Homework 2. Naive parsing of context free grammars

Motivation

You'd like to test grammars that are being proposed as test cases for CS 132 projects. One way is to test it on actual CS 132 projects, but those projects aren't done yet and anyway you'd like a second opinion in case the student projects are incorrect. So you decide to write a simple parser generator. Given a grammar in the style of Homework 1, your program will generate a function that is a parser. When this parser is given a string whose prefix is a program to parse, it returns the corresponding unmatched suffix, or an error indication if no prefix of the string is a valid program.

The key notion of this assignment is that of a matcher. A matcher is a function that inspects a given string of terminals to find a match for a prefix that corresponds to a nonterminal symbol of a grammar, and then checks whether the match is acceptable by testing whether a given acceptor succeeds on the corresponding suffix. For example, a matcher for `awkish_grammar` below might inspect the string `["3";"+";"4";"-"]` and find two possible prefixes that match, namely `["3";"+";"4"]` and `["3"]`. The matcher will first apply the acceptor to the first prefix `["3";"+";"4"]`, along with the corresponding suffix `["-"]`. If this is accepted, the matcher will return whatever the acceptor returns. Otherwise, the matcher will apply the acceptor to the second prefix `["3"]`, along with the corresponding suffix `["+";"4";"-"]`, and will return whatever the acceptor returns. If a matcher finds no matching prefixes, it returns the special value `None`.

As you can see by mentally executing the example, matchers sometimes need to try multiple alternatives and to backtrack to a later alternative if an earlier one is a blind alley.

An acceptor is a function that accepts a rule list and a suffix by returning some value wrapped inside the `Some` constructor. The acceptor rejects the rule list and suffix by returning `None`. For example, the acceptor `(function | "+"::t -> Some ("+"::t) | _ -> None)` accepts any rule list but accepts only suffixes beginning with `"+"`. Such an acceptor would cause the example matcher to fail on the prefix `["3";"+";"4"]` (since the corresponding suffix begins with `"-"`, not `"+"`) but it would succeed on the prefix `["3"]`.

By convention, an acceptor that is successful returns `Some s`, where `s` is a tail of the input suffix (because the acceptor may have parsed more of the input, and has therefore consumed some of the suffix). This allows the matcher's caller to retrieve an indication of where the matched prefix ends (since it ends just before the suffix starts). Although this behavior is crucial for the internal acceptors used by your code, it is not required for top-level acceptors supplied by test programs: a top-level acceptor needs only to return a `Some x` value to succeed.

Whenever there are several rules to try for a nonterminal, you should always try them left-to-right. For example, `awkish_grammar` below contains this:

```
| Expr ->
   [[N Term; N Binop; N Expr];
    [N Term]]
```

and therefore, your matcher should attempt to use the rule "Expr → Term Binop Expr" before attempting to use the simpler rule "Expr → Term".

If you can build a matcher, it should be relatively easy to build a parser, which yields a parse tree that corresponds to its input fragment.

Definitions

symbol, right hand side, rule
: same as in Homework 1.

alternative list
: A list of right hand sides. It corresponds to all of a grammar's rules for a given nonterminal symbol. By convention, an empty alternative list `[]` is treated as if it were a singleton list `[[]]` containing the empty symbol string.

production function
: A function whose argument is a nonterminal value. It returns a grammar's alternative list for that nonterminal.

grammar
: A pair, consisting of a start symbol and a production function. The start symbol is a nonterminal value.

fragment
: a list of terminal symbols, e.g., `["3"; "+"; "4"; "xyzzy"]`.

acceptor
: a function whose argument is a fragment *frag*. If the fragment is not acceptable, it returns `None`; otherwise it returns `Some x` for some value *x*.

matcher
: a curried function with two arguments: an acceptor *accept* and a fragment *frag*. A matcher matches a prefix *p* of *frag* such that *accept* (when passed the corresponding suffix) accepts the corresponding suffix (i.e., the suffix of *frag* that remains after *p* is removed). If there is such a match, the matcher returns whatever *accept* returns; otherwise it returns `None`.

parse tree
: a data structure representing a parse tree in the usual way. It has the following OCaml type:

```
type ('nonterminal, 'terminal) parse_tree =
  | Node of 'nonterminal * ('nonterminal, 'terminal) parse_tree list
  | Leaf of 'terminal
```

If you traverse a parse tree in preorder left to right, the leaves you encounter contain the same terminal symbols as the parsed fragment, and each internal node of the parse tree corresponds to a rule in the grammar, traversed in a leftmost derivation order.

parser
: a function from fragments to parse trees. Parsers consume the entire input, unlike matchers, which may consume only an initial prefix of the input.

