1. What is an (abstract) syntax tree and how is it related to other parts in compilation?

AST is a tree structure that consists of nodes that are different BNF symbols. These nodes have some information about the symbol, e.g its value, name, linenumber and most importantly, its relation to other symbols. Like we can have a while loop node that has children that are nodes declaring what statements we are running inside the loop and what expression defines when the loop stops. AST can be very very deep because in programming languages we can nest things. Like a loop that has function call inside, that has another function call, ...

AST is made from lexer provided tokens that are matched with different BNF rules and after that we run semantic checks to the AST to see if the code is doing legal things. Like calling a function with correct variable type.

2. How is the AST generated using the PLY tool? I.e., what things are needed in the code and how are they related to syntactic rules of the language and the tree?

PLY tool inputs the lexer tokens that are matched into different symbols with BNF rules that are written. When matching these rules the AST nodes are created because matching the BNF rules is already a process where the code finds out the relations, nested things etc. The actual nodes are made with a custom node class ASTnode, that will contain all needed details.

3. Explain in English what kind of tree is formed in your code from the following syntactic elements:
    1. Variable definitions
    2. While loop
    3. Procedure call

1. Variable definition has different child(singular) and children nodes: Child node for the name of the function, child node for the return type and a child node for the body of the function. The children nodes are var_defs list (vars defined in the function, can be none) and formal args list that contains the parameter nodes that define name and type of the parameters.

2. While loop contains the condition(expression) about continuing the looping, and statements list (stmts) that are executed every loop. Every statement is its own subtree than contains all the things the statement consists of.

3. Procedure call node has one child for the name of the proc that is called, and a list of arguments that are given to the procedure function.

All nodes also have the linenumber!

4. Answer the following based on the syntax definition and your implementation:
    1. In which cases is it possible in your implementation to end up with a tree with "empty" child attributes (somewhere in the tree there is a place for a child node (or nodes), but there is none)? I.e., in which situations you end up with tree nodes with `child_...` attribute being None, or `children_...` attribute being an empty list?
    2. Are there places in your implementation where you were able to "simplify" the tree by omitting trivial/non-useful nodes or by collecting a recursive repeating structure into a list of child nodes?

1. This NONE node can happen in every place where my BNF syntax has something called opt_XXX, Meaning that XXX is optional and if it is not provided in the PostHaste code, then there is an empty child(ren) node.

These situations are:

- optional variable definitions inside of a func or proc definition
- procedure function not returning anything
- func or proc def not taking any parameters (opt_formals)
- func or proc call not giving any arguments (opt_args)
- optional otherwise of an unless statement
- optional definitions of the program.

2. I am not sure about this question really. I feel like there is nothing "special" in my implementation, pretty straightforward way of doing everything. No unnecessary nodes that much.

5. What did you think of this assignment? What was difficult? What was easy? Did you learn anything useful?

I think the assignment was pretty fun and has great logical thinking needed. I feel like it was not too large but very close to it, meaning that if there was a bit more different BNF syntax needed the thing would go out of control because I am lacking a good automatic way of testing the changes I make (not breaking something that worked earlier). In the earlier steps I did use auto test to compare my implementation to the example outputs but I had to reverse-engineer everything to make everything 100% same for the automatic tests to work (compare my output to example output char by char basically).

The example tree uses id_name when defining a variable and var_rvalue later on, I didn't realize how to make this so I use just id_name all the time. Not sure if this creates some problems because there needs to be an additional check then to see if some "id_name" is definition or not.

I also couldn't make the date semantic checks well. I made check for initialization and assignment but not the addition/subtraction ones.

Other semantic checks should be working, and I decided to do the checks in separate visits to make my life a bit easier.