# A TD-beam recognizer for MCFG

E. Stabler, Dec 2011 stabler@ucla.edu

These notes present OCaml implementations of top-down recognizers for MCFGs as described in [10]. I have tried to keep the code simple, uncluttered, intuitive. For readability, the types of non-trivial functions are explicitly indicated.

This document describes the 'development' version of the code for interactive use at the top-level, included in the `/dev` directory. The final compiled version is obtained simply by splitting out the grammar and the makeQ functor, and adding a top-level read-recognize loop.

## 1 Representing MCFG for parsing

MCFGs were defined by Seki & al. [9], where a category can have a tuple of string yields and a string function is associated with each rule. A prolog-like notation for these grammars [3, 2, 5] indicates the string functions with variables, making the rules easier for humans to read. A different notation, bottom-up analysis by Albro [1, 4], indicates the string function using the order of constituents and components on the right: the string components of the left-side is given ass a function of the components on the right hand side. For top-down analysis, using the rules from left to right, it is more convenient to specify the right hand side string components as a function of the left side: for each category on the right side, we can list its components numbered according to their position on the left side. This is easily illustrated with an example.

The string function for each rule is defined by specifying, for each rhs category $A$ with arity $n$, the position of each string component of that category with a pair $(k, i)$, indicating that this string component is the $i$'th element of the $k$'th string component of the lhs $A$. And after each rule, we write its probability with a floating point number.

(1)
```
S -->  AC(0,0)(0,2) BD(0,1)(0,3), 1.
AC -->  A(0,0) C(1,0) AC(0,1)(1,1), 0.5
AC -->  A(0,) C(1,), 0.5
BD -->  B(0,0) D(1,0) BD(0,1)(1,1), 0.5
BD -->  B(0,) D(1,), 0.5
A --> "a", 1.
B --> "b", 1.
C --> "c", 1.
D --> "d", 1.
```

For example, the index pairs of the first rule show that this is the rule that would be represented with variables in a prolog-like notation this way:

```
S(x00x01x02x03) :- AC(x00,x02), BD(x01,x03).
```

And the second rule corresponds to:

```
AC(x00x01,x10x11) :- A(x00), C(x01), AC(x01,x11).
```

When a pair $(k, i)$ provides the only element of the $k$'th component of the LHS (in every such case of course $i = 0$), we do not need to indicate its position (with a 0) in that component because there are no other components that need to be ordered with respect to it.

```
AC(x0,x1) :- A(x0), C(x1).
```

In cases like this, instead of pairs $(k, i)$ with $i = 0$ we let $i$ be empty. (For type consistency, this empty is represented with a negative int in the OCaml representation.)

Numbering the categories `S=0, AC=1, BD=2, A=3, B=4, C=5, D=6`, and using the "type constructor" `IN` for nonterminal rewrites, `IT` for terminal rewrites, and `IE` for empty productions, we give OCaml the grammar in this "internal" format:

(2)
```
let g1 = [
((0, IN [(1, [(0, 0); (0, 2)]); (2, [(0, 1); (0, 3)])]), 1.);
((1, IN [(3, [(0, 0)]); (5, [(1, 0)]); (1, [(0, 1); (1, 1)])]), 0.5);
((1, IN [(3, [(0, -1)]); (5, [(1, -1)])]), 0.5);
((2, IN [(4, [(0, 0)]); (6, [(1, 0)]); (2, [(0, 1); (1, 1)])]), 0.5);
((2, IN [(4, [(0, -1)]); (6, [(1, -1)])]), 0.5);
((3, IT "a"), 1.);
((4, IT "b"), 1.);
((5, IT "c"), 1.);
((6, IT "d"), 1.)
];;
```

Appendix A.2 defines a "pretty-printer" for grammars in this internal, numbered format. It generates the notation in (1) from the representation in (2). A translator from Albro's notation to this format (2) appears in Appendix A.3.

## 2 MCFG recognizer: tracking linear order

The key to recognizing MCFG languages is careful attention to the string functions associated with each rule. As explained in [10], inspired by the earlier work [6, 11], the relative linear order of each string component is explicitly specified here by an index. Each rule that breaks a component into subcomponents extends that index with 1 integer to indicate its relative position. Sorting the predictions according to these indices, the collection of predictions is not a 'stack' any more but a 'priority queue'. And the recognizer is otherwise a CF top down beam recognizer [8].

Priority queues created by the functor MakeQ (described in Appendix A.1) are actually used for two things. One type of queue called an ICatQueue holds a set of predicted categories, sorted by linear order as indicated by indices. And another type of queue, a DQueue, holds the set of partial derivations, sorted by decreasing probability. In effect, the DQueue is "smarter" than a backtrack stack, since, at each step, it serves up not an arbitrary derivation but a most probable derivation.

