CS131 Compilers: Writing Assignment 3
Due Sunday, May 1, 2022 at 23:59pm

Tian Haoyuan - 2020533013

This assignment asks you to prepare written answers to questions on semantic analysis. Each of the questions has a short answer. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work. and you should indicate in your submission who you worked with, if applicable. Written assignments are turned in at the start of lecture. You should use the Latex template provided at the course web site to write your solution.

I worked with: nobody

Example for type rule in tex:

$$\frac{O[\text{Bo/x}][\text{Ob/x}](\text{x}) = \text{Ob}}{O[\text{Bo/x}][\text{Ob/x}] \vdash \text{x: Ob}}$$

1. (4x2 pts) **3-AC.** a = b*(minus c) + b*(minus c)

   - Give 3-address code.
   - Give quadruples table.
   - Give triples table.
   - Give indirect triples table.

- $t_1 = minus\ c$
  $t_2 = b * t_1$
  $t_3 = minua\ c$
  $t_4 = b * t_3$
  $a = t_2 + t_4$
  $a = t_5$

	op	result	arg1	arg2
(1)	minus	t_1	c	
(2)	*	t_2	b	t_1
(3)	minus	t_3	c	
(4)	*	t_4	b	t_3
(5)	+	t_5	t_2	t_4
(6)	=	a	t_5	

	op	arg1	arg2
(1)	minus	c	
(2)	*	b	(1)
(3)	minus	c	
(4)	*	b	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

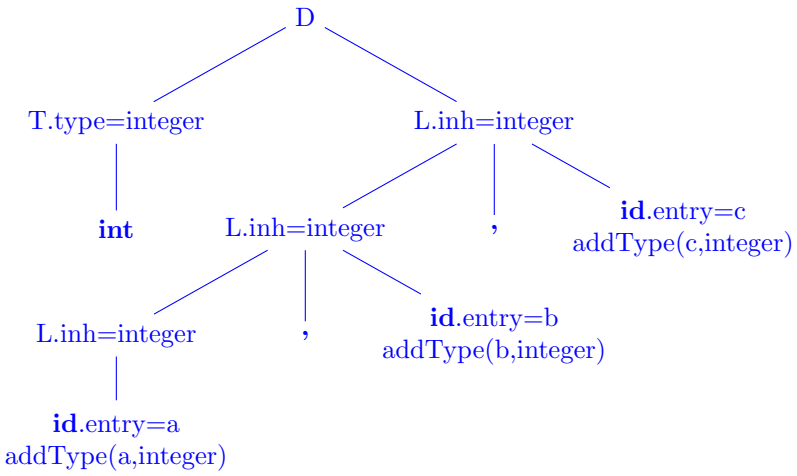
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

	op	arg1	arg2
(11)	minus	c	
(12)	*	b	(11)
(13)	minus	c	
(14)	*	b	(13)
(15)	+	(12)	(14)
(16)	=	a	(15)

