What is **vim**?

- It is **vi** i**M**proved!

  ... but what is vi?

- It is a *visual* text editor.

  ... what the heck is a nonvisual text editor??!?!

- Let's take a look at one!
  - ed - The original unix **ed**itor.

Example`ed` session

- The **ed** editor was developed in August 1969
  - Same month as Woodstock!
- Ken Thompson was the original author
- We're only looking at it for historical context that tells us two things:
  1. Ken Thompson and Dennis Ritchie created the Unix operating system and the C programming language in ed. Think about that...
  2. Having a visual mode that shows you the contents of your file as you are working on it is a MAJOR improvement

```
$ ed
a
This is an example of the ed editor.
You are now adding lines to a buffer.
End mode by a line with a period.
.
w some.txt
130
1p
This is an example of the ed editor.
1,3p
This is an example of the ed editor.
You are now adding lines to a buffer.
End mode by a line with a period.
2d
1,2p
This is an example of the ed editor.
End mode by a line with a period.
q
```

a: append mode

. on its own line ends append mode

w <filename> writes the contents to the file

1p prints the first line

1,3p prints lines 1 through 3

2d deletes the second line

q quits

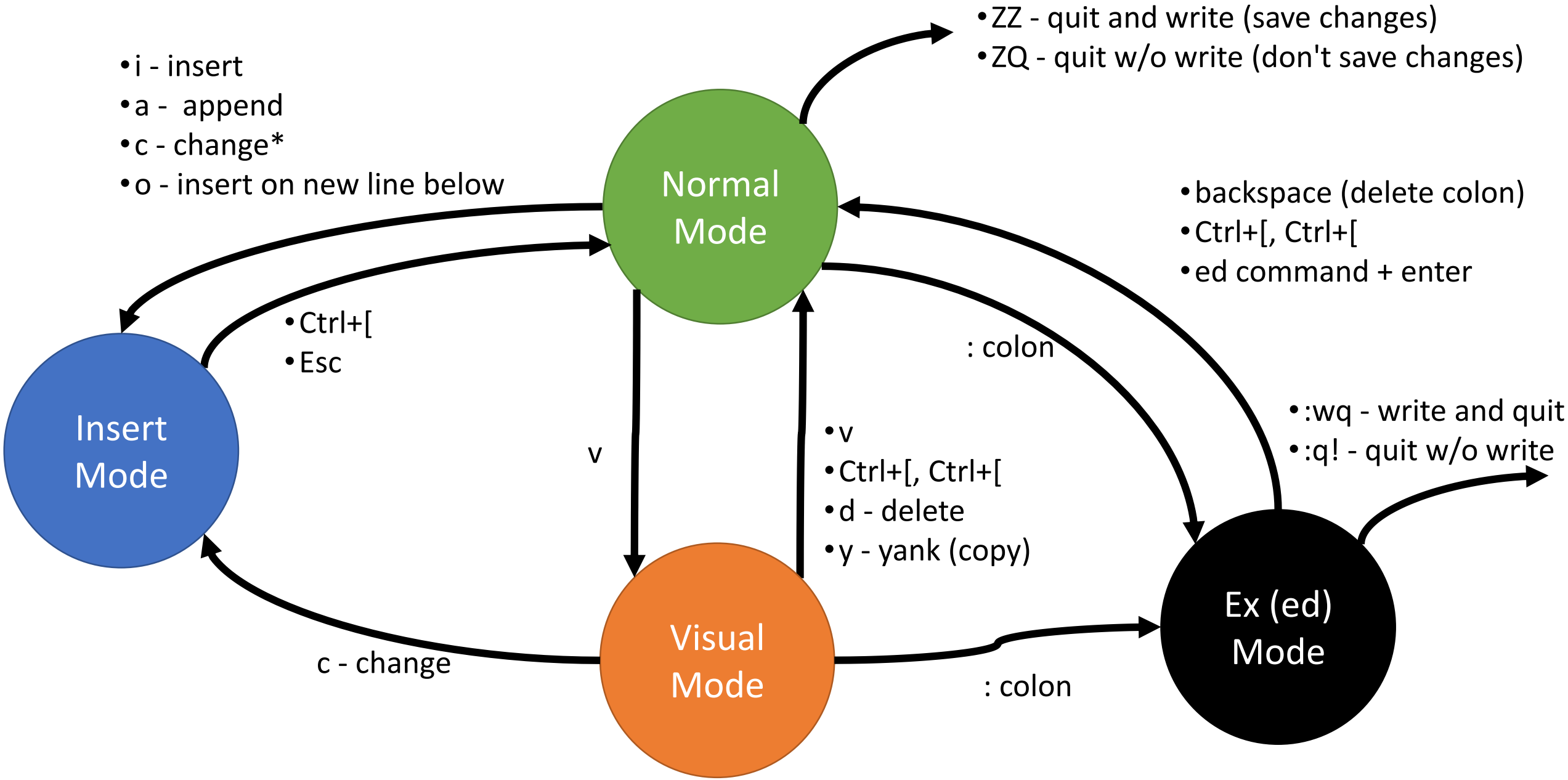
What is **vim**?

