Functional Programming

by Łukasz Stafiniak

Email: `lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl`
Web: `www.ii.uni.wroc.pl/~lukstafi`

Lecture 9: Compiler

Compilation. Runtime. Optimization. Parsing.

Andrew W. Appel *"Modern Compiler Implementation in ML"*
E. Chailloux, P. Manoury, B. Pagano *"Developing Applications with OCaml"*
Jon D. Harrop *"OCaml for Scientists"*
Francois Pottier, Yann Regis-Gianas "*Menhir Reference Manual*"

If you see any error on the slides, let me know!

OCaml Compilers

- OCaml has two primary compilers: the bytecode compiler `ocamlc` and the native code compiler `ocamlopt`.
  - Natively compiled code runs about 10 times faster than bytecode – depending on program.
- OCaml has an interactive shell called *toplevel* (in other languages, *repl*): `ocaml` which is based on the bytecode compiler.
  - There is a toplevel `ocamlnat` based on the native code compiler but currently not part of the binary distribution.
- There are “third-party” compilers, most notably `js_of_ocaml` which translates OCaml bytecode into JavaScript source.
  - On modern JS virtual machines like V8 the result can be 2-3x faster than on OCaml virtual machine (but can also be slower).

- Stages of compilation:

preprocessing
compiling
assembling
linking

Source program
Source or abstract syntax tree program
Assembly program
Machine instrucitons
Executable code

- Programs:

`ocaml`	toplevel loop
`ocamlrun`	bytecode interpreter (VM)
`camlp4`	preprocessor (syntax extensions)
`ocamlc`	bytecode compiler
`ocamlopt`	native code compiler
`ocamlmktop`	new toplevel constructor
`ocamldep`	dependencies between modules
`ocamlbuild`	building projects tool
`ocamlbrowser`	graphical browsing of sources

- File extensions:

`.ml`	OCaml source file
`.mli`	OCaml interface source file
`.cmi`	compiled interface
`.cmo`	bytecode-compiled file
`.cmx`	native-code-compiled file
`.cma`	bytecode-compiled library (several source files)
`.cmxa`	native-code-compiled library
`.cmt`/`.cmti`/`.annot`	type information for editors
`.c`	C source file
`.o`	C native-code-compiled file
`.a`	C native-code-compiled library

- Both compilers commands:

`-a`	construct a runtime library
`-c`	compile without linking
`-o`	name_of_executable specify the name of the executable
`-linkall`	link with all libraries used
`-i`	display all compiled global declarations
`-pp`	command uses command as preprocessor
`-unsafe`	turn off index checking for arrays
`-v`	display the version of the compiler
`-w` list	choose among the list the level of warning message
`-impl` file	indicate that file is a Caml source (.ml)
`-intf` file	indicate that file is a Caml interface (.mli)
`-I` directory	add directory in the list of directories; prefix + for relative
`-g`	generate debugging information

- Warning levels:

`A/a`	enable/disable all messages
`F/f`	partial application in a sequence
`P/p`	for incomplete pattern matching
`U/u`	for missing cases in pattern matching
`X/x`	enable/disable all other messages for hidden object
`M/m`, `V/v`	object-oriented related warnings

- Native compiler commands:

`-compact`	optimize the produced code for space
`-S`	keeps the assembly code in a file
`-inline`	level set the aggressiveness of inlining

- Environment variable `OCAMLRUNPARAM`:

`b`	print detailed stack backtrace of runtime exceptions
`s/h/i`	size of the minor heap/major heap/size increment
`o/O`	major GC speed setting / heap compaction trigger setting

  Typical use, running `prog`: `export OCAMLRUNPARAM='b'; ./prog`

  To have stack backtraces, compile with option -g.

- Toplevel loop directives:

`#quit;;`	exit
`#directory "dir";;`	add `dir` to the "search path"; + for rel.
`#cd "dir-name";;`	change directory
`#load "file-name";;`	load a bytecode `.cmo/.cma` file
`#load_rec "file-name";;`	load the files `file-name` depends on too
`#use "file-name";;`	read, compile and execute source phrases
`#instal_printer pr_nm;;`	register `pr_nm` to print values of a type
`#print_depth num;;`	how many nestings to print
`#print_length num;;`	how many nodes to print – the rest . . .
`#trace func;;/#untrace`	trace calls to `func`/stop tracing

Compiling multiple-file projects

- Traditionally the file containing a module would have a lowercase name, although the module name is always uppercase.
  - Some people think it is more elegant to use uppercase for file names, to reflect module names, i.e. for `MyModule`, use `MyModule.ml` rather than `myModule.ml`.
- We have a project with main module `main.ml` and helper modules `sub1.ml` and `sub2.ml` with corresponding interfaces.

- Native compilation by hand:

```
...:.../Lec9$ ocamlopt sub1.mli
...:.../Lec9$ ocamlopt sub2.mli
...:.../Lec9$ ocamlopt -c sub1.ml
...:.../Lec9$ ocamlopt -c sub2.ml
...:.../Lec9$ ocamlopt -c main.ml
...:.../Lec9$ ocamlopt unix.cmxa sub1.cmx sub2.cmx
main.cmx -o prog
...:.../Lec9$ ./prog
```

- Native compilation using make:

```
PROG := prog
LIBS := unix
SOURCES := sub1.ml sub2.ml main.ml
INTERFACES := $(wildcard *.mli)
OBJS := $(patsubst %.ml,%.cmx,$(SOURCES))
LIBS := $(patsubst %,%.cmxa,$(LIBS))
$(PROG): $(OBJS)
⟨tabulator⟩ocamlopt -o $@ $(LIBS) $(OBJS)
clean: rm -rf $(PROG) *.o *.cmx *.cmi *~
%.cmx: %.ml
⟨tabulator⟩ocamlopt -c $*.ml
%.cmi: %.mli
⟨tabulator⟩ocamlopt -c $*.mli
depend: $(SOURCES) $(INTERFACES)
⟨tabulator⟩ocamldep -native $(SOURCES) $(INTERFACES)
```

  - First use command: `touch .depend; make depend; make`
  - Later just `make`, after creating new source files `make depend`

- Using `ocamlbuild`
  - files with compiled code are created in `_build` directory
  - Command: `ocamlbuild -libs unix main.native`
  - Resulting program is called `main.native` (in directory `_build`, but with a link in the project directory)
  - More arguments passed after comma, e.g.

    ```
    ocamlbuild -libs nums,unix,graphics main.native
    ```
  - Passing parameters to the compiler with `-cflags`, e.g.:

    ```
    ocamlbuild -cflags -I,+lablgtk,-rectypes hello.native
    ```
  - Adding a `--` at the end (followed with command-line arguments for the program) will compile and run the program:

    ```
    ocamlbuild -libs unix main.native --
    ```

Editors

- Emacs
  - `ocaml-mode` from the standard distribution
  - alternative `tuareg-mode` https://forge.ocamlcore.org/projects/tuareg/
    - cheat-sheet: http://www.ocamlpro.com/files/tuareg-mode.pdf
  - `camldebug` intergration with debugger
  - type feedback with `C-c C-t` key shortcut, needs `.annot` files
- Vim
  - OMLet plugin
    http://www.lix.polytechnique.fr/~dbaelde/productions/omlet.html
  - For type lookup: either https://github.com/avsm/ocaml-annot
    - or http://www.vim.org/scripts/script.php?script_id=2025
    - also? http://www.vim.org/scripts/script.php?script_id=1197

- Eclipse
  - *OCaml Development Tools* http://ocamldt.free.fr/
  - an old plugin OcaIDE http://www.algo-prog.info/ocaide/
- TypeRex http://www.typerex.org/
  - currently mostly as `typerex-mode` for Emacs but integration with other editors will become better
  - Auto-completion of identifiers (experimental)
  - Browsing of identifiers: show type and comment, go to definition
  - local and whole-program refactoring: renaming identifiers and compilation units, `open` elimination
- Indentation tool `ocp-ident` https://github.com/OCamlPro/ocp-indent
  - Installation instructions for Emacs and Vim
  - Can be used with other editors.

- Some dedicated editors
  - OCamlEditor http://ocamleditor.forge.ocamlcore.org/
  - `ocamlbrowser` inspects libraries and programs
    - browsing contents of modules
    - search by name and by type
    - basic editing, with syntax highlighting
  - Cameleon http://home.gna.org/cameleon/ (older)
  - Camelia http://camelia.sourceforge.net/ (even older)

Imperative features in OCaml

OCaml is **not** a *purely functional* language, it has built-in:

- Mutable arrays.

  ```
  let a = Array.make 5 0 in
  a.(3) <- 7; a.(2), a.(3)
  ```

  - Hashtables in the standard distribution (based on arrays).

    ```
    let h = Hashtbl.create 11 in
    Hashtbl.add h "Alpha" 5; Hashtbl.find h "Alpha"
    ```

    Takes initial size of the array.

- Mutable strings. (Historical reasons...)

  ```
  let a = String.make 4 'a' in
  a.[2] <- 'b'; a.[2], a.[3]
  ```

  - Extensible mutable strings `Buffer.t` in standard distribution.

- Loops:
  - `for` i = a `to`/`downto` b `do` body `done`
  - `while` condition `do` body `done`

- Mutable record fields, for example:

  ```
  type 'a ref = { mutable contents : 'a }
  ```
  Single, mutable field.

