

Several of the questions in this problem set refer to the following input character strings:

$I1 = ((32+100)*(16-4))$ ,  $I2 = ((32+(10*10))*(16-4))$ , and  $I3 = ((32+(100*16))-4)$ .

1. \*Input character strings and token streams.

\* For each input character string  $I_k$  ( $1 \leq k \leq 3$ ), write down the corresponding token stream  $S_k$  that `SamTokenizer` would generate. What are the compression ratios for each input string, where *compression ratio* is defined as the number of characters in the input string divided by the number of resulting tokens?

1	I1:
2	(
3	(
4	32
5	+
6	100
7	)
8	*
9	(
10	16
11	-
12	4
13	)
14	)

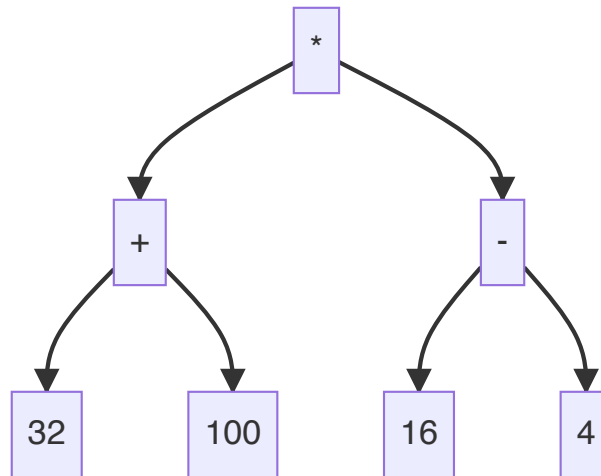
12 tokens, 13/17

Same calculation methods for other two

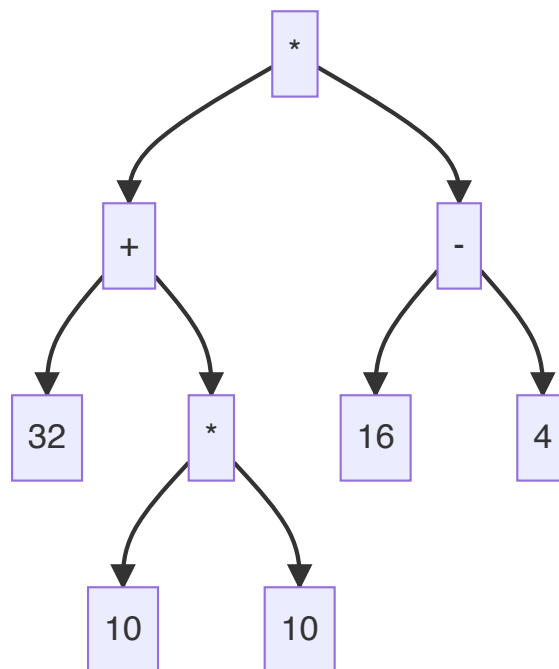
2. \*Abstract syntax trees.

\* For each input character string  $I_k$  ( $1 \leq k \leq 3$ ), show its corresponding abstract syntax tree  $T_k$  when parsed with the expression grammar of LiveOak. In what way are  $T1$  and  $T2$  the same? In what way are they different?

**I1:**

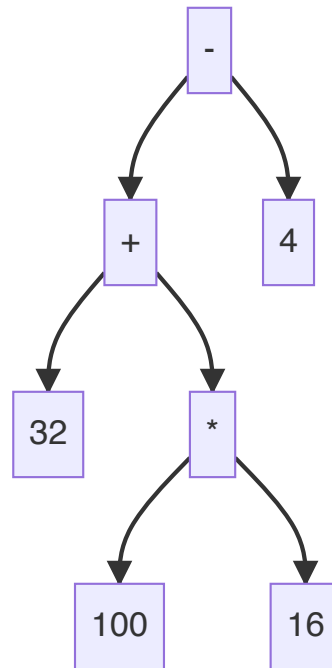


I2:



I1 and I2 are similar in level 0-1, but different at node "\*" starting level 2 and down. It is because the expression added a branch in that slot.

