Simple Earley Parsing

Tom Ridge

2017-11-24

1 Introduction

This is a version of Earley's algorithm which was essentially derived from scratch, paying attention to correctness and performance issues.

This code is the among the fastest implementations of Earley in a high-level language that I am aware of. For example, for the grammar

```
E -> E E E | "1" | eps
```

with an input "111..." of length 400, the parse takes about 3s on a single core on current commodity Intel hardware. By way of comparison, Haskell's Happy parser takes two minutes to process an input of size 60, and cannot handle much longer lengths at all (and I doubt many other "general" parser implementations can either).

The problem with these other implementations is that they seem to be buggy as far as performance goes, and these bugs tend to manifest when using "horrible" grammars like the above, and long inputs.

This code is meant to be a reference implementation for Earley-like parsing, thereby providing a base line against which to measure performance of other algorithms. It is also intended to guide implementations of Earley in other languages: just copy the code below.

1.1 About this document

The documentation is in the form of an asciidoc file, which is turned into an html file.

We include source code in the documentation, and these are produced by extracting code between "special comments" in the source code file. The special comments are of the form :ab: i.e., two letters between two colons. The code snippet is then placed in a file snips/xxx/ab-bc (where :bc: is the label following the label :ab:, and xxx is the original filename). These snippets are then included in this documentation file.

2 Quick introduction to this variant of Earley's algorithm

Earley's algorithm is a general parsing algorithm for all context free grammars. We assume that the reader understands the words "nonterminal", "terminal" and "symbol". A grammar is a list of rules, where each rule is of the form $(X \to \alpha)$, where X is a nonterminal and α is a list of symbols.

Traditional Earley works with "items" of the form $(X \to \alpha.\beta, j)$. The algorithm works in stages. At stage k the algorithm is looking at the input string at position k. The existence of an item $(X \to \alpha.\beta, j)$ at stage k means that, starting from input position j, using rule $(X \to \alpha\beta)$, it was possible to parse the symbols α to match the input between position j and k.

From now on we will use verbatim rather than LaTeX math.

Our version of the algorithm works with items of the form `X -> i,as,k,bs`. This means the same as above: starting from position `i` it was possible to parse the list of symbols `as` by consuming the input upto position `k`. Compared to the traditional presentation, we have placed `i` before `as`, and made `k` explicit and placed it after `as`. This emphasizes that `as` matched the input between `i` and `k`.

We assume the reader is familiar with the functioning of Earley's original algorithm.

This version is similar. Compared to Earley, we have paid close attention to the datastructures and representations. We have also made a few optimizations that perhaps have not been noticed before. We operate in stages, with the current stage denoted `k`.

2.1 Notational conventions

We make use of notational conventions. Usually the following is true:

- `i,j,k` are ints; `i` is the start position for an item; `k` is the current stage
- `X,Y` are nonterminals
- `T` is a terminal
- `s` is a symbol

2.2 Earley items

Traditional Earley works with a single notion of item. We extend this to the following:

- `X -> i,as,k,bs` - the "traditional" item; we refer to this as a "nonterminal" item, or just "item"

- `i,X,k` - a complete (nonterminal) item; arises from nonterminal items of the form `X -> i,as,k,[]`, by omitting the irrelevant `as` and `[]` components. The meaning is: the nonterminal `X` matches the input between `i` and `k`.
- `k,T,j` - a complete (terminal) item; arises from matching a terminal against the input from position `k` to `j`.
- `X -> i,as,k,S bs` - a "traditional" item, but where the final component is not empty and starts with symbol `S`; we refer to this as a "blocked" item, because progress depends on parsing the symbol `S` against the input starting from position `k` (we sometimes say the item is blocked on `k,S`); if the parse of `S` is successful, we get a complete item `k,S,j` which we can cut against the original item to get a new item of the form `X -> i,as S,j,bs`.

2.3 The essential parsing step

At the heart of all general parsing algorithms is the following step.

Suppose we have an item `X -> i,as,k,S bs` (here, S is a symbol). This means that the sequence `as` matched the input from position `i` to `k`. Suppose we also have an item `S -> k,cs,j,[]` (i.e., the symbol `S` matched the input from position `k` to `j`). Then we are justified in concluding that `X -> i,as S,j,bs` (i.e., the sequence `as S` matches the input from position `i` to `j`). This rule can be expressed as follows:

```
X -> i,as,k,S bs     S -> k,...,j,[]
---------------------------------- Cut
X -> i,as S,j,bs
```

This should be read as: given the two things above the line, we can conclude with the thing below the line, and the reasoning step is called "Cut".

