Solver: Goh Jia Wei

Email Address: GOHJ0064@e.ntu.edu.sg

1. (a) (N|S) ([0-9] | [1-8][0-9]) [<*space*>]$^{+}$ ([0-9]|[1-5][0-9])'.'[0-9]^{3}

   (N|S) – Compass direction
   ([0-9]|[1-8][0-9]) – Minute
   [<*space*>]$^{+}$ - "one or more space characters"
   ([0-9]|[1-5][0-9])'.'[0-9]^{3} - Degree

   (b) Deterministic FA. Reason being:
   1. It does not allow 'λ' to label a transition
   2. It does not allow the same character to label transitions from one state to several different states

Input/State	a	b	c	Others
1 (Start)	2	-	-	-
2 (End)	-	3	4	-
3 (End)	-	3	4	-
4 (End)	-	3	4	-

   Input: abcbc – State 1 > 2 > 3 > 4 > 3 > 4
   Ends at accept state, hence Accept.

   Input: abbba – State 1 > 2 > 3 > 3 > 3 > unhandled input 'a'
   Hence Reject.

2. *// Key here is the priority of operators (found in Lecture Notes), similar to the math operators in next question where multiplication rule is prioritized over addition/subtraction rules.*

   Alt-Expr -> Alt-Expr **|** Concat-Expr
   | Concat-Expr
   Concat-Expr -> Concat-Expr Kleene-Expr
   | Kleene-Expr

Kleene-Expr -> Expr*****
| Expr
Expr -> **(**Alt-Expr**)**
| SYMBOL
| λ

(b)

(i) *// Change to right recursion & remove common prefixes*

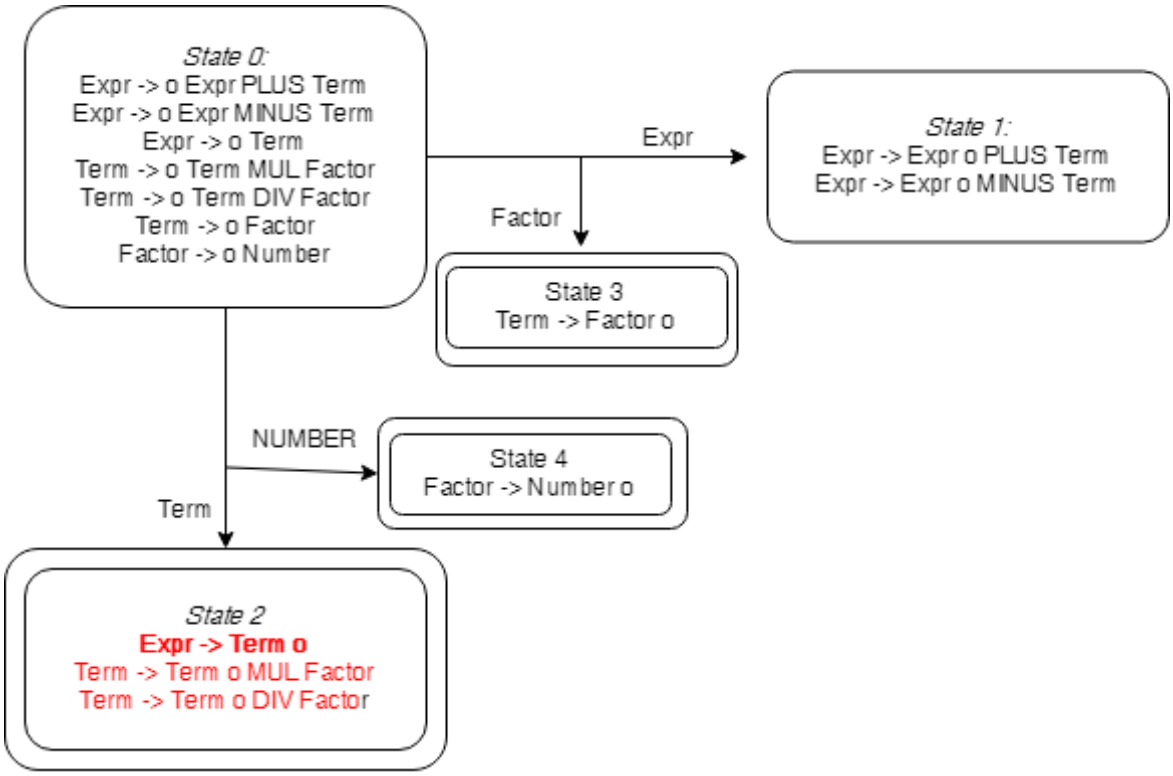
p1: Expr -> Term ExprTail
p2: ExprTail -> PLUS Term ExprTail
p3: | MINUS Term ExprTail
p4: | λ
p5: Term -> Factor TermTail
p6: TermTail -> MUL Factor TermTail
p7: | DIV Factor TermTail
p8: | λ
p9: Factor -> NUMBER