  A record can have both mutable and immutable fields.

  - Modifying the field: `record.field <- new_value`
  - The `ref` type has operations:

    ```
    let (:=) r v = r.contents <- v
    let (!) r = r.contents
    ```

- Exceptions, defined by `exception`, raised by `raise` and caught by `try-with` clauses.

  - An exception is a variant of type `exception`, which is the only open algebraic datatype – new variants can be added to it.

- Input-output functions have no "type safeguards" (no *IO monad*).

Using **global** state e.g. reference cells makes code *non re-entrant*: finish one task before starting another – any form of concurrency is excluded.

Parsing command-line arguments

To go beyond `Sys.argv` array, see `Arg` module:
http://caml.inria.fr/pub/docs/manual-ocaml/libref/Arg.html

Example: configuring a *Mine Sweeper* game.

```
type config = {
   nbcols  : int ; nbrows : int ; nbmines : int }
let default_config = { nbcols=10; nbrows=10; nbmines=15 }
let set_nbcols cf n = cf := {!cf with nbcols = n}
let set_nbrows cf n = cf := {!cf with nbrows = n}
let set_nbmines cf n = cf := {!cf with nbmines = n}
let read_args() =
  let cf = ref default_config in
  let speclist =
   [("-col", Arg.Int (set_nbcols cf), "number of columns");
    ("-lin", Arg.Int (set_nbrows cf), "number of lines");
    ("-min", Arg.Int (set_nbmines cf), "number of mines")] in
  let usage_msg =
    "usage : minesweep [-col n] [-lin n] [-min n]" in
   Arg.parse speclist (fun s -> ()) usage_msg; !cf
```

State of configuration will be updated by given functions.

OCaml Garbage Collection

Representation of values

- Pointers always end with `00` in binary (addresses are in number of bytes).
- Integers are represented by shifting them 1 bit, setting the last bit to `1`.
- Constant constructors (i.e. variants without parameters) like `None`, `[]` and `()`, and other integer-like types (`char`, `bool`) are represented in the same way as integers.
- Pointers are always to OCaml *blocks*. Variants with parameters, strings and OCaml arrays are stored as blocks.
- A block starts with a header, followed by an array of values of size 1 word: either integer-like, or pointers.
- The header stores the size of the block, the 2-bit color used for garbage collection, and 8-bit *tag* – which variant it is.
  - Therefore there can be at most about 240 variants with parameters in a variant type (some tag numbers are reserved).
  - *Polymorphic variants* are a different story.

Generational Garbage Collection

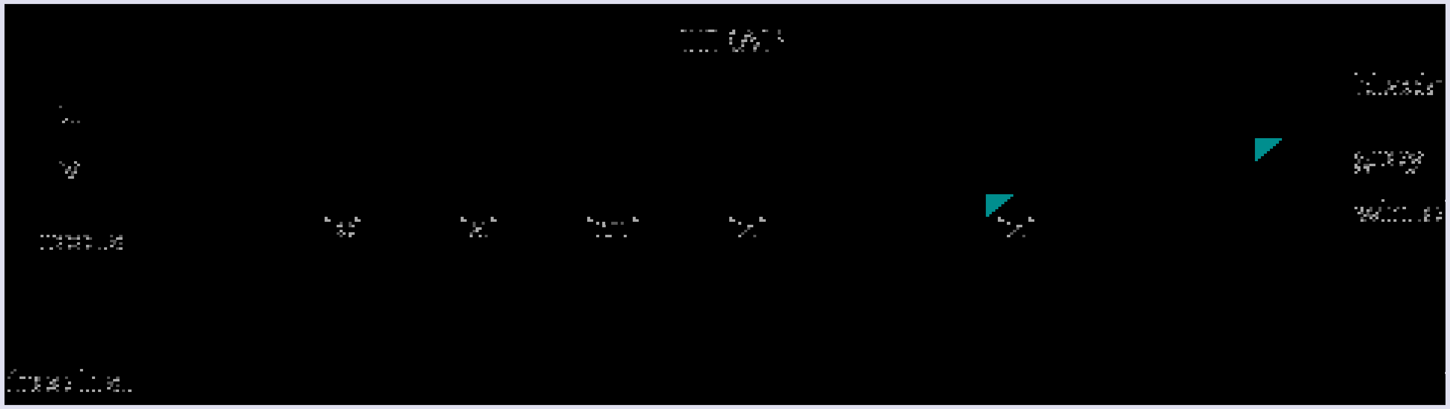
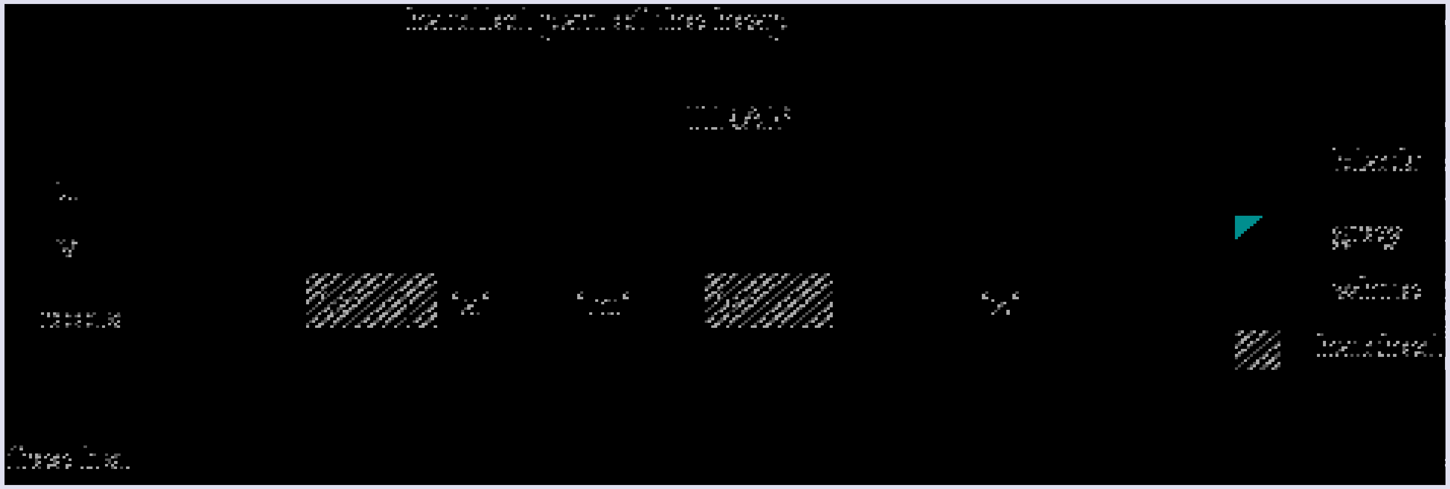
- OCaml has two heaps to store blocks: a small, continuous *minor heap* and a growing-as-necessary *major heap*.
- Allocation simply moves the minor heap pointer (aka. the *young pointer*) and returns the pointed address.
  - Allocation of very large blocks uses the major heap instead.
- When the minor heap runs out of space, it triggers the *minor (garbage) collection*, which uses the *Stop & Copy* algorithm.
- Together with the minor collection, a slice of *major (garbage) collection* is performed to cleanup the major heap a bit.
  - The major heap is not cleaned all at once because it might stop the main program (i.e. our application) for too long.
  - Major collection uses the *Mark & Sweep* algorithm.
- Great if most minor heap blocks are already not needed when collection starts – garbage does **not** slow down collection.

Stop & Copy GC

- Minor collection starts from a set of *roots* – young blocks that definitely are not garbage.
- Besides the root set, OCaml also maintains the *remembered set* of minor heap blocks pointed at from the major heap.
  - Most mutations must check whether they assign a minor heap block to a major heap block field. This is called *write barrier*.
  - Immutable blocks cannot contain pointers from major to minor heap.
    - Unless they are `lazy` blocks.
- Collection follows pointers in the root set and remembered set to find other used blocks.
- Every found block is copied to the major heap.
- At the end of collection, the young pointer is reset so that the minor heap is empty again.

