

```

Q&D top-down parser
(aka LR(0) for arith
expressions:
EBNF grammar: <exp> ::= <op> <exp> <exp>
<op> ::= <term> ('+' | '<term>') *
<term> ::= <factor> ('*' | '<factor>') <factor>
<factor> ::= const | '(' <exp> ')'

int const = 556, EOI;
const_type;
TOKEN_TYPE lookahead;
int lval;

TOKEN_TYPE get_token();
// grabs next token (consumes it)
// get lookahead if it's type
// and set lval if appropriate

Recursive descent parser
(kind of LR(0) parser)

One function for each non-terminal:
job is to read a prefix of the
input that is the yield of a
parse tree rooted with that non-terminal
Precondition: lookahead is just the
first token of that input
Postcondition: lookahead is set
to the first token after the
yield
Semantic value (if any) is returned

void input();
int exp();
int term();
int factor();

```

Feb 16-11:01 AM

```

int exp()
{
    int sum = term();
    while (lookahead == '+')
    {
        lookahead = get_token();
        sum += term();
    }
    return sum;
}

int term()
{
    int product = factor();
    while (lookahead == '*')
    {
        lookahead = get_token();
        product *= factor();
    }
    return product;
}

int factor()
{
    int temp;
    if (lookahead == CONST) {
        temp = lval;
        lookahead = get_token();
        return temp;
    }
    if (lookahead == '(') {
        lookahead = get_token();
        temp = exp();
        if (lookahead == ')') {
            lookahead = get_token();
            return temp;
        }
    }
    return temp;
}

```

Feb 16-11:20 AM

```

for each (id, const or like non-terminal)
    emit(i);
}

void input()
{
    lookahead = get_token();
    if (lookahead != '(' && lookahead != CONST) {
        // error
        emit(i);
    }
    if (lookahead == '(') {
        emit(i);
    }
    if (lookahead == CONST) {
        emit(i);
    }
}

int get_token()
{
    int c = get_char();
    while (c == ' ' || c == '\n') c = get_char();
    if (c == '(' || c == ')') {
        sum = c - '0';
        while (isdigit(c)) {
            sum = sum * 10 + c - '0';
        }
        return (c, sum);
    }
    return (c, sum);
}

```

Feb 16-11:37 AM

```

if (c == EOF)
    return EOI;
return c; // final token
}

FIRST & FOLLOW sets:
Fix a grammar G
G is also its start symbol
only occurs in a single production
S → A
(A is a non-terminal)

G = (N, T, P, S)
den version of G (equivalent):
G' = (N ∪ {S'}, T, P ∪ {S' → S})
Assume deriv grammar with top
prod S' → S
A pruned parse tree of G
is part of a complete parse tree
with root or more sub-tree pruned
Exp'n: any parse tree with S' as
root (not necessarily complete)

The yield of a pruned parse tree
is called a sentential form
E+T+T

```

Feb 16-11:53 AM

Let X be any grammar symbol:

$FIRST(X)$ = the set of all possible terminals that start some yield of a parse tree with root X .

If X is ϵ , ϵ is a yield, then $\epsilon \in FIRST(X)$

$FIRST(X_1 X_2 \dots X_k)$:

```

{
    ret = FIRST(X1);
    if  $\epsilon \in ret$  then
        ret = (ret - { $\epsilon$ }) ∪ FIRST(X2);
    if  $\epsilon \in ret$  then
        ret = (ret - { $\epsilon$ }) ∪ FIRST(X3);
    ...
    if  $\epsilon \in ret$  then
        ret = (ret - { $\epsilon$ }) ∪ FIRST(Xk);
    return ret;
}

```

Feb 16-12:04 PM