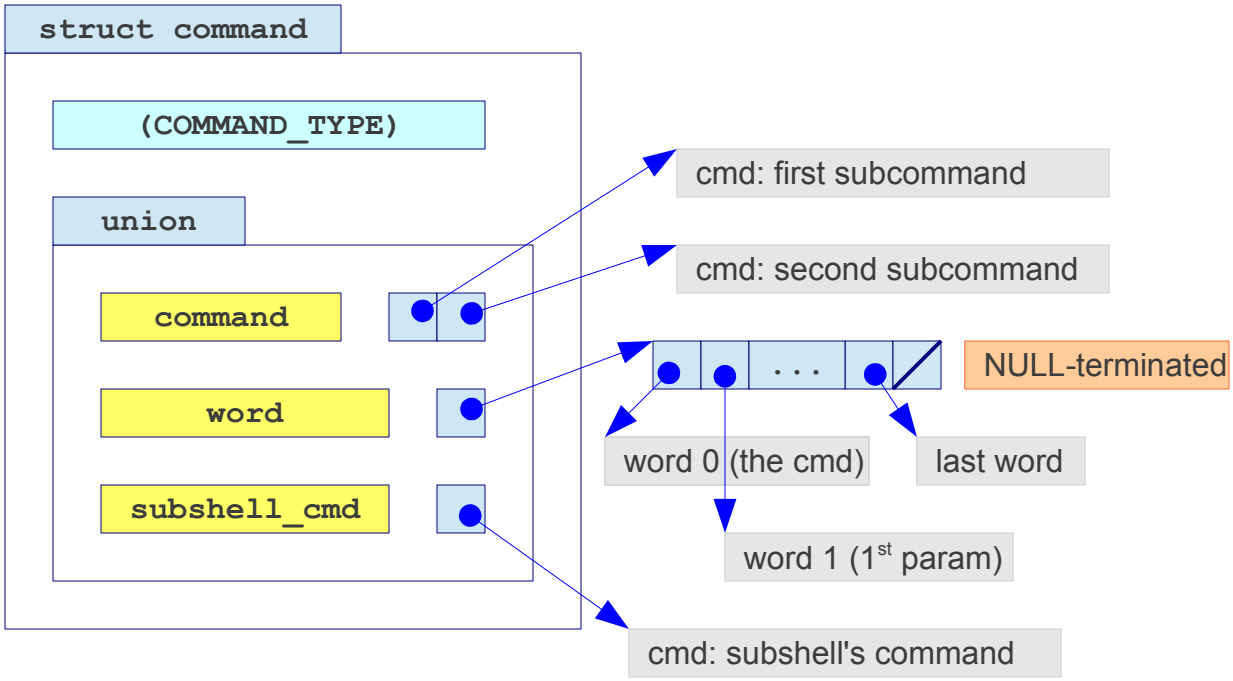
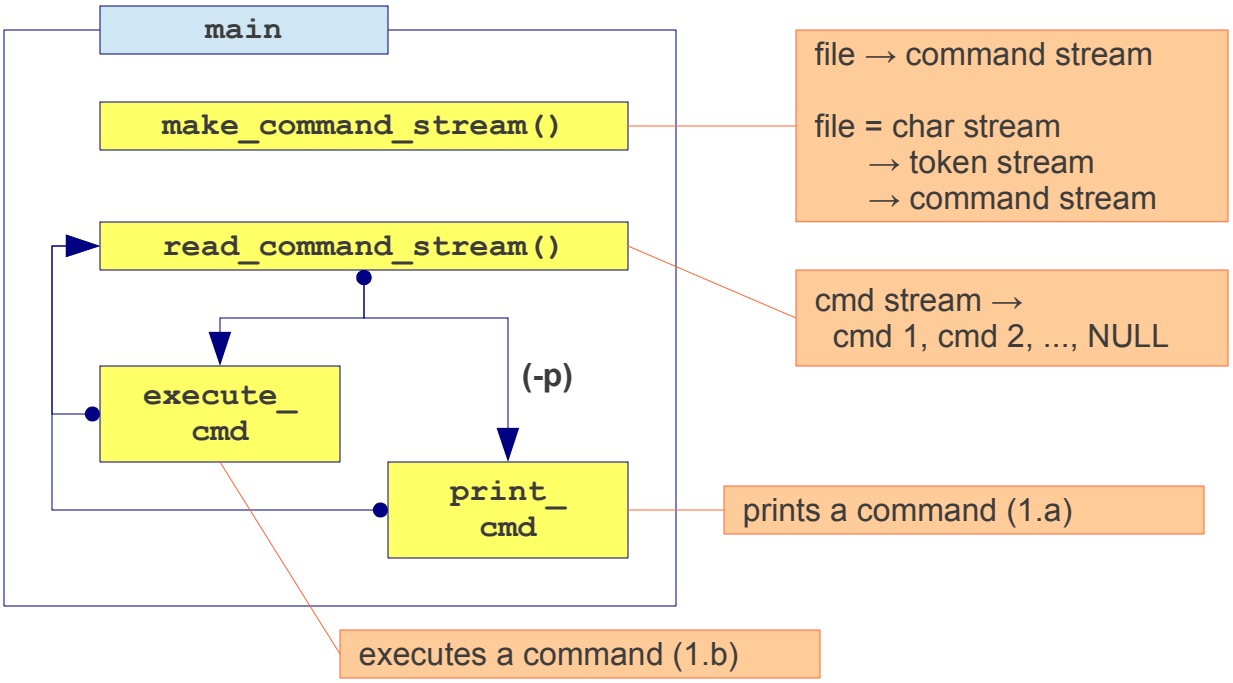