I3:



### 3. \*Parsing and lookahead.

\* Trace the individual steps of parsing and AST generation for input string I1 using a recursive-descent parser driven by the expression grammar of LiveOak. At each step, indicate the current and lookahead tokens, and sketch the shape of the parse tree and AST.

```

1  getExpr('((32+100)*(16-4))'):
2  token = '('
3  lookahead = '('
4  getExpr('(32+100)*(16-4))'):
5  token = '('
6  lookahead = '32'
7  token = '32'
8  exp1 = `PUSHIMM 32`
9  token = '+'
10 binop = `ADD`
11 token = '100'
12 exp2 = `PUSHIMM 100`
13 token = ')'
14 return expr(exp1+exp2+binop)
15 exp1 = return1
16 lookahead = '*'
17 binop = `TIMES`
18 token = '('
19 getExpr('(16-4))'):

```

```

20     token = '('
21     lookahead = '16'
22     token = '16'
23     exp1 = `PUSHIMM 16`
24     token = '-'
25     binop = `SUB`
26     token = '4'
27     exp2 = `PUSHIMM 4`
28     token = ')'
29     return expr(exp1+exp2+binop)
30 exp2 = return2
31 token = ')'
32 return expr(exp1 + exp2 + binop)

```

#### 4. \*Code generation.

\* Show the SaM code generated for input string I2. Indicate the correspondence between code fragments and the AST nodes where they are generated and/or assembled.

```

1  getExpr('( (32+(10+10))*(16-4))'):
2  token = '('
3  lookahead = '('
4
5  getExpr('(32+(10+10))*(16-4))'):
6  token = '('
7  lookahead = '32'
8  token = '32'
9  exp1 = `PUSHIMM 32`
10 token = '+'
11 binop = `ADD`
12 token = '('
13 lookahead = '10'
14
15 getExpr('(10+10)*(16-4))'):
16 token = '('
17 lookahead = '10'
18 token = '10'
19 exp1 = `PUSHIMM 10`
20 token = '+'
21 binop = `ADD`
22 token = '('
23 exp2 = `PUSHIMM 10`
24 token = ')'
25 return expr(exp1+exp2+binop)
26

```

```

27     exp2 = return1
28     token = ')'
29     return expr(exp1+exp2+binop)
30
31 exp1 = return1
32 lookahead = '*'
33 binop = `TIMES`
34 token = '('
35
36     getExpr('(16-4)'):
37         token = '('
38         lookahead = '16'
39         token = '16'
40         exp1 = `PUSHIMM 16`
41         token = '-'
42         binop = `SUB`
43         token = '4'
44         exp2 = `PUSHIMM 4`
45         token = ')'
46         return expr(exp1+exp2+binop)
47
48 exp2 = return2
49 token = ')'
50 return expr(exp1 + exp2 + binop)

```

```

1 // (...) * (...)
2 // (32 + (...))
3 PUSHIMM 32
4 // (10+10)
5 PUSHIMM 10
6 PUSHIMM 10
7 ADD
8 ADD
9
10 // (16 - 4)
11 PUSHIMM 16
12 PUSHIMM 4
13 SUB
14
15 TIMES

```

1. \*Code shape.

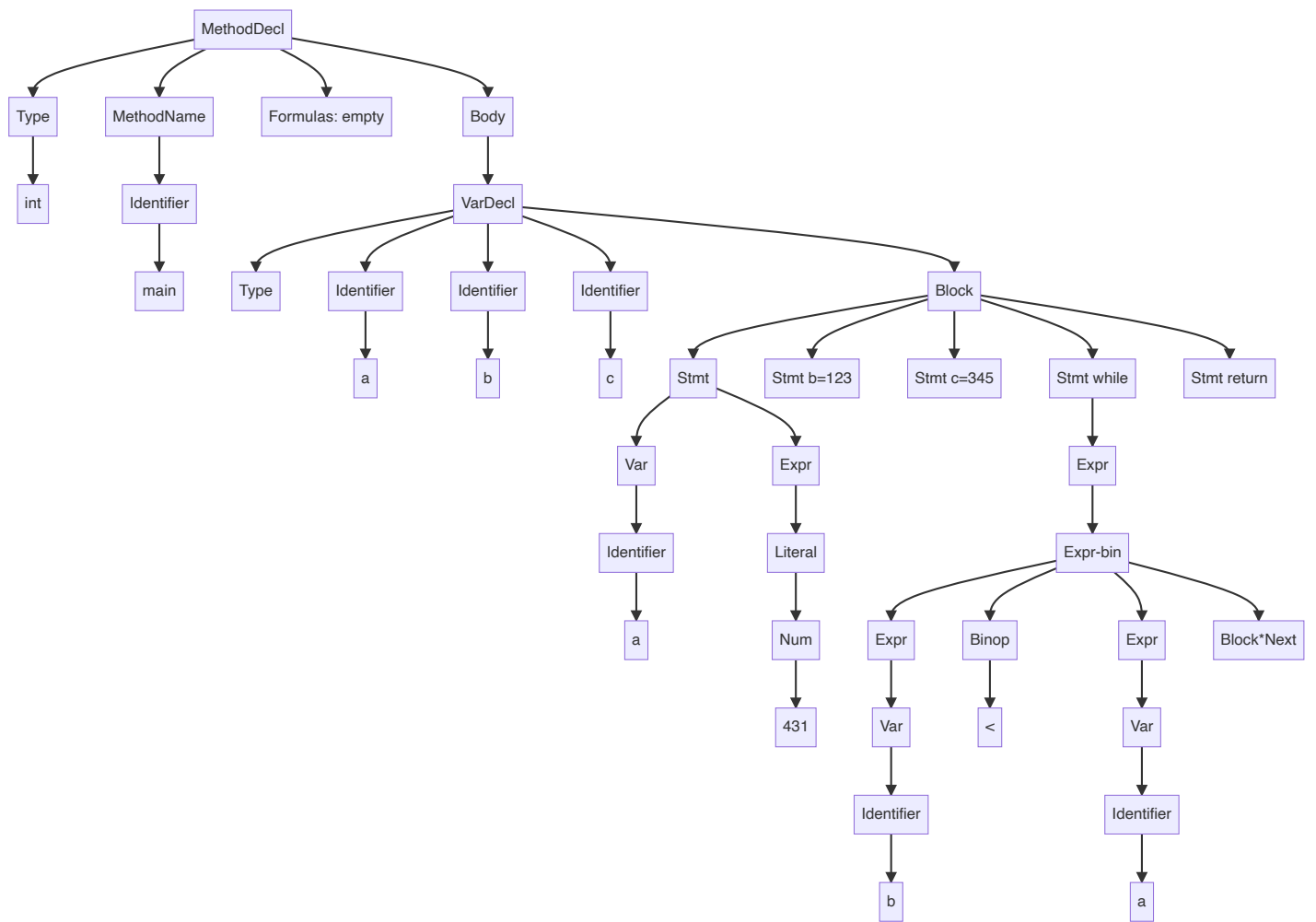
\* Consider the following piece of LiveOak-2 code.

