Linear form of the extended regular expressions

WANG Hanfei

October 20, 2019

Contents

2017 级弘毅班编译原理课程设计第 4 次编程作业 (Linear form of the extended regular expression)

We will use the recursive descent parser convert the extended regular expression to Abstract Syntactic Tree (AST). and then convert the AST to DFA by the derivative (see partial_derivative.pdf) in the next mission.

1 Extended Regular Expression (EREGEX)

3 binary operators added:

1. Difference: `e1 - e2`, `L(e1 - e2) = L(e1) - L(e2)`.
2. Interleave product: `e1 ^ e2`,

   ```
   a ^ b = a b | b a
   ab ^ ba = (a ^ b) (a ^ b)
   a ^ b ^ c = a b c | a c b | b a c | b c a | c a b | c b a
   ```

3. Intersection: `e1 & e2`, `L(e1 & e2) = L(e1)` ∩ `L(e2)`.

the order of precedence from low to high: `|`, `-`, `^`, `&`, `concat`, `*`.

1.1 Grammar of EREGEX

```
reg -> term_or reg'
reg' -> '|' term_or reg' | epsilon

term_or -> term_diff term_or'
term_or' -> '-' term_alt term_or' | epsilon

term_alt -> term_and term_alt'
term_alt' -> '^' term_and term_alt'

term_and -> term term_and'
term_and' -> '&' term term_and' | epsilon

term -> kleene term'
term' -> kleene term' | epsilon

kleene -> fac kleene'
kleene' -> * kleene' | epsilon

fac -> ALPHA | '(' reg ')'
```

the recursive descent parser functions of `term_xxx` and `term_xxx'` can be unified as:

```
expr(op1) -> expr(op2) expr1(op1)
expr1(op1) -> op2 expr(op2) expr1(op1)
           | epsilon

where op1 = |, op2 = -;
      op1 = -, op2 = ^;
      op1 = ^, op2 = &;
      op1 = &, op2 = Seq;
```

so (see `parser.c`)

```
AST_PTR expr(Kind op)
{
  AST_PTR left;
  switch (op) {
  case Or: left = expr(Diff);
    return expr1(Or, left);
  case Diff: left = expr(Alt);
    return expr1(Diff, left);
  case Alt: left = expr(And);
```

```
    return expr1(Alt, left);
  default:  left = term();
    return expr1(And, left);
  }
}

AST_PTR expr1(Kind op, AST_PTR left)
{
  AST_PTR right, tmp;
  char op_ch;
  switch (op) {
  case Or: op_ch = '|'; break;
  case Diff: op_ch = '-'; break;
  case Alt: op_ch = '^'; break;
  default: op_ch = '&';
  }
  if (*current == op_ch ) {
    next_token ();
    right = expr(op);
    tmp = arrangeOpNode(op, left, right);
    return expr1(op, tmp);
  } else
    return left;
}
```

1.2 Structure of AST

See `ast.h` in detail.

```
typedef enum { Or = 1, Diff = 2, Alt = 3, And = 4, Seq = 5,
               Star = 6, Alpha = 7, Epsilon = 8, Empty = 9} Kind;
/* in order of increased precedence */

typedef struct ast {
  Kind op;
  struct ast *lchild, *rchild;
  int hash;
  int nullable; /* = 1, if E is nullable */
  char *exp_string;  /* mostly simplified exp of E */
  int state;   /* for further use!!!
                  state number. trap state is 0,
                  the original exp is 1 */
  LF_PTR lf; /* for further use!!!
                linear form of NFA */
}  AST;
```

1.3 Symbol table

Every parsed subexpression will store in the symbol table for further uses. See `ast.h` in detail.

```
typedef struct exptab {
  struct exptab *next; /* for collision */
  AST_PTR exp;
} *EXPTAB; /* for hash table of expressions */

#define HASHSIZE  8011

/* defined in ast.c */
EXPTAB exptab[HASHSIZE] = {NULL}; /* symbol table */
```

the key is the mostly simplified expression string stored in struct `ast.exp_string`.
the hash function is (in `ast.c`):

```
int hash(char *s)
{
  unsigned int hv = 7, len = strlen(s);
  for (int i = 0; i < len; i++) {
    hv = hv*31 + s[i];
  }
  return (int) (hv % HASHSIZE) ;
}
```

each time a subexpression generated in parsing time, we will check if it is ready in `exptab` by

```
AST_PTR lookup(char *exp_string)
{
  int hv = hash(exp_string);

  EXPTAB t = exptab[hv];

  if (t == NULL) return NULL;

  while (t != NULL) {
    if (strcmp(exp_string, t -> exp -> exp_string) == 0) {
      break;
    }
    t = t -> next;
  }
  if (t == NULL) return NULL;
  return t -> exp;
}
```

if `lookup()` returns `NULL`, it will be stored in `exptab` by

```
AST_PTR insert(AST_PTR exp)
{
  int hv = exp->hash;

  EXPTAB new = (EXPTAB) safe_allocate(sizeof(*new));
  new -> next = exptab[hv];
  new -> exp = exp;
  exptab[hv] = new;

  return exp;
}
```

1.4 Algebraic laws of EREGEX

the derivative's method will use the regex as DFA and NFA states. if 2 regex are equals (`e1 = e2` iff `L(e1) = L(e2)`), its will be the same state. but testing of semantic equality is hard jobs. we will simplify the parsed expression by algebraic laws and test the equality by their `exp_string` (see above `lookup()`).