- **vim (1991)** - A **vi**sual text editor whose lineage traces back to ed/ex, and its direct predecessor **vi** (1976).
  - Still *very* actively developed today! Version 8's last stable release was in 2018.

- It is a modal / stateful text editor.
  - *normal* mode - you are speaking a little command language with keys
  - *insert* mode - your keystrokes are inserting text into the "buffer"
  - *visual* mode - mode allowing selection of text
  - *ex* (ed) mode - a colon : followed by a command is **ed's** language!

- Once you start "speaking" vim's language, you'll feel like a wizard.

State Transitions in **vim**



Exploring **vim**'s Little Command Language

- A **grammar** is a formal specification of the **rules** that define a structured language's **syntax**.

- The same ideas and specifications of grammars apply to little languages just the same as general purpose programming languages.

- Our exploration of grammars will begin more intuitively than formally and more pragmatically than theoretically.
  - In COMP455 you will explore the deep theoretical basis of grammars and their formal boundaries, limitations, and characteristics.

What makes up a grammar?

1. **Terminals** - the elementary symbols of a language (i.e. letters, numbers, whitespace, and reserved words)

2. **Nonterminals** - "syntactic variables" that are replaced by production rules

3. **Production Rules** - a nonterminal "name" for the rule, followed by **->**, and a sequence of terminals, nonterminals, and alternations |

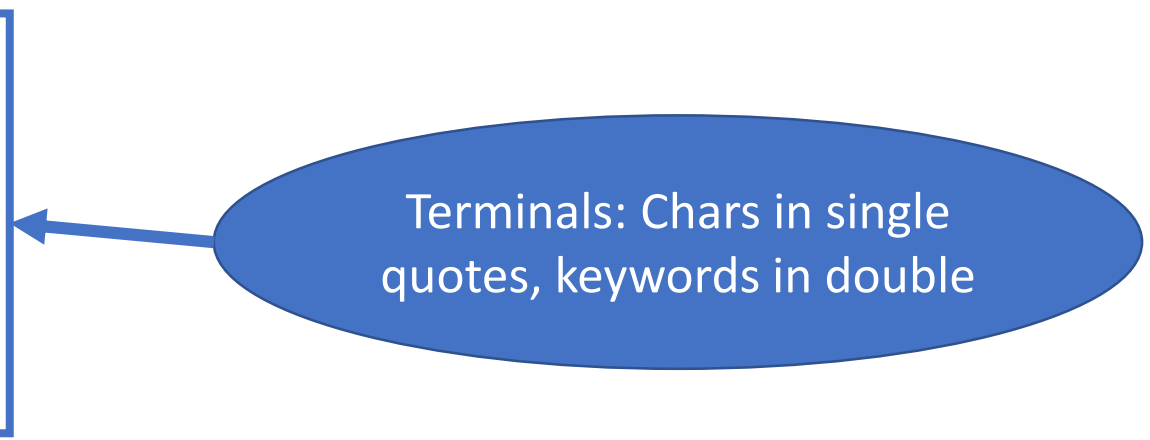
4. **Start symbol** - The nonterminal symbol the grammar starts with

Example Grammar

```
command          ->   cursor_to

cursor_to        ->   location

location         ->   line-below | line-above |
                      char-before | char-after |
                      /* ... other locations ... */

line-below       ->   'j'
line-above       ->   'k'
char-before      ->   'h'
char-after       ->   'l'
```

Example Grammar - Terminals

```
command          ->   cursor_to

cursor_to        ->   location

location         ->   line-below | line-above |
                      char-before | char-after |
                      /* ... other locations ... */

line-below       ->   'j'
line-above       ->   'k'
char-before      ->   'h'
char-after       ->   'l'
```



Example Grammar - Nonterminals

```
command     ->   cursor_to

cursor_to   ->   location

location    ->   line-below | line-above |
                 char-before | char-after |
                 /* ... other locations ... */
```

```
line-below    ->   'j'
line-above    ->   'k'
char-before   ->   'h'
char-after    ->   'l'
```

Nonterminals: Words like variable names.

