Drew Pulliam - DTP180003
CS4337.002

① A. Abstraction

② D. With Shallow Access, it is to place local references in parent's ARI's

③ B. Widening Conversion

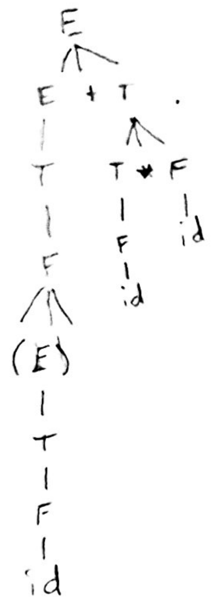
④ E. The Storage of the variable is allocated during compile time

⑤ D. Heap Dynamic

⑥

Drew Pullian - DTP180003
CS4337.002

(7) Parse Tree. id + (id • id)



stack	symbol	input	output
0	(	(id)+(id*id)	s4
04	(	id)+(id•id)	s5
045	(id	)+(id•id)	R6 (F→id)
043	(F	)+(id•id)	Goto(4,5)→3
04	(T	)+id•id	
04	(E	)+id •id	Goto(4,T)→2
0411	(E)	+ +d • id	R2(E→T)
03	F	+id•id	[11,*]→R5
02	T	+id •id	Goto(0,F)=3
01	E	+id - id	Goto(0,T)=2
016	E+	id • id	Goto(2,+)=R2
0165	E+id	•id	Goto(1,+)=S6
016	E+F	•id	(6,id)=S5
0169	E+T	•id	(6,T)=9
01697	E+T•	id	(9,•)=S7
016975	E+T•id	~~$~~	(7,id)=S5
01697	E+T•F	$	(5,$)=R6
0169	E+T	$	T=T•F

01 E $ (0,E)=1

(1,$) ✓

Drew Pulliam - DTP180003

CS4337.002

(2)

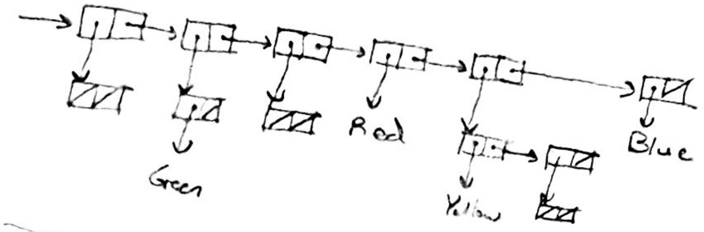
```
path(seattle, omaha).   ; path from seattle to omaha exists
path(seattle, dallas).
:
; declare all paths first (didn't show here for space)

; rules
flight(X, Y) :-
        path(X, Y).
flight(X, Y) :-
        path(X, Z),
        flight(Y, Z).
```

(9) Yes this grammar is ambiguous because it doesn't specify difference between if and if/ which leads to non-unique parse trees

```
<if_stmt> → if <logic_expr> then <stmt>
          | if <logic_expr> then <stmt> else <if_stmt>
<if_else_stmt> → if <logic_expr> then <stmt> else <stmt>
```

Drew Pullian - DTP180003
CS4337.002

(10) cons '(()(Green)()) '(Red (Yellow ()) Blue))
= (()(Green)() Red (Yellow()) Blue)



(11)

```
(define (common-elements list1 list2)
    (cond ((null? list1) '())            <— check if list1 is null, if so, return empty list
        ((member (car list1) list2)      <— check if first element of list1 is member of list2
        (cons (car list1)
              (common-elements (cdr list1) list2)))   <— if yes, return that element + recursive function call on rest of list
        (else (common-elements (cdr list1) list2)))
)
                                         & if no, just return recursive function call on rest of list
```

(common-elements '(() (b) d c) '(a d () b))

check if list 1 is null → not null

check if first item "()" of list1 is member list2 → if is, return "()" + recursive function call

check if next item "(b)" is member list2 → it is not, return recursive function call

check if next "d" is member → it is, return "d" + recursive function

check if next "c" is member → its not, return recursive function call

list 1 is now null, return empty list

exit recursion, final return is "(() d)"

Drew Pulliam - DTP180003

CS 4337.002

(12) the fist 3 lines are simply definitions of girlage, associating letters "a, b, c" with ages (numbers)

- the rule called "lists" is used to create a list of all ages, ignoring names (the letters)
- the rule called "mysum" takes an array of numbers and returns the sum using recursion
- because of the recursion, the base case "mysum([],0)." is needed to end the recursion
- we then query the code to find the list of all girlages, and then find the sum of that list using "mysum".

(13) (a) main → fun1 → fun2

inside fun2 visible variables

c, d, e → defined fun2
b → defined fun1
a → defined main

(b) main → fun3 → fun3* → fun2

visible variables inside fun2

c, d, e → defined fun2
f → defined fun3* (the second time fun3 was called)
a, b → defined main

(c) main → fun1 → fun3 → fun1*

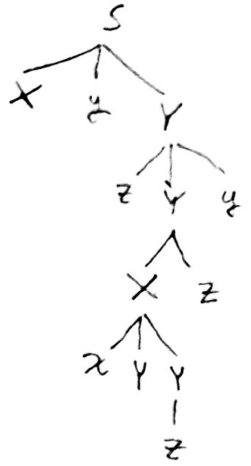
visible variables inside fun1* (second time it was called)

b, c, d → def fun1*
e, f → def fun3
a → def main

Drew Pulliam · DTP180003

CS 4337.0U2

(14) Parse tree



S → X y Y

→ Y y z Y y

→ X y z X z y

→ Y y z x Y Y z y

→ X y z x Y z z y

Y → z Y y

Y → X z

X → x Y Y

Y → z

(15) instead of going in numerical order I'm going by depth

point 5 in main() : p

point 1 in fun1() : t, s, r=p ← passed from main()

point 2 in fun1(): t, s, r=p

point 3 in fun2() : y, x=s ← passed from fun1()

point 4 in fun2() : y, x=s

fun3() also contains q=y ← passed from fun2()