

2021 — 2022 学年《编译原理》第 3-4 次编程作业

AST and derivatives of the extended regular expressions

WANG Hanfei

October 10, 2021

Contents

1	扩展正则表达式 (ERESEX)	2
1.1	ERESEX 的文法	2
1.2	抽象语法树 (AST)	3
1.3	符号表	3
1.4	Algebraic laws of ERESEX	4
1.5	Nullability	5
1.6	AST	5
1.7	交换律和结合律	8
1.8	测试	8
1.9	TODO	10
2	偏导数 (assignment 04)	11
2.1	线性形式	11
2.2	线性形式的计算	12
2.2.1	Example 1. NFA of $x^*(y xx)^*$	12
2.2.2	Example 2. DFA of $x^*(y xx)^*$	13
2.3	算法	14
3	测试	16
3.1	ex1: $(a b)^*(babab(a b)^*bab bba(a b)^*bab)(a b)^*$	16
3.2	ex2: $((a^*b^*a^*b^*)^*(a^*b^*a^*b^*)^*(a^*b^*a^*b^*)^*(a^*b^*a^*b^*)^*)^*$	16
3.3	ex3: $(a^*b^*a b^*a^*b)^*$	16
3.4	ex4: $(ba^*b^* ab^*a^*)^*$	16
3.5	ex5: $((ab ba)^*aa (ab ba)^*bb)^*(ab ba)^*$	17
3.6	ex6: $(aa bb)^*((ab ba)(aa bb)^*(ab ba)(aa bb)^*)^*$	17
3.7	ex7:	17
3.8	ex8:	17
3.9	ex9:	17
4	Discussions	17
5	TODO	18
5.1	Implement <code>linear_form()</code> in <code>nfa.c</code>	18
5.2	ERESEX(Bonus)	18
5.3	LR parser for ERESEX	18

2018 级软件工程专业《编译原理》第 3-4 次编程作业
正则表达式的抽象语法树与求导

本次作业 (03) 将使用递下降分析法对正则表达式进行语法分析, 并输出抽象语法树. 在此基础上对正则表达式进行求导 (derivative), 其对应的线性形式即可看成是 NFA 或 DFA 的状态转移, 从而完成正则表达式到 DFA 的直接转换 (assignment 04) (see [partial_derivative.pdf](#)).

1 扩展正则表达式 (ERESEX)

增加了 3 个二元运算:

1. 差: $e_1 - e_2, L(e_1 - e_2) = L(e_1) - L(e_2).$
2. 交错乘积: $e_1 \wedge e_2, L(e_1 \wedge e_2) = \{ s_1 t_1 \cdots s_n t_n \mid s_1 \cdots s_n \in L(e_1) \wedge t_1 \cdots t_n \in L(e_2) \}$

$$\begin{aligned} a \wedge b &= a b \mid b a \\ ab \wedge ba &= (a \wedge b) (a \wedge b) \\ a \wedge b \wedge c &= a b c \mid a c b \mid b a c \mid b c a \mid c a b \mid c b a \end{aligned}$$

3. 交: $e_1 \& e_2, L(e_1 \& e_2) = L(e_1) \cap L(e_2).$

优先级由低到高排列: $\mid, -, \wedge, \&, \text{concat}, *$.

1.1 ERESEX 的文法

消除左递归后的文法:

```
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_concat -> kleene term_concat'
term_concat' -> kleene term_concat' | epsilon

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

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

设非终结符 term_xxx 和 $\text{term_xxx}'$, 其中 xxx is or, alt, and, or concat. 则文法可统一为:

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

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

这样其对应的递归调用函数 (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 抽象语法树 (AST)

详见 [ast.h](#):

```

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, it will use to determine
                    if NFA is final */
    char *exp_string; /* mostly simplified exp of E, use to
                       determine if 2 regex are equals */
    int state; /* for assignment 04!!!
                state number. trap state is 0,
                the original exp is 1 */
    LF_PTR lf; /* for assignment 04!!!
                linear form of NFA */
} AST;

```

1.3 符号表

语法分析或后续求导过程中出现的子正则表达式都进符号表 ([ast.h](#)):

```

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 (`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) ;
}

```

接口函数:

```

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

求导法直接求 FA 是把正则表达式最为 FA 的状态, 其求解过程需要判断两个状态是否相等, 即两个正则表达式是否等价 ($e1 = e2$ iff $L(e1) = L(e2)$). 但严格按数学定义来判断相等, 只能用其最小状态 DFA 同构, 即本问题求 DFA, 因此行不通. 在此我们用正则表达式的代数变换, 对所分析的正则表达式, 利用代数定律求其最简代数形式. 若最简代数形式 $\sim exp_string \sim$ (see above `lookup()`) 相等, 则相等. 正则表达式的代数定律如下:

1. 空的化简:

$$x \emptyset = \emptyset x = x \& \emptyset = \emptyset \& x = x \wedge \emptyset = \emptyset \wedge x = \emptyset.$$

$$\emptyset - x = \emptyset, x - \emptyset = x.$$

2. 空串的吸收:

$$x \varepsilon = \varepsilon x = x \wedge \varepsilon = \varepsilon \wedge x = x.$$

$$x \mid \emptyset = \emptyset \mid x = x.$$

3. 交换律:

$$x \mid y = y \mid x$$

连续的并运算 \mid 必须调整为左结合且对应的运算符表达式严格按字典序排列. e.g.

$$(c \mid b) \mid (e \mid (f \mid a)) = (((((a \mid b) \mid c) \mid d) \mid e)) \mid f$$

这时 `exp_string` 是 `"a|b|c|d|e|f"`.

4. 连接运算的结合律:

$$(xy)z = x(yz).$$

因 $(xy)z$ 和 $x(yz)$ 的 `exp_string` 是相同的, 即 `"xyz"`, 因此不必对 $x(yz)$ 调整为最左结合. 运算 $\&$ 和 \wedge 也不需调整.

5. 幂等率 (\mid 和 $\&$):

$$x \mid x = x, x \& x = x.$$

6. Kleene 闭包: $x^{**} = x^*$

7. 分配率:

$$(x \mid y)z = xz \mid yz, x(y \mid z) = xy \mid xz.$$

为了加速求导法的收敛, 必须用分配率:

$$(x \mid y)z = (xz \mid yz), x(y \mid z) = xy \mid xz.$$

1.5 Nullability

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

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

1.6 AST

以下 AST 构造函数没有使用交换律和结合率进行化简 (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; /* no associates to any state */
    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 交换律和结合律

请实现交换律化简正则表达式函数 `AST_PTR arrangeOpNode(Kind op, AST_PTR tree1, AST_PTR tree2)`, 其中 `op` 为 `|`, `^` 或 `&`. 要求对连续的 `|` 均转换为最左结合的 `|` 运算.

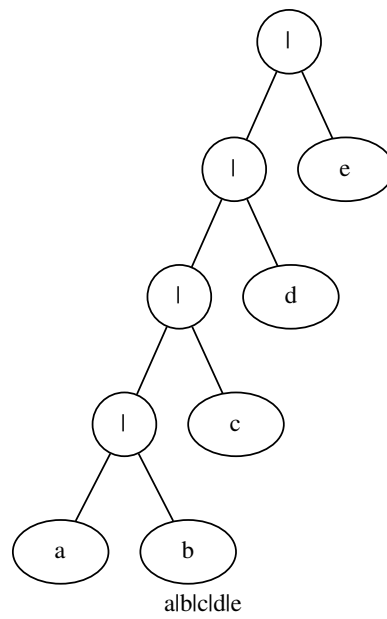
实现函数 `AST_PTR arrangeSeqNode(AST_PTR tree1, AST_PTR tree2)`, 用结合率化简正则表达式. 是的输入 `"(a|b)c"`, 化简后输出 `"ac|bc"`; 输入 `"a(b|c)"`, 输出 `"ab|ac"`.