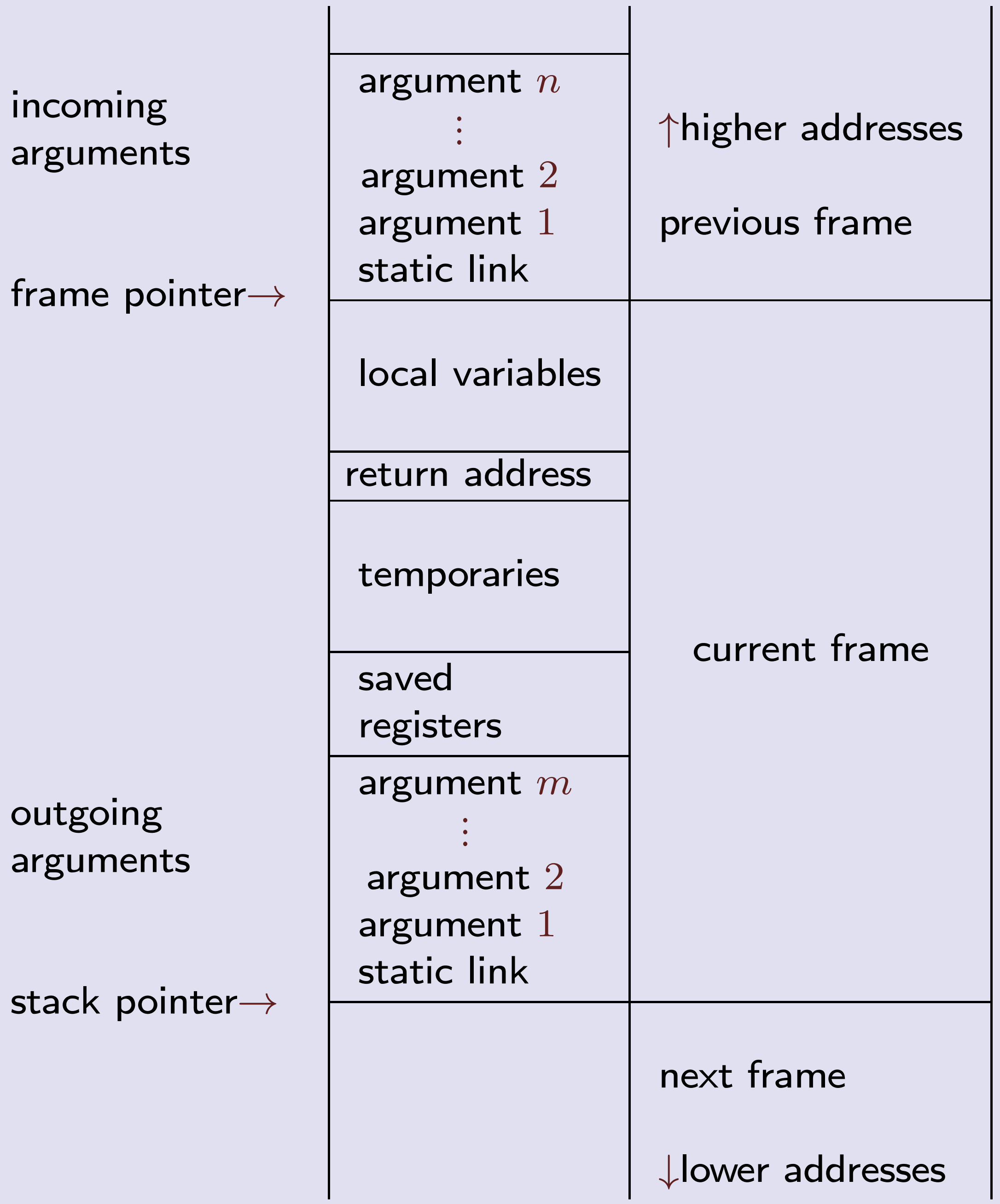
Mark & Sweep GC

- Major collection starts from a separate root set – old blocks that definitely are not garbage.
- Major garbage collection consists of a *mark* phase which colors blocks that are still in use and a *sweep* phase that searches for stretches of unused memory.
  - Slices of the mark phase are performed by-after each minor collection.
  - Unused memory is stored in a *free list*.
- The "proper" major collection is started when a minor collection consumes the remaining free list. The mark phase is finished and sweep phase performed.
- Colors:
  - **gray**: marked cells whose descendents are not yet marked;
  - **black**: marked cells whose descendents are also marked;
  - **hatched**: free list element;
  - **white**: elements previously being in use.

-
  ```
  # let u = let l = ['c'; 'a'; 'm'] in List.tl l ;;
  val u : char list = ['a'; 'm']
  # let v = let r = ( ['z'] , u ) in match r with p -> (fst p) @ (snd p) ;;
  val v : char list = ['z'; 'a'; 'm']
  ```
- 
- 

Stack Frames and Closures

- The nesting of procedure calls is reflected in the *stack* of procedure data.
- The stretch of stack dedicated to a single function is *stack frame* aka. *activation record*.
- *Stack pointer* is where we create new frames, stored in a special register.
- *Frame pointer* allows to refer to function data by offset – data known early in compilation is close to the frame pointer.
- Local variables are stored in the stack frame or in registers – some registers need to be saved prior to function call (*caller-save*) or at entry to a function (*callee-save*). OCaml avoids callee-save registers.
- Up to 4-6 arguments can be passed in registers, remaining ones on stack.
  - Note that *x86* architecture has a small number of registers.
- Using registers, tail call optimization and function inlining can eliminate the use of stack entirely. OCaml compiler can also use stack more efficiently than by creating full stack frames as depicted below.

- 
incoming arguments
frame pointer→
outgoing arguments
stack pointer→
argument n
⋮
argument 2
argument 1
static link
local variables
return address
temporaries
saved registers
argument m
⋮
argument 2
argument 1
static link
↑higher addresses
previous frame
current frame
next frame
↓lower addresses

- *Static links* point to stack frames of parent functions, so we can access stack-based data, e.g. arguments of a main function from inside `aux`.
- A ***closure*** represents a function: it is a block that contains address of the function: either another closure or a machine-code pointer, and a way to access non-local variables of the function.
    - For partially applied functions, it contains the values of arguments and the address of the original function.
- *Escaping variables* are the variables of a function `f` – arguments and local definitions – which are accessed from a nested function which is part of the returned value of `f` (or assigned to a mutable field).
    - Escaping variables must be either part of the closures representing the nested functions, or of a closure representing the function `f` – in the latter case, the nested functions must also be represented by closures that have a link to the closure of `f`.

Tail Recursion

- A function call `f x` within the body of another function g is in *tail position* if, roughly "calling `f` is the last thing that g will do before returning".
- Call inside `try ... with` clause is not in tail position!
  - For efficient exceptions, OCaml stores *traps* for `try`-`with` on the stack with topmost trap in a register, after `raise` unwinding directly to the trap.
- The steps for a tail call are:
  1. Move actual parameters into argument registers (if they aren't already there).
  2. Restore callee-save registers (if needed).
  3. Pop the stack frame of the calling function (if it has one).
  4. Jump to the callee.
- Bytecode always throws `Stack_overflow` exception on too deep recursion, native code will sometimes cause *segmentation fault*!
- `List.map` from the standard distribution is **not** tail-recursive.

Generated assembly

- Let us look at examples from
  http://ocaml.org/tutorials/performance_and_profiling.html

Profiling and Optimization

- Steps of optimizing a program:
  1. Profile the program to find bottlenecks: where the time is spent.
  2. If possible, modify the algorithm used by the bottleneck to an algorithm with better asymptotic complexity.
  3. If possible, modify the bottleneck algorithm to access data less randomly, to increase *cache locality*.
     - Additionally, *realtime* systems may require avoiding use of huge arrays, traversed by the garbage collector in one go.
  4. Experiment with various implementations of data structures used (related to step 3).
  5. Avoid *boxing* and polymorphic functions. Especially for numerical processing. (OCaml specific.)
  6. *Deforestation*.
  7. *Defunctorization*.

Profiling

- We cover native code profiling because it is more useful.
  - It relies on the "Unix" profiling program `gprof`.
- First we need to compile the sources in profiling mode: `ocamlopt -p` ...
  - or using `ocamlbuild` when program source is in `prog.ml`:

    ```
    ocamlbuild prog.p.native --
    ```
- The execution of program `./prog` produces a file `gmon.out`
- We call `gprof prog > profile.txt`
  - or when we used `ocamlbuild` as above:

    ```
    gprof prog.p.native > profile.txt
    ```
  - This redirects profiling analysis to `profile.txt` file.

- The result `profile.txt` has three parts:
  1. List of functions in the program in descending order of the time which was spent within the body of the function, excluding time spent in the bodies of any other functions.
  2. A hierarchical representation of the time taken by each function, and the total time spent in it, including time spent in functions it called.
  3. A bibliography of function references.
- It contains C/assembly function names like `camlList__assoc_1169`:
  - Prefix `caml` means function comes from OCaml source.
  - `List__` means it belongs to a `List` module.
  - `assoc` is the name of the function in source.
  - Postfix `_1169` is used to avoid name clashes, as in OCaml different functions often have the same names.
- Example: computing words histogram for a large file, `Optim0.ml`.

Imperative programming example.

Lecture 6 read_lines function would stack-overflow because of the try-with clause.

```
let read_words file =
  let input = open_in file in
  let words = ref [] and more = ref true in
  try
    while !more do
      Scanf.fscanf input "%[^a-zA-Z0-9’]%[a-zA-Z0-9’]"
        (fun b x -> words := x :: !words; more := x <> "")
    done;
    List.rev (List.tl !words)
  with End_of_file -> List.rev !words
```

Inefficient map update.

```
let empty () = []
let increment h w =
  try
    let c = List.assoc w h in
    (w, c+1) :: List.remove_assoc w h
  with Not_found -> (w, 1)::h
let iterate f h =
  List.iter (fun (k,v)->f k v) h
```

```
let histogram words =
  List.fold_left increment (empty ()) words

let _ =
  let words = read_words "./shakespeare.xml" in
  let words = List.rev_map String.lowercase words in
  let h = histogram words in
  let output = open_out "histogram.txt" in
  iterate (Printf.fprintf output "%s: %dn") h;
  close_out output
```

- Now we look at the profiling analysis, first part begins with:

```
 %   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
37.88      8.54     8.54 306656698    0.00     0.00  compare_val
19.97     13.04     4.50   273169     0.00     0.00  camlList__assoc_1169
 9.17     15.10     2.07 633527269    0.00     0.00  caml_page_table_lookup
 8.72     17.07     1.97   260756    0.00   0.00 camlList__remove_assoc_1189
 7.10     18.67     1.60 612779467    0.00     0.00  caml_string_length
 4.97     19.79     1.12 306656692     0.00    0.00  caml_compare
 2.84     20.43     0.64                             caml_c_call
 1.53     20.77     0.35    14417     0.00     0.00  caml_page_table_modify
 1.07     21.01     0.24     1115     0.00     0.00  sweep_slice
 0.89     21.21     0.20      484     0.00     0.00  mark_slice
```

- `List.assoc` and `List.remove_assoc` high in the ranking suggests to us that `increment` could be the bottleneck.
  - They both use comparison which could explain why `compare_val` consumes the most of time.

