Leon Zhang  
llz210000  
CS 4337.501

Homework 4

1. Consider the following attribute computation functions and predicate functions associated with the rules above.
   1. There are two such statements. Specify these statements informally with an English sentence for each. Do not directly reference any of the attributes, predicate functions, or attribute computation functions. For convenience here are some definitions you can use  
      Informal statements:
      1. Every production list must contain a “main” production and it should be the first statement. This is enforced by getting a list of all production ids: <prod\_list>.ids = {<prod.id} and checking if “main” production exists: Predicate: “main” ∈ <prod\_list>ids, within the combined list: <prod\_list>[0].ids = {<prod>.id} U <prod\_list>[1].ids.   
         Attributes involved:   
         Synthesized: <prod\_list>.ids because its value is determined by {<prod>.id}. Additionally, <prod>.id is also synthesized because its value comes from ID.id   
         Intrinsic: ID.id, because its value is known before the sentence is parsed.
      2. In the body of a production, all items must have the same id as the body. This is enforced by setting an item’s id to the value of the body id: <body\_item>.id = <body>.id.  
         Attributes:  
         Synthesized: <body\_item>.id, <body.id>, these are synthesized because their values come from <body.id> and ID.id respectively.  
         Intrinsic: ID.id because of reasons stated above.
   2. For the following programs determine if they satisfy the static semantics.
      1. Yes, satisfies the static semantics.
      2. No, main is not the first production.
2. For the following programs, what will be printed when it program is ran? Briefly explain why.
   1. Output: 1 1. Following the instructions given by the program output, x is assigned value of 1 before being printed followed by a print statement for a space. Then, the instructions go back to an earlier print statement for x which is still 1 and the program halts, making the output 1 1.
   2. Output: 2 3 3 5. Following the instructions given by the program value, x is assigned to value 2 before being printed followed by a space. Then, it is incremented to 3 before being printed again with a space. This is repeated once more followed by a series of jumps to line numbers from the stack. This leads to x being incremented to 5 before being printed and the program halting.
3. Recall the attributed grammar from question 1. Consider the following denotational semantics. For the following programs, what is the mathematical meaning?
   1. Main -> {10}   
      main -> {step, show} = 4 + 92 = 96  
      step -> {2, 2} = {4}  
      show -> {100, 2, -10} = {92}  
      Mathematical meaning: {(main, 96)}  
      Main ID corresponds with value of 10. & corresponds to 2, so step -> &,& corresponds to {2,2}, 2+2 = 4. Similar logic for show, = 92. Because main is adding both, value of 96.
   2. Compute -> {-10, 2, 100, comput2} = 278  
      compute2 -> {-10, 2, 100, compute3} = 186  
      compute3 -> {-10,2,100,2} = 94  
      main -> {2,2,2,100,2,2, compute} = 390  
      Mathematical meaning: {(main, 390)}  
      Computer -> \*,&,%,compute 2 corresponds to {-10,2,100, value of computer2}, the value which is {-10,2,100, compute3}, etc. Calculating them gives values of 278, 186, 94, and 390 respectively for compute 1, 2, 3, and main.
4. Submitted in separate files. All ran on a previous code assignment to translate infix expressions to postfix expressions and evaluate them.
   1. Mutator-Fuzzer – I wrote this fuzzer to generate random valid infix expressions to run into my converter to postfix and evaluate them. My main takeaway from this fuzzer was my error handling in the program was very low quality, especially for faulty inputs. When I did the prefix analyzer for this Paradigms class, the input handling was very strict and defined. However, for this program written for a previous class, I did not have very defined input handling before passing the postfix string to the evaluator. This led to many cases where the mutator replaced numbers with symbols and didn’t validate the expression before trying to evaluate it, making it crash during the evaluation. In a better program, it would validate the infix expression before even running anything else to save time and resources.
   2. Generation-Fuzzer – While I knew my program could not handle improper inputs well, I did not expect it to fail so much with proper infix expressions. In this generator fuzzer, I made the random generated infix expressions have a lot of filler parenthesis and spaces to see how my program would handle them. For example, even though the expression: 1 + (((3 \* 8)) – 7 ) is completely valid, there is no reason to have the parenthesis. I found that my input handling did not handle parenthesis well at all, and expressions with many valid parentheses failed. However, I did not expect very basic parenthesis to fail too. Throughout various tests, I discovered that expressions starting with a parenthesis really struggled to pass through the input and something in my program was wrong. Even expressions such as (1+1) were getting errors. While this means there is something wrong in my code, in previous tests, it was handling cases with parenthesis fine so this was a surprise to me.
   3. Protocol-Fuzzer – difficult to find different application of protocol fuzzing for my program.