CSE P 501 23au Exam 11/30/23 **Sample Solution**

Question 1. (16 points) Regular expressions. JSON is a well-known language for encoding values as strings. One kind of value that can be expressed is arrays of strings. An array of strings consists of zero or more values surrounded by square brackets (`[` and `]`) and separated by commas, with no embedded spaces. The individual string values (for this problem) consist of upper- and lower-case letters and digit characters 0 through 9, surrounded by double quote characters (`"`).

Examples in this set: `[]`, `[""]` (empty string), `["foo"]`, `["1","two","3"]`, `["f0o","b4r",""]`

Examples that are not in the set: `["foo",]` (extra comma at end), `[,"foo","bar"]` (extra comma at beginning), `[,]` (extra comma), `[foo]` (string not quoted), `"foo","bar"` (no brackets), `["foo", "bar"]` (space between strings), `["under_score"]` (illegal character in string)

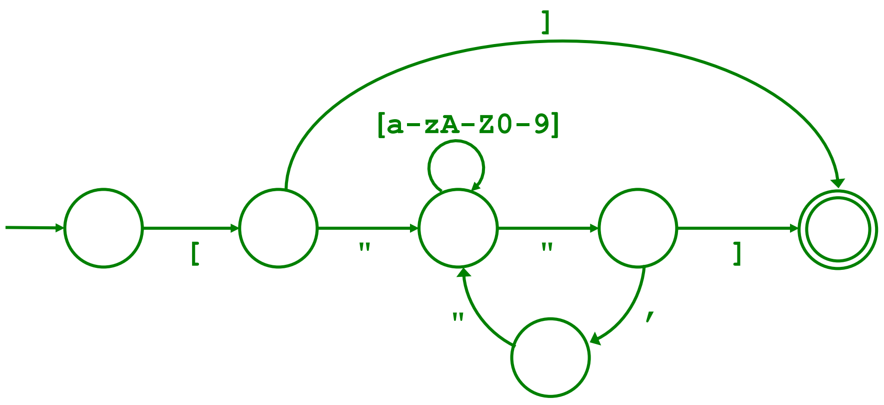
Fine print: You must restrict yourself to the basic regular expression operations covered in class and on homework assignments: *rs* , *r*|*s* , *r** , *r+* , *r?*, character classes like `[a-cxy]` and `[^aeiou]`, abbreviations *name=regexp*, and parenthesized regular expressions. No additional operations that might be found in the "regexp" packages in various Unix programs, scanner generators like JFlex, or programming language libraries are allowed.

(a) (6 points) Give a regular expression that generates all strings in this language. If you need to distinguish between a regular expression operator like `[]` in `[abc]` and literal bracket characters in the input alphabet, underline the literal ones: `["foo"]`.

item = "[a-zA-Z0-9]*" **([ ] here are for character classes)**

[(item(,item)*)?] **([ ] here are literal characters)**

(b) (10 points) Draw a DFA that accepts all strings in this language.



Question 2. (14 points) Scanners. Continuing our exploration of what a MiniJava scanner would do with various kinds of input text, we used our MiniJava scanner to read each of the following four input lines and turn them into tokens. (The lines are numbered for reference. The numbers before each if are not part of the input read by the scanner.)

```
1. if (class / iffy <=4) { _var = -17 } else { /* > case */ }
2. if(class / if fy <= 4) { _var = -17 } else { /* other case */ }
3. if(class / iffy <= 4) { _var = -17 }else { //* > case */ }
4. if (class / iffy < =4) { _ var = - 17 } else { }
```

The MiniJava scanner produced identical token streams for two of these four input lines. The output for the other two input lines produced different token streams.

(a) (4 points) Which two input lines produced the same sets of tokens?

1 and 4

(b) (10 points) Below, list in order the tokens that are returned by a MiniJava scanner for one of the two input lines that had the same output (i.e., what tokens were returned for the input lines that matched, but you only should write that sequence once). If there are any characters in the input line that produced an error, you should write ERROR(#) in the token stream to show where the illegal character was encountered, but using the actual character instead of #. Your scanner output should include all tokens and errors present – i.e., don't stop when (or if) the first error is encountered. You may use any reasonable token names (e.g., LPAREN, ID(x), etc.) as long as your meaning is clear.

