Contents

1 Module `Grammar` : Main module of Pacomb

PaComb implements a representation of grammar with semantical action. A semantical action is a value returned as result of parsing. Parsing is performed by compiling the grammar to combinators implemented in the `Combinator` module. This library offers "scanner less" parsing, but the `Lex` module provide a notion of terminals and blanks which allows for easy way to write grammars in two phases as usual.

Defining languages using directly the Grammar module leads to cumbersome code. This is why Pacomb propose a ppx extension that can be used with the compilation flag `-ppx pacombPpx`. Here is an example:

```
[%%parser
   type p = Atom | Prod | Sum
   let rec
     expr p = Atom < Prod < Sum
            ; (p=Atom) (x::FLOAT)                        => x
            ; (p=Atom) '(' (e::expr Sum) ')'             => e
            ; (p=Prod) (x::expr Prod) '*' (y::expr Atom) => x*.y
            ; (p=Prod) (x::expr Prod) '/' (y::expr Atom) => x/.y
            ; (p=Sum ) (x::expr Sum ) '+' (y::expr Prod) => x+.y
            ; (p=Sum ) (x::expr Sum ) '-' (y::expr Prod) => x-.y
]
```

The extension `[%%parser ...]` extends structure with new let bindings defining grammars. This applies both for `let` and `let rec` the latter being reserved to recursive grammars. We also provide an extension `[%grammar]` for expression that corresponds to grammars, i.e. the right-hand side of binding in the `[%%parser]` extension.

Here is the BNF for these right-hand-side, with its semantics

```
grammar ::= rule                                                  itself
       | grammar ; rule                                      Grammar.alt
rule ::= qitems => expr                                A rule with its action
       | expr < ... < expr                            priority order see below
qitems ::= ()                                                Grammar.empty
       | non_empty_qitems                                           itself
non_empty_qitems ::= qitem
       | non_empty_qitems qitems                                Grammar.seq
qitem ::= item | (lid :: item)           give a name if used in the action
item ::= '...'                                  Grammar.term(Lex.char ())
       | "..."                                Grammar.term(Lex.string ())
       | INT                                      Grammar.term(Lex.int ())
       | FLOAT                                  Grammar.term(Lex.float ())
       | RE(exp)       Grammar.term(Lex.regexp (Regexp.from_string exp))
       | exp                                                        itself
```

- non recursive let bindings correspond to just a name for the grammar.
- recursive let bindings correspond either to
- `declare_grammar + set_grammar` (if no paramater)
- `grammar_familly + setting the grammar` is a parameter is given.

Anything which does not coresponds to this grammar will we keeped unchanged in the structure as ocaml code (like the type definition in the example above. A mutually recursive definition can also mix te definition of grammars (parametric of not) with the definition of normal ocaml values.

1.1 Type

`type 'a grammar`

type of a grammar with semantical action of type `'a` .

`type 'a t = 'a grammar`

An abbreviation

1.2 Grammar contructors

`val print_grammar : ?def:bool -> Stdlib.out_channel -> 'a grammar -> unit`

`print_grammar ch g` prints the grammar `g` of the given output channel. if `def=false` (the default is `true`) it will print the transformed grammar prior to compilation.

`val fail : unit -> 'a grammar`

`fail ()` is a grammar that parses nothing (always fails)

`val empty : 'a -> 'a grammar`

`empty a` accepts the empty input and returns `a`

`val test : bool -> unit grammar`

`test b` is `if b then empty () else fail ()`. Very usefull in grammar family at the beginning of a rule

`val term : ?name:string -> 'a Lex.terminal -> 'a grammar`

`term t` accepts the terminal `t` and returns its semantics. See module `Lex`

`val appl : ?name:string -> 'a grammar -> ('a -> 'b) -> 'b grammar`

`appl g f` parses with `g` and apply `f` to the resulting semantics

`val alt : 'a grammar -> 'a grammar -> 'a grammar`

`alt g1 g2` parses with `g1` and if it fails then `g2`

```
val seq : 'a grammar ->
  'b grammar -> ('a -> 'b -> 'c) -> 'c grammar
```

`seq g1 g2 f` parse with g1 and then with g2 for the rest of the input, uses `f` to combine both semantics