Assignment

1. To warm up, notice that the format of grammars is different in this assignment, versus Homework 1. Write a function `convert_grammar` *gram1* that returns a Homework 2-style grammar, which is converted from the Homework 1-style grammar *gram1*. Test your implementation of `convert_grammar` on the test grammars given in Homework 1. For example, the top-level definition `let awksub_grammar_2 = convert_grammar awksub_grammar` should bind `awksub_grammar_2` to a Homework 2-style grammar that is equivalent to the Homework 1-style grammar `awksub_grammar`.
2. As another warmup, write a function `parse_tree_leaves tree` that traverses the parse tree `tree` left to right and yields a list of the leaves encountered.
3. Write a function `make_matcher` *gram* that returns a matcher for the grammar *gram*. When applied to an acceptor *accept* and a fragment *frag*, the matcher must try the grammar rules in order and return the result of calling *accept* on the suffix corresponding to the first acceptable matching prefix of *frag*; this is not necessarily the shortest or the longest acceptable match. A match is considered to be acceptable if *accept* succeeds when given the suffix fragment that immediately follows the matching prefix. When this happens, the matcher returns whatever the acceptor returned. If no acceptable match is found, the matcher returns `None`.

4. Write a function `make_parser gram` that returns a parser for the grammar gram. When applied to a fragment frag, the parser returns an optional parse tree. If frag cannot be parsed entirely (that is, from beginning to end), the parser returns `None`. Otherwise, it returns `Some tree` where `tree` is the parse tree corresponding to the input fragment. Your parser should try grammar rules in the same order as `make_matcher`.
5. Write one good, nontrivial test case for your `make_matcher` function. It should be in the style of the test cases given below, but should cover different problem areas. Your test case should be named `make_matcher_test`. Your test case should test a grammar of your own.
6. Similarly, write a good test case `make_parser_test` for your `make_parser` function using your same test grammar. This test should check that `parse_tree_leaves` is in some sense the inverse of `make_parser gram`, in that when `make_parser gram frag` returns `Some tree`, then `parse_tree_leaves tree` equals `frag`.
7. Assess your work by writing an after-action report that explains why you decided to write `make_parser` in terms of `make_matcher`, or vice versa, or neither; and if it's "neither" then briefly explain the approach that you took to avoid duplication in the two functions. Also, explain any weaknesses in your solution in the context of its intended application. If possible, illustrate weaknesses by test cases that fail with your implementation. This report should be a simple ASCII plain text file that consumes a page or so (at most 100 lines and 80 columns per line, and at least 50 lines, please). See Resources for oral presentations and written reports for advice on how to write assessments; admittedly much of the advice there is overkill for the simple kind of report we're looking for here.

Unlike Homework 1, we are expecting some weaknesses here, so your assessment should talk about them. For example, we don't expect that your implementation will work with all possible grammars, but we would like to know which sort of grammars it will have trouble with.

As with Homework 1, your code may use the `Pervasives` and `List` modules, but it should use no other modules. Your code should be free of side effects. Simplicity is more important than efficiency, but your code should avoid using unnecessary time and space when it is easy to do so.

Submit

We will test your program on the SEASnet Linux servers as before, so make sure that `/usr/local/cs/bin` is at the start of your path, using the same technique as in Homework 1.

Submit three files:

- `hw2.ml` should define `convert_grammar`, `parse_tree_leaves`, `make_matcher` and `make_parser` along with any auxiliary types and functions needed to define `make_matcher`.
- `hw2test.ml` should contain your test cases along with any auxiliaries need for them.
- `hw2.txt` should hold your assessment.

