Exam 2

601.428/628 Compilers and Interpreters

November 7, 2022

Complete all questions.

Time: 75 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: Solution

Print name: ____________

Date: ____________

Question 1. [25 points] The following attributed context-free grammar uses the nonterminal symbol $\boxed{\text{E}}$ and the terminal symbols $\boxed{+ \; - \; \star \; / \; \textit{num}}$.

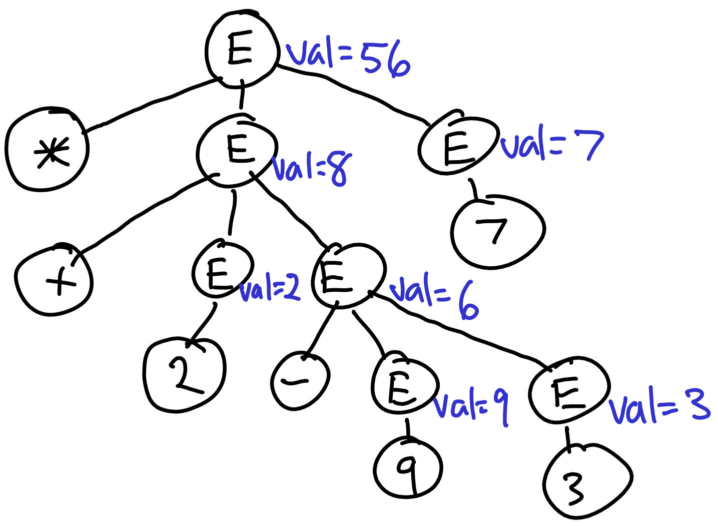
Production	Attribute grammar rules
$E \rightarrow \textit{num}$	$E.\textit{val} \leftarrow$ `valueof(`*num*`)`
$E_0 \rightarrow + \; E_1 \; E_2$	$E_0.\textit{val} \leftarrow E_1.\textit{val} + E_2.\textit{val}$
$E_0 \rightarrow - \; E_1 \; E_2$	$E_0.\textit{val} \leftarrow E_1.\textit{val} - E_2.\textit{val}$
$E_0 \rightarrow \star \; E_1 \; E_2$	$E_0.\textit{val} \leftarrow E_1.\textit{val} \times E_2.\textit{val}$
$E_0 \rightarrow / \; E_1 \; E_2$	$E_0.\textit{val} \leftarrow E_1.\textit{val} \; / \; E_2.\textit{val}$

Note that subscripts are used to distinguish occurrences of E in a production. Also, assume that the lexeme of a *num* terminal symbol is a sequence of digits, and that the `valueof` function returns the numeric value of a *num* symbol based on its lexeme.

(a) Show a derivation for the input $\boxed{\star \; + \; 2 \; - \; 9 \; 3 \; 7}$.

Working string	Production
$\underline{E}$	$E \rightarrow * \; E \; E$
$* \; \underline{E} \; E$	$E \rightarrow + \; E \; E$
$* \; + \; \underline{E} \; E \; E$	$E \rightarrow num$
$* \; + \; 2 \; \underline{E} \; E$	$E \rightarrow - \; E \; E$
$* \; + \; 2 \; - \; \underline{E} \; E \; E$	$E \rightarrow num$
$* \; + \; 2 \; - \; 9 \; \underline{E} \; E$	$E \rightarrow num$
$* \; + \; 2 \; - \; 9 \; 3 \; \underline{E}$	$E \rightarrow num$
$* \; + \; 2 \; - \; 9 \; 3 \; 7$	

(b) Show a parse tree for the derivation you found in (a). Annotate each node in the parse tree to show the value of the *val* attribute, as computed by the attribute rules. For example, if a node for the E nonterminal was derived from the terminal symbols `- 9 3`, the value of its *val* attribute should be 6.