In the terms of the last section, we take a "blocked" item `X -> i,as,k,S bs` and a complete item `k,S,j` and cut them to give an item `X -> i,as S,j,bs`.

2.4 From the "essential step" to Earley's algorithm

Roughly the idea is as follows: from an initial nonterminal `X`, keep adding more and more items until you end up with a complete item `(0,X,l)`, where `l` is the length of the input. This is a straightforward "stupid" algorithm. With appropriate datastructures, this is $O(n^3)$.

Earley adds a single optimization: process the input character by character. At stage `k`, process all items of the form `X -> i,as,k,bs` (and similar) before moving on to stage `k+1`. It seems plausible that in practice this will run faster than the "stupid" algorithm above.

To arrive at the code below, I took these ideas and then examined the steps that were needed in minute detail, taking care to retain good performance and (hopefully) correctness.

3 Code

3.1 Preliminaries

We define some basic library types and functions. Note that for sets and maps we use records rather than (stdlib) functors. This is to allow us to choose the implementation at runtime: if the input size is small we can use arrays; otherwise we can fall back to hashmaps or maps.

```
module List_ = struct

  let fold_left_ ~step ~init_state xs =
    List.fold_left
      (fun a b -> step ~state:a b)
      init_state
      xs

  let with_each_elt = fold_left_
end



module Set_ops = struct
  type ('e,'t) set_ops = {
    add: 'e -> 't -> 't;
    mem: 'e -> 't -> bool;
    empty: 't;
    is_empty: 't -> bool;
    elements: 't -> 'e list;
  }
end



module Map_ops = struct
  type ('k,'v,'t) map_ops = {
    map_add: 'k -> 'v -> 't -> 't;
    map_find:'k -> 't -> 'v;
    map_empty:'t;
    map_remove:'k -> 't -> 't;
  }
end



open Set_ops


open Map_ops
```

Note that the type `('e,'t) set_ops` involves a type of elements `'e`, and a type `'t` for the set itself. Similarly, the type `('k,'v,'t) map_ops` is a map from keys `'k` to values `'v`, implemented by type `'t`.

3.2 Input signature

The code is parameterized over the signature `S_`, which defines all the types and operations we assume. We use a signature so that we can instantiate the generic code with various different implementations. For example, the test code uses integers to represent items.

```
module type S_ = sig
```

First we declare the types for nonterminals, terminals and symbols.

```
  type i_t = int
  type k_t = int
  type j_t = int
  type nt
  type tm
  type sym
  val sym_case: nt:(nt -> 'a) -> tm:(tm -> 'a) -> sym -> 'a
  val _NT: nt -> sym
```

Next we give the type for (nonterminal) items, together with associated operations. An item is like a record, of the form `{nt;i;as;k;bs}`, indicating that the sequence of symbols `as` could be parsed between `i` and `k`; this corresponds to an attempt to parse the production `nt -> as bs` starting at position `i`. Note that we keep the type abstract, and values of type `nt_item_ops` provide the usual record projection functions.

```
  type nt_item
  type nt_item_ops = {
    dot_nt: nt_item -> nt;
    dot_i: nt_item -> i_t;
    dot_k: nt_item -> k_t;
    dot_bs: nt_item -> sym list;
  }
```

Next we have various types for

- sets of items
- triples `(i,X,k)` of a nonterminal `X` and two integers `i,k`, which corresponds to a parsed item of the form `X -> ...` between `i` and `k` in the input (a "complete" nonterminal item); note that because `k` is the "current stage", it suffices to record only `(i,X)` rather than `(i,X,k)`.
- maps from nonterminals, integers, and terminals

```
  type nt_item_set
  val nt_item_set_ops: (nt_item,nt_item_set) set_ops
  val nt_item_set_with_each_elt:
    f:(state:'a -> nt_item -> 'a) -> init_state:'a -> nt_item_set -> 'a

  type ixk = (i_t * nt)  (* i X k *)
  type ixk_set
  val ixk_set_ops: (ixk,ixk_set) set_ops

  type map_nt
  val map_nt_ops: (nt,nt_item_set,map_nt) map_ops

  type map_int
  val map_int_ops : (int,nt_item_set,map_int) map_ops

  type map_tm
  val map_tm_ops : (tm,int list option,map_tm) map_ops
```