`val seq1 : 'a grammar -> 'b grammar -> 'a grammar` usefull derivations from `seq`

```
val seq2 : 'a grammar -> 'b grammar -> 'b grammar
val seqf : 'a grammar -> ('a -> 'b) grammar -> 'b grammar
val dseq : 'a grammar ->
  ('a -> 'b grammar) -> ('b -> 'c) -> 'c grammar
```

`dseq g1 g2 f)` is a dependant sequence, the grammar `g2` used after `g1` may depend upon the semantics of `g1`. This is not very efficient as the grammar `g2` must be compiled at parsing time. It is a good idea to memoize `g2`

```
val lpos : (Position.t -> 'a) grammar -> 'a grammar
```

`lpos g` is identical to `g` but passes the position just before parsing with `g` to the semantical action of `g`

```
val rpos : (Position.t -> 'a) grammar -> 'a grammar
```

`rpos g` is identical to `g` but passes the position just after parsing with `g` to the semantical action of `g`

```
val seqf_pos :
  'a grammar ->
  (Position.t -> 'a -> Position.t -> 'b) grammar -> 'b grammar
```

variants of seqf with the position of the first iterm

```
val seqf_lpos :
  'a grammar ->
  (Position.t -> 'a -> 'b) grammar -> 'b grammar
val seqf_rpos :
  'a grammar ->
  ('a -> Position.t -> 'b) grammar -> 'b grammar
val seq2_pos :
  'a grammar ->
  (Position.t -> Position.t -> 'b) grammar -> 'b grammar
```

variants of seq2 with the position of the first iterm

```
val seq2_lpos : 'a grammar ->
  (Position.t -> 'b) grammar -> 'b grammar
val seq2_rpos : 'a grammar ->
  (Position.t -> 'b) grammar -> 'b grammar
val cache : 'a grammar -> 'a grammar
```

`cache g` avoid to parse twice the same input with g by memoizing the result of the first parsing. Using `cache` allows to recover a polynomial complexity

```
val layout :
  ?old_before:bool ->
```

```
  ?new_before:bool ->
  ?new_after:bool ->
  ?old_after:bool -> 'a grammar -> Lex.blank -> 'a grammar
```

`layout g b` changes the blank function to parse the input with the grammar `g`. The optional parameters allow to control which blanks are used at the bounndary. Both can be used in which case the new blanks are used second before parsing with `g` and first after.

1.3 Definition of recursive grammars

```
val declare_grammar : string -> 'a grammar
```

to define recursive grammars, one may declare the grammar first and then gives its value. `declare_grammar name` creates an undefined grammar with the given name

```
val set_grammar : 'a grammar -> 'a grammar -> unit
```

`set_grammar g1 g2` set the value of `g1` declared with `declare_grammar`. will raise `Invalid_argument` if `g1` was not defined using `declare_grammar` or if it was already set.

```
val fixpoint : ?name:string ->
  ('a grammar -> 'a grammar) -> 'a grammar
```

`fixpoint g` compute the fixpoint of `g`, that is a grammar `g0` such that `g0 = g g0`

```
val grammar_family :
  ?param_to_string:('a -> string) ->
  string -> ('a -> 'b grammar) * (('a -> 'b grammar) -> unit)
```

`grammar_family to_str name` returns a pair `(gs, set_gs)`, where `gs` is a finite family of grammars parametrized by a value of type `'a`. A name `name` is to be provided for the family, and an optional function `to_str` can be provided to print the parameter and display better error messages.

```
(* Declare the grammar family *)
let (gr, set_gr) = grammar_family to_str name in

... code using grammars of gr to define mutually recursive grammars ...
... the grammars in gr cannot be used in "left position" ...
... (same restriction as for declare_grammar ...

(* Define the grammar family *)
let _ = set_gr the_grammars

... now the new family can be used ...
```

1.4 Compilation of a grammar and various

`val compile : 'a grammar -> 'a Combinator.t`

`compile g` produces a combinator that can be used to actually do the parsing see the `Combinator` module

`val grammar_info : 'a grammar -> bool * Charset.t`

`grammar_info g` returns `(b,cs)` where `b` is true is the grammar accepts the empty input and where `cs` is the characters set accepted at the beginnning of the input.

