Homework 1. Fixpoints and grammar filters

[131 home > Homework]

Introduction

You are a reader for Computer Science 181, which asks students to submit grammars that solve various problems. However, many of the submitted grammars are trivially wrong, in several ways. Here is one. Some grammars contain blind-alley rules, that is, grammar rules for which it is impossible to derive a string of terminal symbols. Blind-alley rules do not affect the language or parse trees generated by a grammar, so in some sense they don't make the answers wrong, but they're noise and they make grading harder. You'd like to filter out the noise, and just grade the useful parts of each grammar.

You've heard that OCaml is a good language for writing compilers and whatnot, so you decide to give it a try for this application. While you're at it, you have a background in fixed point and periodic point theory, so you decide to give it a try too.

Definitions

fixed point
: (of a function `f`) A point `x` such that `f x = x`. In this description we are using OCaml notation, in which functions always have one argument and parentheses are not needed around arguments.

computed fixed point
: (of a function `f` with respect to an initial point `x`) A fixed point of `f` computed by calculating `x`, `f x`, `f (f x)`, `f (f (f x))`, etc., stopping when a fixed point is found for `f`. If no fixed point is ever found by this procedure, the computed fixed point is not defined for `f` and `x`.

periodic point
: (of a function `f` with period `p`) A point `x` such that `f (f ... (f x)) = x`, where there are `p` occurrences of `f` in the call. That is, a periodic point is like a fixed point, except the function returns to the point after `p` iterations instead of 1 iteration. Every point is a periodic point for `p`=0. A fixed point is a periodic point for `p`=1.

computed periodic point
: (of a function `f` with respect to a period `p` and an initial point `x`) A periodic point of `f` with period `p`, computed by calculating `x`, `f x`, `f (f x)`, `f (f (f x))`, etc., stopping when a periodic point with period `p` is found for `f`. The computed periodic point need not be equal to `x`. If no periodic point is ever found by this procedure, the computed periodic point is not defined for `f`, `p`, and `x`.

symbol
: A symbol used in a grammar. It can be either a nonterminal symbol or a terminal symbol; each kind of symbol has a value, whose type is arbitrary. A symbol has the following OCaml type:

```
type ('nonterminal, 'terminal) symbol =
  | N of 'nonterminal
  | T of 'terminal
```

right hand side
: A list of symbols. It corresponds to the right hand side of a single grammar rule. A right hand side can be empty.

rule
: A pair, consisting of (1) a nonterminal value (the left hand side of the grammar rule) and (2) a right hand side.

grammar

A pair, consisting of a start symbol and a list of rules. The start symbol is a nonterminal value.

Assignment

You will warm up by modeling sets using OCaml lists. The empty list represents the empty set, and if the list `t` represents the set T, then the list `h::t` represents the set {`h`}∪T. Although sets by definition do not contain duplicates, the lists that represent sets can contain duplicates. Another set of warmup exercises will compute fixed and periodic points. Finally, you can write a function that filters blind alleys.

1. Write a function `subset a b` that returns true iff $a \subseteq b$, i.e., if the set represented by the list `a` is a subset of the set represented by the list `b`. Every set is a subset of itself. This function should be generic to lists of any type: that is, the type of `subset` should be a generalization of `'a list -> 'a list -> bool`.
2. Write a function `equal_sets a b` that returns true iff the represented sets are equal.
3. Write a function `set_union a b` that returns a list representing a∪b.
4. Write a function `set_intersection a b` that returns a list representing a∩b.
5. Write a function `set_diff a b` that returns a list representing $a-b$, that is, the set of all members of *a* that are not also members of *b*.
6. Write a function `computed_fixed_point eq f x` that returns the computed fixed point for `f` with respect to `x`, assuming that `eq` is the equality predicate for `f`'s domain. A common case is that `eq` will be `(=)`, that is, the builtin equality predicate of OCaml; but any predicate can be used. If there is no computed fixed point, your implementation can do whatever it wants: for example, it can print a diagnostic, or go into a loop, or send nasty email messages to the user's relatives.
7. Write a function `computed_periodic_point eq f p x` that returns the computed periodic point for `f` with period `p` and with respect to `x`, assuming that `eq` is the equality predicate for `f`'s domain.
8. Write a function `while_away s p x` that returns the longest list [x; s x; s (s x); ...] such that `p e` is true for every element `e` in the list. That is, if `p x` is false, return `[]`; otherwise if `p (s x)` is false, return `[x]`; otherwise if `p (s (s x))` is false, return `[x; s x]`; and so forth. For example, `while_away ((+) 3) ((>) 10) 0` returns [0; 3; 6; 9]. Your implementation can assume that `p` eventually returns `false`.
9. Write a function `rle_decode lp` that decodes a list of pairs `lp` in run-length encoding form. The first element of each pair is a nonnegative integer specifying the repetition length; the second element is the value to repeat. For example, `rle_decode [2,0; 1,6]` should return [0; 0; 6] and `rle_decode [3,"w"; 1,"x"; 0,"y"; 2,"z"]` should return ["w"; "w"; "w"; "x"; "z"; "z"].
10. OK, now for the real work. Write a function `filter_blind_alleys g` that returns a copy of the grammar `g` with all blind-alley rules removed. This function should preserve the order of rules: that is, all rules that are returned should be in the same order as the rules in `g`.
11. Supply at least one test case for each of the above functions in the style shown in the sample test cases below. When testing the function `F` call the test cases `my_F_test0`, `my_F_test1`, etc. For example, for `subset` your first test case should be called `my_subset_test0`. Your test cases should exercise all the above functions, even though the sample test cases do not.