We need to record the blocked items less than `k` (where `k` is the "current" position in the input). This is a map from an integer `i<k`, to a map from a nonterminal to a set of items, type `bitms_lt_k`.

```
  type bitms_lt_k  (* int -> nt -> nt_item_set; implement by array *)
  type bitms_lt_k_ops = (int,map_nt,bitms_lt_k) map_ops
  (* passed in at run time val bitms_lt_k_ops: (int,map_nt,bitms_lt_k) map_ops *
```

We also need to record the items that we need to process at stage `k'>k`. This is a map from `k'>k` to a set of items.

```
  type todo_gt_k
  val todo_gt_k_ops: (int,nt_item_set,todo_gt_k) map_ops
```

Finally we have the operation `cut`, which takes an item `{nt;i;as;k;bs}` where `bs` is of the form `X::bs'`, and an int `j` which indicates that symbol `X` could be parsed between `k` and `j`, and produces the new item `{nt;i;as';j;bs'}`, where `as'` is `as` with the parsed symbol `X` appended.

```
  type cut = nt_item -> j_t -> nt_item

  (* FIXME why are these here? *)
  val debug_enabled : bool
  val debug_endline : string -> unit
end
```

3.3 Functor declaration

The code is parameterized by the signature `S_` and defined within a functor.

```
module Make = functor (S:S_) -> struct
```

3.4 State type

The algorithm executes in stages. At each stage `k`, the state of the algorithm is captured as a record.

```
  open S
  open Profile

  type bitms_at_k = map_nt  (* bitms blocked at k,X *)
  let bitms_at_k_ops = map_nt_ops

  (* state at k changes at k *)
  (* at stage k, there are items that have not been processed (todo
     items) and those that have already been processed (done items) *)
  type state = {
    k:k_t;  (* current stage *)
    todo:nt_item list;  (* todo items at stage k *)

    todo_done:nt_item_set;
    (* todo and done items at stage k; per k; used by add_todo to
       avoid adding an item more than once *)

    todo_gt_k:todo_gt_k;
```

```
  (* todo items at later stages *)
  (* impl: forall k' > k; empty for large k' > k_current and not
     needed for j<k *)

  (* blocked items are items that are "waiting" for some other parse
     to complete before they can continue; they are split into those at
     the current stage, bitms_at_k, and those at earlier stages,
     bitms_lt_k *)
  bitms_lt_k:bitms_lt_k;
  bitms_at_k:bitms_at_k;

  ixk_done:ixk_set;
  (* Completed items are of the form X -> i,as,k,[]; we record only
     i,X,k *)
  (* impl: per k; array (int/i) with values a set of nt?; set of nt
     implemented by binary upto 63/64 bits *)

  ktjs:map_tm;
  (* Terminals T require parsing the input from k; a successful
     terminal parse will match the input between position k and j; the
     corresponding terminal item is (k,T,j). This map stores those
     items (k,T,j) *)
  (* impl: per k; array (tm) with values a list of int *)
}
```

Maps from int are typically indexed by `k`.

NOTE Typically eg `todo_gt_k` would be sparse and, after the first few k, have no entries (since this gets filled when a terminal completes).

3.5 Auxiliary functions: blocked items

The `bitms` function retrieves the blocked items corresponding to an index `k` and a nonterminal `X`. The `add_bitm_at_k` function adds an item at the current stage.

```
let bitms ~bitms_lt_k_ops s0 (k,x) : nt_item_set =
  match (k=s0.k) with
  | true -> (s0.bitms_at_k |> bitms_at_k_ops.map_find x)
  | false -> (s0.bitms_lt_k |> bitms_lt_k_ops.map_find k |> map_nt_ops.map_find x)

(* nt_item blocked on nt at k FIXME nt is just the head of bs; FIXME
   order of nitm and nt (nt is the key) *)
let add_bitm_at_k nitm nt s0 : state =
  { s0 with
    bitms_at_k =
      let m = s0.bitms_at_k in
      let s = map_nt_ops.map_find nt m in
      let s' = nt_item_set_ops.add nitm s in
      let m' = map_nt_ops.map_add nt s' m in
      m' }
```

