CS 4337 — Programming Language Paradigms, Homework 3

https://iniyanjoseph.com | iniyanijoseph@gmail.com

Consider a programming language where an identifier is defined as a sequence of uppercase and lowercase letters but starting with an uppercase letter; and an integer is defined as a sequence of digits (0, 1, 2, ...9), but does not start with the digit 0.

Use EBNF to define the syntax for identifier and integer

Integer

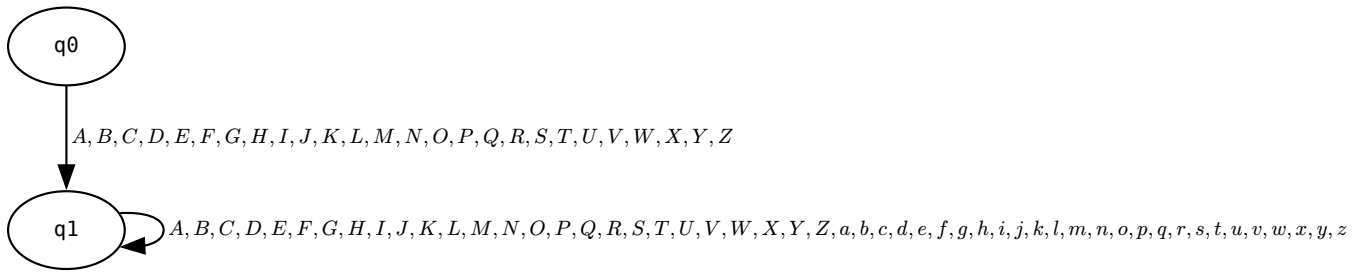
```
<nonzero> -> 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digit> -> 0 | nonzero
<integer> -> <nonzero> {<digit>}
```

Identifier

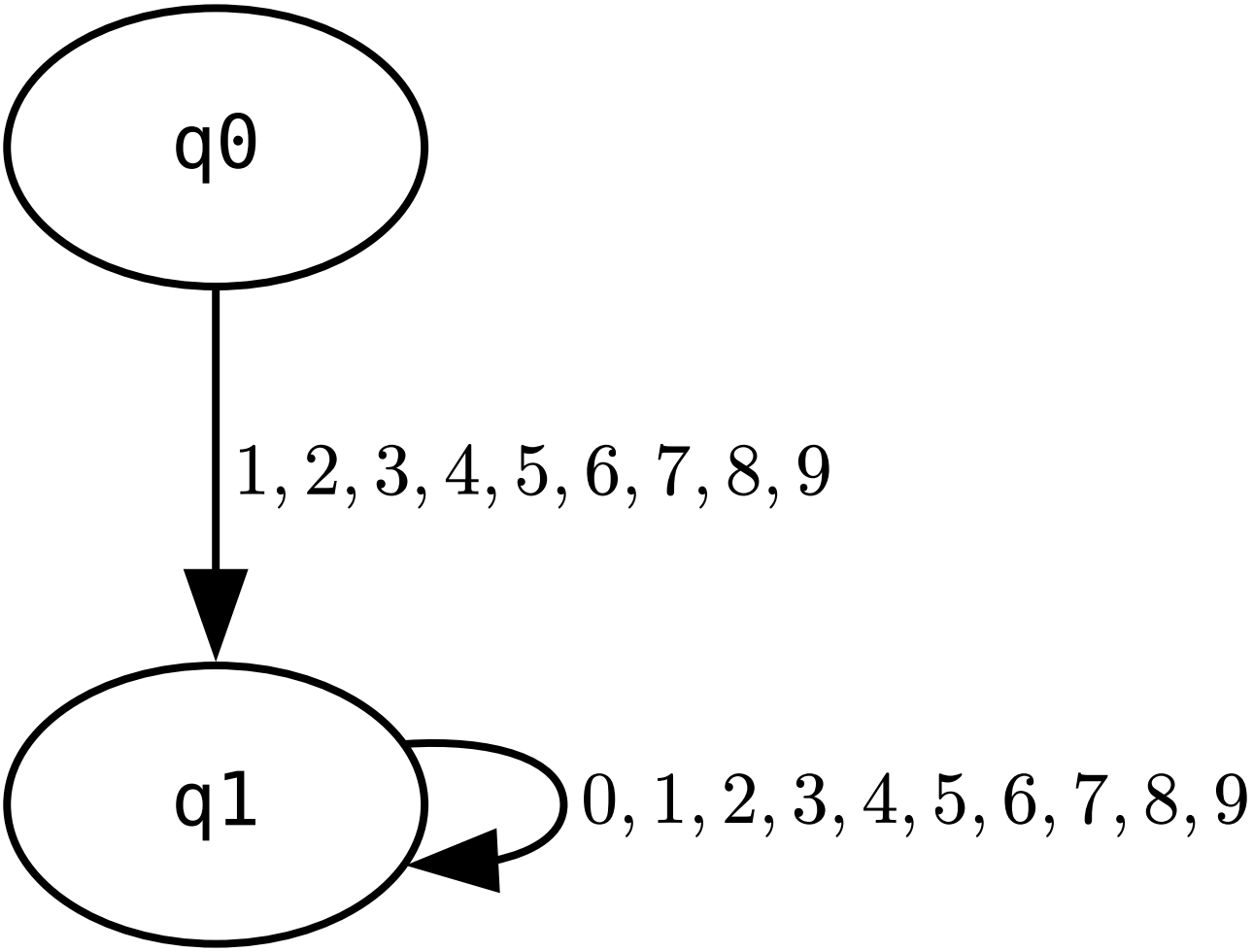
```
<upper> -> A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T
| U | V | W | X | Y | Z
<lower> ->  a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s |
t | u | v | w | x | y | z
<identifier> -> <upper> {(<upper> | <lower>)}
```