Your code should follow these guidelines:

1. Your code may use the `Pervasives` and `List` modules, but it should use no other modules other than your own code.
2. It is OK (and indeed encouraged) for your solutions to be based on one another; for example, it is fine for `filter_blind_alleys` to use `equal_sets` and `computed_fixed_point`.
3. Your code should prefer pattern matching to conditionals when pattern matching is natural.
4. Your code should be free of side effects such as loops, assignment, input/output, `incr`, and `decr`. Use recursion instead of loops.
5. Simplicity is more important than efficiency, but your code should avoid using unnecessary time and space when it is easy to do so. For example, instead of repeating a expression, compute its value once and reuse the computed value.
6. The test cases below should work with your program. You are unlikely to get credit for it otherwise.

Submit

Submit two files. The file `hw1.ml` should implement the abovementioned functions, along with any auxiliary types and functions; in particular, it should define the `symbol` type as shown above. The file `hw1test.ml` should contain your test cases. Please do not put your name, student ID, or other personally identifying information in your files.

Sample test cases

See hw1sample.ml for a copy of these tests.

```
let subset_test0 = subset [] [1;2;3]
let subset_test1 = subset [3;1;3] [1;2;3]
let subset_test2 = not (subset [1;3;7] [4;1;3])

let equal_sets_test0 = equal_sets [1;3] [3;1;3]
let equal_sets_test1 = not (equal_sets [1;3;4] [3;1;3])

let set_union_test0 = equal_sets (set_union [] [1;2;3]) [1;2;3]
let set_union_test1 = equal_sets (set_union [3;1;3] [1;2;3]) [1;2;3]
let set_union_test2 = equal_sets (set_union [] []) []

let set_intersection_test0 =
  equal_sets (set_intersection [] [1;2;3]) []
let set_intersection_test1 =
  equal_sets (set_intersection [3;1;3] [1;2;3]) [1;3]
let set_intersection_test2 =
  equal_sets (set_intersection [1;2;3;4] [3;1;2;4]) [4;3;2;1]

let set_diff_test0 = equal_sets (set_diff [1;3] [1;4;3;1]) []
let set_diff_test1 = equal_sets (set_diff [4;3;1;1;3] [1;3]) [4]
let set_diff_test2 = equal_sets (set_diff [4;3;1] []) [1;3;4]
let set_diff_test3 = equal_sets (set_diff [] [4;3;1]) []

let computed_fixed_point_test0 =
  computed_fixed_point (=) (fun x -> x / 2) 1000000000 = 0
let computed_fixed_point_test1 =
  computed_fixed_point (=) (fun x -> x *. 2.) 1. = infinity
let computed_fixed_point_test2 =
  computed_fixed_point (=) sqrt 10. = 1.
let computed_fixed_point_test3 =
  ((computed_fixed_point (fun x y -> abs_float (x -. y) < 1.)
                         (fun x -> x /. 2.)
                         10.)
   = 1.25)

let computed_periodic_point_test0 =
  computed_periodic_point (=) (fun x -> x / 2) 0 (-1) = -1
let computed_periodic_point_test1 =
  computed_periodic_point (=) (fun x -> x *. x -. 1.) 2 0.5 = -1.

(* An example grammar for a small subset of Awk, derived from but not
   identical to the grammar in
   <http://web.cs.ucla.edu/classes/winter06/cs132/hw/hw1.html>.  *)

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

let awksub_rules =