- Next we look at the interesting pieces of the second part: data about the `increment` function.
    - Each block, separated by `------` lines, describes the function whose line starts with an index in brackets.
    - The functions that called it are above, the functions it calls below.

```
index % time    self  children    called     name
-----------------------------------------------
                0.00    6.47  273169/273169  camlList__fold_left_1078 [7]
[8]     28.7    0.00    6.47  273169         camlOptim0__increment_1038 [8]
                4.50    0.00  273169/273169  camlList__assoc_1169 [9]
               1.97    0.00  260756/260756  camlList__remove_assoc_1189 [11]
```

- As expected, `increment` is only called by `List.fold_left`. But it seems to account for only 29% of time. It is because compare is not analysed correctly, thus not included in time for `increment`:

```
-----------------------------------------------
                1.12   12.13 306656692/306656692     caml_c_call [1]
[2]     58.8    1.12   12.13 306656692         caml_compare [2]
                8.54    3.60 306656692/306656698     compare_val [3]
```

Algorithmic optimizations

- (All times measured with profiling turned on.)
- `Optim0.ml` asymptotic time complexity: $\mathcal{O}(n^2)$, time: 22.53s.
  - Garbage collection takes 6% of time.
    - So little because data access wastes a lot of time.
- Optimize the data structure, keep the algorithm.

```
let empty () = Hashtbl.create 511
let increment h w =
  try
    let c = Hashtbl.find h w in
    Hashtbl.replace h w (c+1); h
  with Not_found -> Hashtbl.add h w 1; h
let iterate f h = Hashtbl.iter f h
```

`Optim1.ml` asymptotic time complexity: $\mathcal{O}(n)$, time: 0.63s.

- Garbage collection takes 17% of time.

- Optimize the algorithm, keep the data structure.

  ```
  let histogram words =
    let words = List.sort String.compare words in
    let k,c,h = List.fold_left
      (fun (k,c,h) w ->
        if k = w then k, c+1, h else w, 1, ((k,c)::h))
      ("", 0, []) words in
    (k,c)::h
  ```

  `Optim2.ml` asymptotic time complexity: $\mathcal{O}(n \log n)$, time: 1s.

  - Garbage collection takes 40% of time.

- Optimizing for cache efficiency is more advanced, we will not attempt it.
- With algorithmic optimizations we should be concerned with **asymptotic complexity** in terms of the $\mathcal{O}(\cdot)$ notation, but we will not pursue complexity analysis in the remainder of the lecture.

Low-level optimizations

- Optimizations below have been made *for educational purposes only*.
- Avoid polymorphism in generic comparison function (=).

```
let rec assoc x = function
    [] -> raise Not_found
  | (a,b)::l -> if String.compare a x = 0 then b else assoc x l
let rec remove_assoc x = function
  | [] -> []
  | (a, b as pair) :: l ->
      if String.compare a x = 0 then l else pair :: remove_assoc x l
```

`Optim3.ml` (based on `Optim0.ml`) time: 19s.

  - Despite implementation-wise the code is the same, as `String.compare` = `Pervasives.compare` inside module `String`, and `List.assoc` is like above but uses `Pervasives.compare`!
  - We removed polymorphism, no longer `caml_compare_val` function.
  - Usually, adding type annotations would be enough. (Useful especially for numeric types `int`, `float`.)

- **Deforestation** means removing intermediate data structures.

```
let read_to_histogram file =
  let input = open_in file in
  let h = empty () and more = ref true in
  try
    while !more do
      Scanf.fscanf input "%[^a-zA-Z0-9']%[a-zA-Z0-9']"
        (fun b w ->
          let w = String.lowercase w in
          increment h w; more := w <> "")
    done; h
  with End_of_file -> h
```

`Optim4.ml` (based on `Optim1.ml`) time: 0.51s.

  - Garbage collection takes 8% of time.
    - So little because we have eliminated garbage.

- **Defunctorization** means computing functor applications by hand.
  - There was a tool `ocamldefun` but it is out of date.
  - The slight speedup comes from the fact that functor arguments are implemented as records of functions.

Comparison of data structure implementations

- We perform a rough comparison of association lists, tree-based maps and hashtables. Sets would give the same results.
- We always create hashtables with initial size 511.
- 10^7 operations of: adding an association (creation), finding a key that is in the map, finding a key out of a small number of keys not in the map.
- First row gives sizes of maps. Time in seconds, to two significant digits.

create:	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
assoc list	0.25	0.25	0.18	0.19	0.17	0.22	0.19	0.19	0.19	
tree map	0.48	0.81	0.82	1.2	1.6	2.3	2.7	3.6	4.1	5.1
hashtable	27	9.1	5.5	4	2.9	2.4	2.1	1.9	1.8	3.7

create:	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}
tree map	6.5	8	9.8	15	19	26	34	41	51	67	80	130
hashtable	4.8	5.6	6.4	8.4	12	15	19	20	22	24	23	33

found:	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
assoc list	1.1	1.5	2.5	4.2	8.1	17	30	60	120	
tree map	1	1.1	1.3	1.5	1.9	2.1	2.5	2.8	3.1	3.6
hashtable	1.4	1.5	1.4	1.4	1.5	1.5	1.6	1.6	1.8	1.8

found:	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}
tree map	4.3	5.2	6	7.6	9.4	12	15	17	19	24	28	32
hashtable	1.8	2	2.5	3.1	4	5.1	5.9	6.4	6.8	7.6	6.7	7.5

not found:	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
assoc list	1.8	2.6	4.6	8	16	32	60	120	240	
tree map	1.5	1.5	1.8	2.1	2.4	2.7	3	3.2	3.5	3.8
hashtable	1.4	1.4	1.5	1.5	1.6	1.5	1.7	1.9	2	2.1

not found:	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}
tree map	4.2	4.3	4.7	4.9	5.3	5.5	6.1	6.3	6.6	7.2	7.5	7.3
hashtable	1.8	1.9	2	1.9	1.9	1.9	2	2	2.2	2	2	1.9

- Using lists makes sense for up to about 15 elements.
- Unfortunately OCaml and Haskell do not encourage the use of efficient maps, the way Scala and Python have built-in syntax for them.

Parsing: ocamllex and Menhir

- *Parsing* means transforming text, i.e. a string of characters, into a data structure that is well fitted for a given task, or generally makes information in the text more explicit.
- Parsing is usually done in stages:
  1. *Lexing* or *tokenizing*, dividing the text into smallest meaningful pieces called *lexemes* or *tokens*,
  2. composing bigger structures out of lexemes/tokens (and smaller structures) according to a *grammar*.
     - Alternatively to building such hierarchical structure, sometimes we build relational structure over the tokens, e.g. *dependency grammars*.
- We will use `ocamllex` for lexing, whose rules are like pattern matching functions, but with patterns being *regular expressions*.
- We will either consume the results from lexer directly, or use *Menhir* for parsing, a successor of `ocamlyacc`, belonging to the *yacc*/*bison* family of parsers.

Lexing with *ocamllex*

- The format of lexer definitions is as follows: file with extension `.mll`

```
{ header }
let ident1 = regexp ...
rule entrypoint1 [arg1... argN] =
  parse regexp { action1 }
        | ...
        | regexp { actionN }
and entrypointN [arg1? argN] =
  parse ...
and ...
{ trailer }
```

  - Comments are delimited by `(*` and `*)`, as in OCaml.
  - The `parse` keyword can be replaced by the `shortest` keyword.
  - ”Header”, “trailer”, “action1”, ... “actionN” are arbitrary OCaml code.
  - There can be multiple let-clauses and rule-clauses.

- Let-clauses are shorthands for regular expressions.
- Each rule-clause `entrypoint` defines function(s) that as the last argument (after `arg1`... `argN` if `N>0`) takes argument `lexbuf` of type `Lexing.lexbuf`.
  - `lexbuf` is also visible in actions, just as a regular argument.
  - `entrypoint1`... `entrypointN` can be mutually recursive if we need to read more before we can return output.
  - It seems `rule` keyword can be used only once.
- We can use `lexbuf` in actions:
  - `Lexing.lexeme lexbuf` – Return the matched string.
  - `Lexing.lexeme_char lexbuf n` – Return the nth character in the matched string. The first character corresponds to n = 0.
  - `Lexing.lexeme_start`/`lexeme_end lexbuf` – Return the absolute position in the input text of the beginning/end of the matched string (i.e. the offset of the first character of the matched string). The first character read from the input text has offset 0.
- The parser will call an `entrypoint` when it needs another lexeme/token.

