Expression Grammar

```
expression → term {”+” term}.
term → factor {”+” factor}.
factor → ”(” expression ”)” | identifier.
```

Corresponding Parser

```
void expression() {
    term();
    while (sym==plus) {
        next(); // consume token
        term();
    }
}

void term() {
    factor();
    while (sym==times) {
        next(); // consume token
        factor();
    }
}

void factor() {
    if (sym==identifier)
        next(); // consume token
    else if (sym==leftparen) {
        next(); // consume token
        expression();
        if (sym==rightparen)
            next(); // consume token
        else error();
    } else error();
}
```

Expression Grammar

```
expression → term {"+" term}.
term → factor {"+" factor}.
factor → "(" expression ")" | number.
```

Interpreter

```
int expression() {
    int val = term();
    while (sym==plus) {
        next();
        val += term();
    }
    return val;
}

int term() {
    int val = factor();
    while (sym==times) {
        next();
        val *= factor();
    }
    return val;
}

int factor() {
    int val;
    if (sym==number)
        val = Scanner.val; next();
    else if (sym==leftparen) {
        next();
        val = expression();
        if (sym==rightparen) next(); else error();
    } else error();
    return val;
}
```

Expression Grammar

```
expression → term {"+" term}.
term → factor {"+" factor}.
factor → "(" expression ")" | number | identifier.
```

Compiler for 0-Address Machine

```
void expression() {
    term();
    while (sym==plus) {
        next();
        term();
        put(addop);
    }
}

void term() {
    factor();
    while (sym==times) {
        next();
        factor();
        put(mulop);
    }
}

void factor() {
    if (sym==number)
        put(loadimmediate+Scanner.val); next();
    if (sym==identifier)
        put(load+LookupAddress(Scanner.ident)); next();
    else if (sym==leftparen) {
        next();
        expression();
        if (sym==rightparen) next(); else error();
    } else error();
}
```

Instruction Assembly: Format F1

6	5	5	16
op	a	b	c

Assembly Routine

```
void PutF1(int op, int a, int b, int c) {
    buf[pc++] = op ≪ 26 | a ≪ 21 | b ≪ 16 | c & 0xffff;
}
```

Code for a RISC Machine Using a “Register Stack” for Intermediate Results

```
void expression() {
    term();
    while (sym==plus) {
        next();
        term();
        putF1(ADD, sp−1, sp−1, sp−−);
    }
}

void term() {
    factor();
    while (sym==times) {
        next();
        factor();
        PutF1(MUL, sp−1, sp−1, sp−−);
    }
}

void factor() {
    if (sym==identifier) {
        PutF1(LDW, ++sp, basereg, LookupAddress(Scanner.ident)));
        next();
    else if (sym==number) {
        PutF1(ADDI, ++sp, 0, Scanner.val);
        next();
    } else if (sym==leftparen) ...
}
```

Delayed Code Generation: Data Structure for Intermediate Results

```
class Result{
    int kind; // const, var, reg, condition
     int value; // value if it is a constant
     int address; // address if it is a variable
     int regno; // register number if it is a reg
}
```

Package a Factor Into a Result

```
Result factor() { Result x = new Result();
    if (Scanner.token==identifier) {
        x.kind = var;
        x.address = LookupAddress(Scanner.ident));
        next();
    } else if (Scanner.token==number) {
        x.kind = const;
        x.value = Scanner.intval;
        next();
    } else if (Scanner.token==leftparen) {
        ...
    } else SyntaxError()
    return x;
}
```

Load a Result Into Register (No Matter What Kind Of Result It Is Currently)

The auxiliary function AllocateReg uses a register reservation table (which can simply be an array of Boolean) to keep track of which registers are in use. If the value is constant zero, we use register zero instead.

```
void load(Result x) {
    if (x.kind==var) {
        x.regno = AllocateReg();
        PutF1(LDW, x.regno, basereg, x.address);
        x.kind = reg;
    } else if (x.kind==const) {
        if (x.value==0) x.regno = 0;
        else{
            x.regno = AllocateReg();
            PutF1(ADDI, x.regno, 0, x.value);
        }
        x.kind = reg;
    }
}
```

Expression & Term

```
Result expression() { Result x, y; int op;
    x = term();
    while ((Scanner.token==plus)||(Scanner.token==minus)) {
        op = Scanner.token; next();
        y = term();
        Compute(op, x, y);
    } return x;
}

Result term() { Result x, y; int op;
    x = factor();
    while ((Scanner.token==times)||(Scanner.token==div)) {
        op = Scanner.token; next();
        y = factor();
        Compute(op, x, y);
    } return x;
}
```

Compute Avoids Generating Code When Both Operands Are Constant

Two arrays opCode and opCodeImm are used to translate between Scanner token values and actual processor opcodes. Register zero is always zero, but it must not be the target of an operation, so if the first argument is a constant zero, we may need to allocate a register for it after all.

```
void Compute(int op, Result x, y) {
    if ((x.kind==const)&&(y.kind==const) {
        if (op==plus) x.value += y.value;
        else if(op==minus) x.value -= y.value;
        else if(op==times) x.value *= y.value;
        else if(op==div) x.value /= y.value;
    } else {
        load(x);
        if (x.regno == 0) { x.regno=AllocateReg(); PutF1(ADD, x.regno, 0, 0); }
        if (y.kind==const)
            PutF1(opCodeImm[op], x.regno, x.regno, y.value);
        else {
            load(y);
            PutF1(opCode[op], x.regno, x.regno, y.regno);
            DeAllocate(y);
        }
    }
}
```