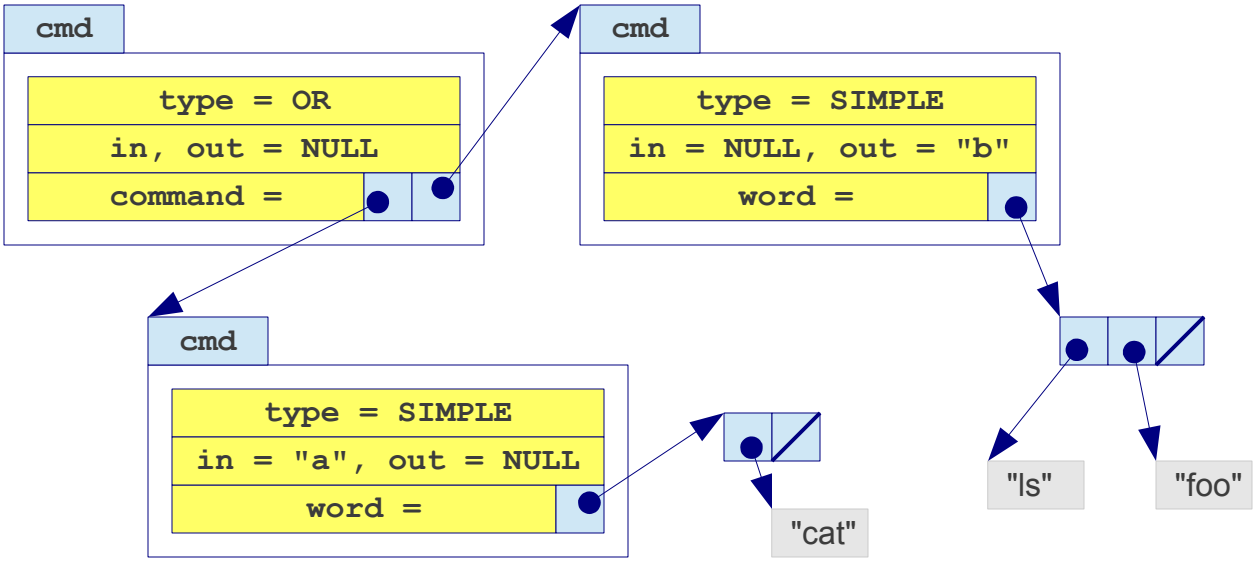