1. Empty is reduced:
   `x` $\emptyset$ = $\emptyset$ `x = x &` $\emptyset$ = $\emptyset$ `& x = x ^` $\emptyset$ = $\emptyset$ `^ x =` $\emptyset$.
   $\emptyset$ `- x =` $\emptyset$, `x -` $\emptyset$ `= x`.

2. Epsilon is absorbed:
   `x` ε = ε `x = x ^` ε = ε `^ x = x`.
   `x |` $\emptyset$ = $\emptyset$ `| x = x`.

3. commutative law:
   `x | y = y | x`

   the parsed `Or` expression should be arranged as left associative expression with their hash values in increased order. e.g.

   `(c | b ) | (e | (f | g)) = (((((a | b) | c) | d) | e) | f`

   and the `exp_string` is `"a|b|c|d|e|f"`.

4. associative law for concatenation:
   `(xy)z = x(yz).`

   because the `exp_string` of `(xy)z` and `x(yz)` are the same : `"xyz"`, so the arrangement of left associatiion for x(yz) to (xy)z is not needed! so for `&` and `^`.

5. idempotent law for `|` and `&`:
   `x | x = x, x & x = x.`

6. law for Kleene star:
   `x** = x`

7. distributive law:
   `(x | y)z = xz | yz, x(y | z) = xy | xz.`

   for the derivative's method converge fastly, we should convert `(x | y)z` to `(xz | yz)` and so `x(y | z)`.

1.5 Nullability

a regex `x` is nullable iff ε in `L(x)`. so

x	N(x)
x	0
ε	1
$\emptyset$	0
x | y	N(x) || N(y)
xy	N(x) && N(y)
x*	1
x - y	N(x) && !N(y)
x ^ y	N(x) && N(y)
x & y	N(x) && N(y)

1.6 AST constructions

the following AST constructors implent the simplifications without commutative and associative laws (in `ast.c`): .

```
AST_PTR mkEpsilon (void)
{
  AST_PTR tree_tmp;

  tree_tmp = lookup ("");

  if (tree_tmp != NULL) return tree_tmp;

  tree_tmp = (AST_PTR) safe_allocate(sizeof(*tree_tmp));
  tree_tmp->op = Epsilon;
  tree_tmp->exp_string = strdup("");
```

```
  tree_tmp->hash = hash(tree_tmp->exp_string);
  tree_tmp->nullable = 1;
  tree_tmp->state = -1;
  tree_tmp->lf = NULL;
  tree_tmp->lchild = NULL;
  tree_tmp->rchild = NULL;
  return insert(tree_tmp);
}

AST_PTR mkEmpty (void)
{
  AST_PTR tree_tmp;

  tree_tmp = lookup ("");

  if (tree_tmp != NULL) return tree_tmp;

  tree_tmp = (AST_PTR ) safe_allocate(sizeof(*tree_tmp));
  tree_tmp->op = Empty;
  tree_tmp->exp_string = strdup("");
  tree_tmp->hash = hash(tree_tmp->exp_string);
  tree_tmp->nullable = 0;
  tree_tmp->lf = NULL;
  tree_tmp->state = -1;
  tree_tmp->lchild = NULL;
  tree_tmp->rchild = NULL;
  return insert(tree_tmp);
}



AST_PTR mkOpNode(Kind op, AST_PTR tree1, AST_PTR tree2)
{
  char *exp_string = (char *)safe_allocate(strlen(tree1->exp_string) +
                                           strlen(tree2->exp_string) + 6);
  char *lp1="", *rp1="", *lp2="", *rp2="";
  char *op_string;
  AST_PTR tree_tmp;

  switch (op) {
  case Alt: op_string = "^"; break;
  case Diff: op_string = "-"; break;
  case And: op_string = "&"; break;
  case Or: op_string = "|"; break;
  default: op_string = "";
  }

  if (op == Seq || op == Alt) {
    if (tree1->op == Epsilon) return tree2;
    if (tree1->op == Empty) return tree1;
    if (tree2->op == Epsilon) return tree1;
    if (tree2->op == Empty) return tree2;
  }

  if (op == And) {
    if (tree1->op == Epsilon) return tree1;
    if (tree1->op == Empty) return tree1;
    if (tree2->op == Epsilon) return tree2;
```

```
    if (tree2->op == Empty) return tree2;
  }
  if (op == Diff) {
    if (tree1->op == Empty) return tree1;
    if (tree2->op == Empty) return tree1;
  }

  if (tree1 == tree2) {
    if (op == Or || op == And) return tree1;
    if (op == Diff) return mkEmpty();
  }

  if (op == Or)
    sprintf(exp_string,"%s%s%s", tree1->exp_string,
            op_string, tree2->exp_string);
  else {
    if (op == Diff && tree2->op == Diff) {
      lp2 ="("; rp2 = ")";
    } else {
    if (tree1->op < op) {
      lp1 ="("; rp1=")";
    }
    if (tree2->op < op) {
      lp2 ="("; rp2 = ")";
    }
    }
    sprintf(exp_string,"%s%s%s%s%s%s%s", lp1, tree1->exp_string, rp1,
            op_string, lp2, tree2->exp_string, rp2);
  }
  tree_tmp = lookup (exp_string);

  if (tree_tmp != NULL)  {
    free(exp_string);
    return tree_tmp;
  }

  tree_tmp = (AST_PTR ) safe_allocate(sizeof *tree_tmp);
  tree_tmp->hash = hash(exp_string);
  tree_tmp->op = op;
  tree_tmp->exp_string = exp_string;
  tree_tmp->nullable = (op == Or?tree1->nullable || tree2->nullable:
                       (op == Diff? tree1->nullable*(tree1->nullable - tree2->nullable)
                        : tree1->nullable && tree2->nullable));
  tree_tmp->lf = NULL;
  tree_tmp->state = -1;
  tree_tmp->lchild = tree1;
  tree_tmp->rchild = tree2;

  return insert(tree_tmp);
}


AST_PTR mkStarNode(AST_PTR tree)
{
  char *exp_string = (char *) safe_allocate(strlen(tree->exp_string) + 4);
  char *lp = "", *rp = "";
  AST_PTR tree_tmp;
```

```
  if (tree->op == Star || tree->op == Epsilon ||
      tree->op == Empty) return tree;

  if (tree->op == Or && tree->lchild->op == Epsilon) return mkStarNode(tree->rchild);

  if (tree->op != Alpha) {
    lp = "("; rp = ")";
  }

  sprintf(exp_string, "%s%s%s%c", lp, tree->exp_string, rp, '*');

  tree_tmp = lookup (exp_string);

  if (tree_tmp != NULL)  {
    free(exp_string);
    return tree_tmp;
  }

  tree_tmp = (AST_PTR) safe_allocate(sizeof(*tree_tmp));

  tree_tmp->hash = hash(exp_string);
  tree_tmp->op = Star;
  tree_tmp->exp_string = exp_string;
  tree_tmp->nullable = 1;
  tree_tmp->state = -1;
  tree_tmp->lf = NULL;
  tree_tmp->lchild = tree;
  tree_tmp->rchild = NULL;
  return insert(tree_tmp);
}
```