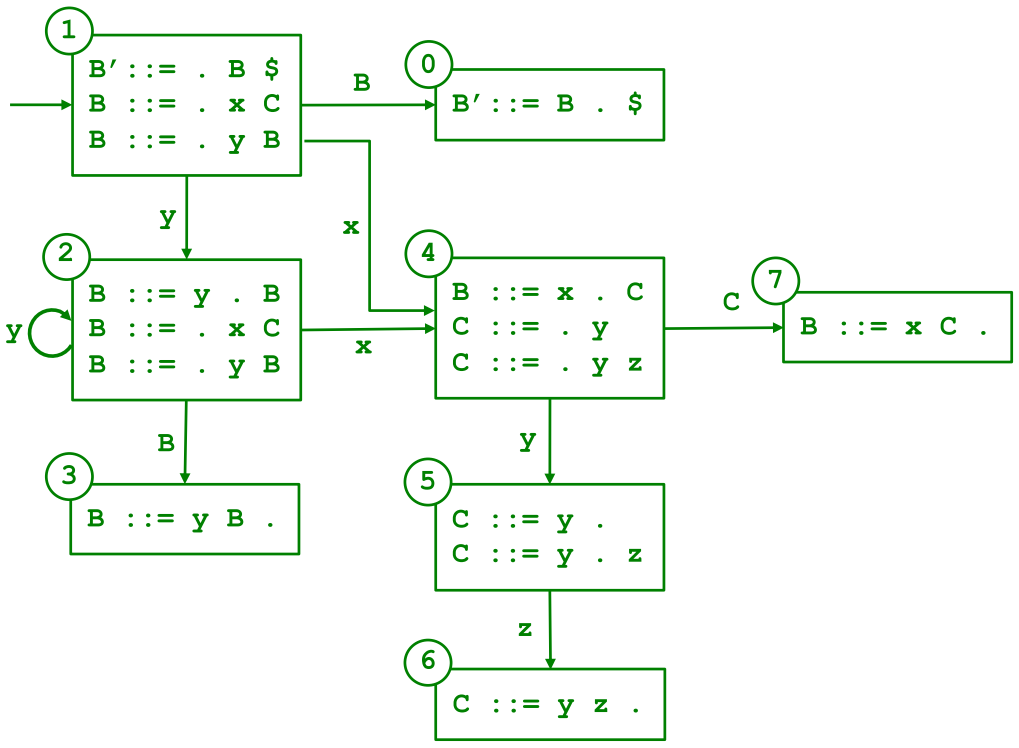
A copy of the MiniJava grammar is attached as the last page of the exam. You should remove it from the exam and use it for reference while you answer this question. You should assume that the scanner processes MiniJava syntax as defined in that grammar, with no extensions or changes to the language. Also recall that the MiniJava language defines an <IDENTIFIER> as a sequence of letters, digits, and underscores, starting with a letter, and uppercase letters are distinguished (different) from lowercase, and an <INTEGER_LITERAL> is a sequence of decimal digits not starting with 0, or the number 0 by itself, denoting a decimal integer value.

IF LPAREN CLASS ERROR(/) ID(`iffy`) LT ASSIGN INT(4) RPAREN LBRACE ERROR(_) ID(`var`) ASSIGN MINUS INT(17) RBRACE ELSE LBRACE RBRACE

Question 3. (30 points) The traditional, just in time for the holidays, parsing question. Consider the following grammar with non-terminals *B* and *C* and terminals `x`, `y`, and `z`. The extra *B′* ::= *B* `$` rule needed to handle end-of-file in an LR parser has been added for you. As usual, whitespace in the grammar is only for readability and is not part of the grammar or the strings generated by it.