The recognizer code is presented in these basic chunks (numbered by page where chunk begins)

3a ⟨*mcfgtdb.ml* 3a⟩≡
  ⟨*mcfgtdb.ml file label* 3b⟩
  ⟨*mcfg cat and input* 3c⟩
  ⟨*mcf td types* 3d⟩
  ⟨*example grammar* 4a⟩
  ⟨*priority queues and more types* 4b⟩
  ⟨*mcfg expansions* 4c⟩
  ⟨*indexing by linear order* 5a⟩
  ⟨*pruning and trimming* 5b⟩
  ⟨*mcfg derivation step* 6a⟩
  ⟨*recursive derivation defined* 6b⟩
  ⟨*example calls to the recognizer* 6c⟩

3b ⟨*mcfgtdb.ml file label* 3b⟩≡
```
(*  file: mcfgtdb.ml
 creator: E Stabler, stabler@ucla.edu
 creation date: 2011-05-12
       updated: 2011-07-06
 purpose: top-down beam recognizer for MCFG
*)
```

The names of the types provides a preliminary indication of our intentions. This chunk also loads the queue functor code and the print functions (defined in Appendix A.2):

3c ⟨*mcfg cat and input* 3c⟩≡
```
type cat = int;;
type input = string list;;
```

3d ⟨*mcf td types* 3d⟩≡
```
type imap = (int * int) list;;
(* to specify string functions, an imap element (k,i) in a string component on the rhs of a rule
   indicates that the string is the i'th element of the k'th string component of the lhs *)
type nrhs = (cat * imap) list;;
type irhs = IT of string | IE | IN of nrhs;;
type iproduction = cat * irhs;;
type iprule = iproduction * float;;
type ing = iprule list;;          (* "internal" grammar *)
type expansion = irhs * float;;
type expansions = expansion list;;
type index = int list;;
type indices = index list;;
type iCat = cat * indices;;       (* the type of predicted categories *)
#use "makeQ.ml";;     (* load queue-making functor - required! *)
```

As the types indicate, grammars will be represented with integer categories, like this:

4a ⟨*example grammar* 4a⟩≡
```
#use "printMCFG.ml";; (* optional print functions *)

(*** begin EXAMPLE **)
let (g1Cat: int->string) = function
  | 0 -> "S" | 1 -> "AC" | 2 -> "BD" | 3 -> "A" | 4 -> "B" | 5 -> "C" | 6 -> "D"
  | _ -> failwith "error:g1Cat";;

let (g1:ing) = [
  ((0, IN [(1, [(0, 0); (0, 2)]); (2, [(0, 1); (0, 3)])]), 1.);
  ((1, IN [(3, [(0, 0)]); (5, [(1, 0)]); (1, [(0, 1); (1, 1)])]), 0.5);
  ((1, IN [(3, [(0, -1)]); (5, [(1, -1)])]), 0.5);
  ((2, IN [(4, [(0, 0)]); (6, [(1, 0)]); (2, [(0, 1); (1, 1)])]), 0.5);
  ((2, IN [(4, [(0, -1)]); (6, [(1, -1)])]), 0.5);
  ((3, IT "a"), 1.);
  ((4, IT "b"), 1.);
  ((5, IT "c"), 1.);
  ((6, IT "d"), 1.)
];;
(*** end EXAMPLE -- with example here, we can put print trace function into def of derive *)
```
We can now define the priority queues we need for the recognizer:

4b ⟨*priority queues and more types* 4b⟩≡
```
let least = let rec least0 sofar = function [] -> sofar | x::xs -> least0 (min sofar x) xs
  in function [] -> failwith "error:least" | x::xs -> least0 x xs;;

module ICatPoset = (* order by comparing least index, an int list *)
  struct
    type t = iCat
    let le = fun ((_,i1):iCat) ((_,i2):iCat) -> compare (least i1) (least i2) < 1
  end;;

module IQ = MakeQ (ICatPoset);;    (* for the queues of predictions *)

type der = input * IQ.t * float;;  (* the type of partial derivations *)

module DerivationPoset = (* order by (decreasing) probability, a float *)
  struct
    type t = der
    let le = fun ((_,_,p1):der) ((_,_,p2):der) -> compare p2 p1 < 1
  end;;

module DQ = MakeQ (DerivationPoset);;  (* for the queue of (partial) derivations *)

#use "printDQ.ml";; (* optional print functions *)
```
With MCFGs in the format described in §1 on page 2 above, it is easy to collect the possible expansions of any category:

4c ⟨*mcfg expansions* 4c⟩≡
```
(* we assume categories numbered sequentially up from 0 *)
let rec numberOfCats = function
  | [] -> 0
  | ((n0, _),_)::more -> max n0 (numberOfCats more);;

(* get expansions of cat from mcfg rules *)
let exps :  ing -> cat -> expansions = fun g cat ->
  List.fold_left (fun l ((c, rhs), p) -> if c=cat then (rhs, p)::l else l) [] g;;
```

So for example, the singleton list of expansions of the start category 0:

```
# exps g1 0;;
- : expansions = [(IN [(1, [(0, 0); (0, 2)]); (2, [(0, 1); (0, 3)])], 1.)]
```

Now the indices that specify linear position are defined as in [10]:

5a ⟨*indexing by linear order* 5a⟩≡

```
  (* OK, now the indexing for linear order -- as in Stabler'11 *)
  let rec extendIndices : indices -> imap -> indices =  fun i0 -> function
    | [] -> []
    | (k,x)::more when x<0 -> (List.nth i0 k)::extendIndices i0 more
    | (k,x)::more -> ((List.nth i0 k)@[x])::extendIndices i0 more;;

  let rec insertPredictions : indices -> IQ.t -> nrhs -> IQ.t = fun i0 iq -> function
    | [] -> iq
    | (cat,ill)::more ->
      insertPredictions i0 (IQ.add (cat, extendIndices i0 ill) iq) more;;
```

And now the pruning and trimming functions, as described in [10]:

5b ⟨*pruning and trimming* 5b⟩≡

```
  (* pruning and trimming functions *)
  exception InitialIndexDiffers;;
  let rec trimIndices: int -> indices -> indices = fun i -> function
    | (x::xs)::more -> if x=i then xs::trimIndices i more else raise InitialIndexDiffers
    | []::_ -> raise InitialIndexDiffers
    | [] -> [];;

  let rec trimmedElements : int -> IQ.t -> iCat list = fun i iq0 ->
    if IQ.is_empty iq0
    then []
    else let ((c,ixs),iq) = IQ.extract_min iq0 in (c,trimIndices i ixs)::trimmedElements i iq;;

  (* we might want to turn this off (i.e. remove call from prunedInsert)..
     certainly the pruning slows performance in many cases *)
  let trimIQ : IQ.t -> IQ.t = fun iq0 ->
    if IQ.is_empty iq0
    then iq0
    else let ((c,ixs),iq1) = IQ.extract_min iq0 in
           if ixs=[]
           then iq0
           else let ix = List.hd ixs in
                  if ix = []
                  then iq0
                  else let i = List.hd ix in (* trim i from every index, else no change on error *)
                         try List.fold_right IQ.add (trimmedElements i iq0) IQ.empty
                         with InitialIndexDiffers -> iq0;;

  let prunedInsert : DQ.t -> float -> der -> DQ.t = (* prune derivation if newP not above bound *)
     fun dq pb (s,iq,newP) ->
       if newP > pb then DQ.add (s,trimIQ iq,newP) dq else dq;;
```

Now the core recognizer functions:

6a ⟨*mcfg derivation step* 6a⟩≡

```
(* recognizer core function *)
let extendDerivation : DQ.t -> float -> indices -> input -> IQ.t -> float -> expansion -> DQ.t =
  fun dq pb i0 input iq p0 -> function
    | IT s, p -> (match input with
        | head::tail -> if s=head then prunedInsert dq pb (tail, iq, p0 *. p) else dq     (* scan *)
        | [] -> dq)
    | IE, p -> prunedInsert dq pb (input, iq, p0 *. p)                          (* scan empty element *)
    | IN nrhs, p -> prunedInsert dq pb (input, insertPredictions i0 iq nrhs, p0 *. p);; (* expand *)
```

6b ⟨*recursive derivation defined* 6b⟩≡

```
let rec derive : ing -> float -> DQ.t -> bool = fun g pb dq0 ->
  if (DQ.is_empty dq0)
  then false
  else
    let ((in0,iq0,p0),dq1) = DQ.extract_min dq0 in
      Printf.fprintf stdout "--\n"; printDQ stdout g1Cat dq0; (* uncomment for trace! *)
      if IQ.is_empty iq0 && in0=[] then true
      else if IQ.is_empty iq0
      then derive g pb dq1
      else
        let ((cat,i0),iq1) = IQ.extract_min iq0 in
          derive g pb (List.fold_left
                         (fun dq x -> extendDerivation dq pb i0 in0 iq1 p0 x)
                         dq1
                         (exps g cat));;

(* create the initial derivation queue and parse --
  minbound is for our first simple pruning rule: analyses with p < this bound are discarded *)
let recognize g minbound input =
  let init  = IQ.add (0,[[]]) IQ.empty in
  let prob = 1. in
    derive g minbound (DQ.add (input,init,prob) DQ.empty);;

(* warning! since this is a TD recognizer, unboundedRecognize can fail to terminate
   if there is no parse and the grammar is left recursive! *)
let unboundedRecognize g input = recognize g (-1.) input;;
```