2. (2x4 pts) **SDD.**

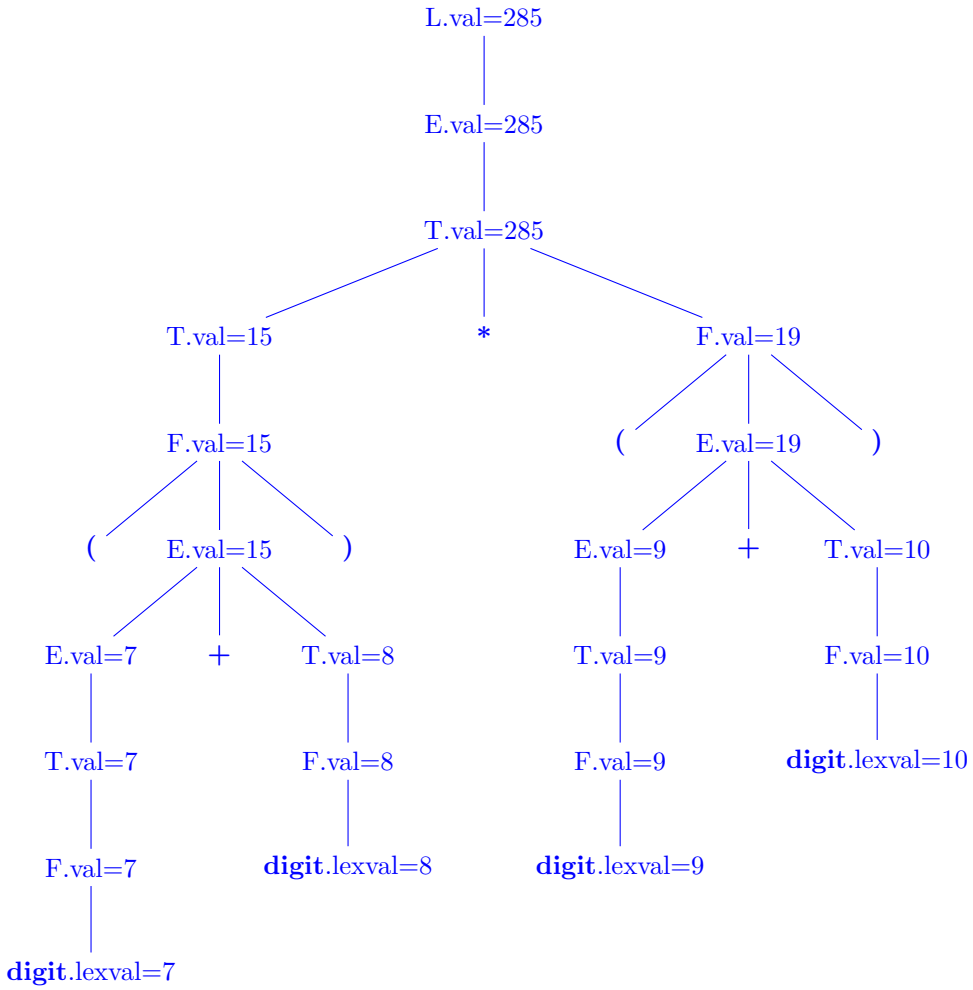
(a) Given Syntax-Directed Definition below construct the annotated parse tree for the input express "int a, b, c".

D → TL	L.inh = T.type
T → int	T.type = integer
T → float	T.type = float
L → L_1, id	L_1.inh = L.inh
	addType(id.entry, L.inh)
L_1 → id	addType(id.entry, L.inh)



(b) Given the Syntax-Directed Definition below with the synthesized attribute *val*, draw the annotated parse tree for the express "(7 + 8)*(9 + 10)".

L → E	L.val = E.val
E → T	E.val = T.val
E → E_1 + T	E.val = E_1.val + T.val
T → F	T.val = F.val
T → T_1 * F	T.val = T_1.val * F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval



3. (4x2 pts) **SDD.** Construct a Syntax-Directed Translation scheme that takes strings of a's, b's and c's as input and produces as output the number of substrings in the input string tha correspond to the pattern a(a—b)*c+(a—b)*b. For example the translation of the input string "abbcabcababc" is 3 ("abbcab", "abcab", "abcabab").

   - Create a CFG that generates all string of a's, b's and c's.
     The CFG: $G = \{\{a, b, c\}, \{S\}, S, P\}$, where the production law P is as follows.

$$\begin{array}{lcl} S & \rightarrow & Sa \\ S & \rightarrow & Sb \\ S & \rightarrow & Sc \\ S & \rightarrow & a \\ S & \rightarrow & b \\ S & \rightarrow & c \end{array}$$