`val grammar_name : 'a grammar -> string`

gives the grammar name

`val give_name : string -> 'a grammar -> 'a grammar`

allows to rename a grammar

2 Module `Combinator` : Combinator library, using continuation

As usual left recursion is not supported, but the library is intended to be used through the `Grammar` module that provides elimination of left recursion. However, a cache combinatr is supported to overcome the cost of backtracking.

2.1 function and type usefull to the end-user

`type 'a combinator`

The type of combinator

`type 'a t = 'a combinator`

Abbreviation

`val give_up : unit -> 'a`

`give_up ()` will reject the current parsing rule from the action code

`exception Parse_error of Input.buffer * int`

Exception raised by the function below when parsing fails

`val handle_exception : ?error:(unit -> 'b) -> ('a -> 'b) -> 'a -> 'b`

`handle_exception fn v` applies the function `fn` to `v` and handles the `Parse_error` exception. In particular, a parse error message is presented to the user in case of a failure, then `error ()` is called. The default `error` is `fun () -> exit 1`.

`val parse_buffer : 'a t -> Lex.blank -> Input.buffer -> 'a`

Parse a whole input buffer. the eof combinator is added at the end of the given combinator

```
val parse_string : 'a t -> Lex.blank -> string -> 'a
```

Parse a whole string

```
val parse_channel : 'a t -> Lex.blank -> Stdlib.in_channel -> 'a
```

Parse a whole input channel

```
val partial_parse_buffer :
  'a t ->
  Lex.blank ->
  ?blank_after:bool -> Input.buffer -> int -> 'a * Input.buffer * int
```

Partial parsing. Beware, the returned position is not the maximum position that can be reached by the grammar

2.2 combinator constructors, normally not needed by the casual user

```
val cfail : 'a t
```

Always fails

```
val cempty : 'a -> 'a t
```

Accepting the empty input only

```
val cterm : 'a Lex.fterm -> 'a t
```

Accepts a given terminal

```
val cseq : 'a t -> 'b t -> ('a -> 'b -> 'c) -> 'c t
```

Sequence of two combinators, parses with the first and then parses the rest of the input with the second combinator. The last function is used to compose the semantics returned by the two combinators.

```
val cdep_seq : 'a t -> ('a -> 'b t) -> ('b -> 'c) -> 'c t
```

`sdep_seq c1 c2 f` is a dependant sequence, contrary to `seq c1 c2 f`, the combinator used to parse after `c1` depends upon the value returned by `c1`. It s a good idea to memoize the function c2.

```
val calt : ?cs1:Charset.t ->
  ?cs2:Charset.t -> 'a t -> 'a t -> 'a t
```

Combinator parsing with the first combinator and in case of failure with the second from the same position. The optionnal charset corresponds to the charaters accepted at the beginning of the input for each combinators. The charset must be Charset.full if the corresponding combinator accept the empty input

```
val capp : 'a t -> ('a -> 'b) -> 'b t
```

Parses with the given combinator and transforms the semantics with the given function

```
val clpos : (Position.t -> 'a) t -> 'a t
```

Parses as the given combinator and give the position to the left of the parsing input as argument to the action

```
val cpush : 'a t -> 'a t
```

To eliminate left recursion, lpos has to be left factored. if lpos is one single combinator, this adds a lot of closures in action code. To solve this problem, lpos is splitted in two combinators, one that pushes the position to a stack and pops after parsing and another that reads the position.

```
val cread : int -> (Position.t -> 'a) t -> 'a t
```

```
val crpos : (Position.t -> 'a) t -> 'a t
```

Same as above with the position to the right

```
val clr : ?cs2:Charset.t ->
  'a t -> ('a -> 'a) t -> 'a t
```

`cls c1 c2` is an optimized version of `let rec r = seq c1 (seq r c2)` which is illegal as it is left recursive and loops. The optional charset indicates the characteres accepted by `c2` at the beginning of input.

```
val cref : 'a t Stdlib.ref -> 'a t
```

Access to a reference to a combinator, use by Grammar.compile for recursive grammars (not for left recursion

```
val clayout :
  ?old_before:bool ->
  ?new_before:bool ->
  ?new_after:bool ->
  ?old_after:bool -> 'a t -> Lex.blank -> 'a t
```

Change the blank function used to parse with the given combinator. we can choose which blank to use at the boundary with the optional parameters.

```
val ccache : 'a t -> 'a t
```

Combinator that caches a grammar to avoid exponential behavior. parsing with the grammar from each position is memoized to avoid parsing twice the same sequence with the same grammar.