Example Grammar - Production Rules

```
command          ->   cursor_to
```

```
cursor_to        ->   location
```

```
location         ->   line-below | line-above |
                      char-before | char-after |
                      /* ... other locations ... */
```

```
line-below       ->   'j'
line-above       ->   'k'
char-before      ->   'h'
char-after       ->   'l'
```

A production rule defines the meaning of a nonterminal.

Example Grammar - Alternation "OR"

```
command          ->    cursor_to

cursor_to        ->    location

location         ->    line-below | line-above |
                       char-before | char-after |
                       /* ... other locations ... */

line-below       ->    'j'
line-above       ->    'k'
char-before      ->    'h'
char-after       ->    'l'
```

The vertical bar is read as **OR**:
A *location* nonterminal can be substituted with any one of line-below **OR** line-above OR char-before OR char-after OR ...

Example Grammar - Start Symbol

```
command      ->   cursor_to

cursor_to    ->   location

location     ->   line-below | line-above |
                  char-before | char-after |
                  /* ... other locations ... */

line-below   ->   'j'
line-above   ->   'k'
char-before  ->   'h'
char-after   ->   'l'
```

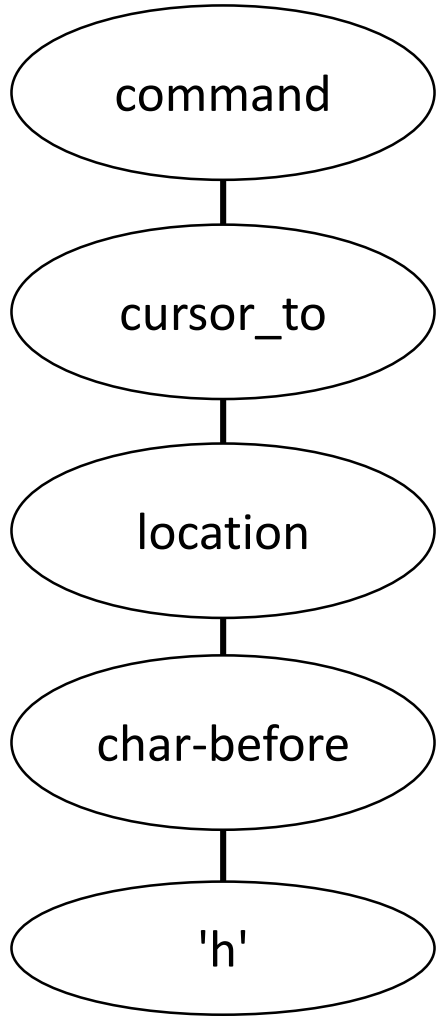
We'll signify the start symbol with an underline. This is what we're ultimately trying to derive.