3.6 Auxiliary functions: todo items

The `pop_todo` function pops an item off the list of items that are "todo" at this stage. // The `add_todo` function adds an item either at the current stage, or at a later stage, depending on the item.

```
let pop_todo s0 =
  match s0.todo with
  | x::xs -> (x,{s0 with todo=xs})
  | _ -> failwith "pop_todo"

(* k is the current stage *)
(* FIXME avoid cost of double lookup by using new ocaml sets with
   boolean rv *)
let add_todo ~nt_item_ops nitm s0 : state =
  let k = s0.k in
  let nitm_k = nitm|>(nt_item_ops.dot_k) in
  match nitm_k > k with
  | true ->
    let nitms = todo_gt_k_ops.map_find nitm_k s0.todo_gt_k in
    let nitms = nt_item_set_ops.add nitm nitms in
    { s0 with todo_gt_k=(todo_gt_k_ops.map_add nitm_k nitms s0.todo_gt_k)}
  | false ->
    (* NOTE this is todo_done at the current stage *)
    match nt_item_set_ops.mem nitm s0.todo_done with
    | true -> s0
    | false ->
      { s0 with todo=(nitm::s0.todo);
                todo_done=nt_item_set_ops.add nitm s0.todo_done}
```

3.7 Auxiliary functions: `(i,X,k)` items

Similarly, we have `add` and `mem` functions for `(i,X,k)` items.

```
let add_ixk_done ix s0 : state =
  { s0 with ixk_done=(ixk_set_ops.add ix s0.ixk_done)}

let mem_ixk_done ix s0 : bool =
  ixk_set_ops.mem ix s0.ixk_done
```

3.8 Auxiliary functions: `find_ktjs`

Finally we have a function to find the set of `j`s corresponding to a parsed terminal item `(k,T,j)`. At stage `k`, we maintain a map from `T` to a list of int (the `j`s). We use an option to distinguish the case where we have not attempted to parse `T` at position `k`, from the case where we have tried to parse, but there were no results.

```
let find_ktjs t s0 : int list option =
  map_tm_ops.map_find t s0.ktjs
```

3.9 Main code, `run_earley` and `step_k`

The main code is parameterized by various set and map operations, `cut`, `new_items` (which provides new items according to the grammar), `input` (the input itself), `parse_tm` (which details

how to parse terminals), `input_length` (the length of the input; the input is not necessarily a string, but can be arbitrary), `init_nt` (the initial nonterminal to start the parse).

```
let counter = ref 0


let run_earley ~nt_item_ops ~bitms_lt_k_ops ~cut ~new_items ~input ~parse_tm
    ~input_length ~init_nt = (
```

We then have some trivial code:

```
  let {dot_nt;dot_i;dot_k;dot_bs} = nt_item_ops in
  let add_todo = add_todo ~nt_item_ops in
  let bitms = bitms ~bitms_lt_k_ops in
```

Finally, we can start the algorithm proper.

We start by popping `nitm` off the todo items at the current stage `k`. We check whether the item is complete (i.e., `bs = []`).

```
  (* step_k ------------------------------------------------------ *)
  let step_k s0 = (
    debug_endline "XXXstep_k";
    assert(log P.ab);
    (*        let _ = (
              counter:=1 + !counter;
              if (!counter mod 1000 = 0) then Gc.full_major() else () )
              in*)
    assert(log P.ac);
    let k = s0.k in
    let bitms = bitms s0 in
    let (nitm,s0) = pop_todo s0 in
    let nitm_complete = nitm|>dot_bs = [] in
    assert(log P.bc);
    (* NOTE waypoints before each split and at end of branch *)
    match nitm_complete with
```

3.10 Complete items

In the complete case, we may or may not have seen the complete item `(i,X,k)` before. We check whether it has already been done or not.

```
    | true -> (
        let (i,x) = (nitm|>dot_i,nitm|>dot_nt) in
        (* possible new complete item (i,X,k) *)
        let already_done = mem_ixk_done (i,x) s0 in
        assert(log P.cd);
        already_done |> function
```

If it has already been done, we do nothing:

```
        | true -> (
            (* the item iXk has already been processed *)
            debug_endline "already_done";
            s0)
```