3 Module `Lex` : Lexing: grouping characters before parsing

It is traditionnal to do parsing in two phases (scanning/parsing). This is not necessary with combinators in general this is still true with Pacomb (scannerless). However, this makes the grammar more readable to use a lexing phase.

Moreover, lexing is often done with a longuest match rule that is not semantically equivalent to the semantics of context free grammar.

This modules provide combinator to create terminals that the parser will call.

It also provide function to eliminate "blank" characteres.

```
type buf = Input.buffer
```

3.1 Types and exception

```
type blank = buf -> int -> buf * int
```

A blank function is just a function progressing in a buffer

```
type 'a fterm = buf -> int -> 'a * buf * int
```

Type of terminal function, similar to blank, but with a returned value

```
type 'a terminal =
{  n : string ;
```

name

```
  f : 'a fterm ;
```

the terminal itself

```
  c : Charset.t ;
```

the set of characters accepted at the beginning of input

```
}
```

The previous type encapsulated in a record

```
type 'a t = 'a terminal
```

Abbreviation

```
exception NoParse
```

exception when failing,

- can be raised (but not captured) by terminal
- can be raised (but not captured) by action code in the grammar, see `Combinator.give_up`
- will be raise and captured by `Combinator.parse_buffer` that will give the most advanced position

3.2 Combinators to create terminals

`val eof : ?name:string -> 'a -> 'a t`

Terminal accepting then end of a buffer only. remark: `eof` is automatically added at the end of a grammar by `Combinator.parse_buffer`. `name` default is `"EOF"`

`val char : ?name:string -> char -> 'a -> 'a t`

Terminal accepting a given char, remark: `char '\255'` is equivalent to `eof`. `name` default is the given charater.

`val test : ?name:string -> (char -> bool) -> char t`

Accept a character for which the test returns `true`. `name` default to the result of `Charset.show`.

`val charset : ?name:string -> Charset.t -> char t`

Accept a character in the given charset. `name` default as in `test`

`val not_test : ?name:string -> (char -> bool) -> 'a -> 'a t`

Reject the input (raises `Noparse`) if the first character of the input passed the test. Does not read the character if the test fails. `name` default to `"^"` prepended to the result of `Charset.show`.

`val not_charset : ?name:string -> Charset.t -> 'a -> 'a t`

Reject the input (raises `Noparse`) if the first character of the input is in the charset. Does not read the character if not in the charset. `name` default as in `not_test`

`val seq : ?name:string -> 'a t -> 'b t -> ('a -> 'b -> 'c) -> 'c t`

Compose two terminals in sequence. `name` default is the concatenation of the two names.

`val seq1 : ?name:string -> 'a t -> 'b t -> 'a t`

`val seq2 : ?name:string -> 'a t -> 'b t -> 'b t`

`val alt : ?name:string -> 'a t -> 'a t -> 'a t`

`alt t1 t2` parses the input with `t1` or `t2`. Contrary to grammars, terminals does not use continuations, if `t1` succeds, no backtrack will be performed to try `t2`. For instance, `seq1 (alt (char 'a' ()) (seq1 (char 'a' ()) (char 'b' ()))) (char 'c' ())` will reject "abc". If both `t1` and `t2` accept the input, longuest match is selected. `name` default to `sprintf "(%s)|(%s)" t1.n t2.n`.

`val option : ?name:string -> 'a -> 'a t -> 'a t`

`option x t` parses the given terminal 0 or 1 time. `x` is returned if 0. `name` defaults to `sprintf "(%s)?" t.n`.

`val appl : ?name:string -> ('a -> 'b) -> 'a t -> 'b t`

Applies a function to the result of the given terminal. `name` defaults to the terminal name.