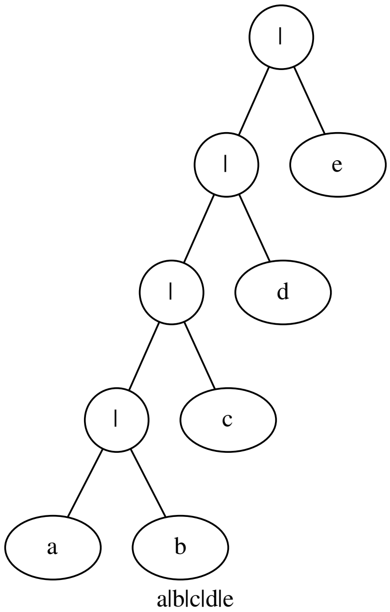
1.7 Commutative and Associative laws

you should implent `AST_PTR arrangeOpNode(Kind op, AST_PTR tree1, AST_PTR tree2)` for the commutative operators `|`, `^` and `&`, with the consecutive `|` will be transformed to the leftmost associative expresion with the operant in increased order; `AST_PTR arrangeSeqNode(AST_PTR tree1, AST_PTR tree2)` for left and right distributive law of `Seq`, so input `"(a|b)c"` will output `"ac|bc"` and input `"a(b|c)"` will output `"ab|ac"`.

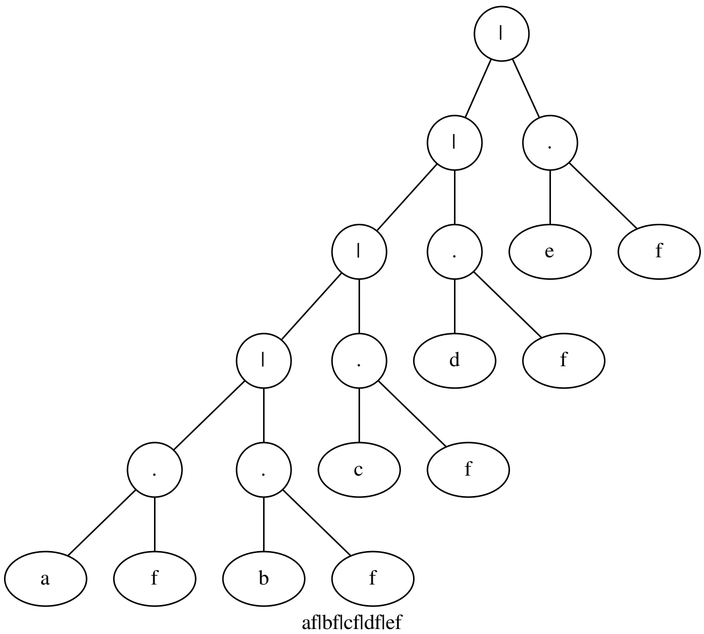
1.8 Testsuite

use the sample exe (`REG2FDFA.EXE` for DOS, `reg2dfa` for Linux) to check the output graphviz file `ast.gv` (`dot -Tpdf -o ast.pdf ast.gv` generates the image).