Create a state diagram (finite state machine) to recognize identifier and integer

Identifier



Integer



Consider the following grammar for simple Boolean expression,

<expr> → <term> { OR <term> }
<term> → <factor> { AND <factor> }
<factor> → NOT <factor> | BOOL | (<expr>)

BOOL represents Boolean literals (e.g., TRUE, FALSE); OR, AND, and NOT are logical operators. In Recursive-Descent Parsing, define the subprograms (in pseudocode) for <expr>, <term>, and <factor>

```
function expr()
  term()
  while nextToken == OR_OP
    lex()
    term()

function term()
  factor()
  while nextToken == AND_OP
    lex()
    factor()

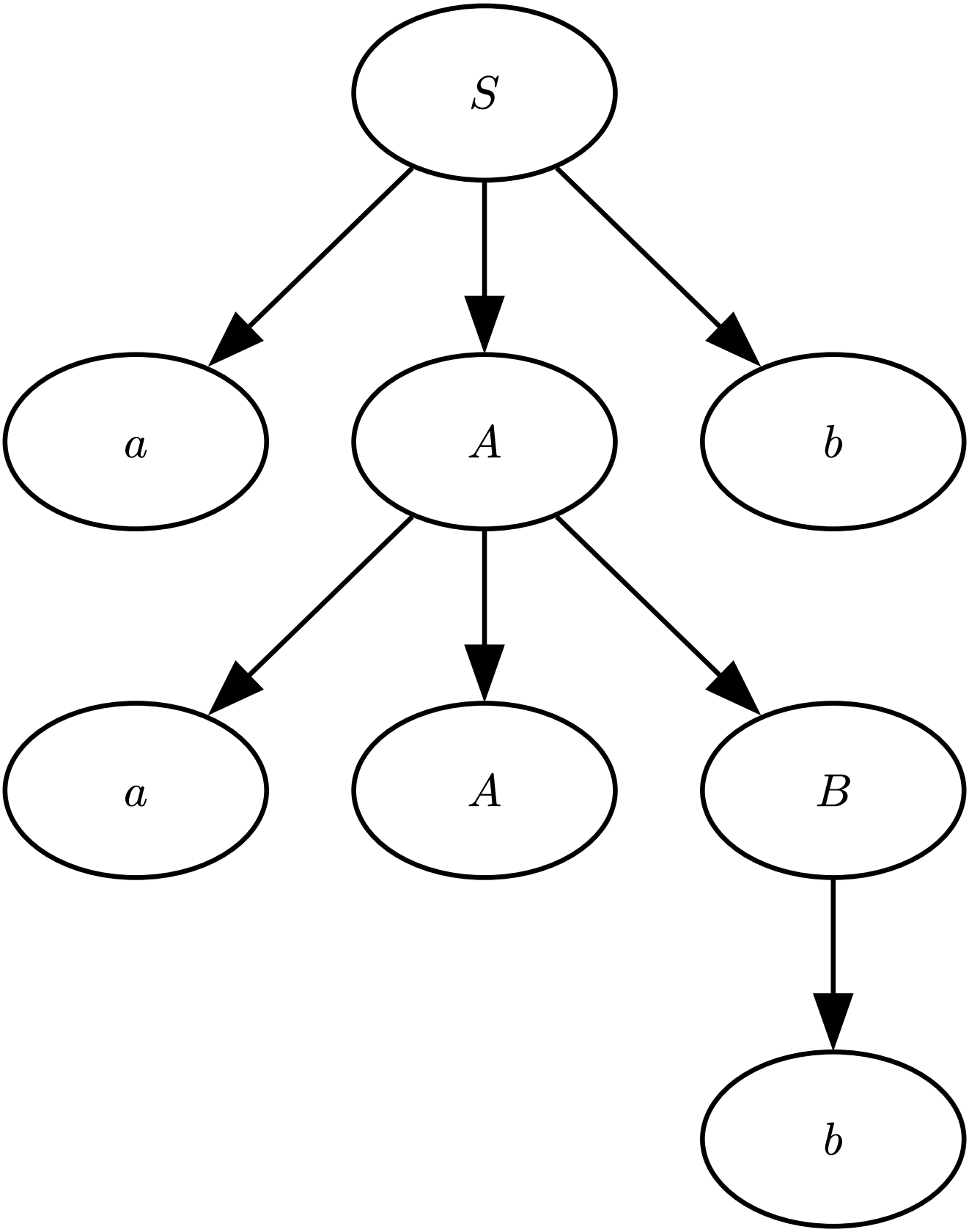
function factor()
  if nextToken == BOOL
    lex()
```

```
else if nextToken == NOT_OP
  lex()
  expr()
else if nextToken == LP_CODE
  lex()
  expr()
  if nextToken == RP_CODE
    lex()
  else
    error()
else
  error()
```