Question 2. [25 points] Consider the following context-free grammar with nonterminal symbols $\boxed{\text{InsList Ins}}$ (start symbol InsList) and terminal symbols $\boxed{\texttt{add sub } \textit{reg}}$:

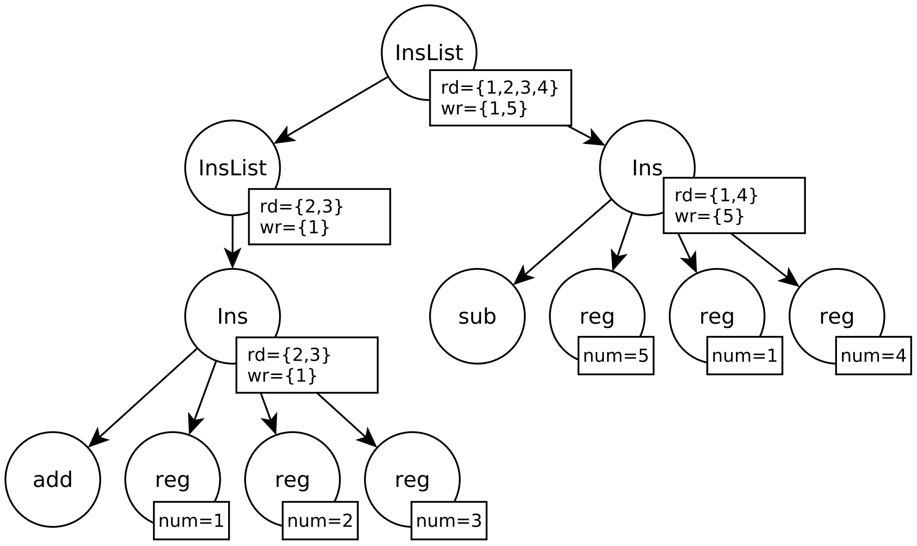
InsList → Ins　　　　　Ins → `add` *reg reg reg*
InsList → InsList Ins　　Ins → `sub` *reg reg reg*

Assume that lexemes for the *reg* terminal symbol have the form `r`n, where n is an integer register number whose value is 1 or greater. The *num* attribute indicates a register number, and the *rd* and *wr* attributes have values which are sets of integers corresponding to the registers that are read and written by `add` and `sub` instructions. Specifically, a register is read if it is the second or third operand of an `add` or `sub` instruction, and a register is written if it is the first operand of an `add` or `sub` instruction.

As an example, after parsing and computation of attributes, the input

```
add r1 r2 r3 sub r5 r1 r4
```

would produce the following attributed parse tree:



On the following page, specify attribute rules for computing the values of the *rd* and *wr* attributes for Ins and InsList nodes. You can assume that a function `regnum` exists to get the register number from a *reg* node. I.e., `regnum(`reg_0`)` would return the register number of the symbol reg_0. Recall that attribute rules are *functional*: there are no mutable variables or state. You may assume that set operations (construction, union, intersection, etc.) are available, since the values of the *rd* and *wr* attributes are sets. Also note that if there are multiple occurrences of a nonterminal symbol in a production, they are distinguished by numeric subscripts.

Production	Attribute grammar rule(s)
InsList → Ins	InsList.*rd* ← Ins.rd
	InsList.*wr* ← Ins.wr
$\text{InsList}_0 \rightarrow \text{InsList}_1$ Ins	$\text{InsList}_0.rd \leftarrow \text{InsList}_1.rd \cup \text{Ins.rd}$
	$\text{InsList}_0.wr \leftarrow \text{InsList}_1.wr \cup \text{Ins.wr}$
Ins → `add` reg_0 reg_1 reg_2	Ins.*rd* ← $\{ \text{regnum}(reg_1), \text{regnum}(reg_2) \}$
	Ins.*wr* ← $\{ \text{regnum}(reg_0) \}$
Ins → `sub` reg_0 reg_1 reg_2	Ins.*rd* ← $\{ \text{regnum}(reg_1), \text{regnum}(reg_2) \}$
	Ins.*wr* ← $\{ \text{regnum}(reg_0) \}$

Question 3. [25 points] This question refers to the C program on the right.

