|  |
| --- |
|  |
|  | To start with, I first focus on the differences between these two styles. In |
|  | Homework1, the grammar is of the form: (start\_symbol, [[lhs1, rhs1]; [lhs2, rhs |
|  | 2]; ...]) The first item in the pair is the starting symbol, which indicates |
|  | where the parsing should begin. The second item is a list of rule pairs. The |
|  | Homework2-grammar is of the form: (start\_symbol, fun x -> [rhs1; rhs2; ...]) |
|  | since the type of the convert function is: val convert\_grammar : 'a \* ('b \* 'c |
|  | ) list -> 'a \* ('b -> 'c list) = <fun>. (However, when I wrote the code, I did |
|  | not write in the form of "fun x -> [...]". Instead, I directly wrote ..., ( |
|  | convert\_rules rules []) where convert\_rules is the recursive helper function |
|  | which takes three arguments.) I find out that when I convert the grammar, the |
|  | first item, which is the start symbol, is the same. What I need to do now is to |
|  | convert the list of rule pairs into a lambda function, which maps from the |
|  | parameter x to the corresponding rhs in our rules. To solve this, I write a |
|  | recursive helper function that can recursively add each rule pair in the |
|  | original list to the lambda function. |
|  |  |
|  | **(\* parse\_prefix \*)** |
|  | To start with, I look at the sample test cases provided on the course website. |
|  | The function parse\_prefix should take three argument: grammar, acceptor, and |
|  | fragment. Fragment is the expression we're going to parse. As mentioned in |
|  | class, acceptor takes a rule list and a suffix of the fragment. It returns the |
|  | value given by the Some construction. Acceptor will be updated on the fly. |
|  | Besides acceptor, we need matchers to deal with the prefix. The matcher and the |
|  | acceptor work together and help us parse the fragment. I drew a simple parsing |
|  | tree to get a feeling of the process: |
|  | (Expr, [N Term; N Binop; N Expr]) |
|  | | |
|  | +------------------------------+---------------------------+ |
|  | | | | |
|  | (Term, [N Num]) (Binop, [T "+"]) (Expr, [N Term]) |
|  | | | |
|  | (Num, [T "1"]) (Term, [N Num]) |
|  | | |
|  | (Num, [T "3"]) |
|  | 1 + 3 |
|  | Then I considered two possible approaches. The first way is to do the parsing |
|  | from the bottom (eg. "1", "+", "3") to the top. We first match tokens to |
|  | terminal symbols from left to right, and then match those terminals to our |
|  | rules. However, it would be easy to miss cases by matching upwards. Meanwhile, |
|  | I could not make sure that the derivation built in this direction has the |
|  | leftmost priority. According to the spec, the derivation should be a leftmost |
|  | derivation. Therefore, I rejected this method. The second way of parsing is to |
|  | do it from the top to the bottom, by expanding the leftmost nonterminals. It |
|  | reminds me of DFS traversal, which is good for backtracking. |
|  |  |
|  | (\* General Ideas \*) |
|  | The parser\_prefix function will start with the starting symbol indicated by the |
|  | first item in the grammar pair. We can also regard this starting symbol as the |
|  | root of our parsing tree. With this starting symbol 'start', and notice that |
|  | the second item in the grammar pair is a function that match a nonterminal to a |
|  | list of possible rules, we can find all possible derivation rules for the |
|  | starting symbol. Let's say the rule list is orig\_rules, then rule candidates |
|  | for our starting symbol is (orig\_rules start). Next, we pass the original rule |
|  | list, the starting symbol (the prefix we're looking at), the rule candidates, |
|  | the acceptor, an empty list of derivation, and the fragment into the matcher |
|  | to figure out the correct parsing. |
|  |  |
|  | Next, we need to construct two matcher functions. As mentioned above, matchers |
|  | are functions that match a prefix of fragments. Inside the matcher function, it |
|  | should call the corresponding acceptor to see whether the suffix can pass the |
|  | acceptor or not. If there is no prefix, return None. If the acceptor returns |
|  | None, we should use the next matching prefix. The prefix might be either a |
|  | terminal symbol or a nonterminal symbol. Meanwhile, for each symbol/prefix, |
|  | there might be more than one rules. We need to take care of all these cases. |
|  |  |
|  | or\_matcher: This matcher is designed to deals with the "disjunction" produced |
|  | by symbols with several rules, since here might be more than one rule to worry |
|  | about. We need to create a choice point for alternatives when parsing. It takes |
|  | the full rule list, the prefix, the rule alternatives, the acceptor, current |
|  | derivation, and the fragment as its input. It calls the and\_matcher to |
|  | recursively check the suffix. If and\_matcher returns None, it means we need |
|  | to pick the next rule in the list of rule alternatives and call or\_matcher |
|  | again. Otherwise, we accept this step of derivation and move on. |