1.8 测试

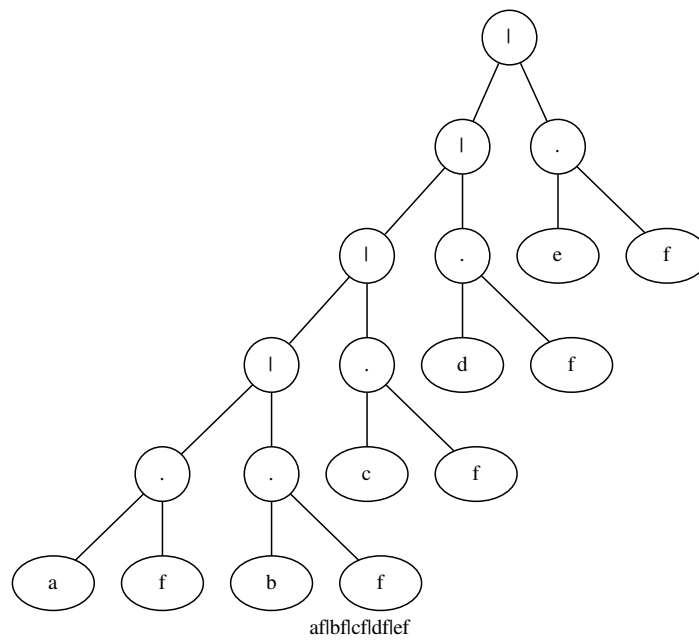
我们借助于 [graphviz](#) 可视化 AST, 函数 `int graphviz_ast_aux(AST_PTR)` 和 `void graphviz_ast(AST_PTR)` 可把 AST 转换成 [graphviz](#) 输入文件格式 (`.gv`). 大家可用样本程序 (`REG2FDFA.EXE` for DOS, `reg2dfa` for Linux) 输入正则表达式, 则程序输出 [graphviz](#) 文件 `ast.gv`. `"dot -Tpdf -o ast.pdf ast.gv"` (Linux) 转换为可视化的 AST 树结构 (`pdf` 文件).

以下是测试用例:

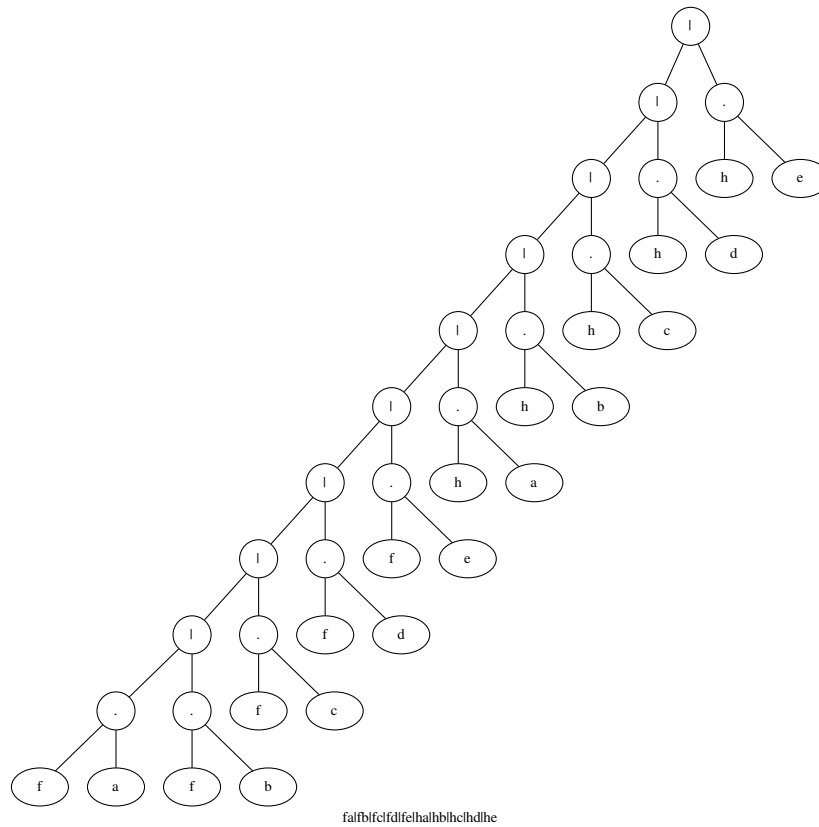
1. `a|((b|d)|(c|e))`
the simplified exp is `a|b|c|d|e`



