Sonoma State University Name: Alana Reyna

Computer Science Department

CS 460 – Fall 2022 – Watts Score: / 100

Final Exam

(Date due: Friday, 16 December 2022, 11:59 pm – *submissionidF.docx)*

| 1. | a. The grammar we used for Project 2 is an LL(1) grammar – what does that mean?  b. How does that affect the design and implementation of your Syntactical Analyzer?  c. Why are first and follow sets necessary when creating a Syntactical Analyzer for an LL(1) grammar?  d. What would your Syntactical Analyzer need to do if the grammar were an LL(2) grammar? |
| --- | --- |
| Answer | 1. LL(1) grammar means that the grammar is context-free. This is a concept we have learned about both in CS460 programming languages and CS454 Theory of Computation. The first L represents that the input will be scanned from left to right and the second L indicates that we will use a leftmost derivation tree parsing technique. The (1) indicates the number of symbols we will be looking ahead to make a parsing decision. A LL(1) grammars are also not ambiguous and not left recursive. 2. This affects the design and implementation of our Syntactical Analyzer because this means that there are a couple things we need to construct our Syntactical Analyzer to meet the requirements of a LL(1) language. First, our Syntactical Analyzer needs to read tokens from left to right as a LL(1) grammar does. Next, we need to create a first and follows set for all the non-terminals of the grammar. This is necessary for the parser to correctly apply the rules necessary in the correct position. In project 2, the compiler needs to be able to identify what token’s should come next when we are at a particular rule. For example, if we are at a non-terminal and the first thing we see is another non-terminal, we need to trace through that non-terminal until we reach a terminal. This is the first terminal we see, which would be the firsts of the non-terminal we started at. This tells the compiler that this is the first terminal to expect and if we get anything else, there is clearly an error. This same concept is applied to create the follows set. 3. The firsts and follows sets for all non-terminals are necessary for the parser to correctly apply the rules necessary in the correct position. In project 2, the compiler needs to be able to identify what token’s should come next when we are at a particular rule. For example, if we are at a non-terminal and the first thing we see is another non-terminal, we need to trace through that non-terminal until we reach a terminal. This is the first terminal we see, which would be the firsts of the non-terminal we started at. This tells the compiler that this is the first terminal to expect and if we get anything else, there is clearly an error. This same concept is applied to create the follows set. 4. If our Syntactical Analyzer were a LL(2) grammar, this means that we need to look ahead by 2 symbols instead of one. If we were to adjust our Syntactical Analyzer for a LL(2) grammar, we would need to adjust our firsts and follows set to accept two terminal symbols instead of one. |
| Score  (of 20) |  |

| 2. | a. What is an Array Mapping Function?  b. What are the benefits of an Array Mapping Function?  The following C++ program uses a struct and a 2-dimensional array to store data.  #include <iostream>  using namespace std;  struct Struct  {  int a;  double b;  char \* c;  };  int main ()  {  Struct Array [5][3];  cout << \*Array[3][1].c << endl;  return 0;  }  c. If the base address of Array is 0x7fffdbe0, what is the address of Array[3][1].c? Explain your calculations. |
| --- | --- |
| Answer | 1. According to our book, specific elements of an array are referenced by means of a two-level syntactic mechanism. The first part is the aggregate name and the second part is possibly a dynamic selector consisting of one or more items known as subscripts or indices. Arrays are sometimes referred to as finite mappings. Array element references and function calls are both mappings. Therefore, array mapping function calls map the parameters to the function definition and eventually function values. An array element references map the subscripts/indices to a particular element of the array. 2. The benefits of an array mapping function is that the subscript value ranges are dynamically bound sometimes. We can access elements of the array through subscripting. Array mapping functions also create a more space efficient array. 3. Array[3][1] = Base address + 1 byte \* ((3 - 0) \* 3 + (1 - 0) =   0x7fffdbe + 1 \*(3\*3+1) = 0x7fffdbe + 1 \* 10 =  2147474400 + 1 \* 10 = 2147474410 |
| Score  (of 20) |  |

