run DFA to determine whether it matches s. */
int
match(DState *start, char *s)
{
	DState *d, *next;
	int c, i;

	d = start;
	for(; *s; s++){
		c = *s & 0xFF;
		if((next = d->next[c]) == NULL)
```

```
				next = nextstate(d, c);
			d = next;
		}
		return ismatch(&d->l);
}

int
main(int argc, char **argv)
{
	int i;
	char *post;
	State *start;

	if(argc < 3){
		fprintf(stderr, "usage: nfa regexp string...\n");
		return 1;
	}

	post = re2post(argv[1]);
	if(post == NULL){
		fprintf(stderr, "bad regexp %s\n", argv[1]);
		return 1;
	}

	start = post2nfa(post);
	if(start == NULL){
		fprintf(stderr, "error in post2nfa %s\n", post);
		return 1;
	}

	l1.s = malloc(nstate*sizeof l1.s[0]);
	l2.s = malloc(nstate*sizeof l2.s[0]);
	for(i=2; i<argc; i++)
		if(match(startdstate(start), argv[i]))
			printf("%s\n", argv[i]);
	return 0;
}
```