Example Grammar - Parsing an Input

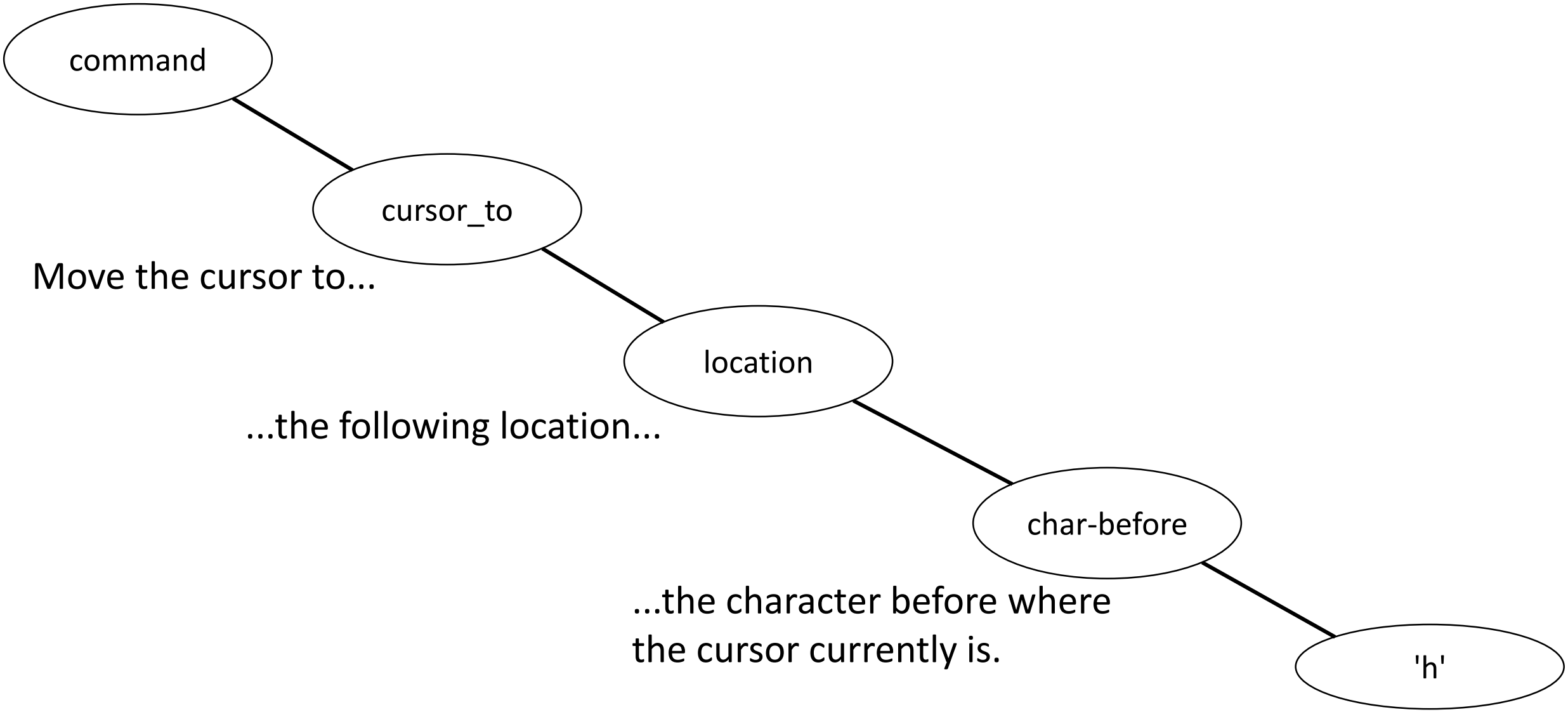
```
command      ->    cursor_to

cursor_to    ->    location

location     ->    line-below | line-above |
                   char-before | char-after |
                   /* ... other locations ... */

line-below   ->    'j'
line-above   ->    'k'
char-before  ->    'h'
char-after   ->    'l'
```

Suppose you enter the input string 'h'. Is there a valid parsing given this grammar?

In other words, is there a valid sequence of replacements we can make of nonterminals, starting from command, that result in the input string?



Intentionally chosen nonterminal names and clever organization of production rules in a grammar helps us derive meaning from inputs...



There are lots of location terminals in **vim**!

```
command     ->   cursor_to

cursor_to   ->   LOCATION
```

To keep the information on the slides manageable, we're going to cheat with this all caps convention that assumes there are additional rules here not shown (in table).

These are the most commonly useful location keys (terminals) in vim's little language.

Location	Terminal
line below	j
line above	k
char left	h
char right	l
first char of line	^
last char of line	$
next word	w
previous word	b
end of next word	e
next occurrence of word	*
previous occurrence of word	#
start of file	gg
end of file	G

Operations carry out actions on your text.

```
command     ->   cursor_to | operation

cursor_to   ->   LOCATION

operation   ->   verb cursor_to

verb        ->   change | delete | yank
change      ->   'c'
delete      ->   'd'
yank        ->   'y'
```

A command is either a cursor_to motion OR an operation.

