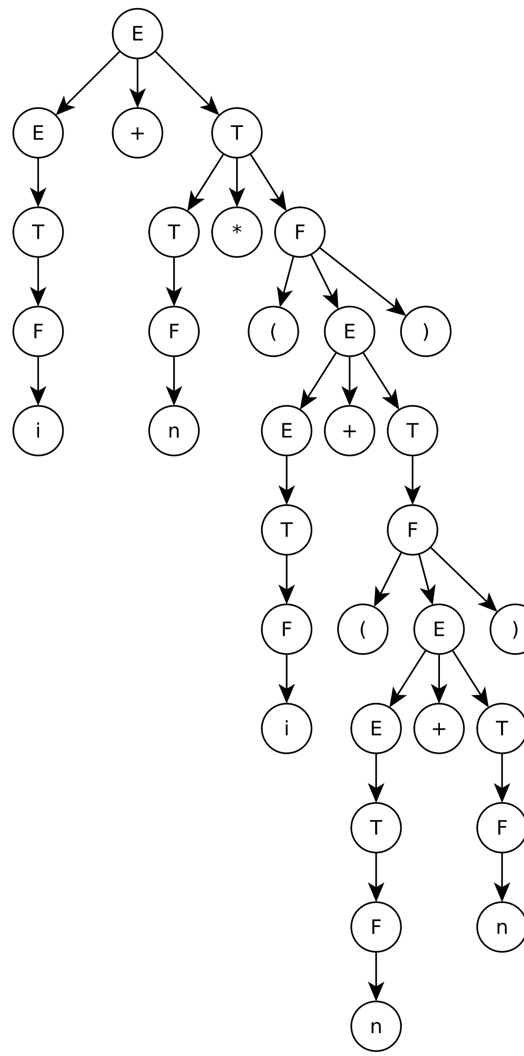
Question 1.

Note: derivation and parse tree will use E, T, and F to mean *expr*, *term*, and *factor*, and i and n to mean **identifier** and **int-literal**. The input string is then $\boxed{\text{i + n * (i + (n + n))}}$.

Derivation:

Working string	Production
$\underline{\text{E}}$	E → E + T
$\underline{\text{E}}$ + T	E →T
$\underline{\text{T}}$ + T	T → F
$\underline{\text{F}}$ + T	F → i
i + $\underline{\text{T}}$	T → T * F
i + $\underline{\text{T}}$ * F	T → F
i + $\underline{\text{F}}$ * F	F → n
i + n * $\underline{\text{F}}$	F → (E)
i + n * ($\underline{\text{E}}$)	E → E + T
i + n * ($\underline{\text{E}}$ + T)	E → T
i + n * ($\underline{\text{T}}$ + T)	T → F
i + n * ($\underline{\text{F}}$ + T)	F → i
i + n * (i + $\underline{\text{T}}$)	T → F
i + n * (i + $\underline{\text{F}}$)	F → (E)
i + n * (i + ($\underline{\text{E}}$))	E → E + T
i + n * (i + ($\underline{\text{E}}$ + T))	E → T
i + n * (i + ($\underline{\text{T}}$ + T))	T → F
i + n * (i + ($\underline{\text{F}}$ + T))	F → n
i + n * (i + (n + $\underline{\text{T}}$))	T → F
i + n * (i + (n + $\underline{\text{F}}$))	F → n
i + n * (i + (n + n))	

Parse tree:

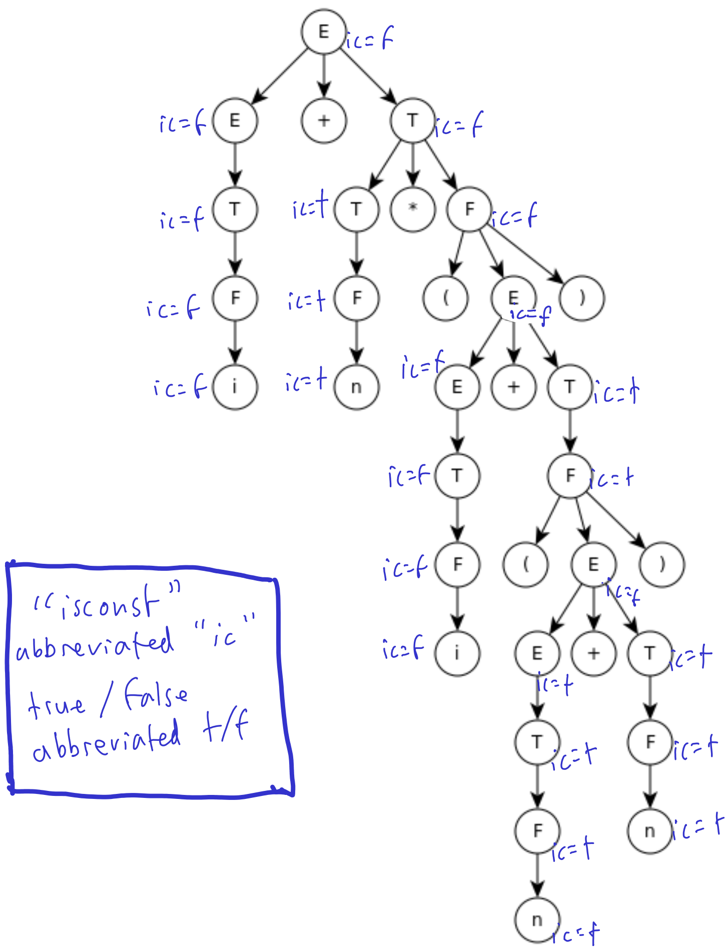


Question 2.

(a) Possible solution:

Grammar rule	Action
$expr_0 \rightarrow expr_1$ **+** $term$	$expr_0$.isconst $\leftarrow$ ($expr_1$.isconst $\wedge$ $term$.isconst)
$expr \rightarrow term$	$expr$.isconst $\leftarrow$ $term$.isconst
$term_0 \rightarrow term_1$ ***** $factor$	$term_0 \leftarrow$ ($term_1$.isconst $\wedge$ $factor$.isconst)
$term \rightarrow factor$	$term$.isconst $\leftarrow$ $factor$.isconst
$factor \rightarrow$ **identifier**	$factor$.isconst $\leftarrow$ false
$factor \rightarrow$ **int-literal**	$factor$.isconst $\leftarrow$ true
$factor \rightarrow$ **(** $expr$ **)**	$factor$.isconst $\leftarrow$ $expr$.isconst

(b) Annotated parse tree:



The isconst attribute is a synthesized attribute, so evaluation is strictly bottom-up (from the leaves towards the root.)

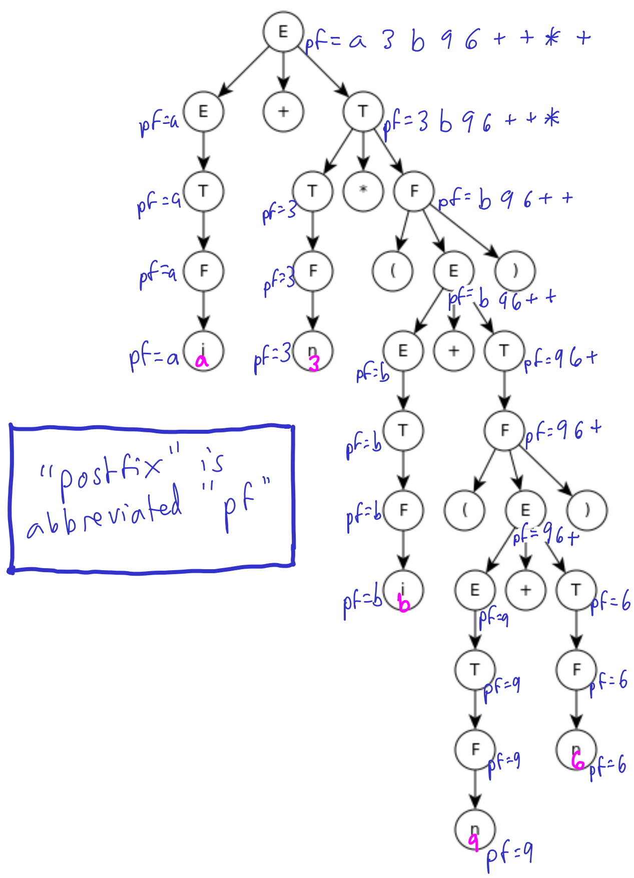
Question 3.