   [Expr, [T"("; N Expr; T")"];
    Expr, [N Num];
```

```
     Expr, [N Expr; N Binop; N Expr];
     Expr, [N Lvalue];
     Expr, [N Incrop; N Lvalue];
     Expr, [N Lvalue; N Incrop];
     Lvalue, [T"$"; N Expr];
     Incrop, [T"++"];
     Incrop, [T"--"];
     Binop, [T"+"];
     Binop, [T"-"];
     Num, [T"0"];
     Num, [T"1"];
     Num, [T"2"];
     Num, [T"3"];
     Num, [T"4"];
     Num, [T"5"];
     Num, [T"6"];
     Num, [T"7"];
     Num, [T"8"];
     Num, [T"9"]]

let awksub_grammar = Expr, awksub_rules

let awksub_test0 =
  filter_blind_alleys awksub_grammar = awksub_grammar

let awksub_test1 =
  filter_blind_alleys (Expr, List.tl awksub_rules) = (Expr, List.tl awksub_rules)

let awksub_test2 =
  filter_blind_alleys (Expr,
      [Expr, [N Num];
       Expr, [N Lvalue];
       Expr, [N Expr; N Lvalue];
       Expr, [N Lvalue; N Expr];
       Expr, [N Expr; N Binop; N Expr];
       Lvalue, [N Lvalue; N Expr];
       Lvalue, [N Expr; N Lvalue];
       Lvalue, [N Incrop; N Lvalue];
       Lvalue, [N Lvalue; N Incrop];
       Incrop, [T"++"]; Incrop, [T"--"];
       Binop, [T"+"]; Binop, [T"-"];
       Num, [T"0"]; Num, [T"1"]; Num, [T"2"]; Num, [T"3"]; Num, [T"4"];
       Num, [T"5"]; Num, [T"6"]; Num, [T"7"]; Num, [T"8"]; Num, [T"9"]])
  = (Expr,
     [Expr, [N Num];
      Expr, [N Expr; N Binop; N Expr];
      Incrop, [T"++"]; Incrop, [T"--"];
      Binop, [T "+"]; Binop, [T "-"];
      Num, [T "0"]; Num, [T "1"]; Num, [T "2"]; Num, [T "3"]; Num, [T "4"];
      Num, [T "5"]; Num, [T "6"]; Num, [T "7"]; Num, [T "8"]; Num, [T "9"]])

let awksub_test3 =
  filter_blind_alleys (Expr, List.tl (List.tl (List.tl awksub_rules))) =
    filter_blind_alleys (Expr, List.tl (List.tl awksub_rules))

type giant_nonterminals =
  | Conversation | Sentence | Grunt | Snore | Shout | Quiet

let giant_grammar =
  Conversation,
  [Snore, [T"ZZZ"];
   Quiet, [];
   Grunt, [T"khrgh"];
   Shout, [T"aooogah!"];
   Sentence, [N Quiet];
```

```
   Sentence, [N Grunt];
   Sentence, [N Shout];
   Conversation, [N Snore];
   Conversation, [N Sentence; T","; N Conversation]]

let giant_test0 =
  filter_blind_alleys giant_grammar = giant_grammar

let giant_test1 =
  filter_blind_alleys (Sentence, List.tl (snd giant_grammar)) =
    (Sentence,
     [Quiet, []; Grunt, [T "khrgh"]; Shout, [T "aooogah!"];
      Sentence, [N Quiet]; Sentence, [N Grunt]; Sentence, [N Shout]])

let giant_test2 =
  filter_blind_alleys (Sentence, List.tl (List.tl (snd giant_grammar))) =
    (Sentence,
     [Grunt, [T "khrgh"]; Shout, [T "aooogah!"];
      Sentence, [N Grunt]; Sentence, [N Shout]])
```

Sample use of test cases

When testing on SEASnet, use one of the machines `lnxsrv07.seas.ucla.edu` and `lnxsrv08.seas.ucla.edu`. Make sure `/usr/local/cs/bin` is at the start of your path, so that you get the proper version of OCaml. To do this, append the following lines to your `$HOME/.profile` file if you use `bash` or `ksh`:

```
export PATH=/usr/local/cs/bin:$PATH
```

or the following line to your `$HOME/.login` file if you use `tcsh` or `csh`:

```
set path=(/usr/local/cs/bin $path)
```

The command `ocaml` should output the version number 4.03.0.