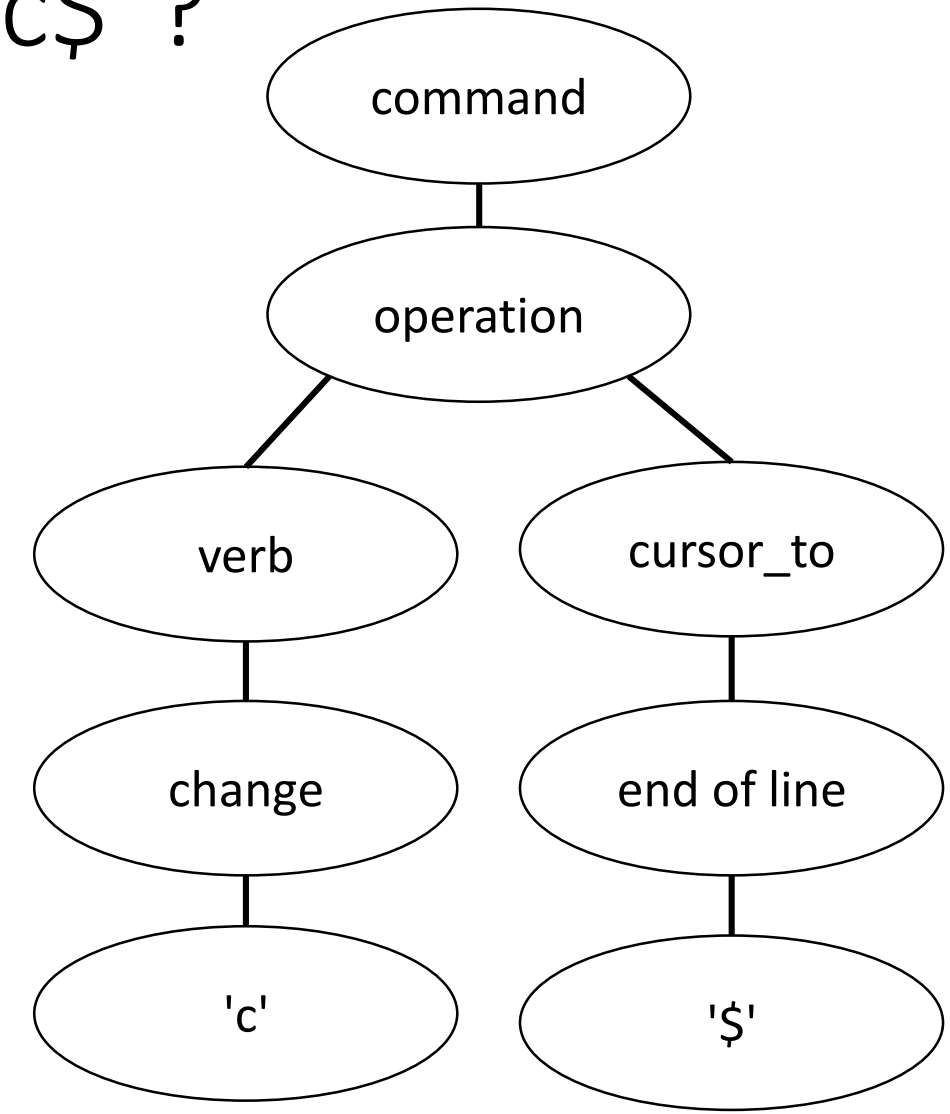
An operation is a verb followed by a cursor_to.

A verb is either:

- Change - removes text, transitions to insert mode
- Delete - removes text
- Yank - copies text

How would the grammar parse "c$"?

```
command     ->    cursor_to | operation

cursor_to   ->    LOCATION

operation   ->    verb cursor_to

verb        ->    change | delete | yank
change      ->    'c'
delete      ->    'd'
yank        ->    'y'
```



"Change from cursor to end of line."