0. *B′* ::= *B* `$` (`$` is end-of-file)
1. *B* ::= `x` *C*
2. *B* ::= `y` *B*
3. *C* ::= `y`
4. *C* ::= `y z`

(a) (16 points) Draw the LR(0) state machine for this grammar. When you finish, you should number the states in the final diagram in whatever order you wish so you can use the state numbers to answer later parts of this question.



(continued)

Question 3. (cont.) Grammar repeated from previous page for reference:

0. $B' ::= B\ \$$ ($ is end-of-file)
1. $B ::= \mathrm{x}\ C$
2. $B ::= \mathrm{y}\ B$
3. $C ::= \mathrm{y}$
4. $C ::= \mathrm{y\ z}$

(b) (6 points) Compute *nullable* and the FIRST and FOLLOW sets for the nonterminals *B* and *C* in the above grammar:

Symbol	nullable	FIRST	FOLLOW
B	**no**	**x, y**	**$**
C	**no**	**y**	**$**

(c) (4 points) Is this grammar LR(0)? Explain why or why not. If it is not, your answer should describe **all** of the reasons why this is so by identifying the relevant state number(s) in your diagram from part (a) and the specific issues in those state(s) (i.e., something like "state 47 has a shift-reduce conflict if the next input is q", but with, of course, state numbers and correct details from your diagram). If the grammar is LR(0) you should explain why. You do not need to write out the full LR(0) parse tables as part of your answer, just explain whether the state machine is LR(0) or not, and why.

No. State 5 has a shift-reduce conflict if the next input symbol is z.

(d) (4 points) Is this grammar SLR? Explain why or why not. As with your answer to the previous part of the question, refer to states by number in the LR diagram in your answer to part (a) if needed and give specific explanations of why the grammar is SLR or why not.

Yes. z is not in FOLLOW(*C*), so in state 5 if the next input is not z we should reduce *C* ::= y. If the next input is z, we should shift to state 6.

Question 4. (8 points) LL parsing. Here is another look at the grammar from the previous question. The extra B' ::= B `$` production needed to handle end-of-file in the LR parser has been omitted since it is not needed here.

0. B ::= `x` C
1. B ::= `y` B
2. C ::= `y`
3. C ::= `y z`

Is this grammar, as written, suitable for constructing a top-down LL(1) predictive parser? If it is, your answer should give a technical explanation why it is. If not, your answer should give a technical explanation listing **all** of the problems with this particular grammar that prevent it from being suitable for a LL(1) predictive parser. You do not need to rewrite the grammar to fix the problems if there are any – just explain why it is or is not suitable for this use.

Hint: Explanations in terms of FIRST/FOLLOW sets calculated in the previous question and other properties needed by a LL(1) predictive parser may be helpful.

No. y is in the FIRST set for both rules 2 and 3 for *C*. With 1-symbol lookahead we cannot pick which of these productions to use in a top-down parser.

Question 5. (16 points) x86-64 code. This question concerns the translation to x86-64 code of a very simple C function to return the largest of three integer arguments.

Suppose we have the following library function that we can use without having to further declare or implement it.

```
// return the largest value of x or y.  If x and y have the
// same value, return that value.
int max(int x, int  y) { ... }
```

For this problem, translate the following function `max3` to x86-64 assembly language, and write your answer on the next page.

```
// return the largest value of a, b, or c.  If there is
// a tie for the largest value, return that largest value.
int max3(int a, int b, int c) {
  return max(max(a,b), c);
}
```

You must use the Linux/gcc x86-64 assembly language, and must follow the x86-64 function call, register, and stack frame conventions:

- Argument registers: %rdi, %rsi, %rdx, %rcx, %r8, %r9 in that order
- Called function must save and restore %rbx, %rbp, and %r12-%r15 if these are used in the function; all other registers may be changed by a function call
- Function result returned in %rax
- %rsp must be aligned on a 16-byte boundary when a call instruction is executed
- %rbp must be used as the base pointer (frame pointer) register for this exam, even though this is not strictly required by the x86-64 specification.
- Pointers, Booleans, and ints are 64 bits (8 bytes) each, as in MiniJava.
- Your x86-64 code must be a straightforward translation of the given code. You may not rewrite or rearrange the code even if it produces equivalent results. However, you can use any reasonable x86-64 code that follows the standard function call and register conventions – you do not need to mimic the code produced by a MiniJava compiler. In particular, these are C functions, not Java methods. Ordinary C calling conventions without object vtables should be used.
- Please include *brief* comments in your code to help us understand what the code is supposed to be doing (which will help us assign partial credit if it doesn't do exactly what you intended.)