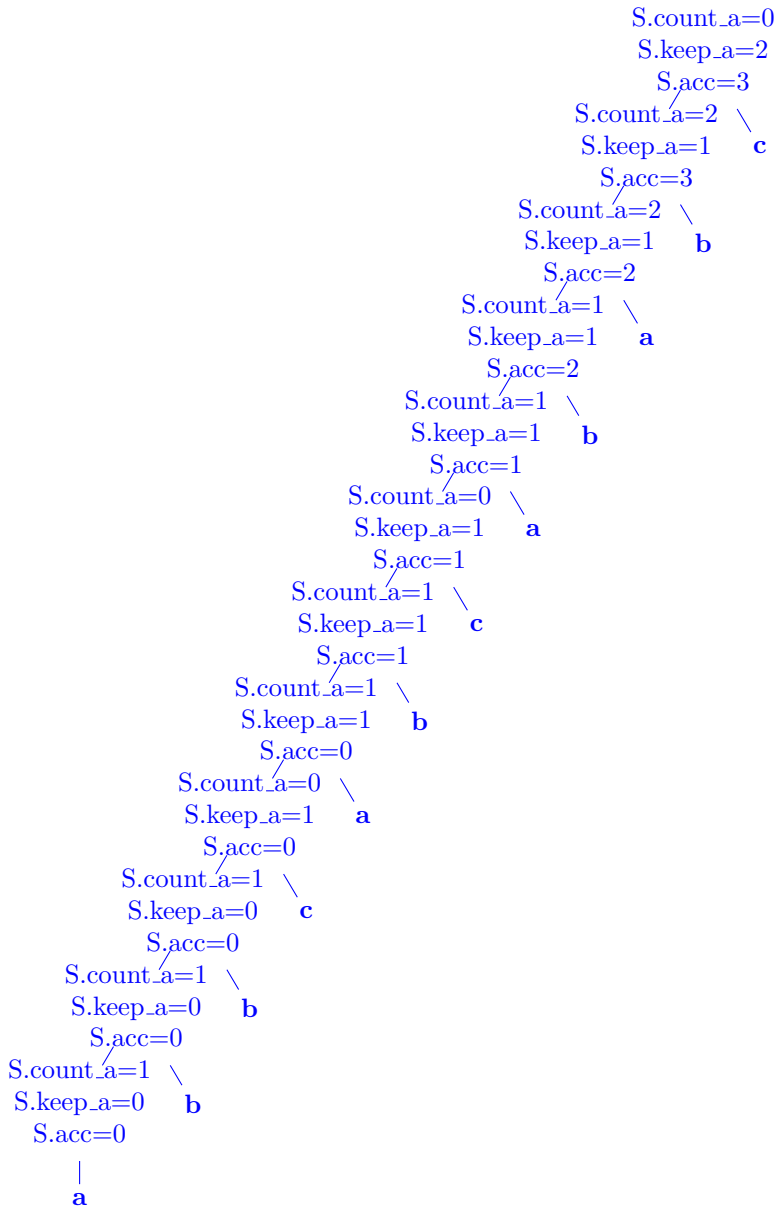
   - Semantic attributes for the grammar symbols.
     Define 3 synthesized attributes for S:
     count_a, which accumulates the number of 'a's that are left to a given 'c';
     keep_a, which keeps the number of 'a's that were left to a given 'c';
     acc, which counts the number substrings that satisfy the requirement.
     Explanation:

In a substring under consideration, since only the number of 'a' matter before a given 'c', so we keep track on the nunmber of 'a' with count_a; after a given 'c' in a substring, only the number of 'b' matters, by multiplying the number of 'a' before that c, so it is necessary to keep the number of 'a' in another attribute keep_a, since count_a would turn to count the number of 'a' for next 'c'.

- For each production of the grammar a set of rules for evaluation of the semantic attributes.

$$
\begin{array}{lcll}
S & \rightarrow & S_1 a & \{S.count\_a = S_1.count\_a + 1; S.keep\_a = S_1.keep\_a; S.acc = S_1.acc;\} \\
S & \rightarrow & S_1 b & \{S.count\_a = S_1.count\_a; S.keep\_a = S_1.keep\_a; S.acc = S_1.acc + S_1.keep\_a;\} \\
S & \rightarrow & S_1 c & \{S.count\_a = 0; S.keep\_a = S_1.count\_a; S.acc = S_1.acc;\} \\
S & \rightarrow & a & \{S.count\_a = 1; S.keep\_a = 0; S.acc = 0;\} \\
S & \rightarrow & b & \{S.count\_a = 0; S.keep\_a = 0; S.acc = 0;\} \\
S & \rightarrow & c & \{S.count\_a = 0; S.keep\_a = 0; S.acc = 0;\}
\end{array}
$$

- Justification that your solution is correct.
Here's the annotated parse tree of string "abbcabcababc".