Our grammar now has two high-level commands!

```
command     ->   cursor_to | operation

cursor_to   ->   LOCATION

operation   ->   VERB cursor_to
```

Verb	Terminal
change	c
delete	d
yank	y

Location	Key
line below	j
line above	k
char left	h
char right	l
first char of line	^
last char of line	$
next word	w
previous word (back)	b
end of next word	e
next occurrence of word	*
previous occurrence of word	#
start of file	gg
end of file	G

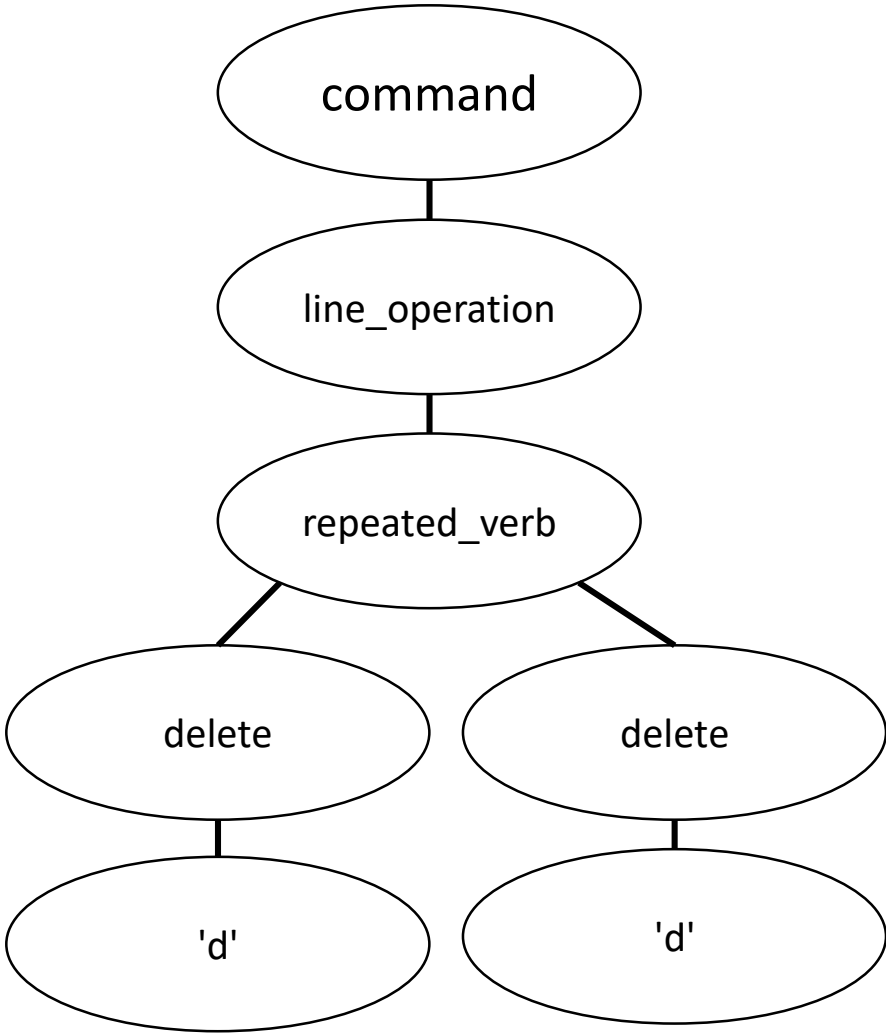
Line operations carry out a verb on a complete line.

```
command          ->   cursor_to | operation | line_operation

cursor_to        ->   LOCATION

operation        ->   VERB cursor_to

line_operation->   repeated_verb

repeated_verb ->   delete delete |
                   change change |
                   yank yank
```

A repeated_verb is either
a delete followed by a delete OR
a change followed by a change OR
a yank followed by a yank.

How would the grammar parse "dd"?

```
command          ->    cursor_to | operation | line_operation

cursor_to        ->    LOCATION

operation        ->    VERB cursor_to

line_operation   ->    repeated_verb

repeated_verb    ->    delete delete |
                       change change |
                       yank yank
```



Notice the language reuses and composes concepts...

```
command          ->    cursor_to | operation | line_operation

cursor_to        ->    LOCATION

operation        ->    VERB cursor_to

line_operation   ->    REPEATED_VERB
```

An operation composes the concept of moving your cursor with an action verb.

It's so common you want to delete or change a whole line there's a convention of repeating a verb twice to do so.

The composition of rules gives you a combinatoric superpower. The number of commands you can carry out is the number of is roughly VERBS x LOCATIONS.

