CSC 461 Exam I Fall 2024

Record your answers using the QUIZ FOR Exam 1.

You will just being given the Letter a,b.c,.. to choose from for each (not the question from exam)

Q1 (4 pts)

I claim that the following grammar is **ambiguous**.

<assign> → <id> = <term>
<expr> → id | <term> - <expr> | (<expr>) | <term> + <expr>
<term> → <expr>
<id> → X|Y

Is **the grammar above** Ambiguous?

a) YES
b) NO

Q2 (4 pts)

A rightmost, leftmost, or neither right nor left most approach can be used in the derivation of a sentence in an unambiguous grammar. If each of these is used on the same sentence of such a language, how many possible different parse trees could be involved?

a) 1
b) 2
c) 3
d) More than 3 are possible

Q [3-7] (10 pts 2-each)

Given the following Extended BNF , which strings are a **valid** or i**nvalid** "LINE"?

- * is zero or more
- + is one or more

<run> → {0|1}*

<squeeze> → $\{010\}^{+} \mid \{101\}^{+}$

<**LINE**> → <squeeze> <run> <squeeze> (*NOTE: This is the **LINE** to check*)

Valid/Invalid "LINE" (V or I)

Q3 10111010

Q4 101101

Q5 010110101101

Q6 000111101

Q7 101111100000

Assume you are using a new language called '**L'** that is imperative and the only storage type for variables is "static" and is implied. Names are statically bound to variables. Initialization of variables are done only when created. You are asked to supply a print list function. YOU go find both a recursive and also a non-recursive version of algorithms that print lists but in other languages on the internet. You can use the logic to try and write them in **L** and are now trying to decide which one might be the best to use. L uses Static scoping and there is a global variable called LIST that holds the list. LIST is an array of integers and the location after the last number in the list is a -1 and can be used to know you are out of numbers.

NOTE: the answers to both question Q8 and Q9 should be in the context of the definition of the language L given above and our study of languages in this class.

Q8 (4 pts)

Of the two functions below, which function can you dismiss **without even doing analysis (so by just simply looking at it without having to work through the logic in the code istself)** of the two because you are using language L? **Printlist()** or **recursive_printlist()** ? Why!

a) Dismiss Recursive_printlist( ) because L can't do recursion.
b) Dismiss Printlist( ) because Static scoping can't handle a Global List variable declaration
c) Dismiss recursive_printlist( ) because it so much harder to understand recursive logic
d) Not given enough initial information about L to dismiss one without looking at how each works in more detail.

Q9 (4 pts) You decide to implement the other algorithm (**the one you did not dismiss in Q8**) and when you call it the first time it works fine and prints the numbers in LIST. However, when you call it more than **once in the same program**, you get no output, or strange output, or the program crashes. Why?

a) The while loop logic does not work even though LIST does have a -1 at end of numbers
b) t is only initialized to 0 one time
c) there is not a good ground condition to end the function
d) There is no way to tell from the def of L given why the code has issues during execution.

***** Function code for Q8 and Q9 on next page**

If you have any questions about syntax … just ask. 😊
Assume the rest of the program will load LIST before calling.

```
//========= Non recursive =====================
void printlist( )
{
 int t = 0;
 while (LIST[t] != -1)            // while not equal to the -1 marker
 {
   cout << LIST[t] << endl;    // output value
   t = t+1;
 }
}
//================  Recursive  ===============
void recursive_printlist(int t )   // assumes first call will pass in value 0 into t
{
    if (LIST[t] == -1)               // exit if -1 marker found
          return;
   cout << LIST[t] << endl;      // output value
   recursive_printlist(t+1);    // recursively call to move through the list
}
```

==========

Q 10 (3 pts) Absolute Addressing can cause issues when if used while programming because

a) It is inefficient when executed.
b) Inserting new code above the targeted address can change the location.
c) Inserting code below the targeted address can change the location.
d) It loses precision when truncated.