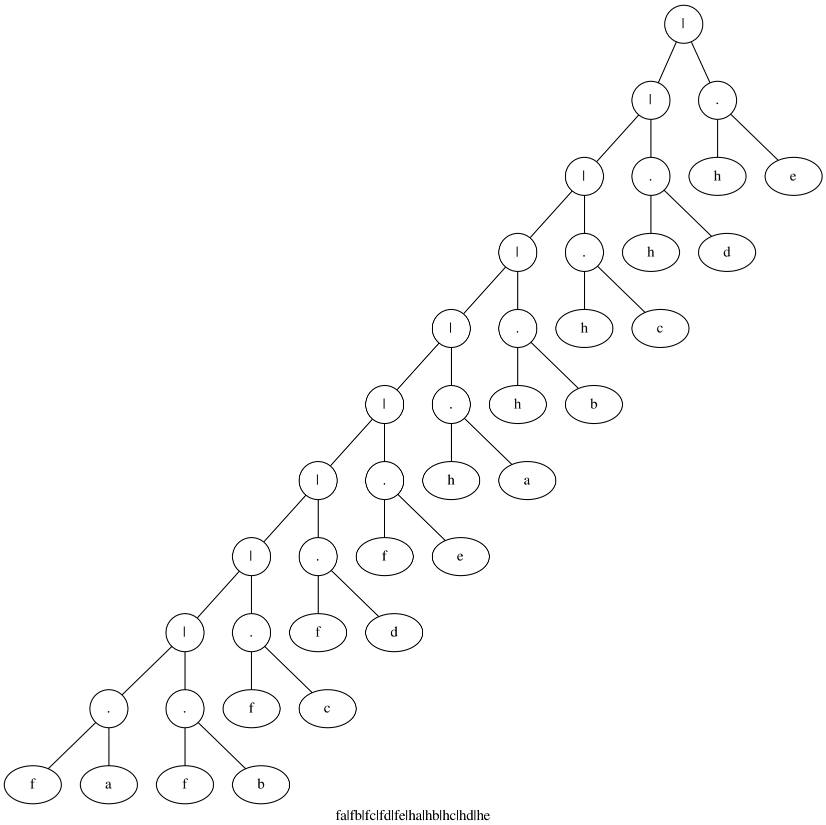
1. `a|((b|d)|(c|e))`
   the simplified exp is `a|b|c|d|e`



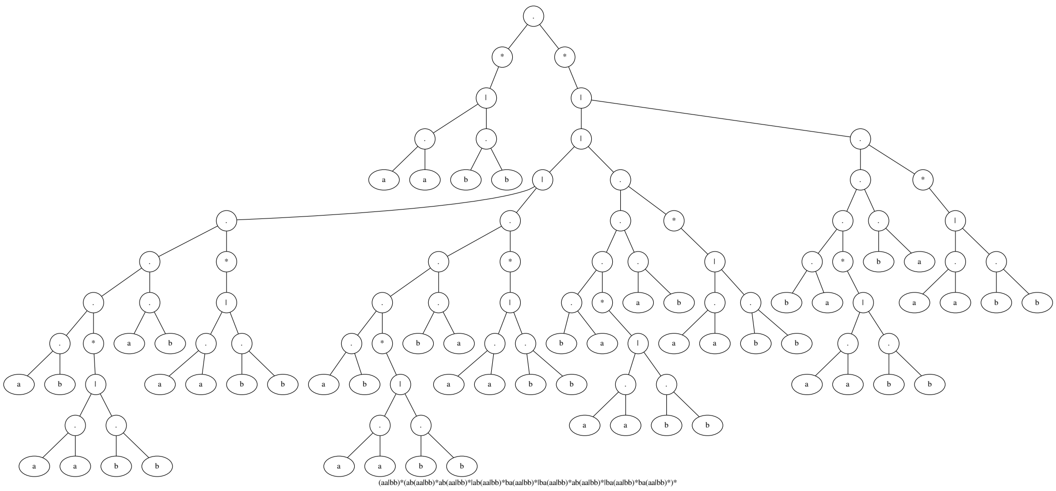
2. `(a|((b|d)|(c|e)))f`
   the simplified exp is `af|bf|cf|df|ef`



3. `(f|h)(a|((b|d)|(c|e)))`
   the simplified exp is `fa|fb|fc|fd|fe|ha|hb|hc|hd|he`



4. `(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*`
   the simplified exp is `(aa|bb)*(ab(aa|bb)*ab(aa|bb)*|ab(aa|bb)*ba(aa|bb)*` `|ba(aa|bb)*ab(aa|bb)*|ba(aa|bb)*ba(aa|bb)*)*`



1.9 TODO

implement `AST_PTR arrangeOpNode(Kind op, AST_PTR tree1, AST_PTR tree2)` and `AST_PTR arrangeSeqNode(AST_PTR tree1, AST_PTR tree2)`, so the exe will output the same `ast.gv` as the sample program.
please send your `ast.c` as attached file to `mailto:hanfei.wang@gmail.com?subject=ID(03)` where the `ID` is your student id number.

2 Partial Derivative

the NFA obtained by Thomson's algorithm has the `O(n)` states where the `n` is the size of the input regex `r` (the number of letters in `r`). this NFA has many ε-transitions. we will use the partial derivative transform the regex to NFA without ε-transitions and with the number of states $\leq$ `n + 1`. the derived DFA has less states than subset construction.

2.1 linear form

let `r` be a regex,

1. *linear form* of `r`:
   `lf(r) = N(r) || a1 r1 | a2 r2 | ... | an rn`,
   where `ai` is an alphabet and `ri` is a regex.

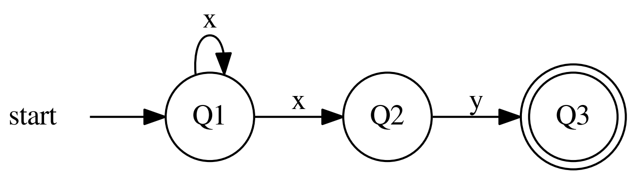
2. `lf(x*xy) = 0 || x x*xy | x y`, where `a1 = a2 = x`, `r1 = x*xy` and `r2 = y`.
   so `ai` and `aj` can be the same letter, so called *undeterministic*. if all `ai` are different, it's *deterministic* linear form.