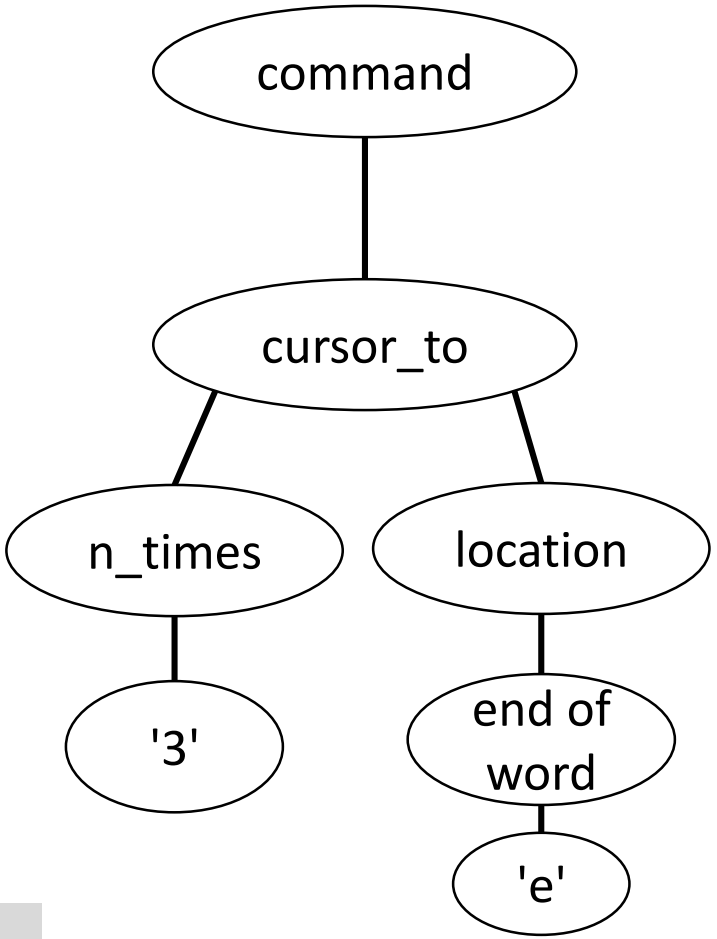
You can repeat / "scale" these commands, too!

```
command          -> cursor_to | operation | line_operation

cursor_to        -> LOCATION | n_times LOCATION

operation        -> VERB cursor_to | n_times VERB cursor_to

line_operation   -> REPEATED_VERB | n_times REPEATED_VERB

n_times          -> POSITIVE_INTEGER
```

We'll look at how to form the grammar of a positive integer out of terminal characters soon. For now assume we can.

How would the grammar parse "3e"?