Some examples:

6c ⟨*example calls to the recognizer* 6c⟩≡

```
(* examples
OK:
  recognize g1 0.0000001 ["a";"b";"c";"d"];;
  unboundedRecognize g1 ["a";"b";"c";"d"];;
NO:
  recognize g1 0.5 ["a";"b";"c";"d"];;
OK: this example is used in Stabler'11
  recognize g1 0.0000001 ["a";"b";"b";"c";"d";"d"];;
NO:
  recognize g1 0.0000001 ["a";"a";"b";"c";"d"];;
*)
```

That's the whole thing!

A session looks like this, evaluating each example listed in the final comment above, at the end of `mcfgtdb.ml`. Note in particular the example grammar (line 66) and its expansions array (line 73). (And compare the pretty-printed version of the grammar in Appendix A.2 on page 12.) The first recognition problem is given to OCaml in line 93, and then the recognizer state is given by listing the probability, the remaining input, and queue of predictions for each analysis, followed by `--` and then the next recognizer state.

```
            Objective Caml version 3.12.1

    # #use "mcfgtdb-dev.ml";;
 5
    # recognize g1 0.0000001 ["a";"b";"c";"d"];;
    --
    1.000000:abcd:S()
    --
10  1.000000:abcd:AC(0)(2) BD(1)(3)
    --
    0.500000:abcd:A(00) AC(01)(21) BD(1)(3) C(20)
    0.500000:abcd:A(0) BD(1)(3) C(2)
    --
15  0.500000:bcd:AC(01)(21) BD(1)(3) C(20)
    0.500000:abcd:A(0) BD(1)(3) C(2)
    --
    0.500000:abcd:A(0) BD(1)(3) C(2)
    0.250000:bcd:A(01) BD(1)(3) C(20) C(21)
20  0.250000:bcd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
    --
    0.500000:bcd:BD(1)(3) C(2)
    0.250000:bcd:A(01) BD(1)(3) C(20) C(21)
    0.250000:bcd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
25  --
    0.250000:bcd:B(10) BD(11)(31) C(2) D(30)
    0.250000:bcd:B(1) C(2) D(3)
    0.250000:bcd:A(01) BD(1)(3) C(20) C(21)
    0.250000:bcd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
30  --
    0.250000:cd:BD(11)(31) C(2) D(30)
    0.250000:bcd:B(1) C(2) D(3)
    0.250000:bcd:A(01) BD(1)(3) C(20) C(21)
    0.250000:bcd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
35  --
    0.250000:bcd:B(1) C(2) D(3)
    0.250000:bcd:A(01) BD(1)(3) C(20) C(21)
    0.250000:bcd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
    0.125000:cd:B(110) BD(111)(311) C(2) D(30) D(310)
40  0.125000:cd:B(11) C(2) D(30) D(31)
    --
    0.250000:cd:C(2) D(3)
    0.250000:bcd:A(01) BD(1)(3) C(20) C(21)
    0.250000:bcd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
45  0.125000:cd:B(110) BD(111)(311) C(2) D(30) D(310)
    0.125000:cd:B(11) C(2) D(30) D(31)
    --
    0.250000:d:D()
    0.250000:bcd:A(01) BD(1)(3) C(20) C(21)
50  0.250000:bcd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
    0.125000:cd:B(110) BD(111)(311) C(2) D(30) D(310)
    0.125000:cd:B(11) C(2) D(30) D(31)
    --
    0.250000::
55  0.250000:bcd:A(01) BD(1)(3) C(20) C(21)
    0.250000:bcd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
```

```
0.125000:cd:B(110) BD(111)(311) C(2) D(30) D(310)
0.125000:cd:B(11) C(2) D(30) D(31)
- : bool = true
# recognize g1 0.5 ["a";"b";"c";"d"];;
--
1.000000:abcd:S()
--
1.000000:abcd:AC(0)(2) BD(1)(3)
- : bool = false
# recognize g1 0.0000001 ["a";"b";"b";"c";"d";"d"];;
--