==========

Q 11 (3 pts) LABEL statements were implemented to break from the Absolute addressing approach to

a) To make the source code more readable.
b) To make the source code more writable.
c) To make the source code more reliable when executed.
d) All of the other answers.

==========

Q 12 (3 pts) An Unambiguous grammar will have

a) Meaningful variable names
b) Will be well commented to help understand the code
c) Only one distinct parse tree
d) Will have a well balanced parse tree

Q 13 (3 pts) The declaration of a local variable with the same name as a non-local variable can **ONLY** hide the non-local variable in the local scope when

a) Static scoping is used
b) Dynamic scoping is used
c) If the non-local would be in the referencing environment if not for the local with the same name
d) All of the others.

Q 14 (3 pts) A globally declared static variable will always be visible everywhere in a program.

a) Only if static scoping is used
b) Only if dynamic scoping is used
c) Only if not hidden by a local variable of the same name
d) Off and on as it is bound and unbound from a memory cell during execution.

Q 15 (3 pts) The usefulness of a Language Generator is

a) To generate useful syntactically correct code for a given language (like a button to be pushed)
b) To determine if a statement is syntactically correct for a given language
c) Provide a set of rules for syntactically correct statements to help users understand the language
d) To generate an unambiguous grammar for a language

Q 16 (3 pts) If the programmer wants to use an array, but 1) have the system manage it and 2) have it reuse memory when it can, they should choose what type of storage binding?

a) A fixed stack-dynamic Array
b) A static Array
c) Explicit heap-dynamic Array
d) Static scoping

==========

Something for each variable storage binding

Q 17 (3 pts) For which variable storage binding is the variable's **LIFETIME** the execution of the program for start to finish?

a) Static
b) Stack-Dynamic
c) Implicit Heap-Dynamic
d) Explicit Heap-Dynamic

Q 18 (3 pts) For which variable storage binding is the variable's **LIFETIME** from Function Invocation to the end of the function?

a) Static
b) Stack-Dynamic
c) Implicit Heap-Dynamic
d) Explicit Heap-Dynamic

Q 19 (3 pts) For which variable storage binding is the variable's **LIFETIME** dependent on the programmer for allocation and deallocation?

a) Static
b) Stack-Dynamic
c) Implicit Heap-Dynamic
d) Explicit Heap-Dynamic

Q 20 (3 pts) For which variable storage binding is the variable's **LIFETIME** a condition for recursion?

a) Static
b) Stack-Dynamic
c) Implicit Heap-Dynamic
d) Explicit Heap-Dynamic

Q 21 (6 pts.)

Our pseudo-code language we developed at the start of the semester has an increment and test statement (+7). Given the **limited number of statements our design allowed (only 20),**

Why was such a statement included since what it does could be accomplish without its inclusion?

a) We had some left-over open statements so why not.
b) It helped when implementing loop logic which is done a lot
c) It was the inverse logic of the -7 statement
d) It got away from absolute Addressing

Q 22 (6 pts.)

Our pseudo-code language we developed at the start of the semester has an increment and test statement (+7). Given the **limited number of statements our design allowed (only 20),**

What design principle was the decision to include the increment and test a very good example of?

a) Impossible Error
b) Regularity
c) Reliability
d) Abstraction

Q 23 (3 pts) When you begin to learn a new language it often seem strange and although eventually it may help you be more productive, at first it actually slows your coding down. How would our understanding of Tools in general explain this?

