Parser of the extended regular expressions

WANG Hanfei

September 16, 2019

Contents

2017 级弘毅班编译原理课程设计第 3 次编程作业 (Parser 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  997

/* 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);
}
```

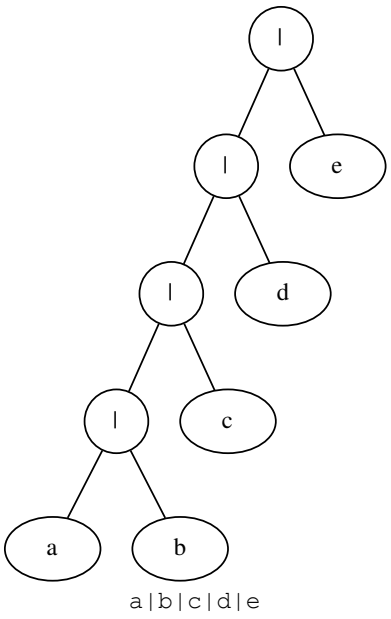
1.7 Commutative and Associative laws

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`".

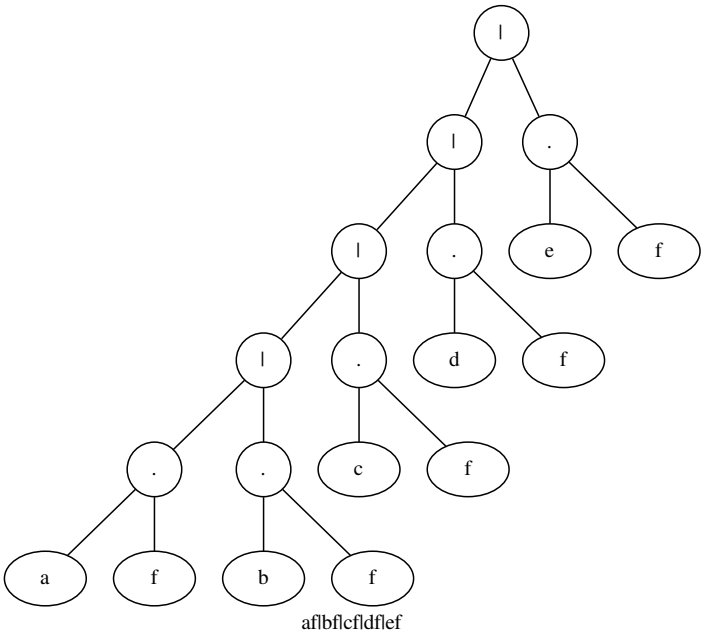
1.8 Testsuite

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`

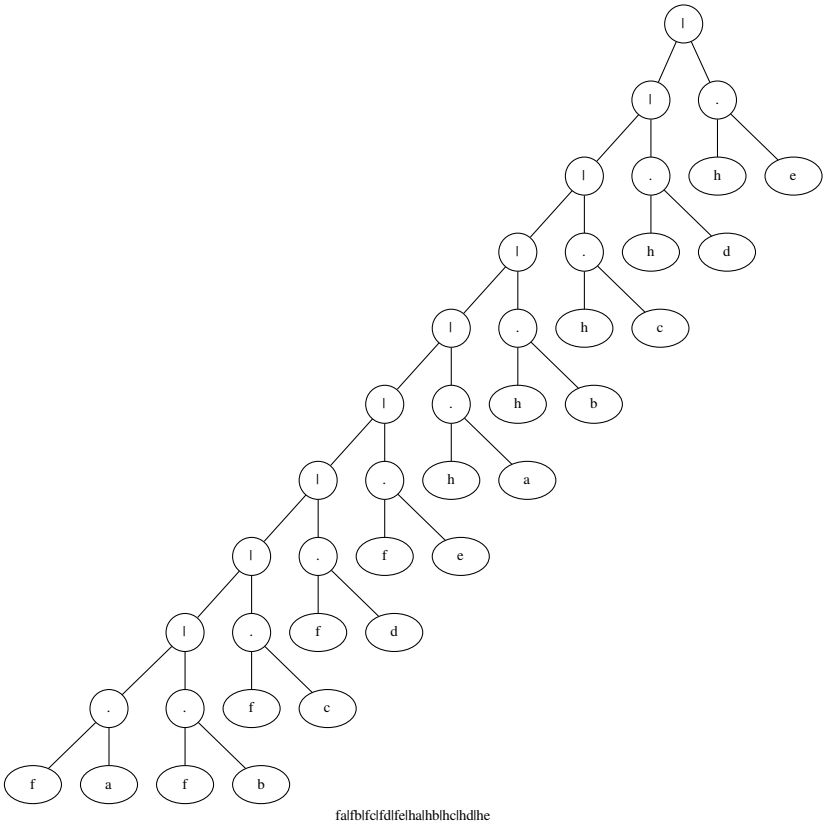


2. `(a|((b|d)|(c|e)))f`
   the simplified exp is `af|bf|cf|df|ef`



af|bf|cf|df|ef

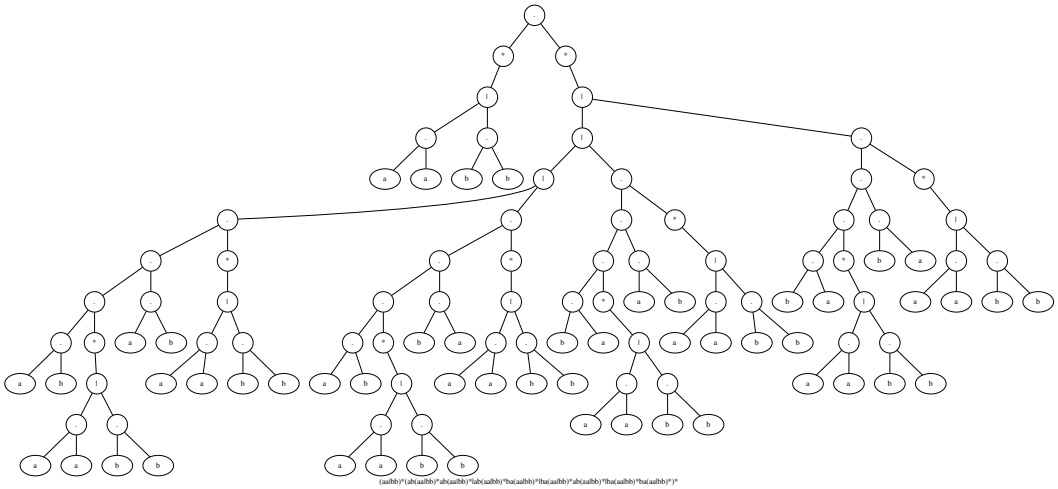
3. `(f|h)(a|((b|d)|(c|e)))`
   the simplified exp is `fa|fb|fc|fd|fe|ha|hb|hc|hd|he`



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.

–hfwang

September 16, 2019