Since $S.count\_a$ counts the number of 'a' of the substring under consideration, and when it encounters 'c', we start to count for the next substring instead of the last one, and we save the number of 'a' to $S.keep\_a$ as a constant. Each time it encounters a 'b', we can find $1 \times S.keep\_a$ substrings that satisfy the requirement. So, keeping an eye on each 'b' in the whole string, which appears as an end of a qualified substring, we can find all of these substrings, that is "abbcab", "abcab", "abcabab".

4. (3x4 pts) **Type checking.** Give the Semantic Rule. All the attributes are synthesized.

   - The syntax directed definition for associating a type to an Expression if *num* stands for an integer number. ↑ means pointer.

   $$\text{E} \rightarrow \text{num} \mid \text{id} \mid \text{E mod E} \mid \text{E[E]} \mid \text{E} \uparrow$$

$E \rightarrow num$	{ $E.type = integer$;}
$E \rightarrow id$	{ $E.type = loopup(id.entry)$;}
$E \rightarrow E_1\ mod\ E_2$	{ if($E_1.type == integer$ && $E_2.type == integer$) { $E.type = integer$; } else{ E.type = type_error; } }
$E \rightarrow E_1[E_2]$	{ if($E_1.type == array(s,t)$ && $E_2.type == integer$) { $E.type = t$; } else{ E.type = type_error; } }
$E \rightarrow E_1 \uparrow$	{ if($E_1.type == pointer(t)$) { $E.type = t$; } else{ E.type = type_error; } }

   - The syntax directed definition for associating a type to a Statement. E is the expression in 5.1.

   $$\text{S} \rightarrow \text{id := E} \mid \text{if E then S} \mid \text{while E do S} \mid \text{S; S}$$

$S \rightarrow id := E$	{ if($id.type == E.type$) { $S.type = void$; } else { S.type = type_error; } }
$S \rightarrow if\ E\ then\ S1$	{ if($E.type == Bool$) { $S.type = S_1.type$; } else { S.type = type_error; } }
$S \rightarrow while\ E\ do\ S1$	{ if($E.type == Bool$) { $S.type = S_1.type$; } else { S.type = type_error; } }
$S \rightarrow S_1; S_2$	{if($S_1.type == void$ &&$S_2.type == void$){$S.type = void$;}else{S.type = type_error;}}

   - The syntax directed definition for associating a type to a function.

   F → fun id(D): T; B
   D → id: T | D; D
   T → char | int | array[num] of T | T
   B → {S}
   S → id(EList)
   EList → E | Elist, E

$F \rightarrow fun\ id(D) : T; B$	{ $id.type = D.type \rightarrow T.type$; }
$D \rightarrow id : T$	{ $D.type = T.type$; $addType(id.entry, T.type)$; }
$D \rightarrow D_1; D_2$	{ $D.type = D_1.type \times D_2.type$; }
$T \rightarrow char$	{ $T.type = char$; }
$T \rightarrow int$	{ $T.type = int$; }
$T \rightarrow array[num]\ of\ T_1$	{ $T.type = list(T_1.type)$; }
$T \rightarrow T_1$	{ $T.type = T_1.type$; }
$B \rightarrow \{S\}$	
$S \rightarrow id(Elist)$	{if($Elist.type == s$ && $id.type = s \rightarrow t$){$S.type = t$;}else{$S.type = type\_error$;}}
$Elist \rightarrow E$	{ $Elist.type = E.type$; }
$Elist \rightarrow Elist_1, E$	{ $Elist.type = Elist_1.type \times E.type$; }

5. (10 pts) **Type Inference.** Consider the following class definitions.

```
class A {
  i: Int
  o: Object
  b: B <- new B
  x: SELF_TYPE
  f(): SELF_TYPE {x}
}
class B inherits A {
  g(b: Bool): Object { (* EXPRESSION *) }
}
```

Assume that the type checker implements the rules described in the lectures and in the Cool Reference Manual. For each of the following expressions, occurring in place of (* EXPRESSION *) in the body of the method *g*, show the static type inferred by the type checker for the expression. If the expression causes a type error, give a brief explanation of why the appropriate type checking rule for the expression cannot be applied.