- The syntax of **regular expressions**
  - `'c'` – match the character `'c'`
  - `_` – match a **single** character
  - `eof` – match end of lexer input
  - `"string"` – match the corresponding sequence of characters
  - [character set] – match the character set, characters `'c'` and ranges of characters `'c'-'d'` separated by space
  - [^character set] – match characters outside the character set
  - [character set 1] # [character set 2] – match the difference, i.e. only characters in set 1 that are not in set 2
  - regexp* – (repetition) match the concatenation of zero or more strings that match regexp
  - regexp+ – (strict repetition) match the concatenation of one or more strings that match regexp

- regexp`?` – (option) match the empty string, or a string matching regexp.
- regexp1 `|` regexp2 – (alternative) match any string that matches regexp1 or regexp2
- regexp1 regexp2 – (concatenation) match the concatenation of two strings, the first matching regexp1, the second matching regexp2.
- `(` regexp `)` – match the same strings as regexp
- `ident` – reference the regular expression bound to ident by an earlier `let ident =` regexp definition
- regexp `as ident` – bind the substring matched by regexp to identifier `ident`.

The precedences are: `#` highest, followed by `*`, `+`, `?`, concatenation, `|`, `as`.

- The type of `as ident` variables can be `string`, `char`, `string option` or `char option`
  - `char` means obviously a single character pattern
  - `option` means situations like `(`regexp `as ident)?` or regexp1|(regexp2 `as ident)`
  - The variables can repeat in the pattern (**unlike** in normal paterns) – meaning both regexpes match the same substrings.
- `ocamllex Lexer.mll` produces the lexer code in `Lexer.ml`
  - `ocamlbuild` will call `ocamllex` and `ocamlyacc/menhir` if needed
- Unfortunately if the lexer patterns are big we get an error:

  transition table overflow, automaton is too big

Example: Finding email addresses

- We mine a text file for email addresses, that could have been obfuscated to hinder our job...
- To compile and run `Emails.mll`, processing a file `email_corpus.xml`:

  `ocamlbuild Emails.native -- email_corpus.xml`

```
{                                                       The header with OCaml code.
  open Lexing                                      Make accessing Lexing easier.
  let nextline lexbuf =        Typical lexer function: move position to next line.
    let pos = lexbuf.lex_curr_p in
    lexbuf.lex_curr_p <- { pos with
      pos_lnum = pos.pos_lnum + 1;
      pos_bol = pos.pos_cnum;
    }
  type state =                         Which step of searching for address we're at:
  | Seek                  Seek: still seeking, Addr (true...): possibly finished,
  | Addr of bool * string * string list          Addr (false...): no domain.
  let report state lexbuf =
    match state with
    | Seek -> ()
    | Addr (false, _, _) -> ()
    | Addr (true, name, addr) ->
      Printf.printf "%d: %s@%sn" lexbuf.lex_curr_p.pos_lnum
        name (String.concat "." (List.rev addr))
}
```

Report the found address, if any.

With line at which it is found.

```
let newline = ('\n' | "\r\n")
let addr_char = ['a'-'z' 'A'-'Z' '0'-'9' '-' '_']
let at_w_symb = "where" | "WHERE" | "at" | "At" | "AT"
let at_nw_symb = '@' | "&#x40;" | "&#64;"
let open_symb = ' '* '(' ' '* | ' '+
let close_symb = ' '* ')' ' '* | ' '+
let at_sep_symb =
  open_symb? at_nw_symb close_symb? |
  open_symb at_w_symb close_symb
```

Regexp for end of line.

Demarcate a possible @ or . symbol.

```
let dot_w_symb = "dot" | "DOT" | "dt" | "DT"
let dom_w_symb = dot_w_symb | "dom" | "DOM"
let dot_sep_symb =
  open_symb dot_w_symb close_symb |
  open_symb? '.' close_symb?
let dom_sep_symb =
  open_symb dom_w_symb close_symb |
  open_symb? '.' close_symb?
let addr_dom = addr_char addr_char
  | "edu" | "EDU" | "org" | "ORG" | "com" | "COM"
```

Obfuscation for last dot.

Restricted form of last part of address.

```
rule email state = parse
| newline
    { report state lexbuf; nextline lexbuf;
      email Seek lexbuf }
| (addr_char+ as name) at_sep_symb (addr_char+ as addr)
    { email (Addr (false, name, [addr])) lexbuf }
```

Check state before moving on.

↙Detected possible start of address.

```
| dom_sep_symb (addr_dom as dom)          Detected possible finish of address.
    { let state =
        match state with
        | Seek -> Seek                     We weren't looking at an address.
        | Addr (_, name, addrs) ->                                      Bingo.
          Addr (true, name, dom::addrs) in
      email state lexbuf }
| dot_sep_symb (addr_char+ as addr)                Next part of address –
    { let state =                                       must be continued.
        match state with
        | Seek -> Seek
        | Addr (_, name, addrs) ->
          Addr (false, name, addr::addrs) in
      email state lexbuf }
| eof                                              End of file – end loop.
     { report state lexbuf }
| _                   Some boring character – not looking at an address yet.
     { report state lexbuf; email Seek lexbuf }
{                                                    The trailer with OCaml code.
  let _ =                        Open a file and start mining for email addresses.
    let ch = open_in Sys.argv.(1) in
    email Seek (Lexing.from_channel ch);
    close_in ch                                         Close the file at the end.
}
```

Parsing with Menhir

- The format of parser definitions is as follows: file with extension `.mly`

`%{ header %}`	OCaml code put in front.
`%parameter < M : signature >`	Parameters make a functor.
`%token < type1 > Token1 Token2`	Terminal productions, variants
`%token < type3 > Token3`	returned from lexer.
`%token NoArgToken`	Without an argument, e.g. keywords or symbols.
`%nonassoc Token1`	This token cannot be stacked without parentheses.
`%left Token3`	Associates to left,
`%right Token2`	to right.
`%type < type4 > rule1`	Type of the action of the rule.
`%start < type5 > rule2`	The entry point of the grammar.
`%%`	Separate out the rules part.
`%inline rule1 (id1, ..., inN) :`	Inlined rules can propagate priorities.
`  | production1 { action1 }`	If production matches, perform action.
`  | production2 | production3`	Several productions
`      { action2 }`	with the same action.

```
%public rule2 :                                   Visible in other files of the grammar.
  | production4 { action4 }
%public rule3 :        Override precedence of production5 to that of productions
  | production5 { action5 } %prec Token1                      ending with Token1
%%                    The separations are needed even if the sections are empty.
trailer                               OCaml code put at the end of generated source.
```

- Header, actions and trailer are OCaml code.
- Comments are `(* ... *)` in OCaml code, `/* ... */` or `// ...` outisde
- Rules can optionally be separated by `;`
- `%parameter` turns the **whole** resulting grammar into a functor, multiple parameters are allowed. The parameters are visible in `%{...%}`.
- Terminal symbols `Token1` and `Token2` are both variants with argument of type type1, called their *semantic value*.
- `rule1`... `ruleN` must be lower-case identifiers.
- Parameters `id1`... `idN` can be lower- or upper-case.

- Priorities, i.e. precedence, are declared implicitly: `%nonassoc`, `%left`, `%right` list tokens in increasing priority (`Token2` has highest precedence).
  - Higher precedence = a rule is applied even when tokens so far could be part of the other rule.
  - Precedence of a production comes from its rightmost terminal.
  - `%left`/`%right` means left/right associativity: the rule will/won't be applied if the "other" rule is the same production.
- `%start` symbols become names of functions exported in the `.mli` file to invoke the parser. They are automatically `%public`.
- `%public` rules can even be defined over multiple files, with productions joined by `|`.

- The syntax of productions, i.e. patterns, each line shows one aspect, they can be combined:

`rule2 Token1 rule3`	Match tokens in sequence with `Token1` in the middle.
`a=rule2 t=Token3`	Name semantic values produced by rules/tokens.
`rule2; Token3`	Parts of pattern can be separated by semicolon.
`rule1(arg1,...,argN)`	Use a rule that takes arguments.
`rule2?`	Shorthand for `option(rule2)`
`rule2+`	Shorthand for `nonempty_list(rule2)`
`rule2*`	Shorthand for `list(rule2)`

- Always-visible "standard library" – most of rules copied below:

```
%public option(X):
  /* nothing */
    { None }
| x = X
    { Some x }
%public %inline pair(X, Y):
  x = X; y = Y
    { (x, y) }
```

```
%public %inline separated_pair(X, sep, Y):
  x = X; sep; y = Y
    { (x, y) }
%public %inline delimited(opening, X, closing):
  opening; x = X; closing
    { x }
%public list(X):
  /* nothing */
    { [] }
| x = X; xs = list(X)
    { x :: xs }
%public nonempty_list(X):
  x = X
    { [ x ] }
| x = X; xs = nonempty_list(X)
    { x :: xs }
%public %inline separated_list(separator, X):
  xs = loption(separated_nonempty_list(separator, X))
    { xs }
```

```
%public separated_nonempty_list(separator, X):
  x = X
    { [ x ] }
| x = X; separator; xs =
separated_nonempty_list(separator, X)
    { x :: xs }
```

- Only *left-recursive* rules are truly tail-recursive, as in:

```
declarations:
| { [] }
| ds = declarations; option(COMMA);
  d = declaration { d :: ds }
```

  - This is opposite to code expressions (or *recursive descent parsers*), i.e. if both OK, first rather than last invocation should be recursive.