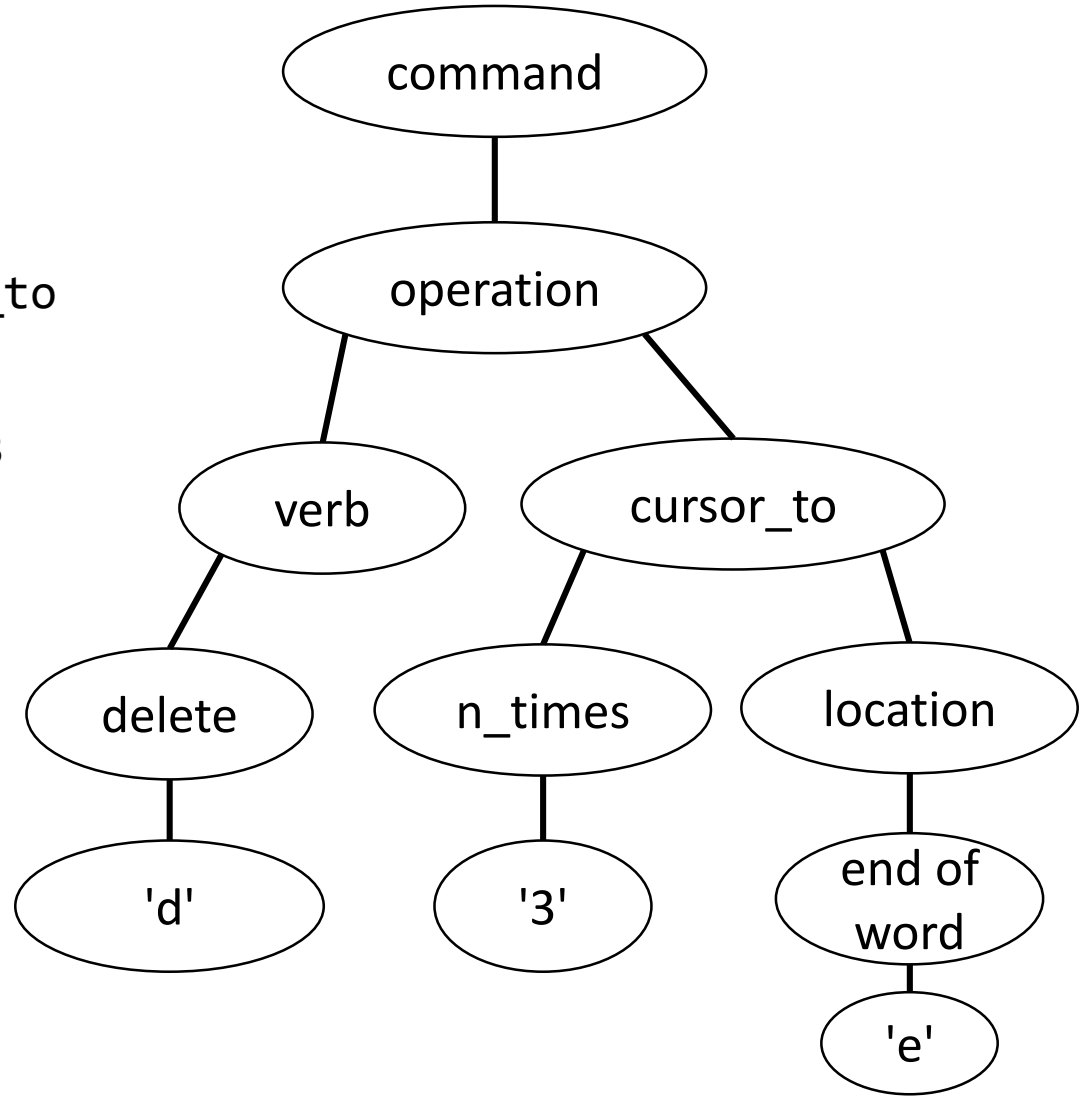
```
command        -> cursor_to | operation | line_operation

cursor_to      -> LOCATION | N_TIMES LOCATION

operation      -> VERB cursor_to | N_TIMES VERB cursor_to

line_operation -> REPEATED_VERB | N_TIMES REPEATED_VERB
```



The command moves the cursor to the end of 3 words forward.

How would the grammar parse "d3e"?

```
command        -> cursor_to | operation | line_operation

cursor_to      -> LOCATION | N_TIMES LOCATION

operation      -> VERB cursor_to | N_TIMES VERB cursor_to

line_operation -> REPEATED_VERB | N_TIMES REPEATED_VERB
```



The command is deletes from the cursor to the 3rd end of word.

Changing to Insert Mode

```
command        -> cursor_to | operation | line_operation | to_insert_mode

cursor_to      -> LOCATION | N_TIMES LOCATION

operation      -> VERB cursor_to | N_TIMES VERB cursor_to

line_operation -> REPEATED_VERB | N_TIMES REPEATED_VERB

to_insert_mode -> insert | insert_below | append
insert         -> 'i'
insert_below   -> 'o'
append         -> 'a'
```

When you're ready to go into insert mode and start typing, there are a few commonly used points to begin inserting new text as shown to the left.

vim Grammar Cheat Sheet

```
command   -> cursor_to | operation | line_operation | TO_INSERT_MODE

cursor_to -> LOCATION | N_TIMES LOCATION

operation -> VERB cursor_to | N_TIMES VERB cursor_to

line_operation -> REPEATED_VERB | N_TIMES REPEATED_VERB
```

To Insert Mode	Key
insert	i
insert new line below	o
insert new line above	O (shift+o)
append after cursor	a

Verb	Key
change	c
delete	d
yank	y

Location	Key
line below	j
line above	k
char left	h
char right	l
first char of line	^
last char of line	$
next word	w
previous word (back)	b
end of next word	e
next occurrence of word	*
previous occurrence of word	#
start of file	gg
end of file	G

1. Try to express verbally what you want to accomplish

2. Then try and express that in the grammar by substituting rules....

- "Move cursor to 5 lines below."
- "Change the entire line."
- "Delete from cursor to the start of the line."