struct command
(COMMAND_TYPE)
union
command
word
subshell_cmd
cmd: first subcommand
cmd: second subcommand
. . .
NULL-terminated
word 0 (the cmd)
last word
word 1 (1st param)
cmd: subshell's command


main
make_command_stream()
read_command_stream()
execute_
cmd
(-p)
print_
cmd
file → command stream
file = char stream
→ token stream
→ command stream
cmd stream →
cmd 1, cmd 2, ..., NULL
prints a command (1.a)
executes a command (1.b)

Example: `cat < a || ls foo > b`



char stream:

```
"  5 * (72 + 3 ) "
```

token stream:

```
NUM(5) TIMES LP NUM(72) PLUS
  NUM(3) RP
```

(BNF) language:

```
sum  → prod | prod PLUS sum
prod → par  | par TIMES prod
par  → NUM  | LP sum RP
```

```
read_sum(queue Q):

  read_prod(Q)

  if Q.next() == PLUS:

    Q.pop()  // process the
             // PLUS sign--
             // remove it from
             // the queue.

    read_sum(Q)  // continue
                 // recursively.
```

```
read_prod(queue Q):
  read_par(Q)
  if Q.next() == TIMES:
    Q.pop()
    read_prod(Q)
```

```
read_par(queue Q):
  if Q.next() == NUM:
    Q.pop()
  else if Q.next() == LP:
    Q.pop()  // left paren.
    read_sum(Q)
    Q.pop()  // right paren.
```

```
result_t is union:

     num_result(int n_1)

  or par_result(result_t s_1)

  or prod_result(result_t q_1,
                 result_t p_1)

  or sum_result(result_t p_1,
                result_t s_1)
```

```
read_sum(queue Q):

  if Q.next() != NUM or LP:
    fail()

  read_prod(Q)

  if Q.empty():
    return  // finished okay.

  else if Q.next() != PLUS:
    fail()

  Q.pop()

  read_sum(Q)
```

char stream:

```
"  5 * (72 + 3 ) "
```

token stream:

```
NUM(5) TIMES LP NUM(72) PLUS
  NUM(3) RP
```

(BNF) language:

```
sum  → prod | prod PLUS sum
prod → par  | par TIMES prod
par  → NUM  | LP sum RP
```

```
result_t read_sum(queue Q):
  result_t p_1, s_2
  p_1 = read_prod(Q)
  if Q.next() == PLUS:
    Q.pop()
    s_2 = read_sum(Q)
    return sum_result(p_1,s_2)
  return p_1
```

```
result_t read_prod(queue Q):
  result_t q_1, p_2
  q_1 = read_par(Q)
  if Q.next() == TIMES:
    Q.pop()
    p_2 = read_prod(Q)
    return prod_result(q_1,p_2)
  return q_1
```

```
________ read_par(queue Q):
  ____________
  if Q.next() == NUM(n):
    Q.pop()
    ____________________
  else if Q.next() == LP:
    Q.pop()
    _____ read_sum(Q)
    Q.pop()
    ________________________
```

char stream for differences:

```
"5 - 3 - 2"
```

we want:

```
(5 - 3) - 2
```

if we parse as in `read_sum()`:

```
5 - (3 - 2)
```

two options:

1. use a more general parsing method
2. read as before, then fix

language for the example:

```
diff → par | diff MINUS par
par  → NUM | LP diff RP
```

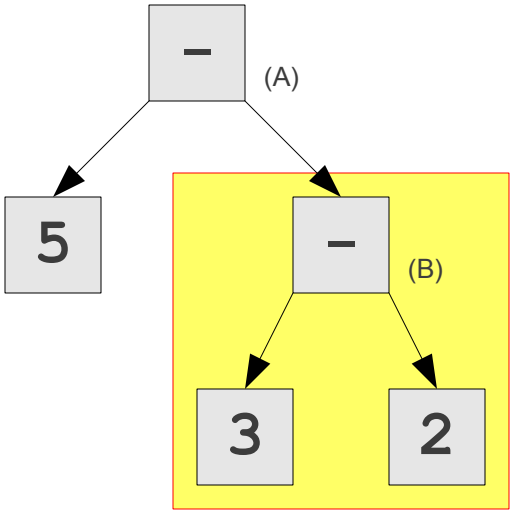
Notice that the grouping is now on the left side–this is left associativity.

We will read as though it were

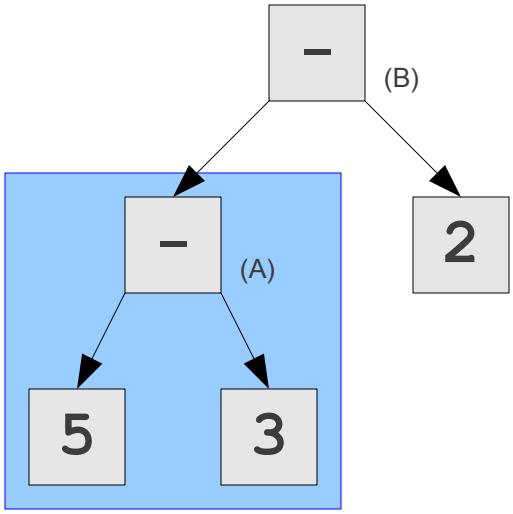
```
diff → par | par MINUS diff
par  → NUM | LP diff RP
```

and then fix it.