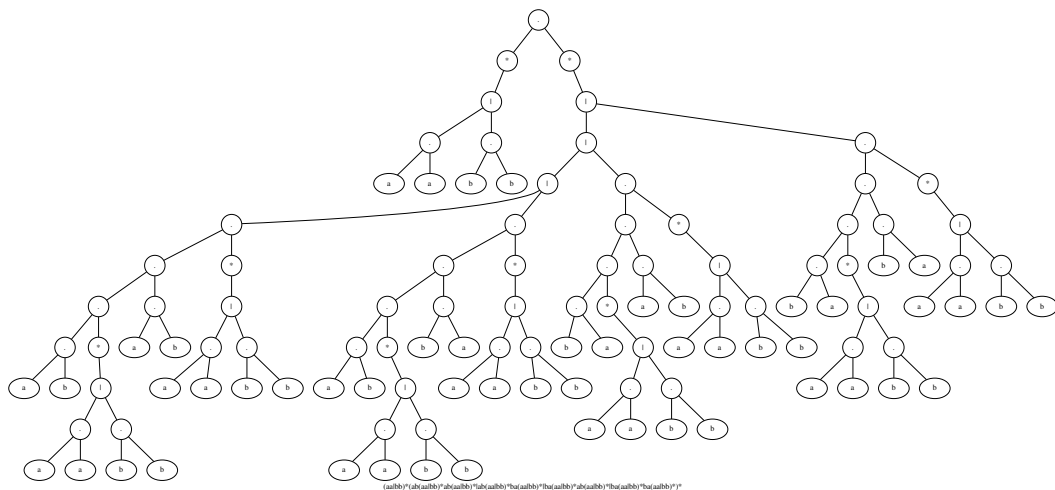
2. $(a|((b|d)|(c|e)))f$
 the simplified exp is $af|bf|cf|df|ef$



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\(03\)](mailto:hanfei.wang@gmail.com?subject=ID(03)) where the ID is your student id number.

2 偏导数 (assignment 04)

用 Thompson 算法转换正则表达式得到 NFA 有 $O(n)$ 个状态, 其中 n 是正则表达式的长度 (用到的字母的个数). 该 NFA 用到了很多 ϵ 转换边. 而偏导数法得到的 NFA 无 ϵ 转换边, 且状态数 $\leq n + 1$. 且由此得到的 DFA, 其状态数比子集构造法的要少, 即更接近最小状态 DFA.

2.1 线性形式

设 r 是正则表达式:

1. *linear form of r*:

$$lf(r) = N(r) \mid a_1 r_1 \mid a_2 r_2 \mid \dots \mid a_n r_n,$$

其中 a_i 是字母, r_i 是正则表达式.

$N(r) = 1$ iff $\epsilon \in L(r)$, 否则 $N(r) = 0$.

2. 如: $lf(x*xy) = 0 \mid x \mid x*xy \mid x \mid y$,

$$N(x*xy) = 0, a_1 = a_2 = x, r_1 = x*xy, r_2 = y.$$

a_i 和 a_j 可以是相同的字母, 即 *undeterministic*.

若所有的 a_i 都是两两不同的, 即 *deterministic linear form*.

3. 我们可利用分配率在求解过程中, 把相同的 a_i 所对应的 r_i 合并即可得到 *deterministic* 形式, 如:

$$lf(x*xy) = 0 \mid x (x*xy \mid y).$$

其中 $(x*xy \mid y)$ 相对于 x 的偏导数.

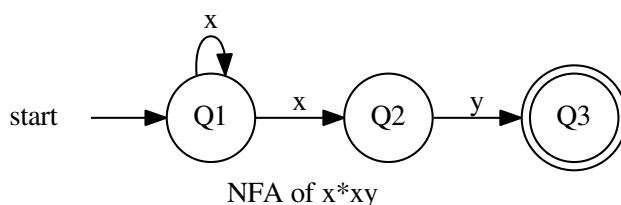
若将 r 和 r_i 看成 NFA 的状态 Q 和 Q_i , 则有 $trans(Q, a_i) = Q_i$. 若 $N(Q_i) = 1$, 状态 Q_i 为接受状态. 对求 Q 的 lf 过程中新产生的状态 Q_i 递归求解其 lf 直到没有新的状态产生, 即可得到 NFA (可证明 NFA 最多有 $n + 1$ 个状态, 即算法收敛, see detail in [partial_derivative.pdf](#)).

如: 设 $Q_1 = x*xy$, 则对应的 NFA:

$$Q_1 = 0 \mid x \mid Q_1 \mid x \mid Q_2 \mid Q_1 = x*xy$$

$$Q_2 = 0 \mid y \mid Q_3 \mid Q_2 = y$$

$$Q_3 = 1 \mid Q_3 = \epsilon$$



若 lf 重组, 则

$$Q_1 = 0 \mid x (Q_1 \mid Q_2).$$

设 $D_1 = Q_1, D_2 = Q_1 \mid Q_2 = x*xy \mid y$. 对 D_2 计算 lf :

$$D_2 = 0 \mid x \mid xx*y \mid x \mid y \mid y \mid \epsilon.$$

重组 D_2 为:

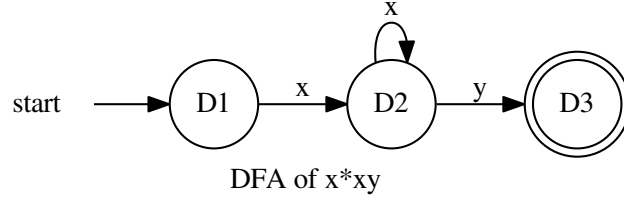
$$D_2 = 0 \mid x (xx*y \mid y) \mid y \mid \epsilon = 0 \mid x \mid D_1 \mid y \mid D_3$$

其中 $D_3 = \epsilon$. 这样对应的 DFA 为:

$$D_1 = 0 \mid x \mid D_2 \mid D_1 = x*xy$$

$$D_2 = 0 \mid x \mid D_2 \mid y \mid D_3 \mid D_2 = x*xy \mid y$$

$$D_3 = 1 \mid D_3 = \epsilon$$



2.2 线性形式的计算

lf 可用以下规则通过对正则表达式的抽象语法树的后序遍历计算出来 (即综合属性):

regex	lf
ε	1 \emptyset
\emptyset	0 \emptyset
x	0 $x \ \varepsilon$
$A \mid B$	$N(A) \vee N(B) \mid (lf(A) \mid lf(B))$ if $lf(A) = a_1 A_1 \mid \dots \mid a_n A_n$, $lf(B) = b_1 B_1 \mid \dots \mid b_m B_m$ then $lf(A) \mid lf(B) = a_1 A_1 \mid \dots \mid a_n A_n \mid b_1 B_1 \mid \dots \mid b_m B_m$
AB	0 $(lf(A))B$ if $N(A) = 0$ if $lf(A) = a_1 A_1 \mid \dots \mid a_n A_n$, then $(lf(A))B = a_1 (A_1 B) \mid \dots \mid a_n (A_n B)$
AB	$N(B) \mid (lf(A))B \mid lf(B)$ if $N(A) = 1$
A^*	1 $(lf(A))A^*$

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

$lf(x^*(y|xx)^*)$
 $= 0((y|xx)^*) \mid (lf(x^*))(y|xx)^* \mid lf((y|xx)^*)$
 $= 1 \mid ((lf(x))x^*)(y|xx)^* \mid lf((y|xx)^*)$
 $= 1 \mid x(x^*(y|xx)^*) \mid lf((y|xx)^*)$
 $= 1 \mid x(x^*(y|xx)^*) \mid (lf(y|xx))(y|xx)^*$
 $= 1 \mid x(x^*(y|xx)^*) \mid (lf(y) \mid lf(xx))(y|xx)^*$
 $= 1 \mid x(x^*(y|xx)^*) \mid (y \ \varepsilon \mid (lf(x))x)(y|xx)^*$
 $= 1 \mid x(x^*(y|xx)^*) \mid (y \ \varepsilon \mid (x \ \varepsilon)x)(y|xx)^*$
 $= 1 \mid x(x^*(y|xx)^*) \mid (y \ \varepsilon \mid x \ x)(y|xx)^*$
 $= 1 \mid x(x^*(y|xx)^*) \mid y(y|xx)^* \mid x \ x(y|xx)^*$
 $= 1 \mid x \ Q1 \mid y \ Q2 \mid x \ Q3$
 (where $Q1 = x^*(y|xx)^*$, $Q2 = (y|xx)^*$, $Q3 = x(y|xx)^*$)

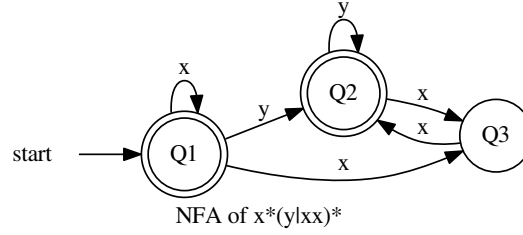
$lf(Q2) = lf((y|xx)^*)$
 $= 1 \mid (lf(y|xx))(y|xx)^*$
 $= 1 \mid (lf(y) \mid lf(xx))(y|xx)^*$
 $= 1 \mid (y \ \varepsilon \mid lf(x)x)(y|xx)^*$
 $= 1 \mid y(y|xx)^* \mid x \ x(y|xx)^*$
 $= 1 \mid y \ Q2 \mid x \ Q3$