|  |  |
|  | and\_matcher: This matcher is a recursive function that deals with the "and" |
|  | cases. We use this function to completely match the suffix with the symbols in |
|  | our rules. If current symbol is a nonterminal, it means we need to go deeper by |
|  | calling or\_matcher to find possible derivations for this symbol. In this case, |
|  | the acceptor function that we pass into or\_matcher needs to be updated. If the |
|  | symbol is a terminal, we can check by matching the head of the current fragment |
|  | with the symbol. If they match, we keep going by recursive calls. Otherwise, |
|  | return None. |
|  |  |
|  | (\* Problems I encountered & Weakness of the parser \*) |
|  | It took me a long time to understand the usage of the acceptor and the way to |
|  | update it on the fly. After I read the TA's response regarding to this issue on |
|  | Piazza, I learned that the "inner acceptor" is formed by calling the matcher |
|  | with the full list of rules, the rest of alternatives and the old acceptor. In |
|  | this way, we can form the curried function with two arguments: a derivation and |
|  | a fragment (since and\_matcher takes five arguments and the last two are left |
|  | blank when we call and\_matcher to form the new acceptor). |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  | (\* Warm up \*) |
|  |  |  | let rec convert\_rules rules lst x = |
|  |  |  | match rules with |
|  |  |  | | [] -> lst (\* the rule list is exhausted \*) |
|  |  |  | | rules\_head :: rules\_tail -> |
|  |  |  | match rules\_head with |
|  |  |  | | (lhs, rhs) -> |
|  |  |  | if lhs = x then convert\_rules rules\_tail (lst@[rhs]) x |
|  |  |  | else |
|  |  |  | convert\_rules rules\_tail lst x |
|  |  |  |  |
|  |  |  | let convert\_grammar gram1 = match gram1 with |
|  |  |  | (\* the first item in gram1 is the start symbol |
|  |  |  | the second item are the list of rules that need to be converted into HW2 form \*) |
|  |  |  | | (start\_symb, rules) -> (start\_symb, (convert\_rules rules [])) |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  | (\* Write a function that returns a matcher for the grammar\*) |
|  |  |  | let append\_dev prev\_dev lhs rule = |
|  |  |  | prev\_dev @ [(lhs, rule)] |
|  |  |  |  |
|  |  |  | (\* the recursive function that deals with the "and" cases \*) |
|  |  |  | (\* use this function to completely match the suffix with the symbols in our rules \*) |
|  |  |  | let rec and\_matcher orig\_rules pick\_rules acceptor derivation fragment = |
|  |  |  | match pick\_rules with |
|  |  |  | (\* no more alternative to examine, |
|  |  |  | plug derivation and fragment into the acceptor, |
|  |  |  | return what acceptor returns \*) |
|  |  |  | | [] -> acceptor derivation fragment |
|  |  |  | (\* If the first symbol of the rule is a nonterminal, parse this symbol \*) |
|  |  |  | | ((N fst) :: rest) -> |
|  |  |  | let new\_acceptor = and\_matcher orig\_rules rest acceptor |
|  |  |  | in (\* In this situation, we need to update the acceptor on the fly \*) |
|  |  |  | (\* Call or\_matcher to find possible derivations for this, applying the new acceptor \*) |
|  |  |  | or\_matcher orig\_rules fst (orig\_rules fst) new\_acceptor derivation fragment |
|  |  |  | (\* If the first symbol is a terminal, match the head of the current fragment with this symbol \*) |
|  |  |  | | ((T fst) :: rest) -> |
|  |  |  | match fragment with |
|  |  |  | | [] -> None (\* If there is no token in fragment, return None \*) |
|  |  |  | (\* If matches, we keep going by recursive calls \*) |
|  |  |  | | (hd :: tl) -> if (hd = fst) then (and\_matcher orig\_rules rest acceptor derivation tl) |
|  |  |  | else None (\* If fails, returns None \*) |
|  |  |  |  |
|  |  |  | (\* the function deals with the "disjunction" produced by symbols with several rules \*) |
|  |  |  | and or\_matcher orig\_rules start pick\_rules acceptor derivation fragment = |
|  |  |  | match pick\_rules with |
|  |  |  | (\* return None when there is no matched alternative \*) |
|  |  |  | | [] -> None |
|  |  |  | | (fst :: rest) -> |
|  |  |  | let check\_suffix = and\_matcher orig\_rules fst acceptor (append\_dev derivation start fst) fragment |
|  |  |  | in (\* call the and\_matcher to recursively check the suffix \*) |
|  |  |  | match check\_suffix with |
|  |  |  | (\* If and\_matcher returns None, |
|  |  |  | pick the next rule in the list of rule alternatives and call or\_matcher again |
|  |  |  | \*) |
|  |  |  | | None -> or\_matcher orig\_rules start rest acceptor derivation fragment |
|  |  |  | | other -> other (\* Otherwise, we accept this step of derivation and move on. \*) |
|  |  |  |  |
|  |  |  | (\* Main function \*) |
|  |  |  | let parse\_prefix grammar acceptor fragment = |
|  |  |  | match grammar with |
|  |  |  | | (start, orig\_rules) -> or\_matcher orig\_rules start (orig\_rules start) acceptor [] fragment |
|  |  |  | (\* orig\_rules is a function, which returns a list of possible rules for a prefix. \*) |