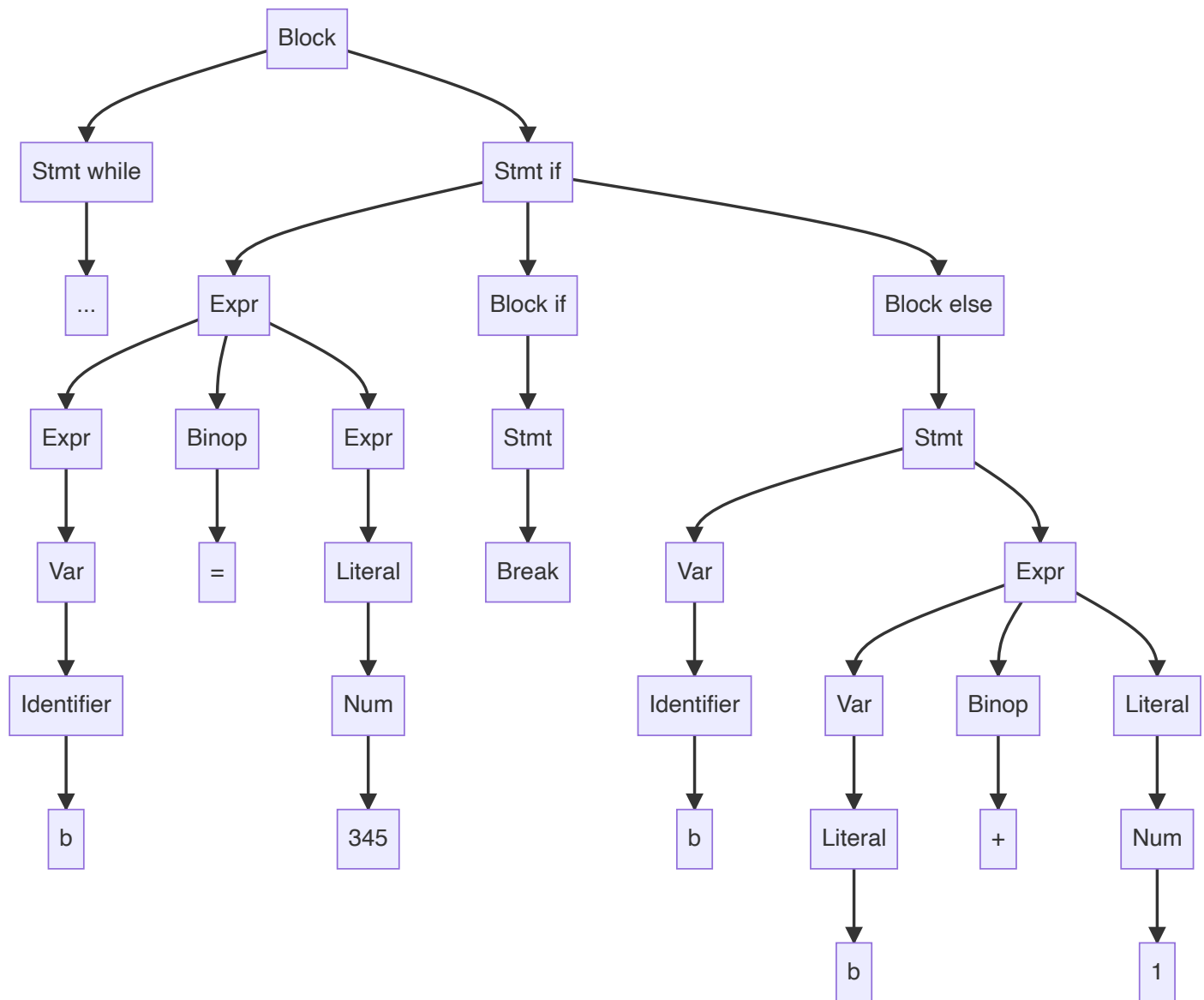
```

1      int main() {
2          int a, b, c;
3
4          a = 431; b = 123; c = 345;
5          while((b<a)) {
6              while((b<c))
7                  b = (b+1);
8              if((b = 345))
9                  break;
10             else
11                 b = (b + 1);
12         }
13         return b;
14     }