Otherwise we have to record `(i,X,k)` in the set of done items for the current stage. Additionally, we have to process (cut!) all blocked items `(Y -> i',as,i,X bs)` against this complete item to produce new items of the form `(Y -> i',as X,k,bs)` which we add to the set of todo items at the current stage.

```
        | false -> (
            (* a new complete item iXk *)
            debug_endline "not already_done";
            let s0 = add_ixk_done (i,x) s0 in
            (* FIXME possible optimization if we work with Y -> {h}
                 as i X bs *)
            bitms (i,x)
            |> nt_item_set_with_each_elt
              ~f:(fun ~state:s bitm -> add_todo (cut bitm k) s)
              ~init_state:s0
            |> fun s ->
            assert(log P.de);
            s))
```

This concludes the handling of complete items.

3.11 Incomplete items

If the item is incomplete, then we have a new blocked item of the form `X -> i,as,k,(S bs')` at stage `k`. The symbol `s` may be a terminal or nonterminal. We case split on which it is.

```
    | false (* = nitm_complete *) -> (
        (* NEW BLOCKED item X -> i,as,k,S bs' on k S *)
        let bitm = nitm in
        let s = List.hd (bitm|>dot_bs) in
        s |> sym_case
```

3.11.1 Incomplete items, blocked on nonterminal

If the symbol is a nonterminal `Y`, then we have an item blocked on `k,Y`. We lookup all blocked items `bitms` at `k,Y`. If we have not processed this item, then `bitms` will be empty. If we have processed any item blocked on `k,Y`, then `bitms` will be nonempty, and we do not need to expand `Y`.

```
          ~nt:(fun _Y ->
              (* X -> i,as,k,Y bs'; check if kY is already done *)
              let bitms = bitms (k,_Y) in
              let bitms_empty = nt_item_set_ops.is_empty bitms in
              (* NOTE already_processed_kY = not bitms_empty *)
              (* NOTE the following line serves to record that we
                 are processing kY *)
              let s0 = add_bitm_at_k bitm _Y s0 in
              assert(log P.fg);
              bitms_empty |> function
```

If `bitms` is not empty, then we have already expanded `Y` at stage `k`. However, we may have new complete items `(k,Y,j)` (where, in fact, `j=k` because we are still at stage k). If we have a complete item `(k,Y,k)` we cut it against the blocked item `X -> i,as,k,(Y bs')` to get a new item `X -> i,as Y,k,bs'`.

```
              | false -> (
                  (* kY has already been done, no need to expand;
                     but there may be a complete item kYk *)
                  debug_endline "not bitms_empty";
```

```
    mem_ixk_done (k,_Y) s0 |> function
    | true -> add_todo (cut bitm k) s0
    | false -> s0)  (* FIXME waypoint? *)
```

If `bitms` is empty, we need to expand the symbol `Y` to get new items.

```
| true -> (
    (* we need to expand Y at k; NOTE that there can
       be no complete items kYk, because this is the
       first time we have met kY *)
    debug_endline "bitms_empty";
    assert (mem_ixk_done (k,_Y) s0 = false);
    new_items ~nt:_Y ~input ~k
    |> List_.with_each_elt
      ~step:(fun ~state:s nitm -> add_todo nitm s)
      ~init_state:s0
    |> fun s ->
    assert(log P.gh);
    s))
```

This completes the handling of incomplete nonterminal items.

3.11.2 Incomplete items, blocked on a terminal

If the symbol is a terminal `T`, we check whether we have any complete items `(k,T,j)`. If we have not yet processed `T` at stage `k`, then `ktjs` will be `None`:

```
~tm:(fun t ->
    (* have we already processed kT ? *)
    find_ktjs t s0 |> fun ktjs ->
    assert(log P.hi);
    ktjs
    |> (function
```

In which case, we process `k,T` by calling the auxiliary `parse_tm` with the terminal, the input, the input length, and the current position `k`. We record the `js` corresponding to successful parses `(k,T,j)`.

```
    | None -> (
        (* we need to process kT *)
        debug_endline "ktjs None";
        debug_endline "processing k T";
        let js = parse_tm ~tm:t ~input ~k ~input_length in
        let ktjs = map_tm_ops.map_add t (Some js) s0.ktjs in
        (js,{s0 with ktjs}))
```

If we have already processed `k,T`, then we just retrieve the `js` from previously. In either case, we have the list `js` of `j` s that we need to process against items blocked on `k,T`.