- Invocations can be nested in arguments, e.g.:

```
plist(X):
| xs = loption(                                      Like option, but returns a list.
  delimited(LPAREN,
            separated_nonempty_list(COMMA, X),
            RPAREN)) { xs }
```

- Higher-order parameters are allowed.

```
procedure(list):
| PROCEDURE ID list(formal) SEMICOLON block SEMICOLON {...}
```

- Example where inlining is required (besides being an optimization)

```
%token < int > INT
%token PLUS TIMES
%left PLUS
%left TIMES                                    Multiplication has higher priority.
%%
expression:
| i = INT { i }                 ↙ Without inlining, would not distinguish priorities.
| e = expression; o = op; f = expression { o e f }
%inline op:                      Inline operator – generate corresponding rules.
| PLUS { ( + ) }
| TIMES { ( * ) }
```

- Menhir is an $\mathrm{LR}(1)$ parser generator, i.e. it fails for grammars where looking one token ahead, together with precedences, is insufficient to determine whether a rule applies.
    - In particular, only unambiguous grammars.
- Although $\mathrm{LR}(1)$ grammars are a small subset of *context free grammars*, the semantic actions can depend on context: actions can be functions that take some form of context as input.
- Positions are available in actions via keywords `$startpos(x)` and `$endpos(x)` where `x` is name given to part of pattern.
    - Do not use the `Parsing` module from OCaml standard library.

Example: parsing arithmetic expressions

- Example based on a Menhir demo. Due to difficulties with `ocamlbuild`, we use option `--external-tokens` to provide `type token` directly rather than having it generated.
- File `lexer.mll`:

```
{
  type token =
    | TIMES
    | RPAREN
    | PLUS
    | MINUS
    | LPAREN
    | INT of (int)
    | EOL
    | DIV
  exception Error of string
}
```

```
rule line = parse
| ([^'n']* 'n') as line { line }
| eof  { exit 0 }
and token = parse
| [' ' 't']        { token lexbuf }
| 'n' { EOL }
| ['0'-'9']+ as i { INT (int_of_string i) }
| '+'  { PLUS }
| '-'  { MINUS }
| '*'  { TIMES }
| '/'  { DIV }
| '('  { LPAREN }
| ')'  { RPAREN }
| eof  { exit 0 }
| _    { raise (Error (Printf.sprintf "At offset %d:
unexpected character.n" (Lexing.lexeme_start lexbuf))) }
```

- File `parser.mly`:

We still need to define tokens,
Menhir does its own checks.

```
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token EOL
%left PLUS MINUS          /* lowest precedence */
%left TIMES DIV           /* medium precedence */
%nonassoc UMINUS          /* highest precedence */
%parameter<Semantics : sig
  type number
  val inject: int -> number
  val ( + ): number -> number -> number
  val ( - ): number -> number -> number
  val ( * ): number -> number -> number
  val ( / ): number -> number -> number
  val ( ~-): number -> number
end>
%start <Semantics.number> main
%{ open Semantics %}
```

```
%%
main:
| e = expr EOL   { e }
expr:
| i = INT       { inject i }
| LPAREN e = expr RPAREN    { e }
| e1 = expr PLUS e2 = expr  { e1 + e2 }
| e1 = expr MINUS e2 = expr { e1 - e2 }
| e1 = expr TIMES e2 = expr { e1 * e2 }
| e1 = expr DIV e2 = expr   { e1 / e2 }
| MINUS e = expr %prec UMINUS { - e }
```

- File calc.ml:

```
module FloatSemantics = struct
  type number = float
  let inject = float_of_int
  let ( + ) = ( +. )
  let ( - ) = ( -. )
  let ( * ) = ( *. )
  let ( / ) = ( /. )
  let (~- ) = (~-. )
end
module FloatParser = Parser.Make(FloatSemantics)
```

```
let () =
  let stdinbuf = Lexing.from_channel stdin in
  while true do
    let linebuf =
      Lexing.from_string (Lexer.line stdinbuf) in
    try
      Printf.printf "%.1fn%!"
        (FloatParser.main Lexer.token linebuf)
    with
    | Lexer.Error msg ->
      Printf.fprintf stderr "%s%!" msg
    | FloatParser.Error ->
      Printf.fprintf stderr
        "At offset %d: syntax error.n%!"
        (Lexing.lexeme_start linebuf)
  done
```

- Build and run command:

  ```
  ocamlbuild calc.native -use-menhir -menhir "menhir
  parser.mly --base parser --external-tokens Lexer" --
  ```

  - Other grammar files can be provided besides `parser.mly`
  - `--base` gives the file (without extension) which will become the module accessed from OCaml
  - `--external-tokens` provides the OCaml module which defines the `token` type

Example: a toy sentence grammar

- Our lexer is a simple limited *part-of-speech tagger*. Not re-entrant.
- For debugging, we log execution in file `log.txt`.
- File `EngLexer.mll`:

```
{
 type sentence = {                    Could be in any module visible to EngParser.
   subject : string;                       The actor/actors, i.e. subject noun.
   action : string;                                     The action, i.e. verb.
   plural : bool;                             Whether one or multiple actors.
   adjs : string list;                             Characteristics of actor.
   advs : string list                             Characteristics of action.
 }
```

```
type token =
| VERB of string
| NOUN of string
| ADJ of string
| ADV of string
| PLURAL | SINGULAR
| A_DET | THE_DET | SOME_DET | THIS_DET | THAT_DET
| THESE_DET | THOSE_DET
| COMMA_CNJ | AND_CNJ | DOT_PUNCT
let tok_str = function ...                              Print the token.
let adjectives =                                   Recognized adjectives.
  ["smart"; "extreme"; "green"; "slow"; "old"; "incredible";
   "quiet"; "diligent"; "mellow"; "new"]
let log_file = open_out "log.txt"          File with debugging information.
let log s = Printf.fprintf log_file "%sn%!" s
let last_tok = ref DOT_PUNCT                        State for better tagging.
```

```
 let tokbuf = Queue.create ()
 let push w =
   log ("lex: "^tok_str w);
   last_tok := w; Queue.push w tokbuf
 exception LexError of string
}
let alphanum = ['0'-'9' 'a'-'z' 'A'-'Z' ''' '-']
rule line = parse
| ([^'\n']* '\n') as l { l }
| eof { exit 0 }
and lex_word = parse
| [' ' '\t']
    { lex_word lexbuf }
| '.' { push DOT_PUNCT }
| "a" { push A_DET } | "the" { push THE_DET }
| "some" { push SOME_DET }
| "this" { push THIS_DET } | "that" { push THAT_DET }
| "these" { push THESE_DET } | "those" { push THOSE_DET }
| "A" { push A_DET } | "The" { push THE_DET }
```

Token buffer, since single word is sometimes two tokens.

Log lexed token.

For line-based interface.

Skip whitespace.

End of sentence.

"Keywords".

```
| "Some" { push SOME_DET }
| "This" { push THIS_DET } | "That" { push THAT_DET }
| "These" { push THESE_DET } | "Those" { push THOSE_DET }
| "and" { push AND_CNJ }
| ',' { push COMMA_CNJ }
| (alphanum+ as w) "ly"
    {
      if List.mem w adjectives
      then push (ADV w)
      else if List.mem (w^"le") adjectives
      then push (ADV (w^"le"))
      else (push (NOUN w); push SINGULAR)
    }
```

Adverb is adjective that ends in “ly”.

```
| (alphanum+ as w) "s"
    {
      if List.mem w adjectives then push (ADJ w)
      else match !last_tok with
      | THE_DET | SOME_DET | THESE_DET | THOSE_DET
      | DOT_PUNCT | ADJ _ ->
        push (NOUN w); push PLURAL
      | _ -> push (VERB w); push SINGULAR
    }
| alphanum+ as w
    {
      if List.mem w adjectives then push (ADJ w)
      else match !last_tok with
      | A_DET | THE_DET | SOME_DET | THIS_DET | THAT_DET
      | DOT_PUNCT | ADJ _ ->
        push (NOUN w); push SINGULAR
      | _ -> push (VERB w); push PLURAL
    }
```

Plural noun or singular verb.

Noun contexts vs. verb contexts.

```
| _ as w
    { raise (LexError ("Unrecognized character "
                       ^Char.escaped w)) }
{
  let lexeme lexbuf =          The proper interface reads from the token buffer.
    if Queue.is_empty tokbuf then lex_word lexbuf;
    Queue.pop tokbuf
}
```

- File `EngParser.mly`:

```
%{
  open EngLexer                        Source of the token type and sentence type.
%}
%token <string> VERB NOUN ADJ ADV                          Open word classes.
%token PLURAL SINGULAR                                          Number marker.
%token A_DET THE_DET SOME_DET THIS_DET THAT_DET                   “Keywords”.
%token THESE_DET THOSE_DET
%token COMMA_CNJ AND_CNJ DOT_PUNCT
%start <EngLexer.sentence> sentence                            Grammar entry.
%%
```

```
%public %inline sep2_list(sep1, sep2, X):
| xs = separated_nonempty_list(sep1, X) sep2 x=X
    { xs @ [x] }
| x=option(X)
    { match x with None->[] | Some x->[x] }
sing_only_det:
| A_DET | THIS_DET | THAT_DET { log "prs: sing_only_det" }
plu_only_det:
| THESE_DET | THOSE_DET { log "prs: plu_only_det" }
other_det:
| THE_DET | SOME_DET { log "prs: other_det" }
np(det):
| det adjs=list(ADJ) subject=NOUN
    { log "prs: np"; adjs, subject }
vp(NUM):
| advs=separated_list(AND_CNJ,ADV) action=VERB NUM
| action=VERB NUM advs=sep2_list(COMMA_CNJ,AND_CNJ,ADV)
    { log "prs: vp"; action, advs }
```

General purpose.