| 3. | a. What is structured programming? What are its benefits?  b. What are the 3 standard categories of control structures? Give a C++, Python, Java, or Scheme example of a structure for each of the 3 categories.  c. How do “break”, “continue”, “return”, and “exit” change the control flow of structures in each of these categories? |
| --- | --- |
| Answer | 1. Structured programming is a programming paradigm. It allows for the creation of programs with readable code and reusable components. The goal of structured programming is to make programs easier to read and understand. The execution of code follows the order that the code is written. Structured programming avoids using goto statements and reduces spaghetti code. 2. A control structure is a control statement and collection of statements whose execution it controls. The three standard categories are:    1. Sequential: sequential execution of code (one line after another)   Ex: string name = “Dr. Watts”;  string grade = “pass”;  cout << name << “, please let me “ << grade << endl;   * 1. Selection: Decision blocks   Ex: int n = 3;  if (n == 1)  {  cout << "1st place! " << endl;  }  else if (n == 2)  {  cout << "2nd place! " << endl;  }  else if (n == 3)  {  cout << "3rd place! " << endl;  }  else  {  cout << "Didn't even podium... :( " << endl;  }   * 1. Repetition: Looping, repeated piece of code   Ex:  for (int i = 0; i < 10; i++)  {  cout << "loop number: " << i << endl;  }   1. break:    1. sequential : Break does nothing for sequential control structure.    2. selection: Break allows control to flow from one selectable code segment to another. It allows the program to be disrupted after it hits some selection condition. Keep in mind, switch statements are considered selection control structures. When a case is hit and we don’t want to continue through the rest of the cases, we use a break, which breaks us out of the selection control block.    3. Repetition: Break works similarly to selection with repetition. It allows us to break out of a loop when some condition is met.   Continue:  i. sequential : Continue does nothing for sequential control structure on its own. When we have a sequential control block within a selection or repetition control block, it does serve the purpose of skipping over a particular code block when some condition is met.  ii. selection: Continue allows us to skip over a code block when a particular selection statement is reached. If we have a switch case for example, if we reach a particular case with a continue in it, it allows us to skip over the rest of the cases and continue to the next iteration (if there is repetition)  iii. repetition: This is where continue is most useful. When we are in a loop, continue allows us to break the current iteration when some condition is met and skip to the next iteration. For example, if we had a for loop that looped 10 times and we are on the 5th iteration, if we reached a continue statement within an if statement, continue would allow us to skip the rest of the selection statements and continue to the 6th iteration.  return:  i. sequential: Return statements allow us to exit the function (or program if we are in main) once the return statement is reached. If we had some block of sequential code and there was a return somewhere in the middle, it would not run the rest of the code that followed after the return statement. Returns can hold values (in the case of functions where we want the function to return something to the main code that is calling the function) or be empty.  ii. selection: Return statements in a selection will function pretty similarly to the above description. It will exit the function and return some value or could be used to just exit the function or program (when in main). If a return statement is hit in a selection statement, it will not reach any other selection blocks that follow after, it will immediately exit.  iii. repetition: Return statements behave similarly to selection blocks. If we reach a return statement in a repetition block, the program or function will immediately terminate, breaking out of the loop and/or any selection statement that it was in.  exit:  i. sequential: exit statements in a sequential code structure will exit the program or function that it’s in. Exit’s never return a value and just terminate the program.  ii. selection: exit statements work the same all around. Exit in a selection will exit out of the program immediately when it’s hit, breaking the flow of the selection code block. If an exit is hit before the rest of the selection statements are reached, the selection statements following will never be reached.  iii. repetition: exit statements work similarly to selection in repetition. If an exit is hit within a repetition (for loop or while for example), it will break out of the iteration and immediately exit the program. It will not allow the iteration to continue past that exit statement. |
| Score  (of 10) |  |
|  |  |

