Distribute the following questions across the members of your group. You will share your solutions (and most importantly the *method* of your solutions) during the next lecture period. Divide up the questions so you have more than one person working on each of 1 and 2, while a couple people could collaborate on the LR parser (**??**).

All students should review the learning goals on the schedule page for **Syntax Directed Translation**, §7.1–§7.2.

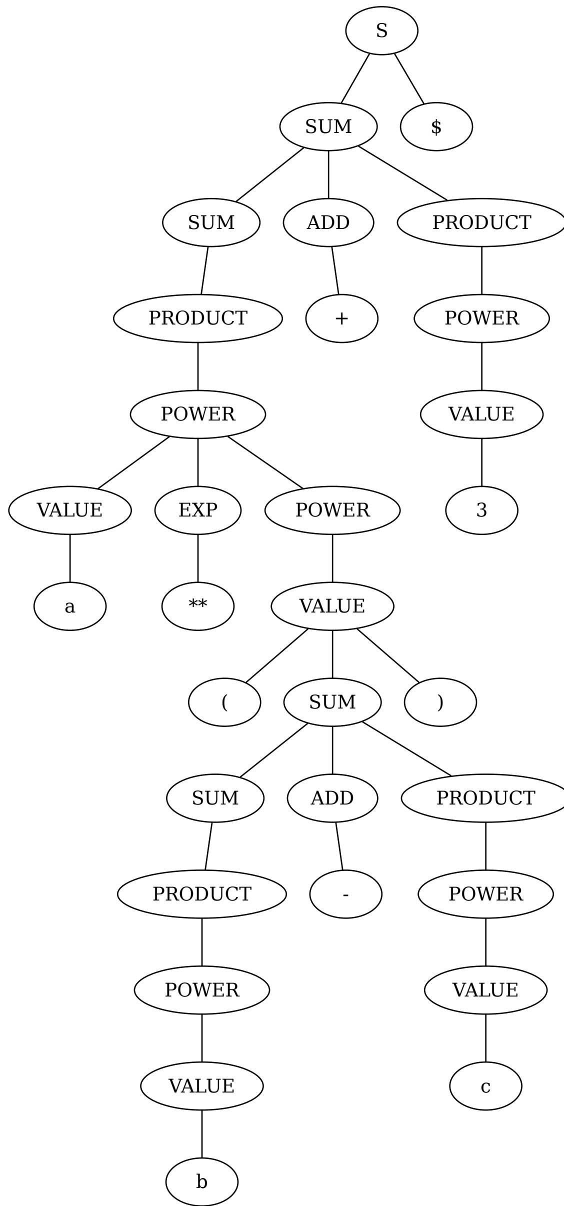
1. Read §7.1–7.3 from the text and explain, clarify or answer the schedule page's **learning goals** for *syntax directed translation*.
2. Read §7.4–7.5 from the text and explain, clarify or answer the schedule page's **learning goals** concerning **abstract syntax trees** (ASTs)
3. Consider the binary operator language for **infix arithmetic expressions**:

#	Rules
1	$S \rightarrow SUM\ \$$
2	$SUM \rightarrow PRODUCT$
3	$SUM \rightarrow SUM\ ADD\ PRODUCT$
4	$PRODUCT \rightarrow POWER$
5	$PRODUCT \rightarrow PRODUCT\ MULT\ POWER$
6	$POWER \rightarrow VALUE$
7	$POWER \rightarrow VALUE\ EXP\ POWER$
8	$VALUE \rightarrow num$
9	$VALUE \rightarrow var$
10	$VALUE \rightarrow (\ SUM\ )$
11	$ADD \rightarrow +$
12	$ADD \rightarrow -$
13	$MULT \rightarrow *$
14	$MULT \rightarrow /$
15	$MULT \rightarrow \%$
16	$EXP \rightarrow **$

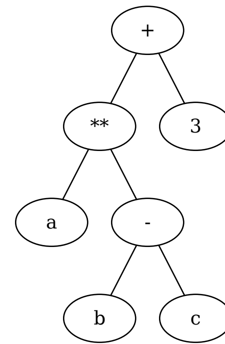
For the input shown with the trees, this would produce concrete parse trees with many nodes (tree on the following page); whereas what we want is a conventional arithmetic expression tree (tree on the next page, also see the first slide of `show_auntsally.pdf`).

Design a scheme (in pseudo code) of SDT that converts the parsed concrete syntax tree into an arithmetic expression tree. Your scheme needs only two semantic actions specific to particular production rules and two "utility" semantic actions (along with the proper assignment of each remaining grammar production rule to one of them).