We use it for "comma-and" lists:

smart, quiet ***and*** *diligent.*

How determiners relate to number.

```
sent(det,NUM):                                   Sentence parameterized by number.
| adjsub=np(det) NUM vbadv=vp(NUM)
    { log "prs: sent";
      {subject=snd adjsub; action=fst vbadv; plural=false;
       adjs=fst adjsub; advs=snd vbadv} }
vbsent(NUM):                                     Unfortunately, it doesn't always work...
| NUM vbadv=vp(NUM)    { log "prs: vbsent"; vbadv }
sentence:                                        Sentence, either singular or plural number.
| s=sent(sing_only_det,SINGULAR) DOT_PUNCT
    { log "prs: sentence1";
      {s with plural = false} }
| s=sent(plu_only_det,PLURAL) DOT_PUNCT
    { log "prs: sentence2";
      {s with plural = true} }
```

```
| adjsub=np(other_det) vbadv=vbsent(SINGULAR) DOT_PUNCT
    { log "prs: sentence3";
      {subject=snd adjsub; action=fst vbadv; plural=false;
       adjs=fst adjsub; advs=snd vbadv} }
| adjsub=np(other_det) vbadv=vbsent(PLURAL) DOT_PUNCT
    { log "prs: sentence4";
      {subject=snd adjsub; action=fst vbadv; plural=true;
       adjs=fst adjsub; advs=snd vbadv} }
```

Because parser allows only one token look-ahead we need to factor-out the “common subset”.

- File `Eng.ml` is the same as `calc.ml` from previous example:

```
open EngLexer
let () =
  let stdinbuf = Lexing.from_channel stdin in
  while true do
    (* Read line by line. *)
    let linebuf = Lexing.from_string (line stdinbuf) in
    try
      (* Run the parser on a single line of input. *)
      let s = EngParser.sentence lexeme linebuf in
      Printf.printf
    "subject=%s\nplural=%b\nadjs=%s\naction=%snadvs=%s\n\n%!"
        s.subject s.plural (String.concat ", " s.adjs)
        s.action (String.concat ", " s.advs)
    with
    | LexError msg ->
      Printf.fprintf stderr "%sn%!" msg
    | EngParser.Error ->
      Printf.fprintf stderr "At offset %d: syntax error.n%!"
          (Lexing.lexeme_start linebuf)
  done
```

- Build & run command:

```
ocamlbuild Eng.native -use-menhir -menhir "menhir
EngParser.mly --base EngParser --external-tokens EngLexer"
--
```

Example: Phrase search

- In lecture 6 we performed keyword search, now we turn to *phrase search* i.e. require that given words be consecutive in the document.
- We start with some English-specific transformations used in lexer:

```
let wh_or_pronoun w =
  w = "where" || w = "what" || w = "who" ||
  w = "he" || w = "she" || w = "it" ||
  w = "I" || w = "you" || w = "we" || w = "they"
let abridged w1 w2 =
  if w2 = "ll" then [w1; "will"]
  else if w2 = "s" then
    if wh_or_pronoun w1 then [w1; "is"]
    else ["of"; w1]
  else if w2 = "d" then [w1^"ed"]
  else if w1 = "o" || w1 = "O"
  then
    if w2.[0] = 'e' && w2.[1] = 'r' then [w1^"v"^w2]
    else ["of"; w2]
  else if w2 = "t" then [w1; "it"]
  else [w1^"'"^w2]
```

Remove shortened forms like *I'll* or *press'd*.

- For now we normalize words just by lowercasing, but see exercise 8.
- In lexer we *tokenize* text: separate words and normalize them.
  - We also handle simple aspects of *XML* syntax.
- We store the number of each word occurrence, excluding XML tags.

```
{
  open IndexParser
  let word = ref 0
  let linebreaks = ref []
  let comment_start = ref Lexing.dummy_pos
  let reset_as_file lexbuf s =
    let pos = lexbuf.Lexing.lex_curr_p in
    lexbuf.Lexing.lex_curr_p <- { pos with
      Lexing.pos_lnum =  1;
      pos_fname = s;
      pos_bol = pos.Lexing.pos_cnum;
    };
    linebreaks := []; word := 0
  let nextline lexbuf =
    ...
    linebreaks := !word :: !linebreaks
```

General purpose lexer function: start lexing from a file.

Old friend.

Besides changing position, remember a line break.

General purpose lexer function: report a syntax error.

```
  let parse_error_msg startpos endpos report =
    let clbeg =
      startpos.Lexing.pos_cnum - startpos.Lexing.pos_bol in
    ignore (Format.flush_str_formatter ());
    Printf.sprintf
      "File "%s", lines %d-%d, characters %d-%d: %sn"
      startpos.Lexing.pos_fname startpos.Lexing.pos_lnum
      endpos.Lexing.pos_lnum clbeg
      (clbeg+(endpos.Lexing.pos_cnum - startpos.Lexing.pos_cnum))
      report
}
let alphanum = ['0'-'9' 'a'-'z' 'A'-'Z']
let newline = ('n' | "rn")
let xml_start = ("<!--" | "<?")
let xml_end = ("-->" | "?>")
rule token = parse
  | [' ' 't']
      { token lexbuf }
  | newline
      { nextline lexbuf; token lexbuf }
```

```
| '<' alphanum+ '>' as w
    { OPEN w }
| "</" alphanum+ '>' as w
    { CLOSE w }
| "'tis"
    { word := !word+2; WORDS ["it", !word-1; "is", !word] }
| "'Tis"
    { word := !word+2; WORDS ["It", !word-1; "is", !word] }
| "o'clock"
    { incr word; WORDS ["o'clock", !word] }
| "O'clock"
    { incr word; WORDS ["O'clock", !word] }
| (alphanum+ as w1) ''' (alphanum+ as w2)
    { let words = EngMorph.abridged w1 w2 in
      let words = List.map
        (fun w -> incr word; w, !word) words in
      WORDS words }
| alphanum+ as w
    { incr word; WORDS [w, !word] }
| "&amp;"
    { incr word; WORDS ["&", !word] }
```

Dedicated token variants for XML tags.

```
| ['.' '!' '?'] as p
  { SENTENCE (Char.escaped p) }
| "--"
    { PUNCT "--" }
| [',' ':' ''' '-' ';'] as p
    { PUNCT (Char.escaped p) }
| eof { EOF }
| xml_start
    { comment_start := lexbuf.Lexing.lex_curr_p;
      let s = comment [] lexbuf in
      COMMENT s }
| _
    { let pos = lexbuf.Lexing.lex_curr_p in
      let pos' = {pos with
        Lexing.pos_cnum = pos.Lexing.pos_cnum + 1} in
      Printf.printf "%s\n%!"
        (parse_error_msg pos pos' "lexer error");
      failwith "LEXER ERROR" }
```

Dedicated tokens for punctuation so that it doesn't break phrases.

```
and comment strings = parse
  | xml_end
      { String.concat "" (List.rev strings) }
  | eof
      { let pos = !comment_start in
        let pos’ = lexbuf.Lexing.lex_curr_p in
        Printf.printf "%sn%!"
          (parse_error_msg pos pos’ "lexer error: unclosed comment");
        failwith "LEXER ERROR" }
  | newline
      { nextline lexbuf;
        comment (Lexing.lexeme lexbuf :: strings) lexbuf
      }
  | _
      { comment (Lexing.lexeme lexbuf :: strings) lexbuf }
```

- Parsing: the inverted index and the query.

```
type token =
| WORDS of (string * int) list
| OPEN of string | CLOSE of string | COMMENT of string
| SENTENCE of string | PUNCT of string
| EOF
```

```
let inv_index update ii lexer lexbuf =
  let rec aux ii =
    match lexer lexbuf with
    | WORDS ws ->
      let ws = List.map (fun (w,p)->EngMorph.normalize w, p) ws in
      aux (List.fold_left update ii ws)
    | OPEN _ | CLOSE _ | SENTENCE _ | PUNCT _ | COMMENT _ ->
      aux ii
    | EOF -> ii in
  aux ii

let phrase lexer lexbuf =
  let rec aux words =
    match lexer lexbuf with
    | WORDS ws ->
      let ws = List.map (fun (w,p)->EngMorph.normalize w) ws in
      aux (List.rev_append ws words)
    | OPEN _ | CLOSE _ | SENTENCE _ | PUNCT _ | COMMENT _ ->
      aux words
    | EOF -> List.rev words in
  aux []
```

Naive implementation of phrase search

- We need *postings lists* with positions of words rather than just the document or line of document they belong to.
- First approach: association lists and merge postings lists word-by-word.

```
let update ii (w, p) =
  try
    let ps = List.assoc w ii in
    (w, p::ps) :: List.remove_assoc w ii
  with Not_found -> (w, [p])::ii
let empty = []
let find w ii = List.assoc w ii
let mapv f ii = List.map (fun (k,v)->k, f v) ii
let index file =
  let ch = open_in file in
  let lexbuf = Lexing.from_channel ch in
  EngLexer.reset_as_file lexbuf file;
  let ii =
    IndexParser.inv_index update empty EngLexer.token lexbuf in
  close_in ch;
  mapv List.rev ii, List.rev !EngLexer.linebreaks
```

Add position to the postings list of `w`.