1.000000:abbcdd:S()
--
1.000000:abbcdd:AC(0)(2) BD(1)(3)
--
0.500000:abbcdd:A(00) AC(01)(21) BD(1)(3) C(20)
0.500000:abbcdd:A(0) BD(1)(3) C(2)
--
0.500000:bbcdd:AC(01)(21) BD(1)(3) C(20)
0.500000:abbcdd:A(0) BD(1)(3) C(2)
--
0.500000:abbcdd:A(0) BD(1)(3) C(2)
0.250000:bbcdd:A(01) BD(1)(3) C(20) C(21)
0.250000:bbcdd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
--
0.500000:bbcdd:BD(1)(3) C(2)
0.250000:bbcdd:A(01) BD(1)(3) C(20) C(21)
0.250000:bbcdd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
--
0.250000:bbcdd:B(10) BD(11)(31) C(2) D(30)
0.250000:bbcdd:B(1) C(2) D(3)
0.250000:bbcdd:A(01) BD(1)(3) C(20) C(21)
0.250000:bbcdd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
--
0.250000:bcdd:BD(11)(31) C(2) D(30)
0.250000:bbcdd:B(1) C(2) D(3)
0.250000:bbcdd:A(01) BD(1)(3) C(20) C(21)
0.250000:bbcdd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
--
0.250000:bbcdd:B(1) C(2) D(3)
0.250000:bbcdd:A(01) BD(1)(3) C(20) C(21)
0.250000:bbcdd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
0.125000:bcdd:B(110) BD(111)(311) C(2) D(30) D(310)
0.125000:bcdd:B(11) C(2) D(30) D(31)
--
0.250000:bcdd:C(2) D(3)
0.250000:bbcdd:A(01) BD(1)(3) C(20) C(21)
0.250000:bbcdd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
0.125000:bcdd:B(110) BD(111)(311) C(2) D(30) D(310)
0.125000:bcdd:B(11) C(2) D(30) D(31)
--
0.250000:bbcdd:A(01) BD(1)(3) C(20) C(21)
0.250000:bbcdd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
0.125000:bcdd:B(110) BD(111)(311) C(2) D(30) D(310)
0.125000:bcdd:B(11) C(2) D(30) D(31)
--
0.250000:bbcdd:A(010) AC(011)(211) BD(1)(3) C(20) C(210)
0.125000:bcdd:B(110) BD(111)(311) C(2) D(30) D(310)
0.125000:bcdd:B(11) C(2) D(30) D(31)
--
0.125000:bcdd:B(110) BD(111)(311) C(2) D(30) D(310)
0.125000:bcdd:B(11) C(2) D(30) D(31)
```

```
--
0.125000:cdd:BD(111)(311) C(2) D(30) D(310)
0.125000:bcdd:B(11) C(2) D(30) D(31)
--
0.125000:bcdd:B(11) C(2) D(30) D(31)
0.062500:cdd:B(111) C(2) D(30) D(310) D(311)
0.062500:cdd:B(1110) BD(1111)(3111) C(2) D(30) D(310) D(3110)
--
0.125000:cdd:C(2) D(30) D(31)
0.062500:cdd:B(111) C(2) D(30) D(310) D(311)
0.062500:cdd:B(1110) BD(1111)(3111) C(2) D(30) D(310) D(3110)
--
0.125000:dd:D(0) D(1)
0.062500:cdd:B(111) C(2) D(30) D(310) D(311)
0.062500:cdd:B(1110) BD(1111)(3111) C(2) D(30) D(310) D(3110)
--
0.125000:d:D()
0.062500:cdd:B(111) C(2) D(30) D(310) D(311)
0.062500:cdd:B(1110) BD(1111)(3111) C(2) D(30) D(310) D(3110)
--
0.125000::
0.062500:cdd:B(111) C(2) D(30) D(310) D(311)
0.062500:cdd:B(1110) BD(1111)(3111) C(2) D(30) D(310) D(3110)
- : bool = true
#
```

## References

[1] Albro, D. M. An Earley-style parser for multiple context free grammars. UCLA, http://www.linguistics.ucla.edu/people/grads/albro/earley.pdf. Code: http://www.linguistics.ucla.edu/stabler/coding.html, 2002.

[2] Boullier, P. Proposal for a natural language processing syntactic backbone. Tech. Rep. 3242, Projet Atoll, INRIA, Rocquencourt, 1998.

[3] Groenink, A. Literal movement grammars. In *Proceedings of the 7th Meeting of the European Association for Computational Linguistics* (1995), pp. 90–97.

