

# The Grammar

```
expr      ::= base_expr rec_expr
rec_expr  ::= base_expr rec_expr |
base_expr ::= postfix_expr rec_base
rec_base  ::= binop postfix_expr rec_base |
postfix_expr ::= atomic rec_postfix
rec_postfix ::= incrop rec_postfix |
atomic     ::= prefix_expr | lvalue | parenvalue | num
prefix_expr ::= incrop atomic
lvalue     ::= F atomic
parenvalue ::= ( expr )
num        ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
incrop    ::= ++
binop     ::= + | -
```

# First Sets

```
expr          = {++,--,F,(,0,1,2,3,4,5,6,7,8,9}
rec_expr      = {++,--,F,(,0,1,2,3,4,5,6,7,8,9,ε}

base_expr     = {++,--,F,(,0,1,2,3,4,5,6,7,8,9}
rec_base      = {+,-,ε}

postfix_expr  = {++,--,F,(,0,1,2,3,4,5,6,7,8,9}
rec_postfix   = {++,--,ε}

atomic         = {++,--,F,(,0,1,2,3,4,5,6,7,8,9}

prefix_expr   = {++,--}
lvalue         = {F}
parenvalue    = {(,)}

num           = {0,1,2,3,4,5,6,7,8,9}
incrop        = {++,--}
binop         = {+,-}
```

## Follow Sets

```
expr          = {$}
rec_expr      = {$}

base_expr     = {++,--,F,(,0,1,2,3,4,5,6,7,8,9}
rec_base      = {++,--,F,(,0,1,2,3,4,5,6,7,8,9}

postfix_expr = {+, -}
rec_postfix  = {+, -}

atomic        = {++,--}

prefix_expr   = {++,--}
lvalue         = {++,--}
parenvalue    = {++,--}

num           = {++,--}
incrop        = {++,--,F,(,0,1,2,3,4,5,6,7,8,9}
binop         = {++,--,F,(,0,1,2,3,4,5,6,7,8,9}
```

# Predictive Parsing Table

	F	(	num	incrop	binop	\$
expr	expr -> base_expr rec_expr	expr -> base_expr rec_expr	expr -> base_expr rec_expr	expr -> base_expr rec_expr		
rec_expr	rec_expr -> base_expr rec_expr	rec_expr -> base_expr rec_expr	rec_expr -> base_expr rec_expr	rec_expr -> base_expr rec_expr		rec_expr-> ε
base_expr	base_expr -> postfix_expr rec_base	base_expr -> postfix_expr rec_base	base_expr -> postfix_expr rec_base	base_expr -> postfix_expr rec_base		
rec_base					rec_base -> binop postfix_expr rec_base	rec_base -> ε
postfix_expr	postfix_expr -> atomic rec_postfix	postfix_expr -> atomic rec_postfix	postfix_expr -> atomic rec_postfix	postfix_expr -> atomic rec_postfix		
rec_postfix				rec_postfix -> incrop rec_postfix		rec_postfix -> ε
atomic	atomic -> value	atomic -> parenvalue	atomic -> num	atomic -> prefix_expr		
prefix_expr				prefix_expr -> incrop atomic		
Ivalue	Ivalue -> F atomic					
parenvalue		parenvalue -> ( expr )				

# Argument for it being LL(1)

First and foremost, this grammar features neither First/First conflicts, nor First/Follow conflicts. This is evidenced by the fact that there is no overlap within the Predictive Parsing table. However, this may also be derived from the base grammar - no token features two of the same terminal within its first set. Similarly, no nullable token has any overlap between its first and follow sets.

At a glance, it may appear that there is a Follow/Follow conflict within the incrop token. This is because both the atomic token and rec\_postfix token - of which, the incrop token's follow set is derived from - themselves feature {++,--} within their Follow sets. However, I would note that there is no way to construct a string that would both be considered valid within the grammar, and would lead to ambiguity between whether an atomic or a postfix\_expr token follows an incrop. Put even simpler, as incrop is simply an alias for the two terminal characters ++ or --, both of its occurrences within the grammar can be factored out without difficulty. Consider the following amendment to the grammar:

```
rec_postfix ::= ++ rec_postfix | -- rec_postfix |
prefix_expr ::= ++ atomic | -- atomic
```

Here, it becomes readily apparent that there exists no meaningful Follow/Follow conflict, as ++ and -- are both terminals irrelevant to Follow computation.

With this in mind, I will defend the use of incrop as a convenient shorthand for ++ or --, as it bears no influence over whether this grammar is LL(1).

Finally, there is no left recursion within the language. All possible instances of left recursion have been factored into instances of right recursion instead. This is demonstrated by the fact that it is possible to compute the First and Follow sets. Alternatively, one can simply traverse the predictive parsing table to discover that for every row's token, there exists no outcome in which the next token visited is the same as the row's.

In conclusion, because there is no string that can create an ambiguity between first/follow sets, there exists no ambiguity in how the grammar is parsed, and there exists no left recursion within the grammar, this is LL(1).

As a side-note, though this is far less formal of a proof, I believe my recursive descent parser, written in java and based solely off of this grammar, also evidences this. Upon lexing a string, it stores each lexical token parsed in a queue. There is no instance within my code in which I ever check the lexical token beyond whatever is currently at the head of the queue, and my program is capable of parsing any valid string within the above grammar.