```

Work through the actions of the recursive-descent parser and the SaM code generator on this input. Show the structure of the AST and the contents of the symbol table. Indicate how the various code fragments are generated, stitched together, and passed around in the tree representation of the program.





```

1 // MethodDecl
2 // VarDecl int a, b, c
3     ADDSP 3
4     (addToSymbolTable(a,b,c))
5     (numVars = 3)
6
7 // Stmt a = 431
8 // Var a
9     (symbol = symbolTable[a])
10     PUSHOFF symbol.offset // redundant PUSH
11     PUSHIMM 431
  
```



```

12     STOREOFF symbol.offset
13     ADDSP -1 // remove redundant PUSH
14
15 // Stmt b = 123
16     // Var b
17     (symbol = symbolTable[b])
18     PUSHOFF symbol.offset // redundant PUSH
19     PUSHIMM 123
20     STOREOFF symbol.offset
21     ADDSP -1 // remove redundant PUSH
22
23 // Stmt c = 345
24     // Var c
25     (symbol = symbolTable[c])
26     PUSHOFF symbol.offset // redundant PUSH
27     PUSHIMM 345
28     STOREOFF symbol.offset
29     ADDSP -1 // remove redundant PUSH
30
31 // Stmt While
32     // Expr (b<a)
33     // Var b
34     (symbol = symbolTable[b])
35     PUSHOFF symbol.offset
36     // Var a
37     (symbol = symbolTable[a])
38     PUSHOFF symbol.offset
39     LESS
40
41 // Block
42     // Stmt While
43     (Ltop, Lmid = newlabel(), newlabel())
44
45     JUMP Lmid
46
47     Ltop:
48     // Block-Stmt b = (b+1)
49     // Var b
50     (symbol = symbolTable[b])
51     PUSHOFF symbol.offset // redundant PUSH
52     // Expr (b+1)
53     // Var b
54     (symbol = symbolTable[b])
55     PUSHOFF symbol.offset
56     PUSHOFF 1
57     ADD

```

```

58         STOREOFF symbol.offset
59         ADDSP -1 // remove redundant PUSH
60
61     Lmid:
62     // Expr (b<c)
63     // Var b
64         (symbol = symbolTable[b])
65         PUSHOFF symbol.offset
66     // Var c
67         (symbol = symbolTable[c])
68         PUSHOFF symbol.offset
69     LESS
70
71 // Stmt if
72     (Lif, Lbreak = newlabel(), newlabel())
73 // Expr (b = 345)
74     // Var b
75         (symbol = symbolTable[b])
76         PUSHOFF symbol.offset
77     PUSHOFF 1
78     EQUAL
79
80     JUMPC Lif
81
82 // Block else b = (b + 1)
83 // Var b
84     (symbol = symbolTable[b])
85     PUSHOFF symbol.offset // redundant PUSH
86 // Expr (b+1)
87 // Var b
88     (symbol = symbolTable[b])
89     PUSHOFF symbol.offset
90     PUSHOFF 1
91     ADD
92     STOREOFF symbol.offset
93     ADDSP -1 // remove redundant PUSH
94
95     Lif:
96 // Block if
97     // break
98         JUMPC Lbreak
99
100     Lbreak:
101
102 // Stmt return b
103     // Var b

```

```
104      (symbol = symbolTable[b])
105      PUSHOFF symbol.offset
106      JUMP fEnd
107
108 fEnd:
109      STOREOFF -1 // [Q: should return slot always at -1?]
110      ADDSP -numVars
111      JUMPIND
112
```