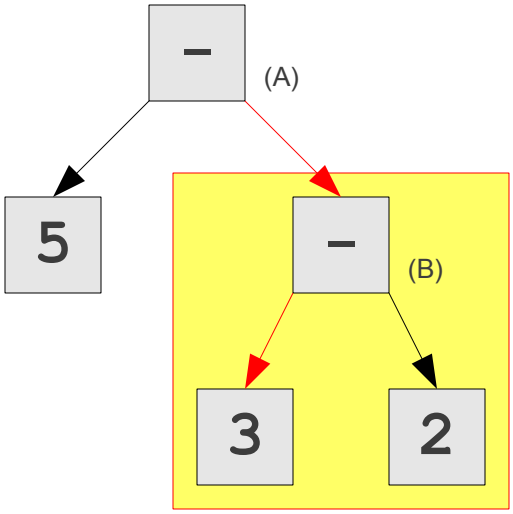
Right associative (before)



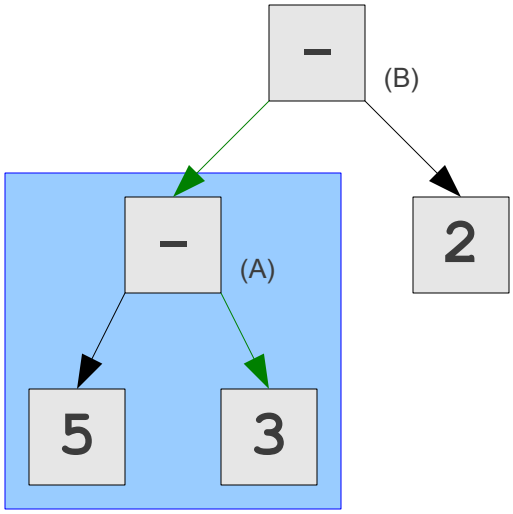
Left associative (after)



Right associative (before)



Left associative (after)



```
result_t read_diff(queue Q):

  result_t p_1, d_2

  p_1 = read_par(Q)

  if Q.next() == MINUS:

    Q.pop()

    d_2 = read_diff(Q)

    return fix_diff(p_1,d_2)

  return p_1
```

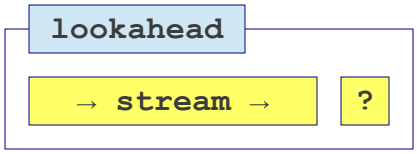
```
result_t fix_diff(result_t a,
                  result_t b):

  // base case: NUM or par.
  if b == num_result(n) or
     b == par_result(d):

    return diff_result(a,b)

  // recursive case:
  // b == diff_result(l,r) .
  else:

    m = fix_diff(a,l)

    return diff_result(m,r)
```

tokens:

```
  NUM(n), TIMES, PLUS, LP, RP

token_t is union:

     num_token(int n)

  or times_token()

  or plus_token()

  or lp_token()

  or rp_token()
```

```
token_t read_token(queue Q):

  if Q.next() == whitespace:
    while Q.next() == ws:
      Q.pop()

  if Q.next() == '0'-'9':
    return read_num_token(Q)

  else if Q.next() == '*':
    return times_token()

  ...

  else:  // unexpected case.
    fail()
```

```
token_t read_num_token(queue Q):
  make a buffer B
  while Q.next() == '0'-'9':
    read Q.next() into buffer B
    Q.pop()
  translate B into an integer n
  return num_token(n)
```




```
class lookahead:

  data:

    stream _s
    item   _next

  methods:

    item next()
    void pop()
```

```
item next():

  if _next == NULL:
    _next = _s.get()
    return _next


void pop():

  if _next != NULL:
    _next = NULL

  else:
    _s.get()
```

Lab 1 syntax elements

words

special tokens: &&, ||, etc.

shell expressions	COMMAND_TYPE
simple commands	SIMPLE
subshells: (*expr*)	SUBSHELL
commands: *expr* < *in* > *out*	? (you need input, output fields)
pipelines: *expr* | *expr*	PIPE
and-ors: *expr* &&, || *expr*	AND, OR
complete commands: *expr* ; *expr*	SEQ