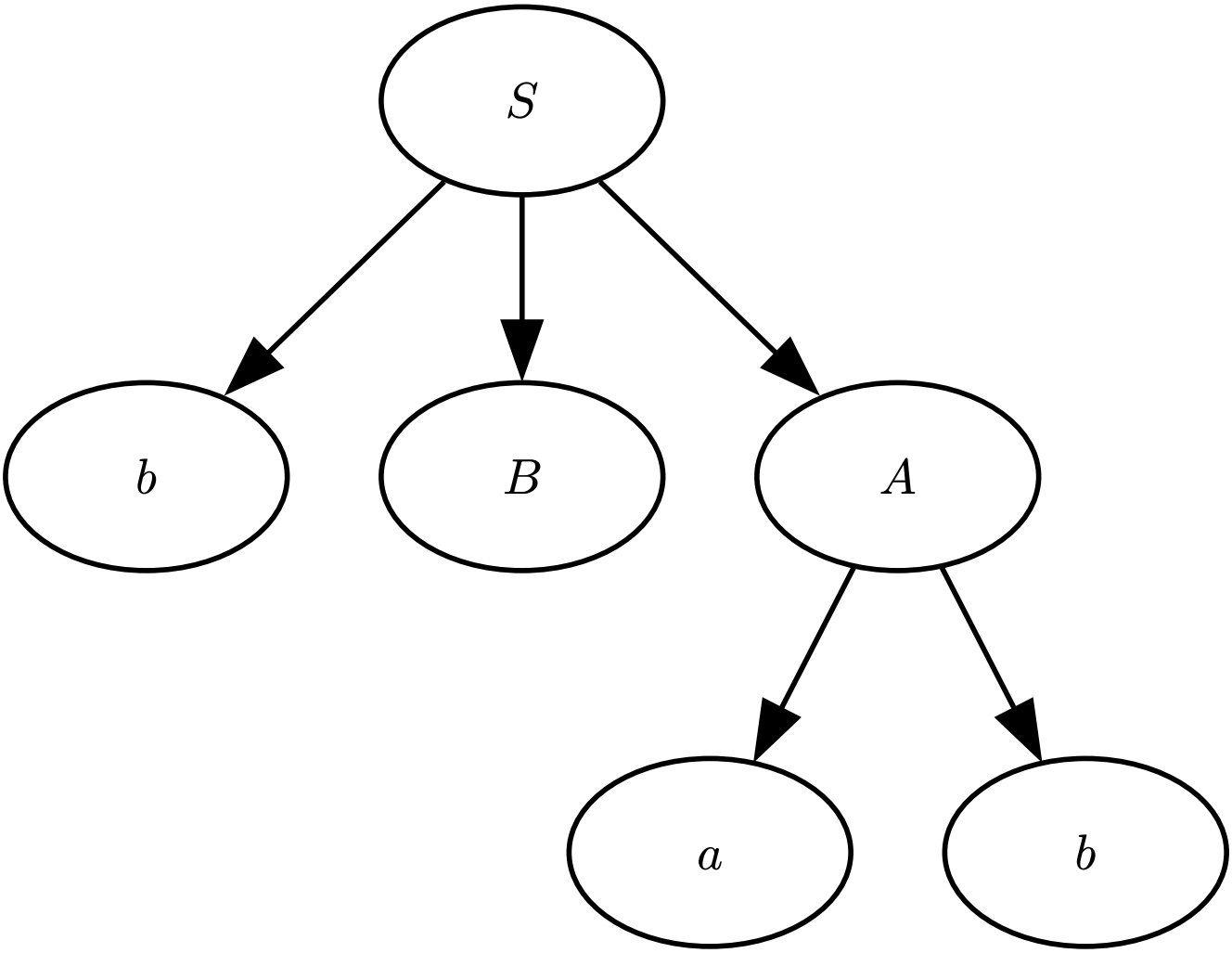
Given the following grammar and the right sentential forms, draw a parse tree and show the phrases, simple phrases, and the handle for each right sentential form.

