Homework 2. Naive parsing of context free grammars

Theoretical background

A *derivation* is a rule list that describes how to derive a phrase from a nonterminal symbol. For example, suppose we have the following grammar with start symbol Expr:

Expr → Term Binop Expr
Expr → Term
Term → Num
Term → Lvalue
Term → Incrop Lvalue
Term → Lvalue Incrop
Term → "(" Expr ")"
Lvalue → $ Expr
Incrop → "++"
Incrop → "−−"
Binop → "+"
Binop → "−"
Num → "0"
Num → "1"
Num → "2"
Num → "3"
Num → "4"
Num → "5"
Num → "6"
Num → "7"
Num → "8"
Num → "9"

Then here is a derivation for the phrase "3" "+" "4" from the nonterminal Expr. After each rule is applied, the resulting list of terminals and nonterminals is given.

rule	after rule is applied
(at start)	Expr
Expr → Term Binop Expr	Term Binop Expr
Term → Num	Num Binop Expr
Num → "3"	"3" Binop Expr
Binop → "+"	"3" "+" Expr
Expr → Term	"3" "+" Term
Term → Num	"3" "+" Num
Num → "4"	"3" "+" "4"

In a *leftmost derivation*, the leftmost nonterminal is always the one that is expanded next. The above example is a leftmost derivation.

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 program to parse, it produces a derivation for that program, or an error indication if the program contains a syntax error and cannot be parsed.

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 derivation and 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 a derivation for 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 a derivation for 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 `(fun d -> function | "+"::t -> Some (d,"+"::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 (d,s)`, where `d` is a rule list that typically contains the acceptor's input rule list as a sublist (because the acceptor may do further parsing, and therefore has applied more rules than before), and `s` is a tail of the input suffix (again, 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 the derivation for the matched prefix, along with an indication 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".

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.

derivation
: a list of rules used to derive a phrase from a nonterminal. For example, the OCaml representation of the example derivation shown above is as follows:

```
[Expr, [N Term; N Binop; N Expr];
 Term, [N Num];
 Num, [T "3"];
 Binop, [T "+"];
 Expr, [N Term];
 Term, [N Num];
 Num, [T "4"]]
```

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

acceptor
: a curried function with two arguments: a derivation *d* and 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 a derivation and 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`.

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. Write a function `parse_prefix` *gram* that returns a matcher for the grammar *gram*. When applied to an acceptor *accept* and a fragment *frag*, the matcher must return the first acceptable match of a prefix of *frag*, by trying the grammar rules in order; this is not necessarily the shortest nor the longest acceptable match. A match is considered to be acceptable if *accept* succeeds when given a derivation and 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`.
3. Write two good, nontrivial test cases for your `parse_prefix` function. These test cases should all be in the style of the test cases given below, but should cover different problem areas. Your test cases should be named `test_1` and `test_2` (note the underscores; this distinguishes your test cases from the standard ones given below). Your test cases should test at least one grammar of your own. You may reuse your test cases for Homework 1 as part of `test_1`, but `test_2` should be new.
4. Assess your work by writing an after-action report that summarizes why you solved the problem the way you did, other approaches that you considered and rejected (and why you rejected them), and 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