3. we can obtain the deterministic lf just by regrouping right factor of lf of the same `ai` by distributive law, so `lf(x*xy) = 0 || x (x*xy | x y)`. the right factor of the deterministic `lf` is so called `partial derivative` relatively to `x`.

if regard `r` and `ri` as an NFA states `Q` and `Qi`, so we havet the NFA transition `trans(Q, ai) = Qi`. if `O(Q) = 1` , the state `Q` is final. we can recursively calculate the `lf` of the new generated `Qi` until no more new state generated (we can proof that there are only at most `n + 1` state generated, so the algorithm converges, see detail in partial_derivative.pdf).

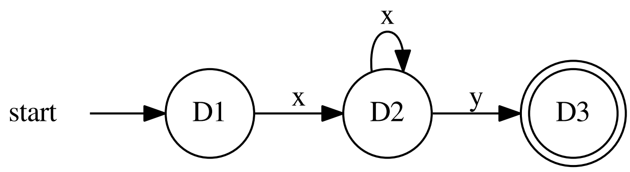
so let `Q1 = x*xy`, we have NFA:

`Q1 = 0 || x Q1 | x Q2 | Q1 = x*xy`
`Q2 = 0 || y Q3 | Q2 = y`
`Q3 = 1 || Q3 =` ε



NFA of x*xy

regrouping the `lf` as `Q1 = 0 || x (Q1 | Q2)`, and let `D1 = Q1` and `D2 = Q1 | Q2 = x*xy | y`, calculate `lf` of D2, `D2 = 0 || x xx*y | x y | y` ε, regrouping D2 as `D2 = 0 || x (xx*y | y) | y` ε `= 0 || x D1 | y D3` where `D3 =` ε. so we have DFA:
`D1 = 0 || x D2 | D1 = x*xy`
`D2 = 0 || x D2 | y D3 | D2 = x*xy|y`
`D3 = 1 || D3 =` ε



DFA of x*xy

2.2 Calculation of the linear form

`lf` can be calculated by post-order tree traversal with the following rules:

regex	lf
`ε`	`1 || ∅`
`∅`	`0 || ∅`
`x`	`0 || x ε`
`A | B`	`N(A)∨N(B) || (lf(A) | lf (B))` if `lf(A) = a1 A1 | ... | an An, lf(B) = b1 B1 | ... | bm Bm` then `lf(A) | lf(B) = a1 A1 | ... | an An | b1 B1 | ... | bm Bm`
`A B`	`0 || (lf(A))B` if `N(A) = 0` if `lf(A) = a1 A1 | ... | an An`, then `(lf(A))B = a1 (A1 B) | ... | an (An B)`
`A B`	`N(B) || (lf(A))B | lf(B)` if `N(A) = 1`
`A*`	`1 || (lf(A))A`

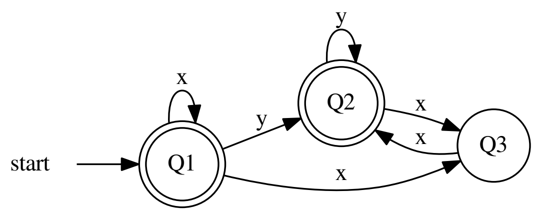
2.2.1 Example 1. NFA of `x*(y|xx)*`

```
  lf(x*(y|xx)*)
= O((y|xx)*) || (lf(x*))(y|xx)* | lf((y|xx)*)
= 1 || ((lf(x))x*)(y|xx)* | lf((y|xx)*)
= 1 || x (x*(y|xx)*) | lf((y|xx)*)
= 1 || x (x*(y|xx)*) | (lf(y|xx))(y|xx)*
= 1 || x (x*(y|xx)*) | (lf(y) | lf(xx))(y|xx)*
= 1 || x (x*(y|xx)*) | (y ε | (lf(x))x)(y|xx)*
= 1 || x (x*(y|xx)*) | (y ε | (x ε)x) (y|xx)*
= 1 || x (x*(y|xx)*) | (y ε | x x )(y|xx)*
= 1 || x (x*(y|xx)*) | y (y|xx)* | x x(y|xx)*
= 1 || x Q1 | y Q2 | x Q3
(where Q1 = x*(y|xx)*, Q2 = (y|xx)*, Q3 = x(y|xx)*)

  lf(Q2) = lf((y|xx)*)
= 1 || (lf(y|xx))(y|xx)*
= 1 || (lf(y) | lf(xx))(y|xx)*
= 1 || (y ε | lf(x)x)(y|xx)*
= 1 || y (y|xx)* | x x(y|xx)*
= 1 || y Q2 | x Q3

  lf(Q2) = lf(x(y|xx)*)
= 0 || (lf(x))(y|xx)*
= 0 || x (y|xx)*
= 0 || x Q3
```

so, we have NFA:

```
Q1 = 1 || x Q1 | y Q2 | x Q3 | Q1 = x*(y|xx)*
```

```
Q2 = 1 || y Q2 | x Q3 | Q2 = (y|xx)*
Q3 = 0 || x Q2 | Q3 = x(y|xx)*
```