[4] Guillaumin, M. Conversions between mildly sensitive grammars. UCLA and École Normale Supérieure. http://www.linguistics. ucla.edu/people/stabler/epssw.htm, 2004.

[5] Kanazawa, M., and Salvati, S. Generating control languages with abstract categorial grammars. In *Proceedings of the 12th Conference on Formal Grammar (FG'07)* (Stanford, California, 2007), L. Kallmeyer, P. Monachesi, G. Penn, and G. Satta, Eds., CLSI Publications.

[6] Mainguy, T. A probabilistic top-down parser for minimalist grammars. http://arxiv.org/abs/1010.1826v1, 2010.

[7] Okasaki, C. *Purely Functional Data Structures.* Cambridge University Press, NY, 1999.

[8] Roark, B. Probabilistic top-down parsing and language modeling. *Computational Linguistics 27*, 2 (2001), 249–276.

[9] Seki, H., Matsumura, T., Fujii, M., and Kasami, T. On multiple context-free grammars. *Theoretical Computer Science 88* (1991), 191–229.

[10] Stabler, E. P. Top-down recognizers for MCFGs and MGs. In *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics (CMCL), 49th Annual Meeting of the Association for Computational Linguistics* (2011), F. Keller and D. Reitter, Eds.

[11] Villemonte de la Clergerie, E. Parsing MCS languages with thread automata. In *Proceedings, 6th International Workshop on Tree Adjoining Grammars and Related Frameworks, TAG+6* (2002).

## A Queue, print, and notation functions

### A.1 makeQ.ml

Filliâtre provides this straightforward implementation of Okasaki's [7] 'leftist heap':

10 ⟨*makeQ.ml* 10⟩≡
```
(*  file: makeQ.ml
 creator: Jean-Christophe.Filliatre@lri.fr
   source: https://groups.google.com/group/fa.caml/msg/526cc7a4a9adc664?dmode=source
    date: 30 Jun 2011 20:14:34 +0200
*)

module MakeQ (X : sig type t val le : t -> t -> bool end) :
sig
  type t
  val empty : t
  val is_empty : t -> bool
  val add : X.t -> t -> t
  exception Empty
  val extract_min : t -> X.t * t
  val merge : t -> t -> t
end
  =  struct

    type t = E | T of int * X.t * t * t

    exception Empty

    let rank = function E -> 0 | T (r,_,_,_) -> r

    let make x a b =
    let ra = rank a and rb = rank b in
      if ra >= rb then T (rb + 1, x, a, b) else T (ra + 1, x, b, a)

  let empty = E

  let is_empty = function E -> true | T _ -> false

  let rec merge h1 h2 = match h1,h2 with
    | E, h | h, E ->
      h
    | T (_,x,a1,b1), T (_,y,a2,b2) ->
      if X.le x y then make x a1 (merge b1 h2) else make y a2 (merge h1 b2)

  let add x h = merge (T (1, x, E, E)) h

  let extract_min = function
    | E -> raise Empty
    | T (_,x,a,b) -> x, merge a b
end;;
```

## A.2 print functions

We define a pretty-printer that makes our grammars more readable, as in this session:

```
# g1;;
- : ing =
[((0, IN [(1, [(0, 0); (0, 2)]); (2, [(0, 1); (0, 3)])]), 1.);
 ((1, IN [(3, [(0, 0)]); (5, [(1, 0)]); (1, [(0, 1); (1, 1)])]), 0.5);
 ((1, IN [(3, [(0, -1)]); (5, [(1, -1)])]), 0.5);
 ((2, IN [(4, [(0, 0)]); (6, [(1, 0)]); (2, [(0, 1); (1, 1)])]), 0.5);
 ((2, IN [(4, [(0, -1)]); (6, [(1, -1)])]), 0.5); ((3, IT "a"), 1.);
 ((4, IT "b"), 1.); ((5, IT "c"), 1.); ((6, IT "d"), 1.)]

# printING g1Cat g1;;
1.000000: S -->  AC(0,0)(0,2) BD(0,1)(0,3)
0.500000: AC -->  A(0,0) C(1,0) AC(0,1)(1,1)
0.500000: AC -->  A(0,) C(1,)
0.500000: BD -->  B(0,0) D(1,0) BD(0,1)(1,1)
0.500000: BD -->  B(0,) D(1,)
1.000000: A --> "a"
1.000000: B --> "b"
1.000000: C --> "c"
1.000000: D --> "d"
- : unit = ()
```

`printING` and associated functions are defined here:

12 ⟨*printMCFG.ml* 12⟩≡
```
  (*  file: printMCFG.ml
   creator: E Stabler, stabler@ucla.edu
   updated: 2011-07-06
  *)
  let printIlist out = List.iter (Printf.fprintf out "%i");;
  let printImap out = List.iter (fun (k,i) ->
      if i<0 then Printf.fprintf out "(%i,)" k else Printf.fprintf out "(%i,%i)" k i);;
  let printInrhs out f =  List.iter (fun (cat,imap) ->
      (Printf.fprintf out " %s" (f cat);printImap out imap));;
  let printIRHS out f = function
    | IE -> Printf.fprintf out "\"\""
    | IT s -> Printf.fprintf out "\"%s\"" s
    | IN nrhs -> printInrhs out f nrhs;;
  let printIPRule out f ((lhs,irhs),p) =
      Printf.fprintf out "%f: %s --> " p (f lhs);
      printIRHS out f irhs;;
  let printING f (g:ing) =
    List.iter (fun x -> (printIPRule stdout f x; Printf.fprintf stdout "\n")) g;;
```

The function `printDQ` is useful for tracing the recognizer steps, as we saw already in the session listing in §2.

13a ⟨*printDQ.ml* 13a⟩≡
```
(*  file: printDQ.ml
 creator: E Stabler, stabler@ucla.edu
 updated: 2011-07-06
*)

let ilist out = fun x ->
        Printf.fprintf out "(";
        List.iter (Printf.fprintf out "%i") x;
        Printf.fprintf out ")";;

let rec printIQ out f = fun iq0 ->
  if IQ.is_empty iq0 then ()
  else let ((c,indices),iq1) = IQ.extract_min iq0 in
         Printf.fprintf out "%s" (f c);
         List.iter (ilist out) indices; Printf.fprintf out " ";
         printIQ out f iq1;;

let rec printDQ out f = fun dq0 ->
  if DQ.is_empty dq0 then ()
  else let ((input,iq,p),dq1) = DQ.extract_min dq0 in
         Printf.fprintf out "%f" p; Printf.fprintf out ":";
         List.iter (Printf.fprintf out "%s") input; Printf.fprintf out ":";
         printIQ out f iq; Printf.fprintf out "\n";
         printDQ out f dq1;;
```

13b ⟨*printTree.ml* 13b⟩≡
```
(*  file: printTree.ml
 creator: E Stabler, stabler@ucla.edu
 creation date: 2011-05-12
   purpose: print utilities for display or tracing
   type tree = T of string * tree list;;
*)

(* pretty-print a tree *)
let rec tab out n = if n <= 0 then () else (Printf.fprintf out " "; tab out (n-1););;

let pptreeOut out tree =
  let rec pptreeOutN out n = function T(s,ts) ->
    (tab out n; Printf.fprintf out "%s\n" s; List.iter (pptreeOutN out (n+4)) ts;) in
    ( Printf.fprintf out "\n"; pptreeOutN out 0 tree; Printf.fprintf out "\n";);;

let pptree tree = pptreeOut stdout tree;;

(* pretty-print a tree as a readable OCaml term *)
let pptermOut out tree =
  let rec pptermOutN out n = function T(s,ts) ->
    ( Printf.fprintf out "\n"; tab out n; Printf.fprintf out "T(\"%s\",[" s;
      List.iter (pptermOutN out (n+4)) ts;
      Printf.fprintf out "]);";
    ) in
    pptermOutN out 0 tree;;

let ppterm tree = ( pptermOut stdout tree; Printf.fprintf stdout ";\n\n"; );;
```

## A.3 translation from Albro MCFG notation

As mentioned in §1, Albro [1] introduced a different notation for MCFGs, one that is also used by Guillaumin [4], based on numbering the categories on the right hand side of each rule and then the arguments of each category. The code presented here translates an Albro-like notation into the notation of §1.

Categories are represented by integers, numbered consecutively from 0, with 0 the start. We have 3 kinds of MCFG rules.

1. A terminal rewrite rule with probability 1. like: `S --> "hi"`, is represented by `((0,T "hi"),1.)` where `0=S` and `T` is the "type constructor" for terminal rewrites.
2. An empty rewrite with probability 1. like: `S --> []`, is represented by `((0,E),1.)`.
3. A nonterminal rewrite like `S --> NP VP,1.  becomes ((0,N [1;2],[0,0;0,1]),1.)` where `0=S 1=NP 2=VP` and `N` is the "type constructor" for nonterminal rewrite rules, and where `[0,0;0,1]` specifies the string map as in the Albro, Guillaumin recognizers.

A list of such rules is converted for the recognizer, changing the string map representation into a form that is easier for the top-down recognizer.