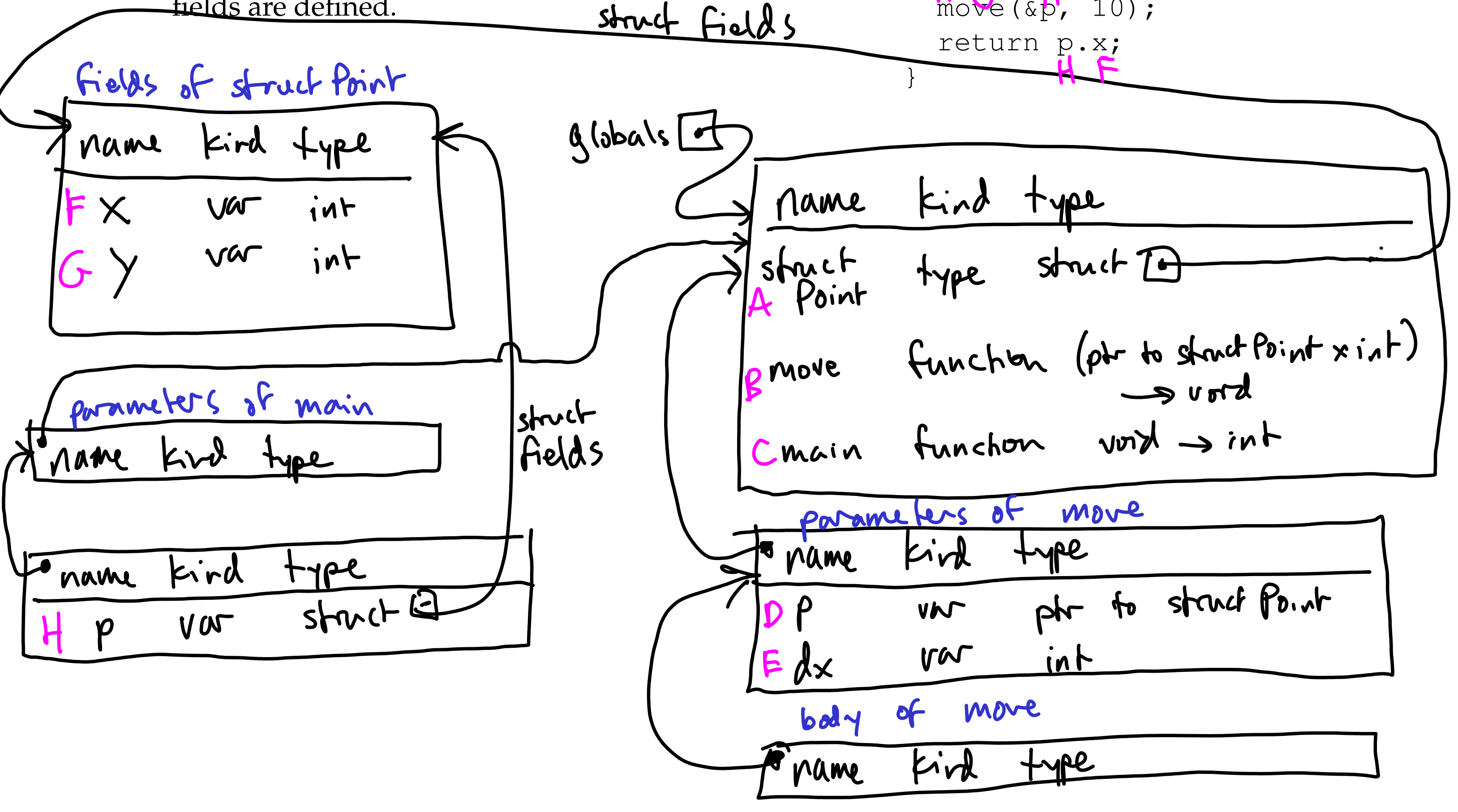
(a) Sketch the contents of each symbol table required to fully represent semantic information about the declarations in the program. For each symbol table entry, be sure to indicate the name (identifier), kind (variable, function, or type), and type representation. If a symbol table has a parent, indicate which symbol table is the parent.

Hint: the representation of a struct type could include a pointer to the symbol table in which its fields are defined.

```
struct Point {
  int x, y;
}

void move(struct Point *p,
          int dx) {
  p->x = p->x + dx;
} D F   D F    E

int main(void) {
  struct Point p;
 H F
  p.x = 2;
  p.y = 3;
 H G  H
  move(&p, 10);
  return p.x;
         H F
}
```



fields of struct Point

	name	kind	type
F	x	var	int
G	y	var	int

globals

	name	kind	type
A	struct Point	type	struct ▢
B	move	function	(ptr to struct Point x int) → void
C	main	function	void → int

parameters of main

name	kind	type

	name	kind	type
H	p	var	struct ▢

parameters of move

	name	kind	type
D	p	var	ptr to struct Point
E	dx	var	int

body of move

name	kind	type

(b) For references to names in the bodies of the `main` and `move` functions, indicate which symbol table entry the name is associated with by labeling each symbol table entry with a unique ID, and then labeling each occurrence of a name in the body of a function with the appropriate ID.