NFA of x*(y|xx)*

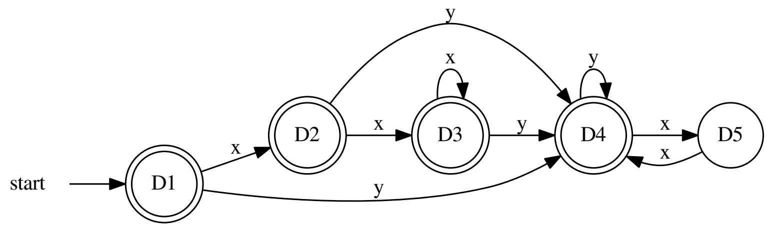
2.2.2 Example 2. DFA of `x*(y|xx)*`

We just regroup the linear form where the same letter appears at most one times in `lf`, the NFA becomes DFA!

```
  lf(D1) = lf(x*(y|xx)*)
= O((y|xx)*) || (lf(x*))(y|xx)* | lf((y|xx)*)
= 1 || x (x*(y|xx)*) | y (y|xx)* | x x(y|xx)*
= 1 || x (x*(y|xx)*|x(y|xx)*) | y (y|xx)* = 1 || x D2 | y D4
(where D1 = x*(y|xx)*, D2 = x*(y|xx)*|x(y|xx)*, D4 = (y|xx)*)

  lf(D2) = lf(x*(y|xx)*|x(y|xx)*)
= 1 || x (x*(y|xx)*|x(y|xx)*|(y|xx)*) | y (y|xx)*
= 1 || x D3 | y D4
(where D3 = x*(y|xx)*|x(y|xx)*|(y|xx)*)

  lf(D3) = lf(x*(y|xx)*|x(y|xx)*|(y|xx)*)
= 1 || x (x*(y|xx)*|x(y|xx)*|(y|xx)*) | y (y|xx)*
= 1 || x D3 | y D4

  lf(D4) = lf((y|xx)*)
= 1 || y (y|xx)* | x (x(y|xx)*)
= 1 || y D4 | x D5
(where D5 = x(y|xx)*)

  lf(D5) = lf(x(y|xx)*)
= 0 || y (y|xx)* | x (x(y|xx)*)
= 0 || x (y|xx)* D4
= 0 || x D4
```

so, we have DFA:

```
D1 = 1 || x D2 | y D4 | D1 = x*(y|xx)*
D2 = 1 || x D3 | y D4 | D2 = x*(y|xx)*|x(y|xx)*
D3 = 1 || x D3 | y D4 | D3 = x*(y|xx)*|x(y|xx)*|(y|xx)*
D4 = 1 || y D4 | x D5 | D4 = (y|xx)*
D5 = 0 || x D4 | D5 = x(y|xx)*
```



DFA of x*(y|xx)*

2.3 Algorithm

the structure of linear form is defined in `ast.h`:

```
typedef struct lf {
  struct lf *next;
  char symbol;
  AST_PTR exp;
} LF;

typedef LF *LF_PTR;
```

we need 2 operations for the calculation of `lf`, abstracts as

```
static LF_PTR (*union_method)();
/* the union of 2 lf */

static LF_PTR (*seq_method)();
/* concatenation of a linear form with a regex */
```

for NFA, `union_method()` should be:

```
LF_PTR lf_union(LF_PTR lf1, LF_PTR lf2)
{
  LF_PTR tmp;
  tmp = lf1 = lf_clone(lf1);
  lf2 = lf_clone(lf2);

  if (lf1 == NULL) return lf2;
  while (tmp != NULL) {
    if (tmp->next == NULL) {
      tmp->next = lf2;
      break;
    }
    tmp = tmp -> next;
  }
  return lf1;
}
/* if
     lf1 = a1 A1 | ... | an An,
     lf2 = b1 B1 | ... | bm Bm
   then
     lf_union(lf1, lf2) = a1 A1 | ... | an An | b1 B1 | ... | bm Bm
*/
```

`seq_method()` should be:

```
LF_PTR lf_concate(LF_PTR lf, AST_PTR exp)
{
  LF_PTR tmp;
  tmp = lf = lf_clone(lf);
  if (tmp == NULL) return NULL;
  while (tmp != NULL) {
    tmp->exp = mkOpNode(Seq, tmp->exp, exp);
    tmp = tmp->next;
  }
  return lf;
}
/* if
     lf = a1 A1 | ... | an An,  exp = B
   then
     lf_concate(lf, lf2) = a1 (A1 B) | ... | an (An B)
*/
```

for DFA, `union_method()` should be:

```
LF_PTR lf_union_plus(LF_PTR lf1, LF_PTR lf2)
{
  LF_PTR head, tmp, new, lf;
  head = lf = lf_clone(lf1);

  if (lf1 == NULL) return lf2;

  while (lf2 != NULL) {
    tmp = lf;
    while (tmp != NULL) {
      if (tmp->symbol == lf2->symbol) {
        tmp->exp = arrangeOpNode(Or, tmp->exp, lf2->exp);
        goto NEXT;
      }
      tmp = tmp->next;
    }
    new = mk_lf(lf2->symbol, lf2->exp);
    new->next = head;
    head = new;
  NEXT:
    lf2 = lf2 -> next;
  }
  return head;
}

/* if
     lf1 = a1 A1 | ... | an An | c1C1 | ... | ciCi,
     lf2 = a1 B1 | ... | an Bn | d1D1 | ....| djDj
   then
     lf_union_plus(lf1, lf2) = a1 (A1 | B1) | ... | an (An | Bn) |
       c1C1 | ... | ciCi | d1D1 | ....| djDj
*/
```

`seq_method()` should be:

```
LF_PTR lf_concate_plus(LF_PTR lf, AST_PTR exp)
{
  LF_PTR tmp;
  tmp = lf = lf_clone(lf);
  if (tmp == NULL) return NULL;
  while (tmp != NULL) {
```

```
    tmp->exp = arrangeSeqNode(Seq, tmp->exp, exp);
      /* apply distributive law */
    tmp = tmp->next;
  }
  return lf;
}
/* if
     lf = a1 (A1 | B1) | ... | an (An | Bn),  exp = C
   then
     lf_concate(lf, lf2) = a1 (A1 C |B1 C) | ... | an (An C | Bn C)
*/
```

so the NFA and DFA can be unified as:

```
void linear_form(AST_PTR exp, int stated)
/* lf calculation of exp,
   if stated is 1, and add allstate all (Ai) and
   recursively call linear_form(Ai, 1) where exp->lf = a1 A1 | ... | an An
*/
{
  /* TODO */
}
```

3 Testsuite

to run the test, just `./reg2dfa exN`, where `N` is number of the test example. `dot -Tpdf -o exN.pdf exN.gv` to see the MDFA graph.