(a) Possible solution:

Grammar rule	Action
$expr_0 \rightarrow expr_1$ **+** $term$	$expr_0$.postfix $\leftarrow$ $expr_1$.psotfix + ␣ + $term$.postfix + ␣ + **+**.lexeme
$expr \rightarrow term$	$expr$.postfix $\leftarrow$ $term$.postfix
$term_0 \rightarrow term_1$ ***** $factor$	$term_0$.postfix $\leftarrow$ $term_1$.postfix + ␣ + $factor$.postfix + ␣ + *****.lexeme
$term \rightarrow factor$	$term$.postfix $\leftarrow$ $factor$.postfix
$factor \rightarrow$ **identifier**	$factor$.postfix $\leftarrow$ **identifier**.lexeme
$factor \rightarrow$ **int-literal**	$factor$.postfix $\leftarrow$ **int-literal**.lexeme
$factor \rightarrow$ **(** $expr$ **)**	$factor$.postfix $\leftarrow$ $expr$.postfix

In attribute rules, the + operator means string concatenation, terminal symbols are assumed to have a "lexeme" property, and ␣ is a string representing a single space character.

(b) Annotated parse tree:



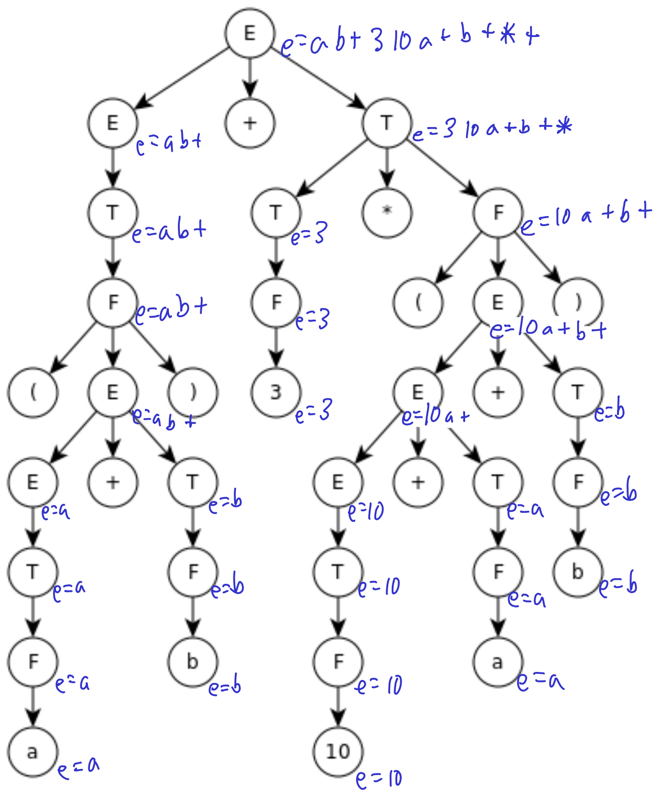
The postfix attribute is a synthesized attribute, so evaluation is strictly bottom-up (from the leaves towards the root.)

Question 4(628).

