CONVERT_GRAMMAR:

My convert_grammar function creates the rule function by mapping the function input (nonterminal symbol) to the input of a recursive function called join_rules. join_rules takes 2 inputs: the list of hw1-formatted rules and a specific nonterminal input symbol. It recursively checks through all of the rules, adding them to a recursively enumerated list of output symbol lists if and only if the start_symbol for that rule is the same as the specific nonterminal input symbol.

PARSE_PREFIX:

My parse_prefix function uses two linked recursive auxiliary functions: check_state and expand_NT. check_state is given a derivation list and a current state (what you get after applying the derivation list to the start symbol), and it recursively checks that the current state matches the given fragment, character by character.

When the first character of the current state is a terminal symbol (this changes each iteration, since each iteration cuts off the first character of the current state), check_state simply checks whether this character matches the first character of the fragment. If so, we cut off the first character of the state and of the fragment, and run check_state again. If not, we return None, since obviously this derivation is incorrect.

When the first character of the current state is a nonterminal symbol, things get more complicated. We apply the rules function to this nonterminal symbol in order to find all the possible ways to replace it in the state, then we call expand_NT to dispatch check_state functions for each of the possible new states (I will explain expand_NT in more detail after I finish describing check_state). If expand_NT returns None, then none of the possible derivations were acceptable, and we return None. Otherwise, we return whatever expand_NT returns.

When the current state is just the empty list ([]), that means that we've successfully verified that each character of our current state (with a specific derivation) is in the given fragment. As such, we then run the acceptor on this derivation and the remaining characters in the given fragment (since we've been recursively cutting off the first character of the fragment, the remaining characters are directly at hand - no extra processing needed).

In addition, if frag is shorter than our state (frag is empty before our current state is), check_state returns none.

In comparison, the expand_NT function is simple. Given a nonterminal symbol, a list of replacements for that symbol, the rest of the current state (starting just after the nonterminal symbol), and some other bookkeeping variables, expand_NT will recursively run through all the possible replacements, creating the corresponding new state, and running check_state on it. If that instance of check_state returns a value != None (successful derivation), it returns that value. If that instance of check_state returns None (failed to match/accept), the expand_NT function will run another instance of itself, allowing it try the next possible new state, until either one is successful, or it runs out of states to try. If it runs out of states/derivations to try, expand_NT returns None.

This implementation of parse_prefix has one notable weakness. It doesn't defend itself against infinite loops of derivation. Since it simply forms a depth first search with a specific order (left to right in nonterminal

```
symbols, and left to right in rule lists), it can risk running forever,
repeating a cycle of derivations with no end. If there exists a cycle of
derivations that can be run infinitely (the first symbol of the replacement
must be a nonterminal for this to work), and any one of those derivations is
earlier in the search order (which is specified in the homework specification)
than the correct derivation, the functions will loop infinitely.
```