As you can tell, the main emphasis in this question is the proper handling of function calls, parameters, and stack frames.

(continued on next page)

Queston 5. (cont.) Write your x86-64 version of function `max3` below. Code repeated below for convenience. If you don't remember the exact assembly code for something, write down your best attempt and add a comment if needed to explain it. We will take that into account when grading.

Hint: be careful about what might happen to registers when function `max` is called.

```
int max3(int a, int b, int c) {
  return max(max(a,b), c);
}
```

One important thing here is to be sure to save the third argument (`c`) to `max3` in memory when calling `max` for the first time. Any function call can potentially alter any or all of the argument registers, so we cannot count on `%rdx` having its previous value after calling `max`.

```
;; incoming arguments %rdi = a, %rsi = b, %rdx = c

max3:   pushq   %rbp            # save frame pointer
        movq    %rsp,%rbp       # set new frame pointer
        subq    $16,%rsp        # allocate stack frame for c
        movq    %rdx,-8(%rbp)   # save c around function call
        call    max             # a, b still in arg registers
        movq    %rax,%rdi       # move max(a,b) to 1st arg
        movq    -8(%rbp),%rsi   # load c into 2nd arg reg
        call    max             # compute final result
        movq    %rbp,%rsp       # restore %rsp, %rbp
        popq    %rbp            #   (could also use leave)
        ret                     # return; result in %rax
```

Question 6. (24 points) Compiler hacking. Now that we have created a MiniJava compiler, we can use it to experiment with new language features. We would like to experiment with a new two-character `?:` expression operator (sometimes called the "Elvis operator") to make it easier to handle potential null pointer values safely. The meaning of the expression *e1* `?:` *e2* is: first evaluate *e1*. If *e1* is not `null`, then the value of the *e1* `?:` *e2* expression is the value of *e1*. If *e1* is `null`, then evaluate *e2*, and the value of the *e1* `?:` *e2* expression is the value of *e2*. Expression *e2* is not evaluated unless *e1* is `null`. (*e2* is allowed to evaluate to `null`, although that doesn't matter for this problem.) For example, in this statement, `result` is assigned the value `default` if `thing.f()` returns `null`:

```
   result = thing.f() ?: default;
```

Note: the basic definition of MiniJava does not include `null` as a constant expression value. You should assume that we have already added `null` to the language, and it represents an 8-byte reference value (pointer) with the value `0x0` (binary zero). Your answer to this question will not need to use the identifier `null`, but you will need to use the fact that a null pointer has the value `0x0`.

Answer the questions below about how this new operator would be added to a MiniJava compiler. There is likely way more space than you will need for some of the answers. The full MiniJava grammar is attached at the end of the exam if you need to refer to it.

(a) (2 points) What new lexical tokens, if any, need to be added to the scanner and parser of our MiniJava compiler to add this new `?:` operator to the original MiniJava language? Just describe any necessary changes and new token name(s) needed. You don't need to give JFlex or CUP specifications or code in this part of the question, but you will need to use any token name(s) you write here in a later part of this question.

Need a new token ELVIS for ?: .

(continued on next page)

Question 6. (cont.) (b) (6 points) Complete the following new AST class to define an AST node type for this new operator. You only need to define instance variables and the constructor. Assume that all appropriate package and import declarations are supplied, and don't worry about visitor code.