(a) `x`

(b) `self = x`

(c) `self = i`

(d) `let x: B <- x in x`

(e)
```
case o of
      o: Int => b;
      o: Bool => o;
      o: Object => true;
esac
```

(a) $SELF\_TYPE_B$

(b) Bool

(c) Error, self and Int cannot be compared, both expresssions should have the same static type, Int for example.

(d) B

(e) Bool

6. (10 pts) **Type Inference.** Show the full type *derivation tree* for the following judgement: (You can use I as the type Int and Ob as the type Object)

$$O[\text{Int/x}] \vdash \texttt{x <- (let x:Object <- x in x = x): Int}$$

$$\dfrac{\dfrac{O[\text{I/x}](\text{x}) = \text{I}}{O[\text{I/x}] \vdash \text{x: I}} \quad \text{Bool} \leq \text{I} \qquad \dfrac{\dfrac{O[\text{I/x}](\text{x}) = \text{I}}{O[\text{I/x}] \vdash \text{x: I}} \quad \text{I} <= \text{Ob} \qquad \dfrac{\dfrac{O[\text{I/x}][\text{Ob/x}](\text{x}) = \text{Ob}}{O[\text{I/x}][\text{Ob/x}] \vdash \text{x: Ob}} \quad \dfrac{O[\text{I/x}][\text{Ob/x}](\text{x}) = \text{Ob}}{O[\text{I/x}][\text{Ob/x}] \vdash \text{x: Ob}}}{O[\text{I/x}][\text{Ob/x}] \vdash \text{x = x: Bool}}}{O[\text{I/x}] \vdash \text{let x:Ob <- x in x = x: Bool}}}{O[\text{I/x}] \vdash \text{x <- (let x:Ob <- x in x = x): I}} \tag{1}$$

7. (10 pts) **Java Array.** The Java programming language includes arrays. The Java language specification states that if s is an array of elements of class S, and t is an array of elements of class T, then the assignment $s = t$ is allowed as long as T is a subclass of S. This typing rule for array assignments turns out to be unsound. (Java works around the fact that this rule is not statically sound by inserting runtime checks to generate an exception if arrays are used unsafely. For this question, assume there are no special runtime checks.)

   Consider the following Java program, which type checks according to the preceeding rule:

```
class Felidae { String name; }

class Cat extends Felidae { void Miaow() { ... } }

class Main {
    static public void main(String argv[]) {
        Cat moe[] = new Cat[5];
        Felidae cute[] = moe;

        /* Insert code here */
    }
}
```

   Add code to the `main` method so that the resulting program is a valid Java program (i.e., it type checks statically and so it will compile), but the program could result in an error being applied to an inappropriate type when executed. Include a brief explanation of how your program exhibits the problem.

```
Felidae single_cat = new Felidae();
cute[0] = single_cat;
moe[0].Miaow();
```

   When type checks are static, this program should be valid since static types are matched in all expressions.
   The problem here is that both Cat and $Felidae$ actually represent the head address of these two arrays, but not just the content of both arrays.
   And it is rudimentary that $Cat \leq Felidae$ according to inheritance, but there is no way to assert $Cat[] \leq Felidae[]$ according to that. In fact, it is not true since we can's replace $cute[0] = single\_cat;$ with $moe[0] = single\_cat;$, where the type of right-hand side of the assignment should conform to the dynamic type of the array. The difference in leff-hand side of the assignment shows that there is no such relationship between $Cat[]$ and $Felidae[]$, thus leading to this problem.

8. (10 pts) **Java Array.** Now that you know why Java arrays are problematic, you decide to add an array construct to Cool with sound typing rules. An array containing objects of type A is declared as being of type Array(A) and one can create arrays in Cool using the new Array[A][e] construct, where e is an expression of type Int, specifying the size of the array. One can access elements in the array using the construct e1[e2] which yields the e2'th element in array e1, and one can insert elements into the array using the notation e1[e2] <- e3. Finally, as in Java, an assignment from one array a to an array b does not make copies of the elements contained in a, but addresses of elements.