`val star : ?name:string -> 'a t -> (unit -> 'b) -> ('b -> 'a -> 'b) -> 'b t`

`star t a f` Repetition of a given terminal 0,1 or more times. The type of function to compose the action allows for `'b = Buffer.t` for efficiency. The returned value is `f ( ... (f(f (a ()) x_1) x_2) ...)  x_nif` `t` returns `x_1` ... `x_n`. The `name` defaults to `sprintf "(%s)*" t.n`

`val plus : ?name:string -> 'a t -> (unit -> 'b) -> ('b -> 'a -> 'b) -> 'b t`

Same as above but parses at least once .

`val string : ?name:string -> string -> 'a -> 'a t`

`string s` Accepts only the given string. Raises `Invalid_argument` if `s = ""`. `name` defaults to `sprintf "%S" s`.

`val int : ?name:string -> unit -> int t`

Parses an integer in base 10. `"+42"` is accepted. `name` defaults to `"INT"`

`val float : ?name:string -> unit -> float t`

Parses a float in base 10. `".1"` is not accepted `"0.1"` is. `name` defaults to `"FLOAT"`

`val keyword : ?name:string -> string -> (char -> bool) -> 'a -> 'a t`

`keyword ~name k cs x = seq ~name (string k ()) (test f ()) (fun _ _ -> x)` usefull to accept a keyword only when not followed by an alpha-numeric char

`val regexp : ?name:string -> Regexp.t -> string t`

create a terminal from a regexp. Returns the whole matched string

`val regexp_grps : ?name:string -> Regexp.t -> string list t`

create a terminal from a regexp. Returns the groups list, last to finish to be parsed is first in the result

3.3 Functions managing blanks

`val noblank : blank`

Use when you have no blank chars

`val blank_charset : Charset.t -> blank`

Blank from a charset

`val blank_terminal : 'a t -> blank`

Blank from a terminal

`val accept_empty : 'a t -> bool`

Test wether a terminal accept the empty string. Such a terminal are illegal in a grammar, but may be used in combinator below to create terminals

4 Module `Regexp` : A small module for efficient regular expressions.

```
type regexp =
  | Chr of char
  | Set of Charset.t
  | Seq of regexp list
  | Alt of regexp list
  | Opt of regexp
  | Str of regexp
  | Pls of regexp
  | Sav of regexp
```

Type of a regular expression.

```
type t = regexp
```

```
exception Regexp_error of Input.buffer * int
```

Exception that is raised when a regexp cannot be read.

```
val print : Stdlib.out_channel -> regexp -> unit
```

```
val accept_empty : regexp -> bool
```

```
val accepted_first_chars : regexp -> Charset.t
```

```
val from_string : string -> regexp
```

```
val read : regexp -> Input.buffer -> int -> string list * Input.buffer * int
```

`read re buf pos` attempts to parse using the buffer `buf` at position `pos` using the regular expression `re`. The return value is a triple of the parsed string, the buffer after parsing and the position after parsing. The exception `Regexp_error(err_buf, err_pos` is raised in case of failure at the given position.

5 Module `Input` : A module providing efficient input buffers with preprocessing.

5.1 Type

```
type buffer
```

The abstract type for an input buffer.

5.2 Reading from a buffer

```
val read : buffer -> int -> char * buffer * int
```

`read buf pos` returns the character at position `pos` in the buffer `buf`, together with the new buffer and position.

`val sub : buffer -> int -> int -> string`

`sub b i len` returns `len` characters from position `pos`. If the end of buffer is reached, the string is filed with eof '\255'

`val get : buffer -> int -> char`

`get buf pos` returns the character at position `pos` in the buffer `buf`.

5.3 Creating a buffer

`val from_file : string -> buffer`

`from_file fn` returns a buffer constructed using the file `fn`.

`val from_channel : ?filename:string -> Stdlib.in_channel -> buffer`

`from_channel ~filename ch` returns a buffer constructed using the channel `ch`. The optional `filename` is only used as a reference to the channel in error messages.

`val from_string : ?filename:string -> string -> buffer`

`from_string ~filename str` returns a buffer constructed using the string `str`. The optional `filename` is only used as a reference to the channel in error messages.