Keep postings lists in increasing order.

```
let find_line linebreaks p =                      Recover the line in document of a position.
  let rec aux line = function
    | [] -> line
    | bp::_ when p < bp -> line
    | _::breaks -> aux (line+1) breaks in
  aux 1 linebreaks
let search (ii, linebreaks) phrase =
  let lexbuf = Lexing.from_string phrase in
  EngLexer.reset_as_file lexbuf ("search phrase: "^phrase);
  let phrase = IndexParser.phrase EngLexer.token lexbuf in
  let rec aux wpos = function                     Merge postings lists for words in query:
    | [] -> wpos                                                     no more words in query;
    | w::ws ->                                    for positions of w, keep those that are next to
      let nwpos = find w ii in                               filtered positions of previous word.
      aux (List.filter (fun p->List.mem (p-1) wpos) nwpos) ws in
  let wpos =
    match phrase with
    | [] -> []                                                       No results for an empty query.
    | w::ws -> aux (find w ii) ws in
  List.map (find_line linebreaks) wpos            Answer in terms of document lines.
```

```
let shakespeare = index "./shakespeare.xml"
let query q =
  let lines = search shakespeare q in
  Printf.printf "%s: lines %s\n%!" q
    (String.concat ", " (List.map string_of_int lines))
```

- Test: 200 searches of the queries:

```
["first witch"; "wherefore art thou";
  "captain's captain"; "flatter'd"; "of Fulvia";
  "that which we call a rose"; "the undiscovered country"]
```

- Invocation: `ocamlbuild InvIndex.native -libs unix --`
- Time: 7.3s

Replace association list with hash table

- I recommend using either *OCaml Batteries* or *OCaml Core* – replacement for the standard library. *Batteries* has efficient `Hashtbl.map` (our `mapv`).
- Invocation: `ocamlbuild InvIndex1.native -libs unix --`
- Time: 6.3s

Replace naive merging with ordered merging

- Postings lists are already ordered.
- Invocation: `ocamlbuild InvIndex2.native -libs unix --`
- Time: 2.5s

Bruteforce optimization: biword indexes

- Pairs of words are much less frequent than single words so storing them means less work for postings lists merging.
- Can result in much bigger index size: $\min\ (W^2,\ N)$ where W is the number of distinct words and N the total number of words in documents.
- Invocation that gives us stack backtraces:

  ```
  ocamlbuild InvIndex3.native -cflag -g -libs unix; export
  OCAMLRUNPARAM="b"; ./InvIndex3.native
  ```
- Time: 2.4s – disappointing.

Smart way: *Information Retrieval* G.V. Cormack et al.

- You should classify your problem and search literature for state-of-the-art algorithm to solve it.
- The algorithm needs a data structure for inverted index that supports:
  - `first(w)` – first position in documents at which `w` appears
  - `last(w)` – last position of `w`
  - `next(w,cp)` – first position of `w` after position `cp`
  - `prev(w,cp)` – last position of `w` before position `cp`
- We develop `next` and `prev` operations in stages:
  - First, a naive (but FP) approach using the `Set` module of OCaml.
    - We could use our balanced binary search tree implementation to avoid the overhead due to limitations of `Set` API.
  - Then, *binary search* based on arrays.
  - Imperative linear search.
  - Imperative *galloping search* optimization of binary search.

The phrase search algorithm

- During search we maintain *current position* cp of last found word or phrase.
- Algorithm is almost purely functional, we use Not_found exception instead of option type for convenience.

```
let rec next_phrase ii phrase cp =   Return the beginning and end position
  let rec aux cp = function         of occurrence of phrase after position cp.
    | [] -> raise Not_found          Empty phrase counts as not occurring.
    | [w] ->                         Single or last word of phrase has the same
      let np = next ii w cp in np, np                 beg. and end position.
    | w::ws ->                       After locating the endp. move back.
      let np, fp = aux (next ii w cp) ws in
      prev ii w np, fp in                     If distance is this small,
  let np, fp = aux cp phrase in               words are consecutive.
  if fp - np = List.length phrase - 1 then np, fp
  else next_phrase ii phrase fp
```

```
let search (ii, linebreaks) phrase =
  let lexbuf = Lexing.from_string phrase in
  EngLexer.reset_as_file lexbuf ("search phrase: "^phrase);
  let phrase = IndexParser.phrase EngLexer.token lexbuf in
  let rec aux cp =
    try                                        Find all occurrences of the phrase.
      let np, fp = next_phrase ii phrase cp in
      np :: aux fp
    with Not_found -> [] in                        Moved past last occurrence.
  List.map (find_line linebreaks) (aux (-1))
```

Naive but purely functional inverted index

```
module S = Set.Make(struct type t=int let compare i j = i-j end)
let update ii (w, p) =
  (try
    let ps = Hashtbl.find ii w in
    Hashtbl.replace ii w (S.add p ps)
  with Not_found -> Hashtbl.add ii w (S.singleton p));
  ii
let first ii w = S.min_elt (find w ii)
let last ii w = S.max_elt (find w ii)
let prev ii w cp =
  let ps = find w ii in
  let smaller, _, _ = S.split cp ps in
  S.max_elt smaller
let next ii w cp =
  let ps = find w ii in
  let _, _, bigger = S.split cp ps in
  S.min_elt bigger
```

The functions raise `Not_found` whenever such position would not exist.

Split the set into elements smaller and bigger than cp.

- Invocation: `ocamlbuild InvIndex4.native -libs unix --`
- Time: 3.3s – would be better without the overhead of `S.split`.

Binary search based inverted index

```
let prev ii w cp =
  let ps = find w ii in
  let rec aux b e =
    if e-b <= 1 then ps.(b)
    else let m = (b+e)/2 in
         if ps.(m) < cp then aux m e
         else aux b m in
  let l = Array.length ps in
  if l = 0 || ps.(0) >= cp then raise Not_found
  else aux 0 (l-1)
let next ii w cp =
  let ps = find w ii in
  let rec aux b e =
    if e-b <= 1 then ps.(e)
    else let m = (b+e)/2 in
         if ps.(m) <= cp then aux m e
         else aux b m in
  let l = Array.length ps in
  if l = 0 || ps.(l-1) <= cp then raise Not_found
  else aux 0 (l-1)
```

We implement binary search separately for `prev` to make sure here we return less than `cp`

and here more than `cp`.

- File: `InvIndex5.ml`. Time: 2.4s

Imperative, linear scan

```
let prev ii w cp =
  let cw,ps = find w ii in
  let l = Array.length ps in
  if l = 0 || ps.(0) >= cp then raise Not_found
  else if ps.(l-1) < cp then cw := l-1
  else (
    if !cw < l-1 && ps.(!cw+1) < cp then cw := l-1;
    while ps.(!cw) >= cp do decr cw done
  );
  ps.(!cw)
let next ii w cp =
  let cw,ps = find w ii in
  let l = Array.length ps in
  if l = 0 || ps.(l-1) <= cp then raise Not_found
  else if ps.(0) > cp then cw := 0
  else (
    if !cw > 0 && ps.(!cw-1) > cp then cw := 0;
    while ps.(!cw) <= cp do incr cw done
  );
  ps.(!cw)
```

For each word we add a cell with last visited occurrence.

Reset pointer if current position is not “ahead” of it.

Otherwise scan starting from last visited.

Reset pointer if current position is not ahead of it.

- End of `index`-building function:

```
  mapv (fun ps->ref 0, Array.of_list (List.rev ps)) ii,...
```

- File: `InvIndex6.ml`
- Time: 2.8s

Imperative, galloping search

```
let next ii w cp =
  let cw,ps = find w ii in
  let l = Array.length ps in
  if l = 0 || ps.(l-1) <= cp then raise Not_found;
  let rec jump (b,e as bounds) j =                    Locate the interval with cp inside.
    if e < l-1 && ps.(e) <= cp then jump (e,e+j) (2*j)
    else bounds in
  let rec binse b e =                                 Binary search over that interval.
    if e-b <= 1 then e
    else let m = (b+e)/2 in
         if ps.(m) <= cp then binse m e
         else binse b m in
  if ps.(0) > cp then cw := 0
  else (
    let b =                                           The invariant is that ps.(b) <= cp.
      if !cw > 0 && ps.(!cw-1) <= cp then !cw-1 else 0 in
    let b,e = jump (b,b+1) 2 in                       Locate interval starting near !cw.
    let e = if e > l-1 then l-1 else e in
    cw := binse b e
  );
  ps.(!cw)
```

- `prev` is symmetric to `next`.
- File: `InvIndex7.ml`
- Time: 2.4s – minimal speedup in our simple test case.