Recall that we are processing item `X -> i,as,k,(T bs')`. Certainly we need to cut this against all the `(k,T,j)` items we have just found. Do we need to deal with any other items blocked on `k,T`? Suppose there was such an item; then it would have been processed at some earlier stage, and at that point it would have been cut against the items `(k,T,j)`. So in fact, we only need to worry about the current blocked item, which we cut against each of the `j` s to get new todo items. FIXME this could be checked with an assert

```
    | Some js ->
        (* we have already processed kT *)
        debug_endline "ktjs Some";
        (js,s0))
  |> fun (js,s0) ->
  assert(log P.ij);
  (* cut (k,T,j) against the current item NOTE each item
     that gets blocked on kT is immediately processed
     against items kTj *)
  js
  |> List_.with_each_elt
    ~step:(fun ~state:s j -> add_todo (cut bitm j) s)
    ~init_state:s0
  |> fun s ->
  assert(log P.jk);
  s))
) (* step_k *)
in
```

This concludes the exposition of the `step_k` function.

3.12 Main code: `loop_k`

At stage `k`, if there are todo items we process them, otherwise we stop.

```
(* loop_k: loop at k ------------------------------------------ *)

let rec loop_k s0 =
  match s0.todo with
  | [] -> s0
  | _ -> loop_k (step_k s0)
in
```

3.13 Main code: `loop`

The main loop repeatedly processes items at `k` before moving on to `k+1`.

```
(* loop ------------------------------------------------------- *)

(* outer loop: repeatedly process items at stage k, then move to
   stage k+1 *)
let rec loop s0 =
  match s0.k >= input_length with
  (* correct? FIXME don't we have to go one further? *)
  | true -> s0
  | false ->
    (* process items *)
    let s0 = loop_k s0 in
    let old_k = s0.k in
    let k = s0.k+1 in
    let todo = todo_gt_k_ops.map_find k s0.todo_gt_k in
    let todo_done = todo in
    let todo = nt_item_set_ops.elements todo in
    let todo_gt_k =
      (* keep debug into around *)
      match debug_enabled with
      | true -> s0.todo_gt_k
      | false -> todo_gt_k_ops.map_remove k s0.todo_gt_k
```

```
      in
      let ixk_done = ixk_set_ops.empty in
      let ktjs = map_tm_ops.map_empty in
      let bitms_lt_k =
        (* FIXME the following hints that bitms_lt_k should be a
           map from k to a map from nt to ... since bitms_at_k is a
           map from nt *)
        bitms_lt_k_ops.map_add old_k s0.bitms_at_k s0.bitms_lt_k
      in
      let bitms_at_k = map_nt_ops.map_empty in
      (* FIXME let all_done = s0.todo_done::s0.all_done in *)
      let s1 =
        {k;todo;todo_done;todo_gt_k;ixk_done;ktjs;bitms_lt_k;bitms_at_k} in
      loop s1
      (* end loop *)
  in
```

3.14 Main code: `result`

Finally, we construct the result by calling `loop`, starting from `k=0`.

```
  (* staged: main entry point ------------------------------------ *)

  (* construct initial context, apply loop *)
  let result : state =
    let k = 0 in
    let init_items = new_items ~nt:init_nt ~input ~k in
    let todo = init_items in
    let todo_done = nt_item_set_ops.empty in
    let todo_gt_k = todo_gt_k_ops.map_empty in
    let ixk_done = ixk_set_ops.empty in
    let ktjs = map_tm_ops.map_empty in
    let bitms_lt_k = bitms_lt_k_ops.map_empty in
    let bitms_at_k = bitms_at_k_ops.map_empty in
    (* let all_done = [] in *)
    let s0 =
      {k;todo;todo_done;todo_gt_k;ixk_done;ktjs;bitms_lt_k;bitms_at_k} in
    loop s0
  in
```

And that is the end of the code.

```
  (* NOTE the result contains the todo_done items at stage
     l=|input|, including any complete items; if we have a complete
     item X -> i,as S,j,[] then we know it arose from a blocked item
     X -> i,as,k,S and a complete item (k,S,j); FIXME so when
     cutting items we at least need to record a map from (S,j) ->
     k

     We could also just retain the todo_done(k) sets, since these
     detail all (complete) items, but then we would have to process
     these sets which might be expensive. *)
  result
) (* run_earley *)

end (* Make *)
```

4 Using the library

There is an example of how to use the library in the file `test.ml`. Note that this includes an optimization (represent items by ints) that introduces some complexity.