S → aAb | bBA
A → ab | aAB
B → aB | b

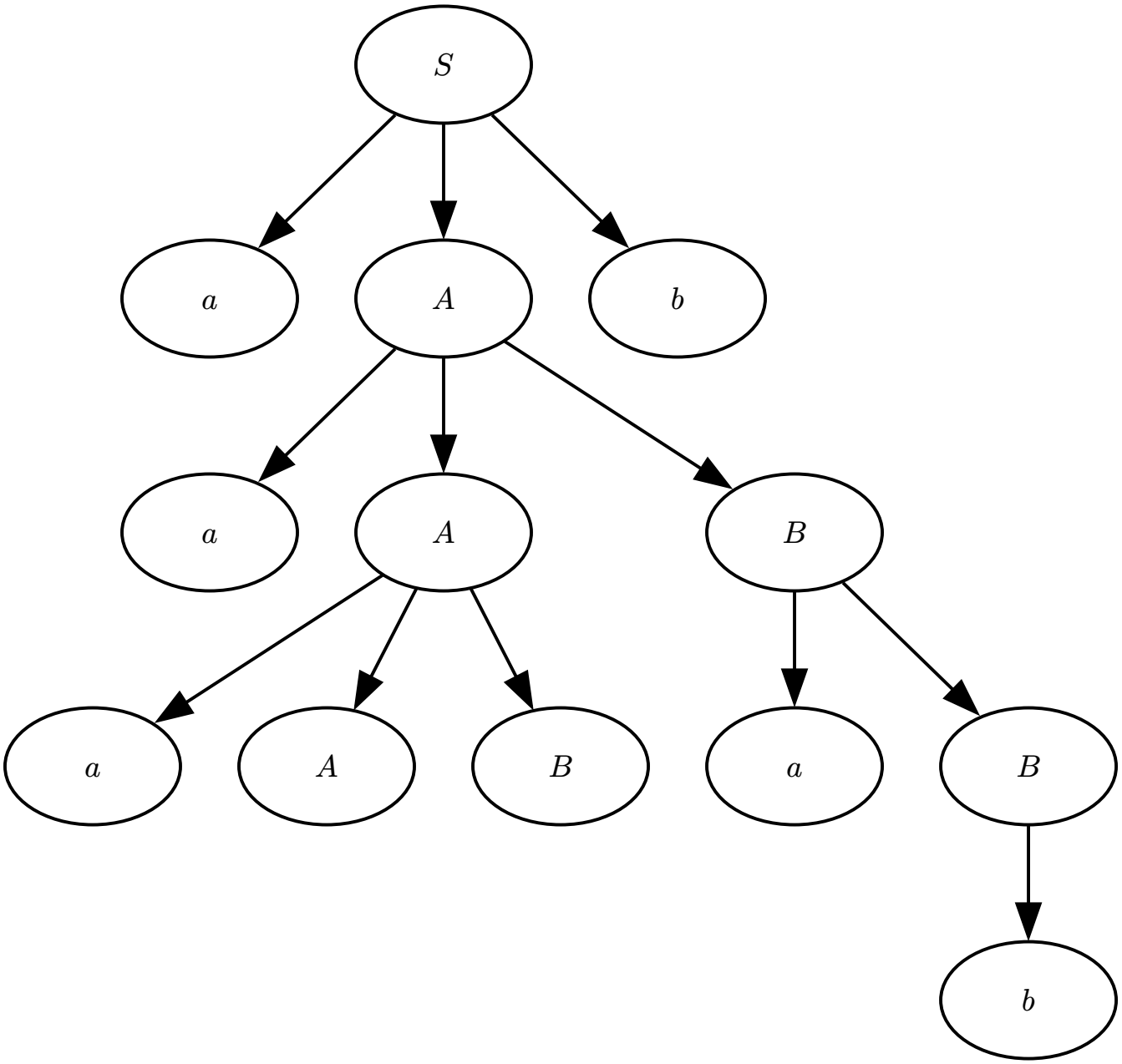
aaAbb


S
a
A
b
a
A
B
b

bBab



aaaABabb



Show a complete parse, including the parse stack contents, input string, and action for the string `id * (id + id)`, using the grammar and parse table in Section 4.5.3

Stack	Input	Action
0	id + id * id $	S5
0id5	+ id * id $	R6
0F3	+ id + id * id $	R4
0T2	+ id + id * id $	R2
0E1	+ id + id * id $	S6
0E1+6	+ id + id * id $	S5
0E1+6id5	id * id $	R6
0E1+6F3	* id $	R4
0E1+6T9	* id $	S7

Stack	Input	Action
0E1+6T9*7	* id $	S5
0E1+6T9*7id5	id $	R6
0E1+6T9*7F10	$	R3
0E1+6T9	$	R1
0E1	$	Accept

