Lecture 2.4

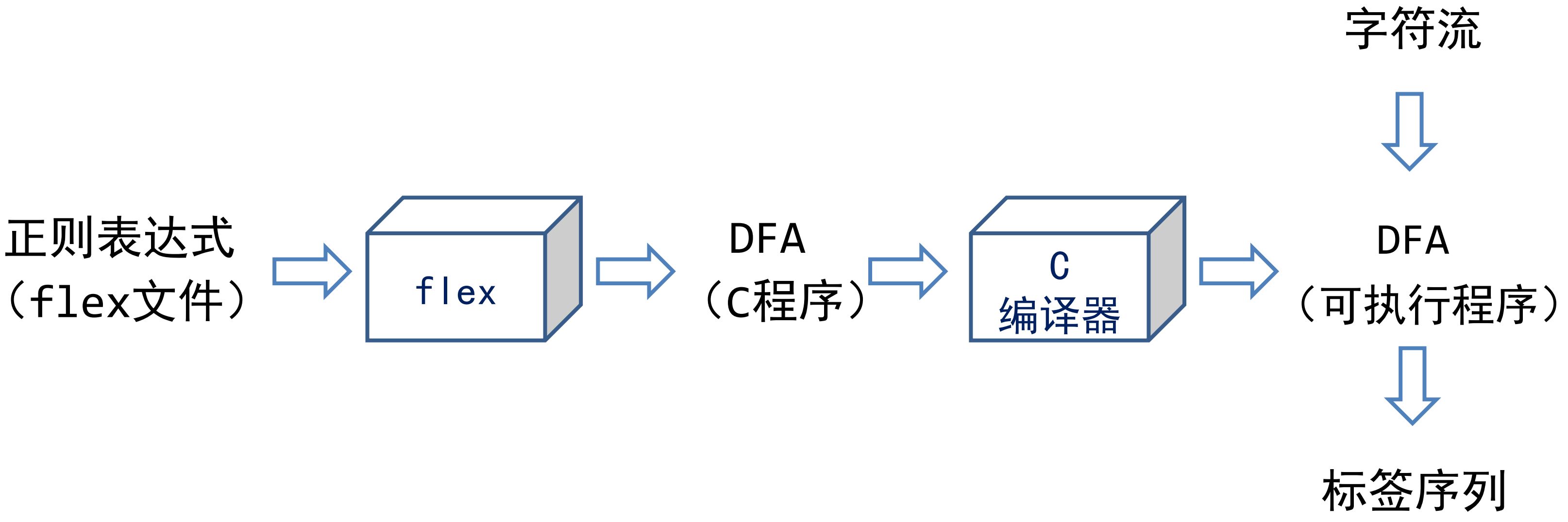
计算器实验

徐 辉

xuh@fudan.edu.cn

Lex

- 词法分析器生成工具：Lex(POSIX)
  - Flex(GNU)：fast lexical analyzer generator
- 通常和语法分析工具YACC(POSIX)/Bison (GNU)配合使用。



Flex文件

定义区

```
%{
    #include <stdio.h>
%}
%option outfile="Lexer.c" header-file="Lexer.h"

DIGIT   [0-9]
DIGITS  {DIGIT}+
FRAC    (\.{DIGITS})?
UNUM    {DIGITS}{FRAC}
%%
```