[This page is blank.]

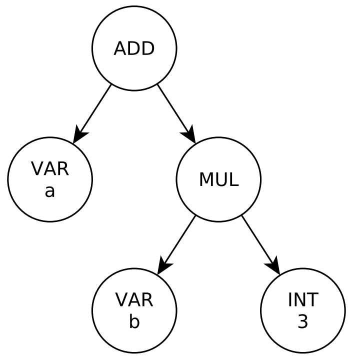
Question 4. [25 points] Consider the following AST node data type:

```
enum ASTTag { ADD, SUB, MUL, DIV, INT, VAR };

struct ASTNode {
  ASTTag tag;            // what kind of AST node is it
  int ival;              // integer value
  std::string varname;   // variable name
  ASTNode *left, *right;
  ASTNode(ASTTag tag, int ival=0) : tag(tag), ival(ival),
    left(nullptr), right(nullptr) { }
  ASTNode(const std::string &varname) : tag(VAR), ival(0),
    varname(varname), left(nullptr), right(nullptr) { }
};
```

Trees constructed using this type represent expressions with operators to perform addition, subtraction, multiplication, and division, and primary expressions (leaf nodes) representing literal integers and references to named variables. For example, the following code would construct the tree shown on the right, representing the infix expression `a + (b * 3)`:

```
ASTNode *ast = new ASTNode(ADD);
ast->left = new ASTNode("a");
ast->right = new ASTNode(MUL);
ast->right->left = new ASTNode("b");
ast->right->right = new ASTNode(INT, 3);
```



Assume that an intermediate representation language supports the following instructions, where R represents a register (`r0`, `r1`, etc.):

Instruction	Meaning
`loadvar` R, *varname*	Load value of variable *varname* into register R
`loadint` R, *intval*	Load value of integer *intval* into register R
`add` $R_{\text{dst}}, R_{\text{src1}}, R_{\text{src2}}$	Store sum $R_{\text{src1}} + R_{\text{src2}}$ in R_{dst}
`sub` $R_{\text{dst}}, R_{\text{src1}}, R_{\text{src2}}$	Store difference $R_{\text{src1}} - R_{\text{src2}}$ in R_{dst}
`mul` $R_{\text{dst}}, R_{\text{src1}}, R_{\text{src2}}$	Store product $R_{\text{src1}} \times R_{\text{src2}}$ in R_{dst}
`div` $R_{\text{dst}}, R_{\text{src1}}, R_{\text{src2}}$	Store quotient $R_{\text{src1}} / R_{\text{src2}}$ in R_{dst}

[Question continues on next page]

Here is a possible translation of the example tree:

```
loadvar r0, a
loadvar r1, b
loadint r2, 3
mul r3, r1, r2
add r4, r0, r3
```

Complete the `codegen` function below, so that it generates a sequence of instructions which carry out the operations represented in the AST passed as the parameter. You may assume that a `nextreg()` function is available which returns an `int` value representing a "fresh" register in which to store the result of evaluating the expression. Also assume that `codegen` will return the register number in which the evaulation result is stored. The function should print the generated instructions using `printf` or `cout`. *Hint*: use recursion.

```
int codegen(ASTNode *ast) {
  int tag= ast->tag;    int r;
  if ( tag == VAR) { r= nextreg();
                     printf(" loadvar r%d, %s\n", r, ast->varname.c_str());}
  else if ( tag== INT) { r= nextreg();
                     printf("loadint r%d, %d\n", r, ast->ival); }
  else { int left= codegen(ast->left), right = codegen(ast->right);
         r= nextreg();
         switch (tag) {
         case ADD: printf("add r%d, ", r); break;
         case SUB: printf("sub r%d, ", r); break;
         case MUL: printf("mul r%d, ", r); break;
         case DIV: printf("div r%d, ", r); break;
         }
         printf ("r%d, r%d\n", left, right );
  }
  return r;
}
```