`val from_fun : ('a -> unit) -> string -> ('a -> string) -> 'a -> buffer`

`from_fun finalise name get data` returns a buffer constructed from the object `data` using the `get` function. The get function is used to obtain one line of input from `data`. The `finalise` function is applied to `data` when the end of file is reached. The `name` string is used to reference the origin of the data in error messages.

`exception Preprocessor_error of string * string`

Exception that can be raised by a preprocessor in case of error. The first string references the name of the buffer (e.g. the name of the corresponding file) and the second string contains the message.

`val pp_error : string -> string -> 'a`

`pp_error name msg` raises `Preprocessor_error(name,msg)`.

```
module type Preprocessor =
  sig
```

`type state`

Type for the internal state of the preprocessor.

`val initial_state : state`

Initial state of the preprocessor.

```
val update :
  state ->
  string -> int -> string -> state * string * int * bool
```

`update st name lnum line` takes as input the state `st` of the preprocessor, the file name `name`, the number of the next input line `lnum` and the next input line `line` itself. It returns a tuple of the new state, the new file name, the new line number, and a boolean. The new file name and line number can be used to implement line number directives. The boolean is `true` if the line should be part of the input (i.e. it is not a specific preprocessor line) and `false` if it should be ignored. The function may raise `Preprocessor_error` in case of error.

```
val check_final : state -> string -> unit
```

`check_final st name` check that `st` indeed is a correct state of the preprocessor for the end of input of file `name`. If it is not the case, then the exception `Preprocessor_error` is raised.

```
end
```

Specification of a preprocessor.

```
module WithPP :
  functor (PP : Preprocessor) ->   sig
```

```
val from_fun : ('a -> unit) -> string -> ('a -> string) -> 'a -> Input.buffer
```

Same as `Input.from_fun` but uses the preprocessor.

```
val from_channel : ?filename:string -> Stdlib.in_channel -> Input.buffer
```

Same as `Input.from_channel` but uses the preprocessor.

```
val from_file : string -> Input.buffer
```

Same as `Input.from_file` but uses the preprocessor.

```
val from_string : ?filename:string -> string -> Input.buffer
```

Same as `Input.from_string` but uses the preprocessor.

```
end
```

Functor for building buffers with a preprocessor.

5.4 Buffer manipulation functions

`val is_empty : buffer -> int -> bool`

`is_empty buf` test whether the buffer `buf` is empty.

`val line_num : buffer -> int`

`line_num buf` returns the current line number of `buf`.

`val line_offset : buffer -> int`

`line_beginning buf` returns the offset of the current line in the buffer `buf`.

`val line : buffer -> string`

`line buf` returns the current line in the buffer `buf`.

`val line_length : buffer -> int`

`line_length buf` returns the length of the current line in the buffer `buf`.

`val utf8_col_num : buffer -> int -> int`

`utf8_col_num buf pos` returns the utf8 column number corresponding to the position `pos` in `buf`.

`val normalize : buffer -> int -> buffer * int`

`normalize buf pos` ensures that `pos` is less than the length of the current line in `str`.

`val filename : buffer -> string`

`filename buf` returns the file name associated to the `buf`.

`val buffer_uid : buffer -> int`

`buffer_uid buf` returns a unique identifier for `buf`.

`val buffer_equal : buffer -> buffer -> bool`

`buffer_eq b1 b2` tests the equality of `b1` and `b2`.

`val buffer_compare : buffer -> buffer -> int`

`buffer_compare b1 b2` compares `b1` and `b2`.

`val buffer_before : buffer -> int -> buffer -> int -> bool`

`leq_bug b1 i1 b2 i2` returns true if the position `b1, i1` is before `b2, i2`. Gives meaningless result if `b1` and `b2` do not refer to the same file.

```
module Tbl :
  sig
```

```
    type 'a t
    val create : unit -> 'a t
    val add : 'a t -> Input.buffer -> int -> 'a -> unit
    val find : 'a t -> Input.buffer -> int -> 'a
    val clear : 'a t -> unit
    val iter : 'a t -> ('a -> unit) -> unit
  end
```