If you put the sample test cases into a file `hw1sample.ml`, you should be able to use it as follows to test your `hw1.ml` solution on the SEASnet implementation of OCaml. Similarly, the command `#use "hw1test.ml";;` should run your own test cases on your solution.

```
$ ocaml
        OCaml version 4.03.0

# #use "hw1.ml";;
type ('a, 'b) symbol = N of 'a | T of 'b
...
# #use "hw1sample.ml";;
val subset_test0 : bool = true
val subset_test1 : bool = true
val subset_test2 : bool = true
val equal_sets_test0 : bool = true
val equal_sets_test1 : bool = true
val set_union_test0 : bool = true
val set_union_test1 : bool = true
val set_union_test2 : bool = true
val set_intersection_test0 : bool = true
val set_intersection_test1 : bool = true
val set_intersection_test2 : bool = true
val computed_fixed_point_test0 : bool = true
val computed_fixed_point_test1 : bool = true
val computed_fixed_point_test2 : bool = true
val computed_fixed_point_test3 : bool = true
val computed_periodic_point_test0 : bool = true
```

```
val computed_periodic_point_test1 : bool = true
type awksub_nonterminals = Expr | Lvalue | Incrop | Binop | Num
val awksub_rules :
  (awksub_nonterminals * (awksub_nonterminals, string) symbol list) list =
  [(Expr, [T "("; N Expr; T ")"]); (Expr, [N Num]);
   (Expr, [N Expr; N Binop; N Expr]); (Expr, [N Lvalue]);
   (Expr, [N Incrop; N Lvalue]); (Expr, [N Lvalue; N Incrop]);
   (Lvalue, [T "$"; N Expr]); (Incrop, [T "++"]); (Incrop, [T "--"]);
   (Binop, [T "+"]); (Binop, [T "-"]); (Num, [T "0"]); (Num, [T "1"]);
   (Num, [T "2"]); (Num, [T "3"]); (Num, [T "4"]); (Num, [T "5"]);
   (Num, [T "6"]); (Num, [T "7"]); (Num, [T "8"]); (Num, [T "9"])]
val awksub_grammar :
  awksub_nonterminals *
  (awksub_nonterminals * (awksub_nonterminals, string) symbol list) list =
  (Expr,
   [(Expr, [T "("; N Expr; T ")"]); (Expr, [N Num]);
    (Expr, [N Expr; N Binop; N Expr]); (Expr, [N Lvalue]);
    (Expr, [N Incrop; N Lvalue]); (Expr, [N Lvalue; N Incrop]);
    (Lvalue, [T "$"; N Expr]); (Incrop, [T "++"]); (Incrop, [T "--"]);
    (Binop, [T "+"]); (Binop, [T "-"]); (Num, [T "0"]); (Num, [T "1"]);
    (Num, [T "2"]); (Num, [T "3"]); (Num, [T "4"]); (Num, [T "5"]);
    (Num, [T "6"]); (Num, [T "7"]); (Num, [T "8"]); (Num, [T "9"])])
val awksub_test0 : bool = true
val awksub_test1 : bool = true
val awksub_test2 : bool = true
val awksub_test3 : bool = true
type giant_nonterminals =
    Conversation
  | Sentence
  | Grunt
  | Snore
  | Shout
  | Quiet
val giant_grammar :
  giant_nonterminals *
  (giant_nonterminals * (giant_nonterminals, string) symbol list) list =
  (Conversation,
   [(Snore, [T "ZZZ"]); (Quiet, []); (Grunt, [T "khrgh"]);
    (Shout, [T "aooogah!"]); (Sentence, [N Quiet]); (Sentence, [N Grunt]);
    (Sentence, [N Shout]); (Conversation, [N Snore]);
    (Conversation, [N Sentence; T ","; N Conversation])])
val giant_test0 : bool = true
val giant_test1 : bool = true
val giant_test2 : bool = true
#
```

$Id: hw1.html,v 1.71 2016/09/22 22:49:49 eggert Exp $