规则区

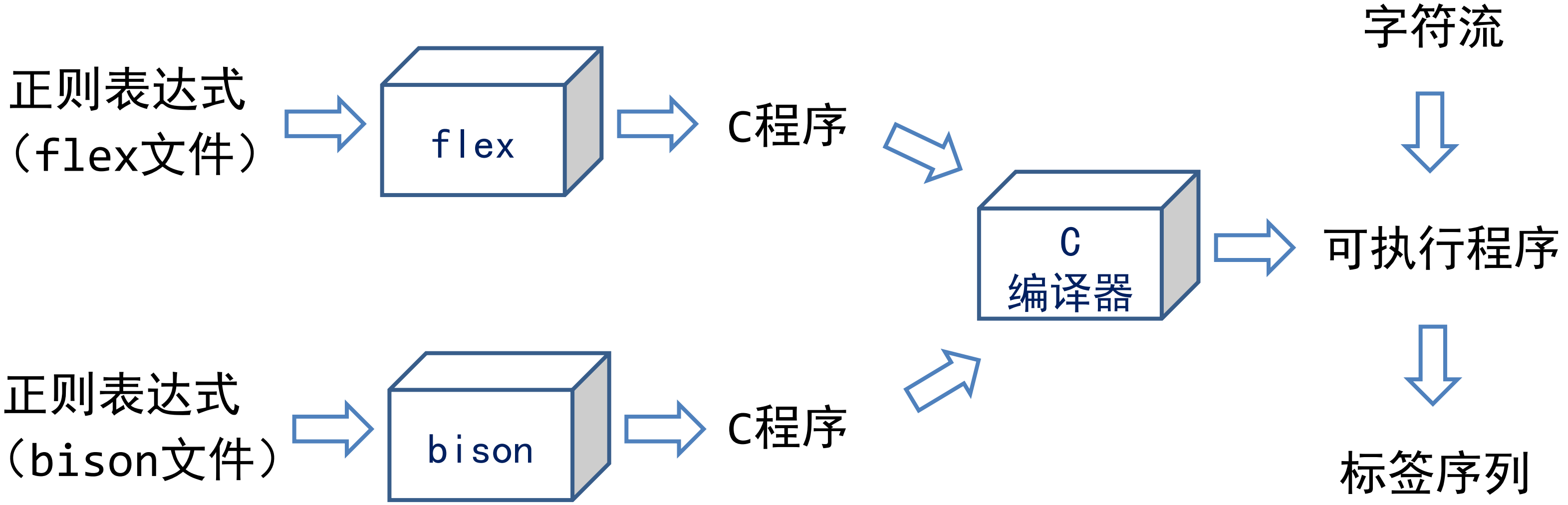
```
"+"          { return ADD; }
"-"          { return SUB; }
"*"          { return MUL; }
"/"          { return DIV; }
"^"          { return EXP; }
"("          { return LPAR; }
")"          { return RPAR; }
{UNUM}       { yylval->value = atof(yytext);
               printf("value = %f\n", yylval->value);
               return UNUM; }
%%
```

代码区

```
int yyerror(const char *msg) {
    fprintf(stderr, "Error: %s\n", msg);
    return 0;
}
```

Bison

- 语法分析工具YACC(POSIX)/Bison (GNU)
  - 默认采用LALR(1)解析
  - 支持LR(1)等方法



https://www.gnu.org/software/bison/manual/html_node/

Bison文件：方式一

```
%define api.pure
%lex-param   { yyscan_t scanner }
%parse-param { Expr **expression }
%parse-param { yyscan_t scanner }

%union {
    double value;
    Expr *expression;
}

%token ADD    "+"
%token SUB    "-"
%token MUL    "*"
%token DIV    "/"
%token EXP    "^"
%token LPAR   "("
%token RPAR   ")"
%token <value> UNUM "unum"

%type <expression> E
%type <expression> E1
%type <expression> E2
%type <expression> E3
%type <expression> OP1
%type <expression> OP2
%type <expression> OP3
%type <expression> NUM
```

```
%%
input
    : E { *expression = $1; }
E
    : E OP1 E1 { $$ = createOperation($1, $2, $3); }
    | E1     { $$ = $1; }
E1
    : E1 OP2 E2 { $$ = createOperation($1, $2, $3); }
    | E2     { $$ = $1; }
E2
    : E3 OP3 E2 { $$ = createOperation($1, $2, $3); }
    | E3     { $$ = $1; }
E3
    : NUM            { $$ = $1; }
    | "(" E ")"     { $$ = $2; }
NUM
    : UNUM            { $$ = createNumber($1); }
    | "-" UNUM        { $$ = createNumber(0-$2); }
OP1
    : "+"            { $$ = setOperator(AddNode); }
    | "-"            { $$ = setOperator(SubNode); }
OP2
    : "*"            { $$ = setOperator(MulNode); }
    | "/"            { $$ = setOperator(DivNode); }
OP3
    : "^"            { $$ = setOperator(ExpNode); }
```