Table to associate value to positions in input buffers

6 Module `Position` : Functions managing positions

```
type pos =
{  name : string ;
```

file's name

```
  line : int ;
```

line number

```
  col : int ;
```

column number

```
  utf8_col : int ;
```

column number with unicode

```
  phantom : bool ;
```

is the postion a "phantom", i.e. not really in the file

```
}
```

Type to represent position

```
type t = pos
```

Abbreviation

```
val phantom : pos
```

a phantom position, used for grammar accepting the empty input

```
val max_pos : pos -> pos -> pos
```

the max of to position (further in the file

```
val compute_utf8_col : bool Stdlib.ref
```

if false (the default) `utf8_col` field is set to `-1` by `get_pos`

```
val get_pos : Input.buffer -> int -> pos
```

Get a position from an input buffer and a column number

7 Module `Earley` : Earley compatible interface (UNFINISHED)

Earley is a parser combinator library implemented using the Earley algorithm. This modules is an UNFINISHED WORK to provide an Earley compatible interface to Pacomb

7.1 Types and exceptions

`type 'a grammar`

Type of a parser (or grammar) producing a value of type `'a`.

`type blank = Input.buffer -> int -> Input.buffer * int`

As `Earley` does scannerless parsing, a notion of `blank` function is used to discard meaningless parts of the input (e.g. comments or spaces). A `blank` function takes as input a `buffer` and a position (represented as an `int`) and returns a couple of a `buffer` and a position corresponding to the next meaningful character.

WARNING: a blank function must return a normalized pair (b,p), which means $0 \leq p <$ Input.line_num b. You can use Input.normalize to ensure this.

`exception Parse_error of Input.buffer * int`

The exception `Parse_error(buf,pos,msgs)` is raised whenever parsing fails. It contains the position `pos` (and the corresponding buffer `buf`) of the furthest reached position in the input.

`val give_up : unit -> 'a`

`give_up ()` can be called by the user to force the parser to reject a possible parsing rule.

`val handle_exception : ?error:(unit -> 'b) -> ('a -> 'b) -> 'a -> 'b`

`handle_exception fn v` applies the function `fn` to `v` and handles the `Parse_error` exception. In particular, a parse error message is presented to the user in case of a failure, then `error ()` is called. The default `error` is `fun () -> exit 1`.

7.2 Atomic parsers

`val char : ?name:string -> char -> 'a -> 'a grammar`

`char ~name c v` is a grammar that accepts only the character `c`, and returns `v` as a semantic value. An optional `name` can be given to the grammar for reference in error messages.

`val string : ?name:string -> string -> 'a -> 'a grammar`

`string s v` is a grammar that accepts only the string `str`, and returns `v` as a semantic value. An optional `name` can be given to the grammar for reference in error messages.

`val keyword : ?name:string -> string -> (char -> bool) -> 'a -> 'a grammar`

`keyword s forbidden v` is simalar to string, but the parsing fails if `forbidden c` returns `true` when `c` is the next available character.

`val eof : 'a -> 'a grammar`

> `eof v` is a grammar that only accepts the end of file and returns `v` as a semantic value. Note that the end of file can be parsed one or more times (i.e. the input ends with infinitely many end of file symbols.

`val any : char grammar`

> `any` is a grammar that accepts a single character (but fails on the end of file) and returns its value.

`val in_charset : ?name:string -> Charset.charset -> char grammar`

> `in_charset cs` is a grammar that parses any character of the `cs` charset, and returns its value. An optional `name` can be given to the grammar for reference in error messages.

`not_in_charset cs` is similar to `in_charset cs` but it accepts the characters that are not in `cs`.

`blank_not_in_charset cs` is the same as `not_in_charset` but testing with blank_test.

`val empty : 'a -> 'a grammar`

> `empty v` is a grammar that does not parse anything and returns `v` as a semantic value. Note that this grammar never fails.

`type 'a fpos = Input.buffer -> int -> Input.buffer -> int -> 'a`

> type for a function waiting for the start and end positions (i.e. buffer and index) of an item, in general resulting from parsing

`empty_pos v` is similar to the above except that the action wait for the position of a complete sequence build using `fsequence` of `sequence`.

For instance, `sequence_position g1 g2 f` below can be defined as `fsequence g1 (fsequence g2 (empty_pos f'))`. where `f' = fun b p b' p' a2 a1 = f b p b' p' a1 a2` to give the result of g1 and g2 in the expected order.