$lf(Q3) = lf(x(y|xx)^*)$
 $= 0 \mid (lf(x))(y|xx)^*$
 $= 0 \mid x(y|xx)^*$
 $= 0 \mid x \ Q2$

so, we have NFA:

$Q1 = 1 \mid x \ Q1 \mid y \ Q2 \mid x \ Q3 \mid Q1 = x^*(y|xx)^*$

$Q2 = 1 \mid y \mid Q2 \mid x \mid Q3 \mid Q2 = (y|xx)^*$
 $Q3 = 0 \mid x \mid Q2 \mid Q3 = x(y|xx)^*$



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

对上例求 lf 过程中相同字母对应的右边因子用 $|$ 运算重组便可直接得到 DFA!

$lf(D1) = lf(x^*(y|xx)^*)$
 $= 0((y|xx)^*) \mid (lf(x^*)) (y|xx)^* \mid lf((y|xx)^*)$
 $= 1 \mid x (x^*(y|xx)^*) \mid y (y|xx)^* \mid x x(y|xx)^*$
 $= 1 \mid x (x^*(y|xx)^* | x(y|xx)^*) \mid y (y|xx)^* = 1 \mid x D2 \mid y D4$
 (where $D1 = x^*(y|xx)^*$, $D2 = x^*(y|xx)^* | x(y|xx)^*$, $D4 = (y|xx)^*$)

$lf(D2) = lf(x^*(y|xx)^* | x(y|xx)^*)$
 $= 1 \mid x (x^*(y|xx)^* | x(y|xx)^* | (y|xx)^*) \mid y (y|xx)^*$
 $= 1 \mid x D3 \mid y D4$
 (where $D3 = x^*(y|xx)^* | x(y|xx)^* | (y|xx)^*$)

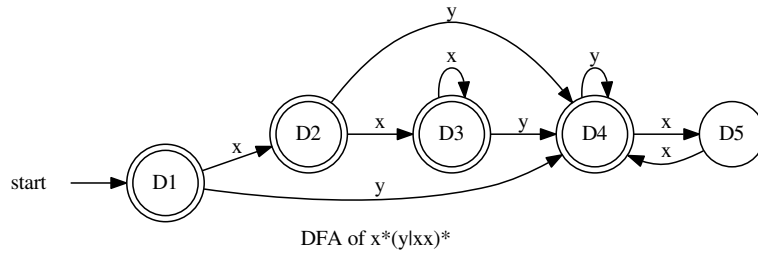
$lf(D3) = lf(x^*(y|xx)^* | x(y|xx)^* | (y|xx)^*)$
 $= 1 \mid x (x^*(y|xx)^* | x(y|xx)^* | (y|xx)^*) \mid y (y|xx)^*$
 $= 1 \mid x D3 \mid y D4$

$lf(D4) = lf((y|xx)^*)$
 $= 1 \mid y (y|xx)^* \mid x (x(y|xx)^*)$
 $= 1 \mid y D4 \mid x D5$
 (where $D5 = x(y|xx)^*$)

$lf(D5) = lf(x(y|xx)^*)$
 $= 0 \mid y (y|xx)^* \mid x (x(y|xx)^*)$
 $= 0 \mid x (y|xx)^* D4$
 $= 0 \mid x D4$

so, we have DFA:

$D1 = 1 \mid x D2 \mid y D4 \mid D1 = x^*(y|xx)^*$
 $D2 = 1 \mid x D3 \mid y D4 \mid D2 = x^*(y|xx)^* | x(y|xx)^*$
 $D3 = 1 \mid x D3 \mid y D4 \mid D3 = x^*(y|xx)^* | x(y|xx)^* | (y|xx)^*$
 $D4 = 1 \mid y D4 \mid x D5 \mid D4 = (y|xx)^*$
 $D5 = 0 \mid x D4 \mid D5 = x(y|xx)^*$



2.3 算法

lf 的数据结构见 [ast.h](#):

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

```
typedef LF *LF_PTR;
```

lf 的计算可抽象为下述两函数 (见 [nfa.c](#)):

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

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

对 NFA, union_method() 为:

```
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() 为:

```

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)
*/

```

对 DFA, union_method() 为:

```

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() 为:

```

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_plus(lf, lf2) = a1 (A1 C | B1 C) | ... | an (An C | Bn C)
*/
/* use distributive law to factorize regex, so more opportunity to
    test if two regex are equals */