Given a cat from rhs identified by its ordinal position as it is in `<map>`, and given a map `iill`, we collect all the tuples `(j,k,p)` where

| | | |
|---|---|---|
| `j` | = | the number of the component `(cat,j)` found |
| `k` | = | the number of the element of `iill` in which it was found |
| `p` | = | the position of `(i,j)` in the `k`'th elementof iill |

14 ⟨*albro2beam.ml* 14⟩≡
```
(*  file: albro2beam.ml
 creator: E Stabler, stabler@ucla.edu
 updated: 2011-07-06
 purpose: convert grammar with Albro-style map function into our format.
*)
(* types for the external grammar notation, int version *)
type rhs = T of string | E | N of cat list * (int*int) list list;;
type production = cat * rhs;;
type prule = production * float;;
type exg = prule list;;

(*** begin EXAMPLE **)
let (g1Cat: int->string) = function (* maps g1 categories to strings, for printing *)
  | 0 -> "S" | 1 -> "AC" | 2 -> "BD" | 3 -> "A" | 4 -> "B" | 5 -> "C" | 6 -> "D"
  | _ -> failwith "error:g1Cat";;

let (xg1:exg) = [
  (0, E), 0.5;
  (0, N ([1; 2], [[0,0;1,0;0,1;1,1]])), 0.5;
  (1, N ([3; 5; 1], [[0,0;2,0];[1,0;2,1]])), 0.5;
  (1, N ([3; 5], [[0,0];[1,0]])), 0.5;
  (2, N ([4; 6; 2], [[0,0;2,0];[1,0;2,1]])), 0.5;
  (2, N ([4; 6], [[0,0];[1,0]])), 0.5;
  (3, T "a"), 1.;
  (4, T "b"), 1.;
  (5, T "c"), 1.;
  (6, T "d"), 1.;
];;
(*** end EXAMPLE -- with example here, we can put print trace function into def of derive *)

let rec jkp cat k p = function
  | ((c,j)::[])::more when cat=c && p=0 -> (j,k,-1)::jkp cat k (p+1) more
  | ((c,j)::cis)::more when cat=c -> (j,k,p)::jkp cat k (p+1) (cis::more)
  | (_::cis)::more -> jkp cat k (p+1) (cis::more)
  | []::more -> jkp cat (k+1) 0 more
  | [] -> [];;

(* example: jkp 2 0 0 [[0,0;2,0];[1,0;2,1]];; *)

(* sort the jkp list to get the components of cat in consecutive order,
   and then delete the component numbers j *)
let kp cat iill =
  List.map (fun (_,x,y) -> (x,y)) (List.sort compare (jkp cat 0 0 iill));;

(* examples, contrast:  kp 2 [[0,0;2,0];[1,0;2,1]];;
                        kp 2 [[0,0;2,1];[1,0;2,0]];; *)

let rec nrhs2in i iill = function
  | [] -> []
  | cat::cats -> (cat,(kp i iill))::nrhs2in (i+1) iill cats;;

let rhs2in = function
  | T s -> IT s
  | E -> IE
  | N (cats,iill) -> IN (nrhs2in 0 iill cats);;

let prule2in ((lhs,rhs),p) = match rhs with
  | T s -> ((lhs,IT s),p)
  | E -> ((lhs,IE),p)
  | N (cats,iill) -> ((lhs,IN (nrhs2in 0 iill cats)),p);;

(* prule2in ((2, N ([4; 6; 2], [[0,0;2,0];[1,0;2,1]])), 0.5);; *)

let g2in = List.map prule2in;;

let printIIe out (i1,i2) = Printf.fprintf out "%i,%i;" i1 i2;;
let printL elementPrinter out list =
  Printf.fprintf out "[";
  List.iter (elementPrinter out) list;
  Printf.fprintf out "]";;
let printIILe out = printL printIIe out;;
let printIILL out = printL printIILe out;;
let printRHS out f = function
  | E -> Printf.fprintf out "\"\""
  | T s -> Printf.fprintf out "\"%s\"" s
  | N (cats,iill) -> List.iter (fun x -> Printf.fprintf out "%s " (f x)) cats; printIILL out iill;;
let printPRule out f ((lhs,rhs),p) =
    Printf.fprintf out "%f: %s --> " p (f lhs);
    printRHS out f rhs;;
let printEXG f (g:exg) =
  List.iter (fun x -> (printPRule stdout f x; Printf.fprintf stdout "\n")) g;;

(*
printEXG g1Cat xg1;;
printING g1Cat (g2in xg1);;
*)
```