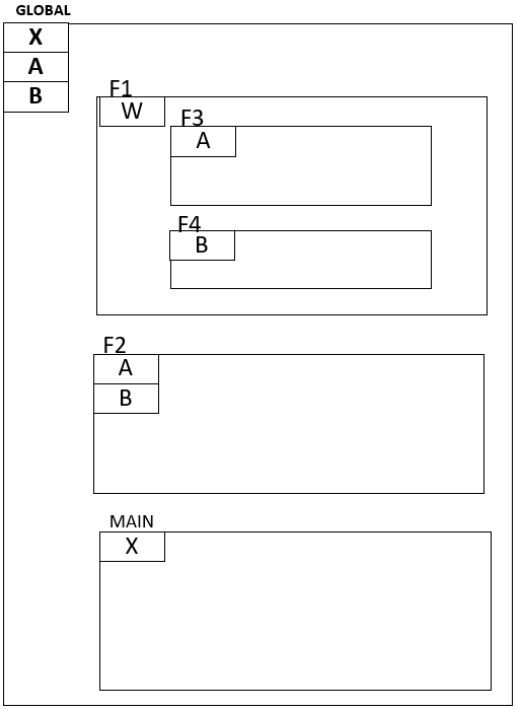
a) We have yet to reach the Objectification phase
b) We have yet to reach the Embodiment phase
c) We are more of a Dystopian than a Utopian
d) We are lazy

Q 24 (3 pts) When learning programming one is often taught to use meaningful Variable Names. Why.

a) To help with the readability of the source code
b) To help the compiler parse the syntax
c) Because the language usually enforces that you do
d) All of the others

Q 25 (3 pts) Early programming languages often only implemented Static variable storge binding which

a) Limited the number of characters a variable name could have
b) Did not allow for recursion
c) Made things run less efficiently
d) All of the others



(Fig 1) Scope Diagram for Program P.

Assume program P always starts in MAIN !

Q 26 (4 pts) In Fig 1, using **STATIC** Scoping, what is the referencing environment if in F4?

a) {Global :(X and A and B) , F1:W, F4:B}
b) {MAIN: X, F1:W, F3:A, F4:B}
c) {Global :A , MAIN: X, F1:W, F4:B}
d) {Global :(X and B) , F1:W, F4:A}
e) Can't tell by looking at fig 1.

Q 27 (4 pts) in Fig 1, assuming the program starts executing in MAIN and then calls F2, which then calls F1 which then calls F3 that then calls F4. Using **STATIC** Scoping, what is the referencing environment if in F4?

a) {Global :(X and A and B) , F1:W, F4:B}
b) {MAIN: X, F1:W, F3:A, F4:B}
c) {Global :A , MAIN: X, F1:W, F4:B}
d) {Global :(X and B) , F1:W, F4:A}
e) Can't tell by looking at fig 1.

Q 28 (4 pts) in Fig 1, assuming the program starts executing in MAIN and then calls F2, which then calls F1 which then calls F3 that then calls F4. Using **DYNAMIC** Scoping, what is the referencing environment if in F4?

a) {Global :(X and A and B) , F1:W, F4:B}
b) {MAIN: X, F1:W, F3:A, F4:B}
c) {Global :A , MAIN: X, F1:W, F4:B}
d) {Global :(X and B) , F1:W, F4:A}
e) Can't tell by looking at fig 1.

Q 29 (4 pts) in Fig 1, assuming the program starts executing in MAIN and then calls F2, which then calls F1 which then calls F3 that then calls F4. Using **STATIC** Scoping, what variable are **"ALIVE"** if in F4?

f) {Global :(X and A and B) , F1:W, F4:B}
g) {Global :(X and A and B) , MAIN:X, F1:W, F3: A, F4:B}
h) {Global :A , MAIN: X, F1:W, F4:B}
i) {Global :(X and B) , F1:W, F4:A}
j) Can't tell by looking at fig 1.

Q 30 (4 pts) in Fig 1, assuming the program starts executing in MAIN and then calls F2, which then calls F1 which then calls F3 that then calls F4. Using **DYNAMIC** Scoping, what variable are **"ALIVE"** if in F4?

a) {Global :(X and A and B) , F1:W, F4:B}
b) {Global :(X and A and B) , MAIN:X, F1:W, F3: A, F4:B}
c) {Global :A , MAIN: X, F1:W, F4:B}
d) {Global :(X and B) , F1:W, F4:A}
e) Can't tell by looking at fig 1.