Compute predict() function for each rule and decide if it is an LL(1) grammar:

New Rules:	SymbolDerives Empty[N]	first()	RuleDerives Empty[p_i]	Follow	predict()
p1	F	NUMBER	F	NA	NUMBER
p2	T	PLUS, NUMBER	F	NA	PLUS, NUMBER
p3	T	MINUS, NUMBER	F	NA	MINUS, NUMBER
p4	T	λ	T	λ	λ
p5	F	NUMBER	F	NA	NUMBER
p6	T	MUL, NUMBER	F	NA	MUL, NUMBER
p7	T	DIV, NUMBER	F	NA	DIV, NUMBER
p8	T	λ	T	λ	λ
p9	F	NUMBER	F	NA	NUMBER

Yes, LL(1) Grammar as recursion and common prefix(es) have been removed.

(ii)



1 Shift-Reduce conflict in State 2, where shifting by 'Term' results in a 'reduce' action for p3, whereas the other rules p4 and p5 are 'shift' actions.

3. (a) Difference between Parse Trees and Abstract Syntax Trees (AST):

	Parse Tree	AST
Role	A parse tree represents a derivation – a conceptual tool for analysing the parsing process.	An AST is a real data structure and stores information for semantic analysis and code generation
Structure	A parse tree contains a node for every token in the program.	Semantically useless symbols and punctuation tokens are omitted in AST.

(b)

(i) Call = ID.id LPAREN OptArguments.args RPAREN
{ return new Call(id, args); };
OptArguments = Exprs.exprs

```
                    { return exprs; }
                   | λ
                    { return new List<Expr>();};
Exprs = Exprs.exprs COMMA Expr.e
        { return exprs.add(e); }
       | Expr.e
        { return new List<Expr>().add(e); };
```

(ii) *// full code for a generic call function can be found in 'namecheck.jrag' and 'typecheck.jrag' in Lab 3 files*.

public void Call.typecheck () {

i. Get Callee and call its *typecheck()* method
   FunctionDeclaration callee = getCallTarget();
ii. If there are arguments (optional):
   1. For every argument, get the argument and call its *typecheck()* method.
   2. For every argument, get the *type* of the argument, and check it against the *type* of the corresponding optional parameter of the same index. If not equal, throw error.

```
public void Call.namecheck() {
    getCallee().namecheck();
    for(Expr argument : getArguments())
        argument.namecheck();
}
```

Argument.namecheck() and *Argument.typecheck()* will call the corresponding *namecheck()* and *typecheck()* methods of the class argument instance belongs to (typically *Expr*), which typically will continue recursively calling *namecheck()* and *typecheck()* on its children.

5 semantic errors:

1. Duplicate function declaration
2. Missing function Declaration
3. Inconsistent types in operands of arguments
4. Duplicate Variable declaration
5. Missing Variable declaration

4. (a)
   (i) T (AST is a representation of source code, fed to interpreters/code generators for further processing)
   (ii) F (JVM bytecode will require many more instructions e.g. load instructions)
   (iii) F

   (b) L0: i = x + 1;
   if y <= i: goto L1;
   j = c * d;
   k = a + b;
   j = k * j;
   y = y – j;
   if y > 10: goto L0;
   y = y + 1;
   goto L0;
   L1: *(terminate);*

   (c)

Instruction	Stack height	Instruction	Stack height
L0: Load 2	1	Store y*	0
Load z	2	Load y*	1
Add	1	Load z	2
Load x	2	Ifgt L2	0
Ifgt L1	0	Load 1	1
Load 2	1	Load x*	2
Load 1	2	Add	1
Ifgt L2	0	Store x*	0
L1: Load 1	1	Load 1	1
Load y	2	Load 0	2
Add	1	Ifgt L0	0
Load y	2		
Mul	1	L2:	0

Unable to remove any store-load pairs.
*// Unable to remove instructions marked by *, as new value of 'y' or 'x' is required in subsequent rounds of execution - removing this pair of instructions before it will cause undesirable side-effects!*

Java code:
[L0] while (z+2 > x):
[L1] Y = Y * (Y +1);

```
If y > z:
        break;
else:
        x = x + 1;
```

[L2] *(terminate)*

(d)