Write an attribute grammar whose BNF basis is that of Example 3.6 in Section 3.4.5 but whose language rules are as follows: Data types cannot be mixed in expressions, but assignment statements need not have the same types on both sides of the assignment operator

```
<assign> -> <var> = <expr>
Semantic Rule:
<var>.actual_type <- <expr>.actual_type

<expr> -> <var>[1] + <var>[2]
Semantic Rule:
<expr>.actual_type <-
  if (<var>[1].actual_type == <var>[2].actual_type)
    then <var>[1].actual_type
  else error("Type mismatch in expression")
Predicate:
  <var>[1].actual_type == <var>[2].actual_type

<expr> -> <var>
Semantic Rule:
<expr>.actual_type <- <var>.actual_type

<var> -> A | B | C
Semantic Rule:
<var>.actual_type <- look-up(<var>.string)
```

Using the virtual machine instructions given in Section 3.5.1.1, give an operational semantic definition of the following, Java do-while

```
do {
  expr_list
} while (condition);
expr_list
loop:
  if condition == 0
    goto out
  else
    expr_list
  goto loop
out: ...
```

Define denotational semantics for the language of octal (base 8) numerals. Use the definition to find the value of "752". You must define M_{oct}as a semantic function that maps syntactic objects written as octal numbers (with no decimal points, i.e., no fractions) to a set of non-negative decimal integer values. For Example, given the octal string "752", then Moct ("752") would return 490 as its equivalent decimal value.

$$M_{\text{oct}('0')} = 0$$
$$M_{\text{oct}('1')} = 1$$
$$M_{\text{oct}('2')} = 2$$
$$M_{\text{oct}('3')} = 3$$
$$M_{\text{oct}('4')} = 4$$
$$M_{\text{oct}('5')} = 5$$
$$M_{\text{oct}('6')} = 6$$
$$M_{\text{oct}('7')} = 7$$