Sample test cases

```
let accept_all string = Some string
let accept_empty_suffix = function
   | _::_ -> None
   | x -> Some x

(* An example grammar for a small subset of Awk.
   This grammar is not the same as Homework 1; it is
   instead the same as the grammar under
   "Theoretical background" above.  *)

type awksub_nonterminals =
  | Expr | Term | Lvalue | Incrop | Binop | Num
```

```
let awkish_grammar =
  (Expr,
   function
     | Expr ->
         [[N Term; N Binop; N Expr];
          [N Term]]
     | Term ->
         [[N Num];
          [N Lvalue];
          [N Incrop; N Lvalue];
          [N Lvalue; N Incrop];
          [T"("; N Expr; T")"]]
     | Lvalue ->
         [[T"$"; N Expr]]
     | Incrop ->
         [[T"++"];
          [T"--"]]
     | Binop ->
         [[T"+"];
          [T"-"]]
     | Num ->
         [[T"0"]; [T"1"]; [T"2"]; [T"3"]; [T"4"];
          [T"5"]; [T"6"]; [T"7"]; [T"8"]; [T"9"]])

let test0 =
  ((make_matcher awkish_grammar accept_all ["ouch"]) = None)

let test1 =
  ((make_matcher awkish_grammar accept_all ["9"])
   = Some [])

let test2 =
  ((make_matcher awkish_grammar accept_all ["9"; "+"; "$"; "1"; "+"])
   = Some ["+"])

let test3 =
  ((make_matcher awkish_grammar accept_empty_suffix ["9"; "+"; "$"; "1"; "+"])
   = None)

(* This one might take a bit longer.... *)
let test4 =
 ((make_matcher awkish_grammar accept_all
     ["("; "$"; "8"; ")"; "-"; "$"; "++"; "$"; "--"; "$"; "9"; "+";
      "("; "$"; "++"; "$"; "2"; "+"; "("; "8"; ")"; "-"; "9"; ")";
      "-"; "("; "$"; "$"; "$"; "$"; "$"; "++"; "$"; "$"; "5"; "++";
      "++"; "--"; ")"; "-"; "++"; "$"; "$"; "("; "$"; "8"; "++"; ")";
      "++"; "+"; "0"])
  = Some [])

let test5 =
  (parse_tree_leaves (Node ("+", [Leaf 3; Node ("*", [Leaf 4; Leaf 5])]))
   = [3; 4; 5])

let small_awk_frag = ["$"; "1"; "++"; "-"; "2"]

let test6 =
  ((make_parser awkish_grammar small_awk_frag)
   = Some (Node (Expr,
                 [Node (Term,
                        [Node (Lvalue,
                               [Leaf "$";
                                Node (Expr,
                                      [Node (Term,
                                             [Node (Num,
```

```
                                                    [Leaf "1"])])])])]);
                         Node (Incrop, [Leaf "++"])]);
                  Node (Binop,
                        [Leaf "-"]);
                  Node (Expr,
                        [Node (Term,
                               [Node (Num,
                                      [Leaf "2"])])])])]))
let test7 =
  match make_parser awkish_grammar small_awk_frag with
    | Some tree -> parse_tree_leaves tree = small_awk_frag
    | _ -> false
```

Sample use of test cases

If you put the sample test cases into a file `hw2sample.ml`, you should be able to use it with something ike the following to test your `hw2.ml` solution on the SEASnet implementation of OCaml. Similarly, the command `#use "hw2test.ml";;` should run your own test cases on your solution.

```
$ ocaml
        OCaml version 4.07.1

# #use "hw2.ml";;
...
val parse_tree_leaves : ('a, 'b) parse_tree -> 'b list = <fun>
...
val make_matcher :
  'a * ('a -> ('a, 'b) symbol list list) ->
  ('b list -> 'c option) ->
  'b list -> 'c option = <fun>
...
val make_parser :
  'a * ('a -> ('a, 'b) symbol list list) ->
  'b list ->
  ('a, 'b) parse_tree option = <fun>
...
# #use "hw2sample.ml";;
val accept_all : 'a -> 'a option = <fun>
val accept_empty_suffix : 'a list -> 'b list option = <fun>
type awksub_nonterminals = ...
val awkish_grammar :
  awksub_nonterminals *
  (awksub_nonterminals -> (awksub_nonterminals, string) symbol list list) =
  (Expr, <fun>)
val test0 : bool = true
val test1 : bool = true
val test2 : bool = true
val test3 : bool = true
val test4 : bool = true
val test5 : bool = true
...
val test6 : bool = true
val test7 : bool = true
#
```

Hint

You can use [a previous Homework 2]() as a hint. It is a tough homework and is not the same problem but there are some common ideas. Look for the sample solution at the end.