A more straightforward implementation is in file `simple_test.ml`. This version takes about twice as long as the "optimized" version. However, this is the one to look at if you want to understand how to use this library for parsing.

5 Informal derivation

From the cut rule, it is clearly important to keep track of items that are blocked on `i,X`. Let's call these `B(i,X)`; `B(i,X)` is a map from `(i,X)` to a set of nonterminal items. In the implementation we distinguish whether the items are blocked at the current stage `B(k,X)` or some previous stage `B(k'<k,X)`.

We also need to keep track of complete items, which are either `C(i,X,k)` or `C(k,T,j)`. In the implementation `C(k,T,j)` is a map from `T` to an optional list of `j`, so that we can check whether we have already attempted to parse `T` at stage k.

At each stage `k` we have a list of `todo(k)` items which we need to process. Processing these items can result in further items being put on the `todo(k)` list (or on the lists for `k'>k`). In the implementation we keep track of all items that are either `todo` or `done`, in order to avoid adding an item more than once to `todo`. FIXME why track done?

Then, for each item in the `todo(k)` list of the form `X -> i,as,k,bs`:

- If item of the form `X -> i,as,k,[]` then process complete item `(i,X,k)`, by cutting it against all items blocked on `(i,X)`. In the implementation we keep track of the set of complete items `(i,X,k)` and only process each once.
- If item of the form `X -> i,as,k,Y bs` then process `Y` at stage `k`, by expanding it using the grammar rules. We also have to add the item to the blocked items `B(k,Y)`. ADDITIONALLY check whether a complete item `(k,Y,k)` has been found, and if so cut it with the item `X -> ...` to get a new todo `X -> i,as Y,k,bs`. In the implementation we only expand `Y` once at stage `k`.
- If item of the form `X -> i,as,k,T bs` then process `T` by parsing the input at position `k` to get a set of terminal items of the form `(k,T,j)`. We then add further todo items `X -> i,as T,j,bs`. NOTE we do not record items blocked on terminals - we process these immediately then they are encountered. In the implementation, we implement `C(k,T,j)` as a map from `T` to an OPTIONAL list of `j` so that we only parse `T` once.

It is fairly easy (!) to convince yourself that this indeed captures all the possible cases that can arise.

6 Slightly more formal derivation

The aim of this section is to attempt to describe the derivation of the implementation from first principles.

We start by restating the "Cut" rule.

```
X -> i,as,k,S bs     S -> k,...,j,[]
---------------------------------- Cut
X -> i,as S,j,bs
```

Now we consider the different types of item at stage k:

- Complete items of the form (i, X, k) or (k, T, j).
- Incomplete items of the form $(X \rightarrow i, \alpha, k, \beta)$.

We also consider the following "work items". Work items serve to label pieces of work which we ideally execute only once (this is the "dynamic programming" aspect of Earley's algorithm).

(1) $W - CUT - COMPLETE(i, X, k)$: Cut complete (i, X, k) with items blocked on (i, X).

(2) $W-CUT-COMPLETE(k, T, j)$: Cut (set of) complete (k, T, j) with item (not items) blocked on (k, T).

(3) $W - CUT - BLOCKED(k, X)$: Cut blocked (i, X) with complete (i, X, k); in general new blocked items are blocked on (k, X), so these need only be cut against (k, X, k) complete items.

(4) $W - EXPAND(k, T)$: Terminal expand: for item blocked on (k, T), expand to get set of (k, T, j) (and then cut as (2) $W - CUT - COMPLETE(k, T, k)$ above).

(5) $W - EXPAND(k, X)$: Non-terminal expand: for item blocked on (k, X), expand X according to the rules of the grammar.

Returning to the different types of item at stage k:

Complete items: Items $(X \rightarrow i, \alpha, k, \beta)$ may be of the form $(X \rightarrow i, \alpha, k, [])$, giving a complete item (i, X, k).

- We need to record whether we have processed an item (i, X, k) at stage k. This can be done using, at each k, a set $DONE - COMPLETE - NT$ of (i, X).
- If we haven't processed the item, we need to process it and record that we have done so in $DONE - COMPLETE - NT$. This is work item $W - CUT - COMPLETE(i, X, k)$. To process it we need to cut it against all items blocked on (i, X). So we need a map $B(i, X)$.
- It may be best to consider the case $i = k$ separately from $i < k$.