(Hint: recall that the AST package in MiniJava contains the following key classes: `ASTNode`, `Exp extends ASTNode`, and `Statement extends ASTNode`. Also remember that each AST node constructor has a `Location` parameter, and the supplied `super(pos);` statement at the beginning of the constructor below is used to properly initialize the superclass with this information.)

```
public class Elvis extends Exp {
  // add any needed instance variables below

     Exp e1, e2;

  // constructor - add parameters and method body below

  public Elvis ( Exp e1, Exp e2, Location pos ){
    super(pos);   // initialize location information in superclass
    this.e1 = e1;
    this.e2 = e2;
  }
}
```

(continued on next page)

Question 6. (cont.) (c) (5 points) Complete the CUP specification below to define a production for this new operator, including associated semantic action(s) needed to parse the new expression and create an appropriate AST node (as defined in part (b) above). You should use any new lexical tokens defined in your answer to part (a) as needed. Use reasonable names for any other lexical tokens that already would exist in the compiler scanner and parser if you need them. We have added additional code to the parser rule for `Expression` below so the CUP specification for the new operator can be written as an independent grammar rule with separate semantic actions.

Hint: recall that the `Location` of an item `foo` in a CUP grammar production can be referenced as `fooxleft`.

```
Expression ::= ...
     | ElvisExp:e  {: RESULT = e; :}
    ...
     ;
ElvisExp ::=
```

```
Exp:e1 ELVIS:e Exp:e2

{: RESULT = new Elvis(e1, e2, exleft); :}


// Note: it would also be ok to the location of any
// component of the grammar rule for the location in
// the new Elvis node constructor.
```

(d) (4 points) Describe the checks that would be needed in the semantics/type-checking part of the compiler to verify that a program containing this new operator is legal. You do not need to give code for a visitor method or anything like that – just describe what language rules (if any) need to be checked for this new operator.

There are a couple of possible requirements for the types of e1 and e2. The simplest (as in the 3-argument ? : operator in Java) would be to require that e1 and e2 be reference types and have the same type, and that type is the result type of the ?: operator. Some languages that include the elvis operator allow e2 to be assignment-compatible with the type of e1, basically being a subclass since this operator is restricted to reference types, and in that case the type of the operator is the type of e1. Any reasonable answer along these lines received full credit.

(continued on next page)

Question 6. (cont.) (e) (7 points) Describe the x86-64 code shape for this new operator that would be generated by a MiniJava compiler. Your answer should be similar in format to the descriptions we used in class for other language constructs. If needed, you should assume that the code generated for an expression will leave the value of that expression in %rax, as in the MiniJava project.

Use Linux/gcc x86-64 instructions and assembler syntax when needed. If you need to make any additional assumptions about code generated by the rest of the compiler you should state them.

Hint: be sure that your code follows the described operation and semantics of the new `?:` operator precisely.

The main thing to watch for here is to be sure to evaluate *e2* only if *e1* evaluates to null (0x0).

```
        Visit e1 to generate code. Leaves result in %rax
        cmpq  $0,%rax       # or tstq %rax,$rax
        jnz   done          # result in %rax is final result if not 0
        Visit e2 to generate code.  Leaves result in %rax
    done:     #end of code for ?:
```

Question 7. (14 points) Value numbering. For the following sequence of code, (i) apply local value numbering to the statements in the block, and (ii) rewrite the code to eliminate redundant expressions using the value numbering information. Your value-numbered version of the code should show both version numbers and value numbers for variables, i.e., x_v^n is version *v* of variable *x* with value number *n*. We suggest you use the table at the bottom to keep track of value numbers for variables and expressions, and to help the reader award partial credit if any errors are encountered, but this is not required. The first two entries in the table are given for you to help get started.