   (a) (2 pts) Give a sound subtype relation for arrays in Cool, i.e., state the conditions under which the subtype relation Array(T)≤ T' is valid.

$$\frac{\text{T'=Array(T")} \quad \text{T"=T}}{\text{Array(T)} \leq \text{T'}}$$

(b) Give sound typing rules that are as permissive as possible for the following constructs:

i. (2 pts) new Array[A][e]

$$\frac{O, M, C \vdash e : Int}{O, M, C \vdash new\ Array[A][e] : Array(A)}$$

ii. (2 pts) e1[e2]

$$\frac{O, M, C \vdash e1 : Array(T) \quad O, M, C \vdash e2 : Int}{O, M, C \vdash\ e1[e2] : T}$$

iii. (4 pts) e1[e2] <- e3. Assume the type of the whole expression is the type of e1.

$$\frac{O, M, C \vdash e1 : Array(T) \quad O, M, C \vdash e2 : Int \quad O, M, C \vdash e3 : T' \quad T' \leq T}{O, M, C \vdash\ e1[e2] \leftarrow e3 : Array(T)}$$

9. (10 pts) **Multi-dimensional Array.** In lecture, we talked about array descriptors, which are data structures containing all the information one needs to access (get the address of) an array element $A[i, j]$ in an implementation that allocates all elements of a new array contiguously. In C, multidimensional arrays are composed of rows of rows, so that $A[i, j]$ (or $A[i][j]$ in C) is located at address $(A_{0,0}) + N \cdot S \cdot i + S \cdot j$, where the array in A is $M \times N$ and each element has size S. Thus, the three constants data address $(A_{0,0})$ (the virtual origin), $N \cdot S$ (the row stride), and S (the column stride) can be precomputed into an array descriptor, which the program can use to generate array accesses and can pass as a parameter to functions that expect to receive the array as a by-reference parameter. Show the RISCV code that you'd use to access array element A[i][j], assuming that the d, t_i, and t_j are registers containing the address of the array descriptor for A, the value of i, and the value of j, respectively.

```
lw a2 0(d)      # a2 is the start address
lw a3 4(d)      # a3 is N*S
lw a4 8(d)      # a4 is S
mul a5 a3 ti
add a6 a2 a5
mul a5 a4 tj
add a6 a6 a5    # a6 is the address of the target
lw a0 0(a6)     # a0 holds the target element
```

10. (3x2 pts) **Multi-dimensional Array.** By constructing appropriate array descriptors, one can give different views of an array. Describe how to compute the constructors to create the following views (we don't need the actual code, just the calculations it must do).

(a) Suppose that a certain array descriptor contains the information (VO, S_1, S_2) for accessing two-dimensional array B. Show how to create a new array descriptor that accesses column number j of B. This will be a one-dimensional array descriptor (having only one stride).
We can access $B[i][j]$ by $VO + S_1 \times i + S_2 \times j$.
Now we would like to access elements in column j, whose start address should be $VO + S_2 \times j$, and the offset between elements should be S_1.
So the descriptor should be $(VO + S_2 \times j, S_1)$.

(b) Show how to create a new array descriptor that accesses the transpose of B.
The start address should keep still, while the offset between rows and the between column should exchange.
So the descriptor should be (VO, S_2, S_1).

(c) Show how to create a new array descriptor (for array view B') that accesses the rows and columns of B in reverse, so that $B'[0,0]$ is the same as the last column of the last row of B.
Let's say the size of the array is $m \times n$.
So the starter element should be $B[m-1][n-1]$, thus making the starter address $VO + S_1 \times (m-1) + S_2 \times (n-1)$.
The offset between rows and columns should be respectively $-S_1$ and $-S_2$.
So the descriptor should be $(VO + S_1 \times (m-1) + S_2 \times (n-1), -S_1, -S_2)$.