Expr.c

```
typedef enum NodeType {
    OpNode,
    AddNode,
    SubNode,
    MulNode,
    DivNode,
    ExpNode,
    ValueNode
} NodeType;

typedef struct StExpr {
    double value;
    NodeType type;
    struct StExpr *op;
    struct StExpr *left;
    struct StExpr *right;
} Expr;
```

```
Expr* setOperator(NodeType type) {
    Expr* b = allocateExpr();
    if (b == NULL)
        return NULL;
    b->type = type;
    b->op = NULL;
    b->left = NULL;
    b->right = NULL;
    return b;
}
```

```
static Expr* allocateExpr() {
    Expr* b = (Expr *)malloc(sizeof(Expr));
    if (b == NULL)
        return NULL;
    b->type = ValueNode;
    b->value = 0;
    b->op = NULL;
    b->left = NULL;
    b->right = NULL;
    return b;
}
Expr* createNumber(double value) {
    Expr* b = allocateExpr();
    if (b == NULL)
        return NULL;
    b->type = ValueNode;
    b->value = value;
    printf("b = %f\n",value);
    return b;
}
Expr* createOperation(Expr *left, Expr *op, Expr *right) {
    Expr* b = allocateExpr();
    if (b == NULL)
        return NULL;
    b->type = OpNode;
    b->op = op;
    b->left = left;
    b->right = right;
    return b;
}
```

计算结果

```
double evaluate(Expr *e) {
    switch (e->type) {
        case ValueNode:
            return e->value;
        case OpNode:
            switch (e->op-type) {
                case AddNode:
                    return evaluate(e->left) + evaluate(e->right);
                case SubNode:
                    return evaluate(e->left) - evaluate(e->right);
                case MulNode:
                    return evaluate(e->left) * evaluate(e->right);
                case DivNode:
                    return evaluate(e->left) / evaluate(e->right);
                case ExpNode:
                    return pow(evaluate(e->left), evaluate(e->right));
                default:
                    printf("Inner Unreachable!\n");
                    return 0;
            }
        default:
            printf("Unreachable!\n");
            return 0;
    }
}
```

main.c

```c
Expr *getAST(const char *expr)
{
    Expr *expression;
    yyscan_t scanner;
    YY_BUFFER_STATE state;
    if (yylex_init(&scanner)) {
        printf("init lexer failure!!!\n");
        return NULL;
    }
    state = yy_scan_string(expr, scanner);
    if (yyparse(&expression, scanner)) {
        printf("parse expression failure!!!\n");
        return NULL;
    }
    yy_delete_buffer(state, scanner);
    yylex_destroy(scanner);
    return expression;
}

int main(void) {
    char expr[256];
    scanf("%s",expr);
    Expr *e = getAST(expr);
    if (e == NULL)
        return -1;
    double result = evaluate(e);
    printf("Result of '%s' is %f\n", expr, result);
    deleteExpr(e);
    return 0;
}
```

方式二：使用操作符优先级

```
%token ADD    "+"
%token SUB    "-"
%token MUL    "*"
%token DIV    "/"
%token EXP    "^"
%token LPAR   "("
%token RPAR   ")“
%token <value> UNUM "unum"

%type <expression> E
%type <expression> NUM

%left "+" "-"
%left "*" "/"
%right "^"
%%
```

```
input
    : E { *expression = $1; }
E
    : E "+" E { $$ = createOperation(AddNode, $1, $3); }
    | E "-" E { $$ = createOperation(SubNode, $1, $3); }
    | E "*" E { $$ = createOperation(MulNode, $1, $3); }
    | E "/" E { $$ = createOperation(DivNode, $1, $3); }
    | E "^" E { $$ = createOperation(ExpNode, $1, $3); }
    | NUM           { $$ = $1; }
    | "(" E ")"     { $$ = $2; }
NUM
    : UNUM            { $$ = createNumber($1); }
    | "-" UNUM        { $$ = createNumber(0-$2); }
```

对应Expr.c

```
typedef enum NodeType {
    OpNode,
    AddNode,
    SubNode,
    MulNode,
    DivNode,
    ExpNode,
    ValueNode
} NodeType;

typedef struct StExpr {
    NodeType type;
    double value;
    NodeType op;
    struct StExpr *left;
    struct StExpr *right;
} Expr;
```

```
static Expr* allocateExpr() {
    Expr* b = (Expr *) malloc (sizeof(Expr));
    if (b == NULL)
        return NULL;
    b->left = NULL;
    b->right = NULL;
    return b;
}
Expr* createNumber(double value) {
    Expr* b = allocateExpr();
    if (b == NULL)
        return NULL;
    b->type = ValueNode;
    b->value = value;
    return b;
}
Expr* createOperation(NodeType op, Expr *left, Expr *right) {
    Expr* b = allocateExpr();
    if (b == NULL)
        return NULL;
    b->op = op;
    b->left = left;
    b->right = right;
    return b;
}
void deleteExpr(Expr *b) {
     if (b == NULL)
         return;
     deleteExpr(b->left);
     deleteExpr(b->right);
     free(b);
}
```