Original	With VNs	Rewritten
a = b * c	$a_0^3 = b_0^1 * c_0^2$	$a_0^3 = b_0^1 * c_0^2$
d = a * e	$d_0^5 = a_0^3 * e_0^4$	$d_0^5 = a_0^3 * e_0^4$
g = c * b	$g_0^3 = c_0^2 * b_0^1$	$g_0^3 = a_0^3$
a = d / e	$a_1^6 = d_0^5 / e_0^4$	$a_1^6 = d_0^5 / e_0^4$
z = g * e	$z_0^5 = g_0^3 * e_0^4$	$z_0^5 = d_0^5$

VN Table

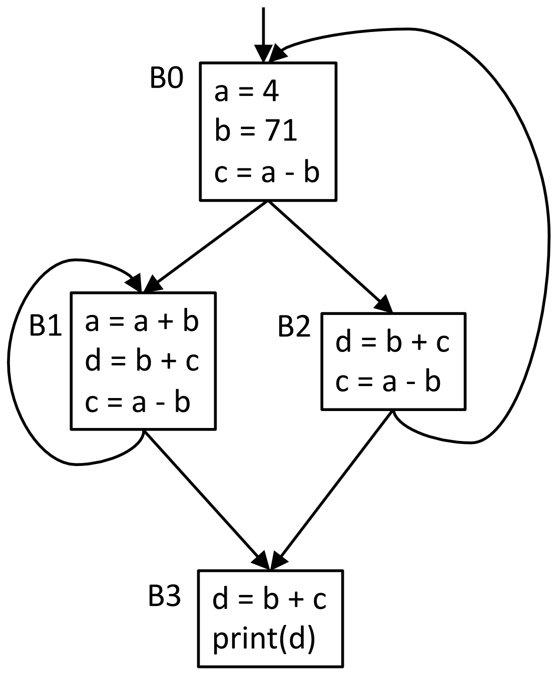
expr	vn
b_0	1
c_0	2
< * 1 2 >	3
a_0	3
e_0	4
< * 3 4 >	5
d_0	5
g_0	3
< / 5 4 >	6
a_1	6
z_0	5

Table continued if needed

expr	vn

Note: For commutative operations like * and +, we sort the value numbers of the operands so we can recognize that, for example, <* 2 1> is the same as <* 1 2>. We can't do that for subtraction and division since those operations are not commutative.

Question 8. (18 points) Dominators and SSA. We would like to convert the following flowgraph to SSA form.



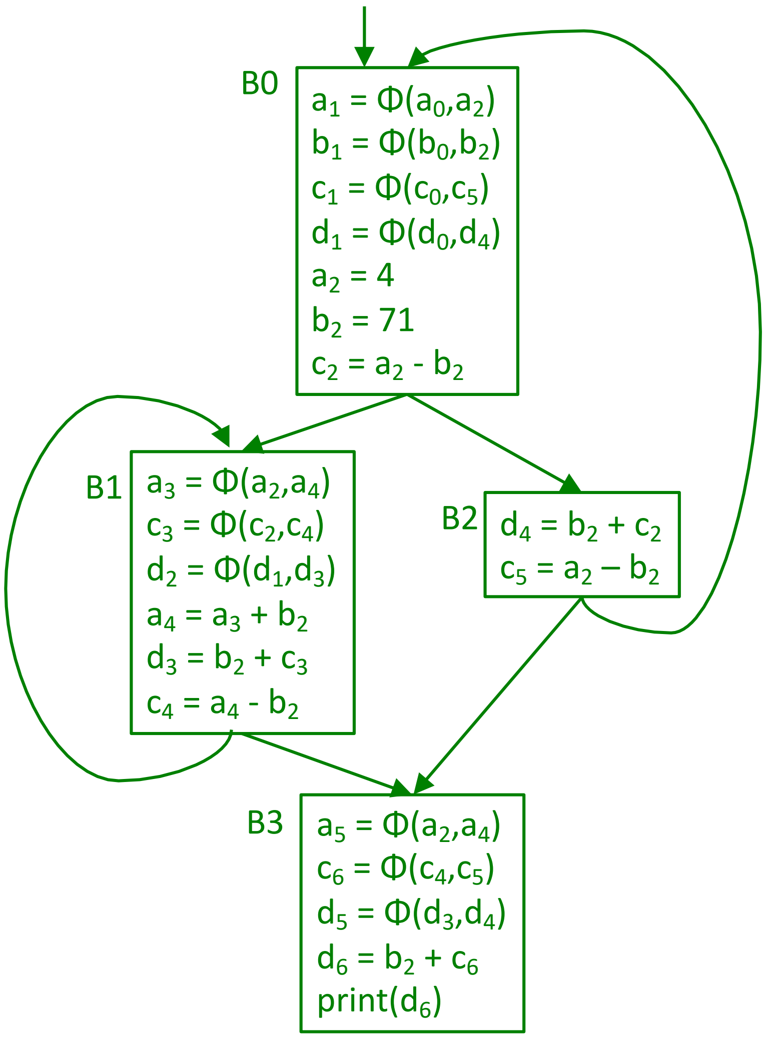
Here are the basic definitions of dominators and related concepts we have seen previously in class. (Note: read this carefully – it is somewhat different than the definitions in prior exams to be consistent with the presentation of SSA in class this quarter.)

