

Given the grammar

- 1 Start ::= Expr
- 2 Expr ::= + Expr Expr
- 3 Expr ::= - Expr Expr
- 4 Expr ::= * Expr Expr
- 5 Expr ::= Opnd
- 6 Opnd ::= **INTCONST**

	First	Follow
Start	+ - * INTCONST	eof
Expr	+ - * INTCONST	eof + - * INTCONST
Opnd	INTCONST	eof + - * INTCONST

(1) Compute First and Follow sets...

(2) Is the grammar LL(1)? Here's the LL(1) parse table...

	eof	+	-	*	INTCONST
Start		1	1	1	1
Expr		2	3	4	5
Opnd					6

No conflicts exist, so the grammar is LL(1). Now let's show how the string "+ 3 * 5 6" is parsed.

- ([eof, **Start**], + 3 * 5 6, + Expr Expr) =>
 ([eof, Expr, Expr, **+**], + 3 * 5 6, next input + pop) =>
 ([eof, Expr, **Expr**], 3 * 5 6, Opnd) =>
 ([eof, Expr, **Opnd**], 3 * 5 6, **INTCONST**) =>
 ([eof, Expr, **INTCONST**], 3 * 5 6, next input + pop) =>
 ([eof, **Expr**], * 5 6, * Expr Expr) =>
 ([eof, Expr, Expr, *****], * 5 6, next input + pop) =>
 ([eof, Expr, **Expr**], 5 6, Opnd) =>
 ([eof, Expr, **Opnd**], 5 6, **INTCONST**) =>
 ([eof, Expr, **INTCONST**], 5 6, next input + pop) =>
 ([eof, **Expr**], 6, Opnd) =>
 ([eof, **Opnd**], 6, **INTCONST**) =>
 ([eof, **INTCONST**], 6, next input + pop) =>
 ([eof], eof, accept).

3. Write a parser that either accepts or errors.

```
driver()
{
    -- initialization
    next_token();
    Start();
}

Start()
{
    if(INTCONST == token
        || PLUS == token
        || MINUS == token
        || TIMES == token) { Expr(); }
    else { error("Bad Start"); }
}

Expr()
{
    if(PLUS == token
        || MINUS == token
        || TIMES == token)
    { next_token();
      Expr();
      Expr();
    } else if(INTCONST == token)
    { Opnd(); }
    else { error("Bad Expr"); }
}

Opnd()
{
    if(INTCONST == token)
    { next_token(); }
    else
    { error("Bad Opnd"); }
}
```

4. Write a parser that acts as a calculator.

```
driver()
{
    -- initialization
    next_token();
    print -> Start();
}

int Start()
{
    if(INTCONST == token
        || PLUS == token
        || MINUS == token
        || TIMES == token)
    { return Expr(); }
    else { error("Bad Start"); }
}

int Expr()
{ int temp;
  if(PLUS == token) {
    temp = Expr();
    return temp + Expr();
  } else if(MINUS == token) {
    temp = Expr();
    return temp - Expr();
  } else if(TIMES == token) {
    temp = Expr();
    return temp * Expr();
  } else if(INTCONST == token) {
    return Opnd();
  } else { error("Bad Expr"); }
}

int Opnd()
{ int temp
  if(INTCONST == token) {
    temp = token.value;
    next_token();
    return temp;
  } else { error("Bad Opnd"); }
}
```