3.1 ex1: `(a|b)*(babab(a|b)*bab|bba(a|b)*bab)(a|b)*`

NFA states: 13, DFA states 21, MDFA states 10.
Thomson & Subset (JFLAP): NFA 68, DFA 62, MDFA 10.

3.2 ex2: `((a*b*a*b*)*(a*b*a*b*)*(a*b*a*b*)*(a*b*a*b*)*)*`

NFA states: 17, DFA states 3, MDFA states 1.
Thomson & Subset (JFLAP): NFA 84, DFA 3, MDFA 1.

3.3 ex3: `(a*b*a|b*a*b)*`

NFA states: 5, DFA states 3, MDFA states 1.
Thomson & Subset (JFLAP): NFA 3, DFA 3, MDFA 1.

3.4 ex4: `(ba*b*|ab*a*)*`

NFA states: 5, DFA states 8, MDFA states 1.
Thomson & Subset (JFLAP): NFA 2, DFA 9, MDFA 1.

3.5 ex5: `((ab|ba)*aa|(ab|ba)*bb)*(ab|ba)*`

NFA states: 12, DFA states 4, MDFA states 2.
Thomson & Subset (JFLAP): NFA 66, DFA 8, MDFA 2.

3.6 ex6: `(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*`

NFA states: 9, DFA states 4, MDFA states 4.
Thomson & Subset (JFLAP): NFA 82, DFA 17, MDFA 4.

3.7 ex7:

`((aa|ab(bb)*ba)*(b|ab(bb)*a)(a(bb)*a)*(b|a(bb)*ba))*(aa|ab(bb)*ba)*(b|ab(bb)*a)(a(bb)*a)*`

NFA states: 64, DFA states 7, MDFA states 4.

3.8 ex8:

`(a(aa)*|aa(aaa)*|aaa(aaaaa)*|aaaaa(aaaaaaa)*|aaa(aaaaaaaaaaa)*|aaa(aaaaaaaaaaaa)*|`
`aaa(aaaaaaaaaaaaa)*)`

NFA states: 54, DFA states 60061, MDFA states 30030.

3.9 ex9:

`(a(aa)*|aa(aaa)*|aaa(aaaaa)*|aaaaa(aaaaaaa)*|aaa(aaaaaaaaaaa)*|aaa(aaaaaaaaaaaa)*|`
`aaa(aaaaaaaaaaaaa)*)*`

NFA states: 54, DFA states 4, MDFA states 1.

if we disacitve distributive law in `lf_concate_plus()`, `rege2dfa` will cause memory exhausted!

4 Discussions

the distributive laws in `lf_concate_plus()` can accelerate the convergence of DFA, and significantly reduce the DFA states. as an example:

`(a(aa)*|aa(aaa)*|aaa(aaaaa)*|aaaaa(aaaaaaa)*)*` will generate the DFA with 8 states. but if we choose `lf_concate()` (without distributive law) as `seq_method()` for DFA, it will work few minnutes to generate 211 DFA states!

but it's not always true! there is the risk of ast grow exponentially if the distributive law is chosen. as an example, the regex of no repeatation of digits `0 - 3` (see `rep0_3.txt`) :

`(1|!)(01)*(0|!)(2(0(10)*(1|!)|1(01)*(0|!)))*(2|!)(3(2((0(10)*(1|!)|1(01)*(0|!))2)*(1|!)`
`(01)*(0|!)|(0(10)*(1|!)|1(01)*(0|!))(2(0(10)*(1|!)|1(01)*(0|!)))*(2|!)))*(3|!)`

with distributive law: DFA 12 states, MDFA 5 states. without distributive law, DFA 13 states.

the regex of no repeatation of digits `0 - 4` (see `rep0_4.txt`):

`(1|!)(01)*(0|!)(2(0(10)*(1|!)|1(01)*(0|!)))*(2|!)(3(2((0(10)*(1|!)|1(01)*(0|!))2)*(1|!)`
`(01)*(0|!)|(0(10)*(1|!)|1(01)*(0|!))(2(0(10)*(1|!)|1(01)*(0|!)))*(2|!)))*(3|!)`
`(4(3((2((0(10)*(1|!)|1(01)*(0|!))2)*(1|!)(01)*(0|!)|(0(10)*(1|!)|1(01)*(0|!))`
`(2(0(10)*(1|!)|1(01)*(0|!)))*(2|!))3)*(1|!)(01)*(0|!)(2(0(10)*(1|!)|1(01)*(0|!)))*`
`(2|!)|(2((0(10)*(1|!)|1(01)*(0|!))2)*(1|!)(01)*(0|!)|(0(10)*(1|!)|1(01)*(0|!))(2(0(10)*`
`(1|!)|1(01)*(0|!)))*(2|!))(3(2((0(10)*(1|!)|1(01)*(0|!))2)*(1|!)(01)*(0|!)|(0(10)*`
`(1|!)|1(01)*(0|!))(2(0(10)*(1|!)|1(01)*(0|!)))*(2|!)))*(3|!)))*(4|!)`

with distributive law: memory exausted! without distributive law, DFA 31 states, MDFA 6 states.