(a) The attribute grammar defining the postfix attribute in Question 3 will work, just change "postfix" to "exprid". The postfix form of an expression has the property of being identical for subtrees which perform identical computations.

(b) Annotated parse tree:

"exprid" abbreviated "e"



As with the previous attribute grammars, the postfix attribute is a synthesized attribute, and can be evaluated strictly bottom-up.

Question 4(428) / Question 5(628).

Note: parse will use E, T, and F to mean *expr*, *term*, and *factor*, and i and n to mean **identifier** and **int-literal**. The input string is then $\boxed{\text{i + n * (i + (n + n))}}$.

Stack	Input string	Action
$	i + n * (i + (n + n)) $	shift i
$ i	+ n * (i + (n + n)) $	reduce F → i
$ F	+ n * (i + (n + n)) $	reduce T → F
$ T	+ n * (i + (n + n)) $	reduce E → T
$ E	+ n * (i + (n + n)) $	shift +
$ E +	n * (i + (n + n)) $	shift n
$ E + n	* (i + (n + n)) $	reduce T → F
$ E + F	* (i + (n + n)) $	reduce F → n
$ E + T	* (i + (n + n)) $	shift *
$ E + T *	(i + (n + n)) $	shift (
$ E + T * (	i + (n + n)) $	shift i
$ E + T * (i	+ (n + n)) $	reduce F → i
$ E + T * (F	+ (n + n)) $	reduce T → F
$ E + T * (T	+ (n + n)) $	reduce E → T
$ E + T * (E	+ (n + n)) $	shift +
$ E + T * (E +	(n + n)) $	shift (
$ E + T * (E + (	n + n)) $	shift n
$ E + T * (E + (n	+ n)) $	reduce F → n
$ E + T * (E + (F	+ n)) $	reduce T → F
$ E + T * (E + (T	+ n)) $	reduce E → T
$ E + T * (E + (E	+ n)) $	shift +
$ E + T * (E + (E +	n)) $	shift n
$ E + T * (E + (E + n	)) $	reduce F → n
$ E + T * (E + (E + F	)) $	reduce T → F
$ E + T * (E + (E + T	)) $	reduce E → E + T
$ E + T * (E + (E	)) $	shift)
$ E + T * (E + (E)	) $	reduce F → (E)
$ E + T * (E + F	) $	reduce T → F
$ E + T * (E + T	) $	reduce E → E + T
$ E + T * (E	) $	shift)
$ E + T * (E)	$	reduce F → (E)
$ E + T * F	$	reduce T → T * F
$ E + T	$	reduce E → E + T
$ E		

Question 5(428) / Question 6(628).

```
PROGRAM scopes;

BEGIN
  VAR x, fac, count : INTEGER;
  VAR factors : ARRAY 20 of INTEGER;

  READ x;
  fac := x DIV 2;
  count := 0;

  WHILE fac > 1 DO
    VAR rem : INTEGER;

    rem := x MOD fac;

    IF rem = 0 THEN
      VAR nextcount : INTEGER;
      factors[count] := fac;
      nextcount := count + 1;
      count := nextcount;
    ELSE
      VAR y : INTEGER;
      y := fac;
      WRITE y;
      WRITE " is not a divisor!\n";
    END;

    fac := fac - 1;
  END;
END.
```

A

depth	kind	name	type	offset
0	VAR	x	INTEGER	0
0	VAR	fac	INTEGER	4
0	VAR	count	INTEGER	8
0	VAR	factors	ARRAY 20 OF INTEGER	12

92 bytes total

B

depth	kind	name	type	offset
1	VAR	rem	INTEGER	92

4 bytes total

C

depth	kind	name	type	offset
2	VAR	nextcount	INTEGER	96

4 bytes total

D

depth	kind	name	type	offset
2	VAR	y	INTEGER	96

4 bytes total

Total storage required is $\boxed{100}$ bytes.
(storage for nextcount and y can be overlapped because they have non-overlapping lifetimes.)