Incomplete items: Alternatively, we may have an item $(X \rightarrow i, \alpha, k, S\ \beta)$.

Case S is a terminal T: Items (k, T, j) arise from incomplete items of the form $(X \rightarrow i, \alpha, k, T\beta)$. When we process such an item, we immediately attempt to parse T at position k, and remember the result, a set of (k, T, j). This is $W - EXPAND(k, T)$.

- At stage k we need to record, for each terminal T, whether we have parsed T from position k, and if so, what were the results. This is best done using, at each k, a map $DONE - TERMINAL - K(T)$ to an optional set of int.
- We then cut these (k, T, j) against the item blocked on k, T. This is $W - CUT - COMPLETE(k, T, j)$.

Case S is a nonterminal Y. This involves recording that the item is blocked, and expanding Y according to the rules of the grammar ($W - EXPAND(k, T)$). It may also involve cutting the blocked item against a complete item (k, Y, k).

- At stage k, we need to record those items that are blocked on a nonterminal Y, using a map $BLOCKED - K(Y)$ from a nonterminal to a set of items. After we finish stage k we need to remember this blocked map as $BLOCKED(k, Y)$ when we process later stages.
- We need to record, for each k, whether we have expanded Y at stage k (so maintain, at each stage, a set $EXPANDED-K$ of those nonterminals we have expanded). If we have not, we expand Y according to the rules of the grammar, and update $EXPANDED-K$. NOTE we can reuse $B(k, X)$ which already record whether an item is blocked on (k, X) (in which case, X would have been expanded).
- For an item blocked on k, Y we may also have a complete item (k, Y, k) (in which case, we have already expanded Y at stage k). If so, we need to cut the blocked item against this (k, Y, k). This is $W - CUT - BLOCKED(k, Y)$.

7 Datastructures

Datastructures specific to stage k:

- $TODO(k)$ - a list of items
  - called `todo` in code
- $TODO - DONE(k)$ - a set of items
  - called `todo_done` in code
- $B(k, X)$ - a map to a set of items
  - called `bitms_at_k`
- $C(i < k, X, k)$ - record complete items encountered (and then processed), implemented as a set of (i, X)
  - called `ixk_done`
- $C(k, X, k)$ - ditto, implemented as a set of nonterminals
  - called `ixk_done` - no distinction between i and k
- $C(k, T, j)$ - implemented as a map from T to optional list of j
  - called `ktjs`

Datastructures at stages $<$ k:

- $B(i < k, X)$
  - called `bitms_lt_k`

Datastructures at stages $>$ k:

- $TODO(i > k)$
  - called `todo_gt_k`

8 Informal proof of soundness and completeness

We prove wrt. the naive algorithm that simply applies the cut rule repeatedly, keeping track of the current set of items.

Soundness is straightforward.

For completeness, we need to argue that every action that is taken by the naive algorithm is also taken by our implementation.

```
X -> i,as,k,S bs     S -> k,...,j,[]
---------------------------------- Cut
X -> i,as S,j,bs
```

Let's restrict to the case that S is a nonterminal Y say. Let's rename the indices.

```
X -> i,as,i',Y bs     Y -> i',...,k,[]
---------------------------------- Cut
X -> i,as Y,k,bs
```

By induction on the execution of the naive algorithm, `X -> i,as,i',Y bs` will have been encountered at stage i', and marked as blocked on i', Y. Similarly (i', Y, k) will have been encountered and recorded in $C(i', Y, k)$ (let's assume $i' < k$). Since $i' < k$, the blocked item is available when the complete item $C(i', Y, k)$ is processed by our algorithm. And our algorithm indeed produces a new item `X -> i,as Y,k,bs` to process further.

Other cases are (presumably!) similar.

For formalization, one option is to annotate the naive algorithm with "execution step" numbers to provide an easy way to phrase the induction.

```
X -> i,as,i',Y bs|u     Y -> i',...,k,[]|v
----------------------------------------- Cut
X -> i,as Y,k,bs|succ(u,v)
```

Here, we require `succ(u,v)` to be greater than both `u` and `v` (so take eg `succ(u,v)=1+u+v`). We induct on this measure.