5 TODO

5.1 Implement `linear_form()` in `nfa.c`

Implement `lf` for the ordinary regex (union, concatenation, and star).

5.2 EREGEX(Bonus)

the `lf` of EREGEX is recursively defined as

<table>
<tr><th>regex</th><th>lf</th></tr>
<tr><td><code>A ^ B</code></td><td><code>N(A)∧N(B) || ((lf(A) ^ B) | (A ^ lf (B)))</code>
where if <code>lf(A) = a1 A1 | ... | an An</code>,
then <code>lf(A) ^ B = a1 (A1 ^ B) | ... | an (An ^ B)</code></td></tr>
<tr><td><code>A & B</code></td><td><code>N(A)∧N(B) || (lf(A)) & (lf(B))</code>
if <code>lf(A) = a1 A1 | ... | an An</code>, <code>lf(B) = a1 B1 | ... | an Bn</code> then
<code>(lf(A)) & (lf(B)) = a1 (A1 & B1) | ... | an (An & Bn)</code></td></tr>
<tr><td><code>A - B</code></td><td><code>N(B)∧¬N(B) || (lf(A)) - (lf(B))</code>
if <code>lf(A) = a1 A1 | ... | an An</code>, <code>lf(B) = a1 B1 | ... | an Bn</code> then
<code>(lf(A)) - (lf(B)) = a1 (A1 - B1) | ... | an (An - Bn)</code></td></tr>
</table>

Implement `linear_form()` for extended regex operations. as test example (`rep0_9B.txt`):

```
(0|1|2|3|4|5|6|7|8|9)*-(0|1|2|3|4|5|6|7|8|9)*(00|11|22|33|44|55|66|77|88|99)(0|1|2|3|4|5|6|7|8|9)*
```

will generate 12 state MDFA where one is trap state.

because the different op `-` will diverged for NFA (e.g. `(a|b)*-(a|b)*ab(a|b)*`). we should disacitve `lf` for NFA if `-` is presented in regex (see `is_minus(exp)` in `nfa.c`).

5.3 LR parser for EREGEX

We can also use YACC to generate LR parser of EREGEX. to add the macro definition in the regex definition, we can add `Eq` of AST type `Kind`:

```
typedef enum { Eq = 0, Or = 1, Diff = 2, Alt = 3, And = 4, Seq = 5,
               Star = 6, Alpha = 7, Epsilon = 8, Empty = 9} Kind;
/* in order of increasing precdence */
```

and YACC grammar:

```
%{
#include <ctype.h>
#include <stdlib.h>
#include "ast.h"

#define YYSTYPE AST_PTR
#define MAX_BUFFER 1024
static char input_buffer[MAX_BUFFER] = "\0";
static char * current = input_buffer;
%}

%token ALPHA
%right ’=’
%left ’|’
%left ’-’
%left ’^’
%left ’&’
%left ALPHA ’(’ ’!’
%left CONCAT
%nonassoc ’?’
%nonassoc ’*’
%nonassoc ’+’

%%
```

```
root : root  line
|
;
line : reg ';' {
  if ($1->op != Eq) {
    printf("the simplified exp is %s\n", $1->exp_string);
    print_tree($1);
    printf("\n");
    reg2nfa($1);
  }
;
reg : ALPHA { $$ = mkLeaf(*current); }
| '!'  { $$ = mkEpsilon(); }
| '(' reg ')' { $$ = $2;  }
| reg '=' reg { $$ = mkEqNode($1, $3); }
| reg '|' reg { $$ = arrangeOpNode(Or, $1, $3); }
| reg '-' reg { $$ = arrangeOpNode(Diff, $1, $3); }
| reg '^' reg { $$ = arrangeOpNode(Alt, $1, $3); }
| reg '&' reg { $$ = arrangeOpNode(And, $1, $3); }
| reg reg %prec CONCAT { $$ = arrangeSeqNode($1, $3); }
| reg '*' { $$ = mkStarNode($1); }
| reg '+' { $$ = arrangeSeqNode($1, mkStarNode($1));
             /* e+ = e e* */ }
| reg '?' { $$ = arrangeOpNode(Or, $1, mkEpsilon());
            /* e? = e | epsilon */ }
;
```

so the the regex of no repeatation of digits can be recursive defined as (`rep0_9A.txt`):

```
A = 1? (0 1)* 0?;
B = 1 (0 1)* 0? | 0 (1 0)* 1?;
C = A(2 B)* 2?;
D = 2 (B 2)* A | B (2 B)* 2?;
E = C(3 D)* 3?;
F = 3(D 3)* C | D (3 D)* 3?;
G = E (4 F)* 4?;
H = 4(F 4)* E | F (4 F)* 4?;
I = G (5 H)* 5?;
J = 5(H 5)* G | H (5 H)* 5?;
K = I (6 J)* 6?;
L = 6(J 6)* I | J (6 J)* 6?;
M = K (7 L)* 7?;
N = 7(L 7)* K | L (7 L)* 7?;
O = M (8 N)* 8?;
P = 8(N 8)* M | N (8 N)* 8?;
Q = O (9 P)* 9?;
Q;
```

if disacitve distributive law, `reg2dfa` will generate 59 state NFA, 1892 state DFA, and 11 state MDFA.

please send your `nfa.c` as attached file to `mailto:hanfei.wang@gmail.com?subject=ID(04)` where the ID is your student id number.

–hfwang

October 20, 2019