```

则递归计算 lf 的函数如下 (TODO) :

```

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

    for lf of AB and nullabe(A) = 1, then

    lf(AB) = (lf(A))B + lf(B)

    so lf(A) and lf(B) are not states for AB. so they don't needed
    attribute the state number, hence the recursive call lf are
    linear_form(A, 0) and linear_form(B, 0).

    the same case for lf(A*).

*/
{
    /* TODO */
}

```

3 测试

to run the test, just `./reg2dfa < exN`, where N is number of the test example. `dot -Tpdf -o mdfa.pdf mdfa.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.
Thompson & 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.
Thompson & 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.
Thompson & Subset (JFLAP): NFA 3, DFA 3, MDFA 1.

3.4 ex4: $(ba^*b^*|ab^*a^*)^*$

NFA states: 5, DFA states 8, MDFA states 1.
Thompson & 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.

Thompson & 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.

Thompson & 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:

$(a(aa)^*|aa(aaa)^*|aaa(aaaaa)^*|aaaaa(aaaaaaa)^*|aaaaa(aaaaaaa)^*|aaa(aaaaaaa)^*|aaa(aaaaaaa)^*|aaa(aaaaaaa)^*)^*$

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

3.9 ex9:

$(a(aa)^*|aa(aaa)^*|aaa(aaaaa)^*|aaaaa(aaaaaaa)^*|aaa(aaaaaaa)^*|aaa(aaaaaaa)^*|aaa(aaaaaaa)^*)^*$

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

if we disactivate distributive law in `lf_concat_plus()`, `rege2dfa` will cause memory exhausted!

4 Discussions

分配率 (`lf_concat_plus()`) 可加速求 DFA 时的收敛速度, 即非常显著地减少 DFA 的状态数, 如:

$(a(aa)^*|aa(aaa)^*|aaa(aaaaa)^*|aaaaa(aaaaaaa)^*)^*$ 的 DFA 状态数为

1.

但若用没有分配率的 `lf_concat()` 作为 `seq_method()`, 则生成的 DFA 有 211 个状态.

但也有例外的情况. 如由 0 - 3 组成的且没有数字重复的正则表达式 (see [rep0_3.txt](#)) :

$(1|!)(01)^*(0|!)(2(0(10)^*(1|!)|1(01)^*(0|!)))^*(2|!)(3(2((0(10)^*(1|!)|1(01)^*(0|!))2)^*(1|!)(01)^*(0|!)|0(10)^*(1|!)|1(01)^*(0|!))(2(0(10)^*(1|!)|1(01)^*(0|!)))^*(2|!)))(3|!)$

使用分配率: DFA 12 states, MDFA 5 states. 不使用: DFA 13 states.

而 0 - 4 的无重复的正则表达式 (see [rep0_4.txt](#)):

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

使用分配率: DFA 的状态数激增导致 memory exhausted! 不使用: 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

regex	lf
$A \hat{\ } B$	$N(A) \wedge N(B) \ \ ((lf(A) \hat{\ } B) \ \ (A \hat{\ } lf(B)))$ where if $lf(A) = a1 \ A1 \ \ \dots \ \ an \ An$, then $lf(A) \hat{\ } B = a1 \ (A1 \hat{\ } B) \ \ \dots \ \ an \ (An \hat{\ } B)$
$A \& B$	$N(A) \wedge N(B) \ \ (lf(A)) \& (lf(B))$ if $lf(A) = a1 \ A1 \ \ \dots \ \ an \ An$, $lf(B) = a1 \ B1 \ \ \dots \ \ an \ Bn$ then $(lf(A)) \& (lf(B)) = a1 \ (A1 \& B1) \ \ \dots \ \ an \ (An \& Bn)$
$A - B$	$N(B) \wedge \neg N(B) \ \ (lf(A)) - (lf(B))$ if $lf(A) = a1 \ A1 \ \ \dots \ \ an \ An$, $lf(B) = a1 \ B1 \ \ \dots \ \ an \ Bn$ then $(lf(A)) - (lf(B)) = a1 \ (A1 - B1) \ \ \dots \ \ an \ (An - Bn)$

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 precdecence */
```

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 repeation 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 = 0 (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\(04\)](mailto:hanfei.wang@gmail.com?subject=ID(04)) where the ID is your student id number.

-hfwang

October 10, 2021