- Every control flow graph has a unique **start node** s0.
- Node *x* **dominates** node *y* if every path from s0 to *y* must go through x.
  - A node *x* dominates itself.
- A node *x* **strictly dominates** node *y* if *x* dominates *y* and $x \neq y$.
- The **dominator set** of a node *x* is the set of nodes *dominated by x*.
  - $|\text{Dom}(x)| \geq 1$
  - (note: sometimes the definition of Dom(x) is given as the set of all nodes that dominate *x*. For SSA it is more convenient to keep track of the set of nodes that *x* dominates.)
- An **immediate dominator** of a node *y*, idom(*y*), has the following properties:
  - idom(*y*) strictly dominates *y* (i.e., dominates *y* but is different from *y*)
  - idom(*y*) does not dominate any other strict dominator of *y*
- The **dominator tree** of a control flow graph is a tree where there is an edge from the immediate dominator of a node x (idom(x)) to node *x*.
- The **dominance frontier** of a node *x* is the set of all nodes *w* such that
  - *x* dominates a predecessor of *w*, but
  - *x* does not strictly dominate *w*

Question 8. (cont.) (a) (8 points) Fill in the following table with information about this flowgraph showing dominance relationships and the dominance frontiers of each node.

Node	Nodes dominated by this node	Successors of dominated nodes	Dominance Frontier of this node
B0	**B0, B1, B2, B3**	**B0, B1, B2, B3**	**B0**
B1	**B1**	**B1, B3**	**B1, B3**
B2	**B2**	**B0, B3**	**B0, B3**
B3	**B3**	**---**	**---**

(b) (10 points) Now redraw the flowgraph in SSA (static single-assignment) form. Your answer should include all of the Φ-functions required by the Dominance Frontier Criteria (or alternatively the path convergence criteria, which places the same set of Φ-functions), but no additional ones. It should include all Φ-functions that satisfy the Dominance Frontier Criteria even if some of those are assignments to variables that are never used (i.e., dead assignments). Answers that have a couple of extraneous Φ-functions will receive appropriate partial credit, but answers that, for example, use a maximal-SSA strategy of placing Φ-functions for all variables at the beginning of every block will not be looked on with favor.


B0
a₁ = Φ(a₀,a₂)
b₁ = Φ(b₀,b₂)
c₁ = Φ(c₀,c₅)
d₁ = Φ(d₀,d₄)
a₂ = 4
b₂ = 71
c₂ = a₂ - b₂
B1
a₃ = Φ(a₂,a₄)
c₃ = Φ(c₂,c₄)
d₂ = Φ(d₁,d₃)
a₄ = a₃ + b₂
d₃ = b₂ + c₃
c₄ = a₄ - b₂
B2
d₄ = b₂ + c₂
c₅ = a₂ – b₂
B3
a₅ = Φ(a₂,a₄)
c₆ = Φ(c₄,c₅)
d₅ = Φ(d₃,d₄)
d₆ = b₂ + c₆
print(d₆)