| 4. | Using the rational class implementation provided for Project 3, the output of the following C++ program is  1. (a,b,c) = (7, -14, 17)  2. (p,q,r) = (3/2, -2, -2)  3. (f,g,h) = (-3, -15, -11)  4. (f,g,h) = (-3, 46, 91)  When the input values are: 5 -14  And the output is  1. (a,b,c) = (-12, 5, -39)  Denominator cannot be 0; exiting program.  When the input values are: -14 5  Explain each line of output.  #include <iostream>  #include "rational.h"  using namespace std;  int g = 4;  int change (int x)  {  return g = x \* g;  }  int main ()  {  int a, b;  cin >> a >> b;  int c = 2 \* a++ + 7 / b + ++a;  cout << "1. (a,b,c) = (" << a << ", " << b << ", " << c << ")\n";  rational p (a, b);  rational q = b/a;  int r = rational(2) \* p++ + rational(7) / q + ++p;  cout << "2. (p,q,r) = (" << p << ", " << q << ", " << r << ")\n";  int f = -3;  int h = g++ + change (f);  cout << "3. (f,g,h) = (" << f << ", " << g << ", " << h << ")\n";  f = -3;  h = change (f) + ++g;  cout << "4. (f,g,h) = (" << f << ", " << g << ", " << h << ")\n";  return 0;  } |
| --- | --- |
| Answer | 1. (a, b, c) = (-12, 5, -39) // c = 2 \* (-13) + 7 / 5 + (-14)   // calling a++ immediately changes the value while ++a does not change the value  // of a until the after the operation is done   1. Denominator cannot be 0; exiting program.   // once we hit the line where we do rational(7) / q (5/-12), the program will fail  // this is because 5 / -12 will give us a value of 0, so when we try to divide a  // rational value of 7 with a 0, it will error out since this is not allowed and will  // terminate the program  // we will not be able to continue realistically with 3 and 4 since the program ends  // with #2   1. (f, g, h) = () 2. (f, g, h) = () |
| Score  (of 20) |  |

| 5. | Using the subset of Scheme defined for our project, define 3 Scheme functions:  **findMaximum** will find and return the maximum numeric value in a list of numeric values. This function should expect a single list as the argument passed to it and should return the minimum value in the list. For example:  (findMaximum ‘(1 3 5 -2 15 -3))  Would return the value 15.  **removeFromList** will remove a specific value from a list. This function should expect to have 2 arguments passed to it; the value to be removed and the list. It should return a copy f the list with the value removed. For example:  (removeFromList 5 ‘(1 3 5 -2 15 -3))  Would return the list (1 3 -2 15 -3). If there are multiple occurrences of the value, it should just remove the first one.  **selectSort** will use the above functions and the selection sort algorithm to sort a list of numerical values into descending order.  The output of the function main:  (define (main)  (display (findMaximum '(-1 -2 5 -3 -4))) (newline)  (display (removeFromList -4 '(-1 2 5 -3 -4 5))) (newline)  (display (selectSort '(1 -1 5 3 -6.2 2.5 -3 4 -1 7 15)))  (newline)  )  Would be:  5  (-1 2 5 -3 5)  (15 7 5 4 3 2.5 1 -1 -1 -3 -6.2)  In addition to pasting the code for your functions below, please submit the 3 functions (findMaximum, removeFromList, and selectSort) in a file called *yourlastname*F.ss. |
| --- | --- |
| Answer | (define (findMaximum L)  (if (null? L) '()  (if (null? (cdr L)) (car L)  (let ((maxVal (car L)))  (if (> (findMaximum (cdr L)) maxVal)  (findMaximum (cdr L))  maxVal  )  )  )  )  )  (define (removeFromList val L)  (if (null? L) '()  (cond ((equal? val (car L)) (cdr L))  (else (cons (car L) (removeFromList val (cdr L))))  )  )  )  (define (selectSort L)  (if (null? L) '()  (let ((maxVal (findMaximum L)))  (cons maxVal (selectSort (removeFromList maxVal L)))  )  )  )  (define (main)  (display (findMaximum '(1 3 5 -2 15 -3)))  (newline)  (display (findMaximum '(-1 -2 5 -3 -4)))  (newline)  (display (removeFromList -4 '(-1 2 5 -3 -4 5)))  (newline)  (display (selectSort '(1 -1 5 3 -6.2 2.5 -3 4 -1 7 15)))  (newline)  )  (main) |
| Score  (of 20) |  |