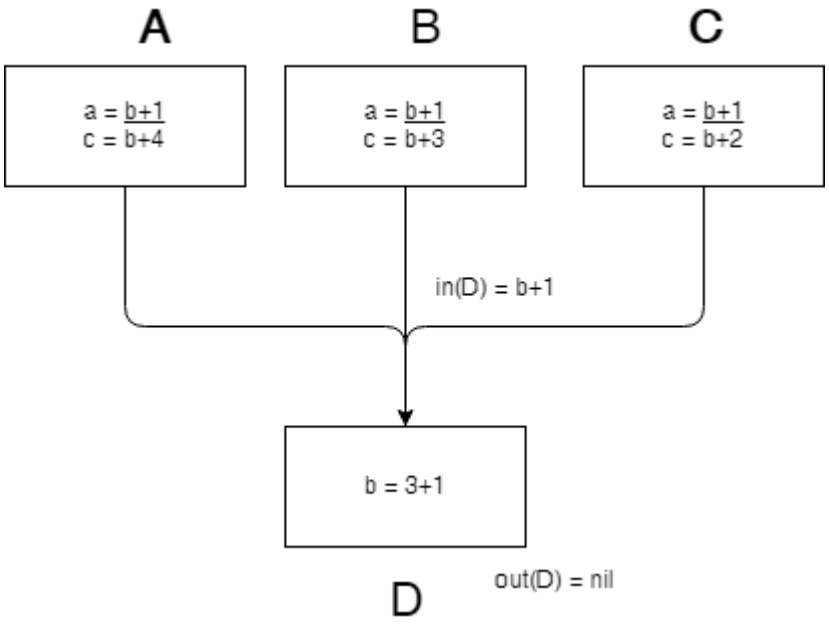
Situation where G is k-colourable: If we have 'k' colours, and if there is some node 'n' in the graph G with less than 'k' neighbours, then the whole graph is k-colourable if the graph G-1 (G without 'n') is k colourable so we can simplify the problem by disregarding 'n'. We can solve this recursively.

Situation where G is not k-colourable: All nodes have at least 'k' number of neighbours. Then, we should choose a spill candidate (generally the node 'n' with largest number of neighbours), remove it to simplify the problem, and solve the simplified problem G-1 (without node 'n'), depending on G-1 is k-colourable or not.

5. (a)

(i) F (variable could be defined or overwritten in the node)
(ii) T (if then, else)
(iii) F (AE is a forward-must analysis, available *after* computation within the node)
(iv) F (again, AE is a forward-must analysis)

(b)



For Available Expression analysis (forward-must):

$$out_A(n) = in_A(n) \setminus \{e \mid vars(e) \cap def(n) \neq \varnothing\} \cup comp\ (n)$$

- Clearly, an expression e is available after a node *n* if
  1. Node *n* computes e, or
  2. e is available before *n* and *n* does not write to any variable in e

As 'b+1' is common in all outputs of nodes A,B,C, but 'b' is overwritten in node D, there are no available expression after D.

(c) *Chapter 6 – Optimization*

Liveness: a local variable is live at a program point if its value may be read before it is re-assigned. In terms of CFG: a variable x is live after a CFG node n if there is a path p from n to EXIT such that there is a node r on p that reads x, and r is not preceded on p by any node that writes x.

This can be broken down into:

- Note that if we already know $out_L(n)$ for some node, $in_L(n)$ is easy to compute: note that a variable *x* is live before *n* if
  1. either *n* reads x,
  2. or x is live after *n* and *n* does not write *x*

Hence,

$$in_L(n) = out_L(n) \setminus def(n) \cup use(n)$$

And,

- There are two cases:
  1. Node *n* is the EXIT node
     $out_L(n) = \varnothing$: no variable is live at the end of a method
  2. Node *n* has at least one successor node:
     Let succ(*n*) be the set of successor nodes of *n*
     Note that a variable x is live after *n* if it is live before any successor of n
     Thus we define $out_L(n) = \cup \{ in_L(m) \mid m \in succ(n) \}$

(d) Common subexpression elimination increases the live range of local variables (i.e. the number of nodes where they are live). This makes register allocation harder, so the performance gain of avoiding recomputation may be lost because more memory accesses are needed.

For example, given the code:

*a = b * c + g;*

*d = b * c * e;*

We might replace the code with the following and perform Common Subexpression Elimination using a temporary variable *tmp* stored in the register.
*tmp = b * c;*
a = tmp + g;
*d = tmp * e;*

This is only if the cost of accessing *tmp* is faster than recomputing b*c, which may not be the case if register spilling is involved.

6. (a)
   (i) F (Analysis Phase produces AST; Synthesis Phase produces executable code)
   (ii) F (traditionally, compiled native code runs faster than interpreting code line by line.)

   (b)

	Input	Output
Lexer	Character stream	Stream of Token types and Lexemes
Parser	Stream of Token types and Lexemes	Parse Tree
Semantic Analyzer	Parse Tree	Abstract Syntax Tree
Code Generator	Abstract Syntax Tree	Executable Code
Optimizer	Executable Code	Optimized Code