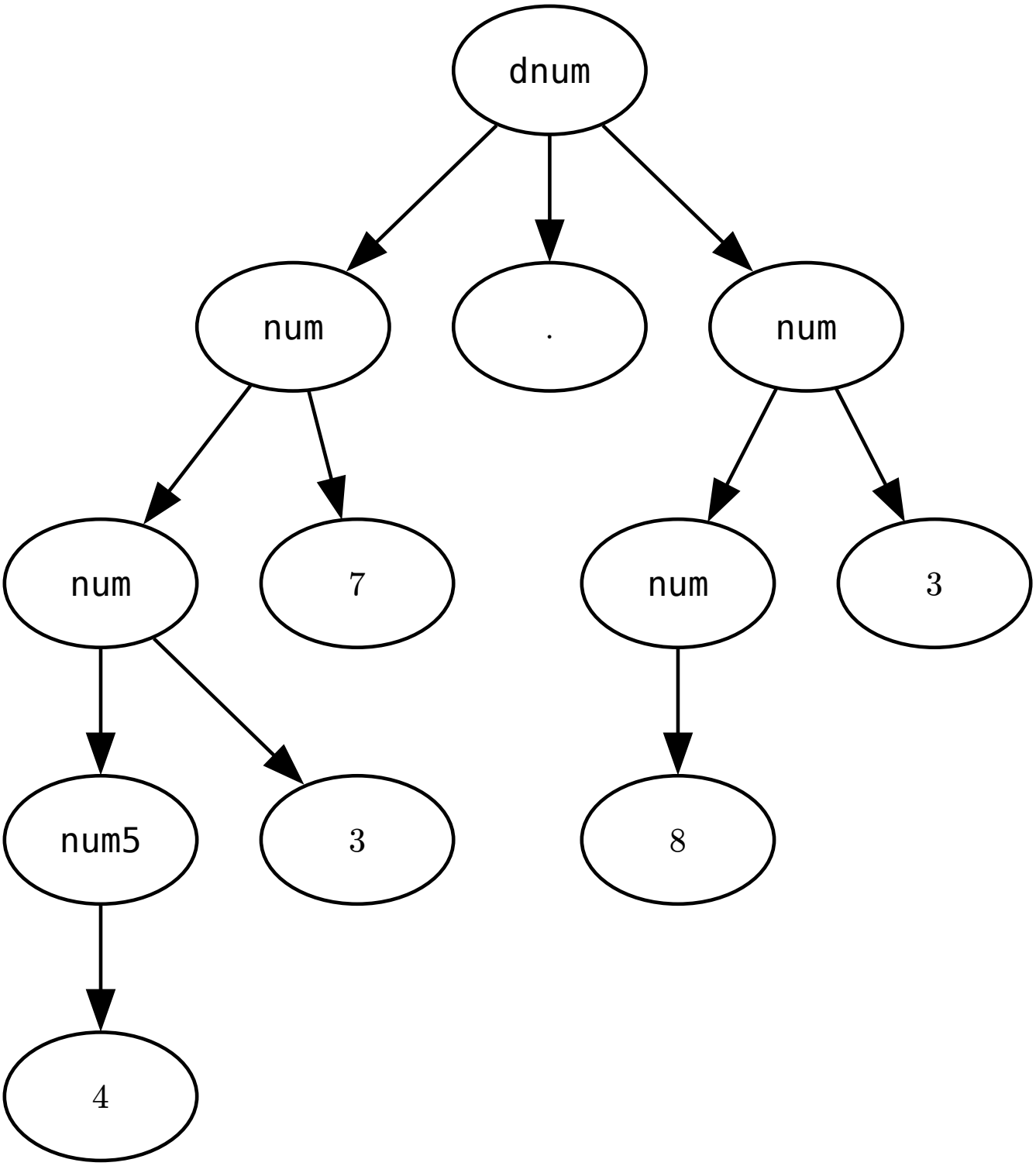
$$M_{\text{oct}(<\text{oct_num}>\,'0')} = 8 * M_{\text{oct}(<\text{oct_num}>)} + 0 \quad M_{\text{oct}(<\text{oct_num}>\,'1')} = 8 * M_{\text{oct}(<\text{oct_num}>)} + 1$$
$$M_{\text{oct}(<\text{oct_num}>\,'2')} = 8 * M_{\text{oct}(<\text{oct_num}>)} + 2 \quad M_{\text{oct}(<\text{oct_num}>\,'3')} = 8 * M_{\text{oct}(<\text{oct_num}>)} + 3$$
$$M_{\text{oct}(<\text{oct_num}>\,'4')} = 8 * M_{\text{oct}(<\text{oct_num}>)} + 4 \quad M_{\text{oct}(<\text{oct_num}>\,'5')} = 8 * M_{\text{oct}(<\text{oct_num}>)} + 5$$
$$M_{\text{oct}(<\text{oct_num}>\,'6')} = 8 * M_{\text{oct}(<\text{oct_num}>)} + 6 \quad M_{\text{oct}(<\text{oct_num}>\,'7')} = 8 * M_{\text{oct}(<\text{oct_num}>)} + 7$$

Given the following BNF that defines decimal numbers (with decimal points)

```
dnum → num.num
num → num digit | digit
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

And the following Attribute grammar that computes the value of a given decimal string: (Hint: The count attribute is used to count the number of digits to the right of the decimal point.)

Draw a parse tree with denoted objects that would compute the decimal value of the decimal string "437.83" (Note that the decimal value of the string "437.83" is 437.83).



Compute the weakest precondition for each of the following

`a = 2 * b + 1; b = a – 3; {b < 0}`

$$a - 3 < 0 \Rightarrow a < 3$$
$$2 * b + 1 < 3 \Rightarrow b < 2$$
$$\{b < 2, a < 3\}$$

`if (a == b) b = 2 * a + 1 else b = 2 * a; {b > 1}`

$$2 * a > 1 \Rightarrow a > \frac{1}{2}$$

$2 * a + 1 > 1 \Rightarrow a > 0$; Alternatively, since $a == b, 2 * b + 1 = 2 * b + 1 > 1 \Rightarrow a = b = 0$

$\{a > 0\}$