Concrete Syntax Tree



Conventional Expression Tree

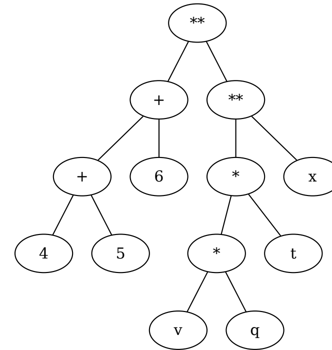


Input: `a ** ( b - c ) + 3`

4. Consider the binary operator language for **prefix arithmetic expressions**:

#	Rules
1	$S \rightarrow PrefixExpr\ \$$
2	$PrefixExpr \rightarrow (\ Op\ Args\ )$
3	$Args \rightarrow Arg\ Arg\ MoreArgs$
4	$MoreArgs \rightarrow Arg\ MoreArgs$
5	$MoreArgs \rightarrow \lambda$
6	$Arg \rightarrow num$
7	$Arg \rightarrow var$
8	$Arg \rightarrow PrefixExpr$
9	$Op \rightarrow LeftOp$
10	$Op \rightarrow RightOp$
11	$LeftOp \rightarrow +$
12	$LeftOp \rightarrow *$
13	$RightOp \rightarrow **$

Conventional Expression Tree



Input: `( ** ( + 4 5 6 ) ( * v q t ) x )`

For the input shown, this would produce concrete parse trees with many nodes (below) whereas what we want is a conventional arithmetic expression tree (above).

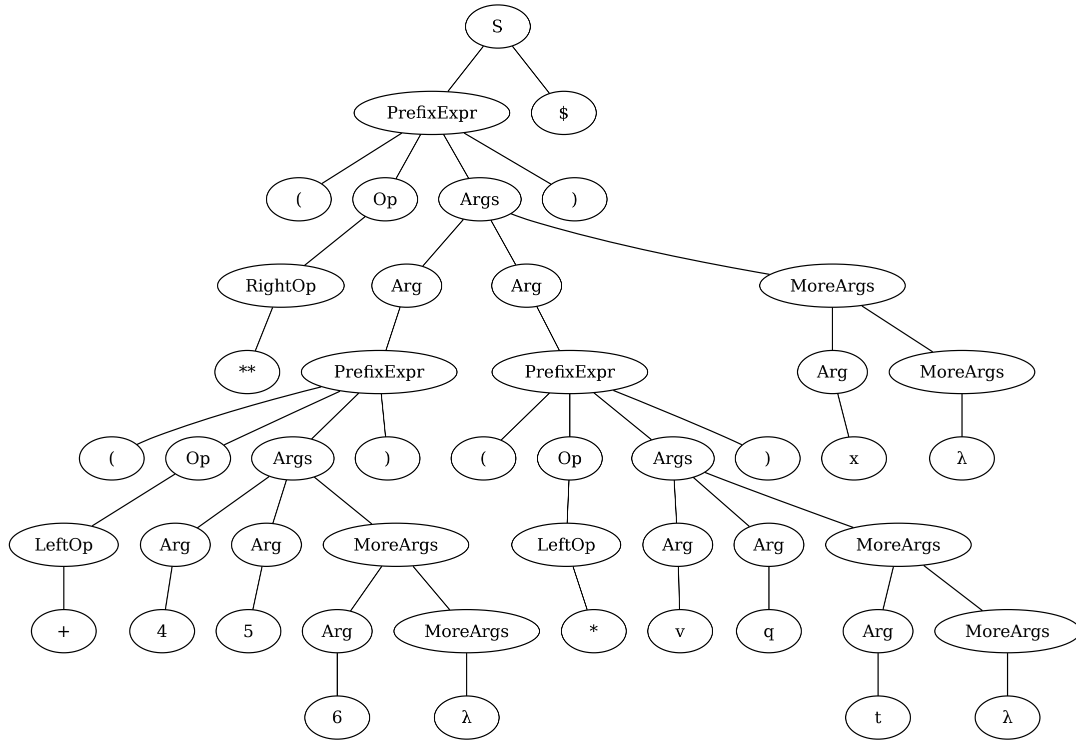
Design a scheme (in pseudo code) of SDT that converts the parsed concrete syntax tree into an arithmetic expression tree. Take care that your solution makes addition and multiplication **left associative** and exponentiation (`**`) **right associative**.

Also, explain to your group:

(a) Could the grammar have been designed differently to avoid the need for SDT? (Be careful not to change the language!)

(b) How does this grammar "encode" our middle-school friend: "*Please Excuse My Dear Aunt Sally?*", aka: order of operations.

Concrete Syntax Tree


S
PrefixExpr
$
(
Op
Args
)
RightOp
Arg
Arg
MoreArgs
**
PrefixExpr
PrefixExpr
Arg
MoreArgs
(
Op
Args
)
(
Op
Args
)
x
λ
LeftOp
Arg
Arg
MoreArgs
LeftOp
Arg
Arg
MoreArgs
+
4
5
Arg
MoreArgs
*
v
q
Arg
MoreArgs
6
λ
t
λ