`val fail : unit -> 'a grammar`

> `fail ()` is a grammar that always fail, whatever the input.

`black_box fn cs accept_empty name` is a grammar that uses the function `fn` to parses the input buffer. `fn buf pos` should start parsing `buf` at position `pos`, and return a couple containing the new buffer and position of the first unread character. The character set `cs` must contain at least the characters that are accepted as first character by `fn`, and no less. The boolean `accept_empty` must be true if the function accept the empty string. The `name` argument is used for reference in error messages. Note that the functon `fn` should use `give_up ()` in case of a parse error.

WARNING: fn must return a triple (x,b,p) when (b,p) is normalized, which means $0 \leq p <$ Input.line_num b. You can use Input.normalize to ensure this.

`debug msg` is a dummy grammar that always succeeds and prints `msg` on `stderr` when used. It is useful for debugging.

`val regexp : ?name:string -> string -> string array grammar`

> `regexp ?name re` is a grammar that uses the regexp `re` to parse the input buffer. The value returnes is the array of the contents of the groups.

7.3 Blanks management

```
val no_blank : blank
```

no_blank is a blank function that does not discard any character of the input buffer.

blank_regexp re builds a blank from the regexp re.

blank_grammar gr bl produces a blank function using the grammar gr and the blank function bl. It parses as much of the input as possible using the grammar gr with the blank function bl, and returns the reached position.

change_layout ~old_blank_before ~new_blank_after gr bl replaces the current blank function with bl, while parsing using the grammar gr. The optional parameter old_blank_before (true by default) forces the application of the old blank function, before starting to parse with gr. Note that the new blank function is always called before the first terminal of gr. Similarly, the opt- -ional parameter new_blank_after (true by default) forces a call to the new blank function after the end of the parsing of gr. Note that the old blank function is always called after the last terminal.

change_layout ~oba gr bl same as abobe but with no blank. It keeps the first char prediction and is therefore more efficient

7.4 Support for recursive grammars

```
val declare_grammar : string -> 'a grammar
```

declare_grammar name returns a new grammar that can be used in the definition of other grammars, but that cannot be run on input before it has been initialized with set_grammar. The name argument is used for reference to the grammar in error messages.

```
val set_grammar : 'a grammar -> 'a grammar -> unit
```

set_grammar gr grdef set the definiton of grammar gr (previously declared with declare_grammar) to be grdef. Invalid_argument is raised if set_grammar is used on a grammar that was not created with declare_grammar. The behavious is undefined if a grammar is set twice with set_grammar.

7.5 Parsing functions

```
val parse_buffer : 'a grammar -> blank -> Input.buffer -> 'a
```

parse_buffer gr bl buf parses the buffer buf using the grammar gr and the blank function bl. The exception Parse_error may be raised in case of error.

```
val parse_string : ?filename:string -> 'a grammar -> blank -> string -> 'a
```

parse_string ~filename gr bl str parses the string str using the grammar gr and the blank function bl. An optional filename can be provided for reference to the input in error messages. The exception Parse_error may be raised in case of error.

```
val parse_channel :
  ?filename:string ->
  'a grammar -> blank -> Stdlib.in_channel -> 'a
```

`parse_channel ~filename gr bl ch` parses the contenst of the input channel `ch` using the grammar `gr` and the blank function `bl`. A `filename` can be provided for reference to the input in case of an error. `parse_channel` may raise the `Parse_error` exception.

```
val parse_file : 'a grammar -> blank -> string -> 'a
```

`parse_file gr bl fn` parses the file `fn` using the grammar `gr` and the blank function `bl`. The exception `Parse_error` may be raised in case of error.

```
val partial_parse_buffer :
  'a grammar ->
  blank ->
  ?blank_after:bool -> Input.buffer -> int -> 'a * Input.buffer * int
```

`partial_parse_buffer gr bl buf pos` parses input from the buffer `buf` starting a position `pos`, using the grammar `gr` and the blank function `bl`. A triple is returned containing the new buffer, the position that was reached during parsing, and the semantic result of the parsing. The optional argument `blank_after`, `true` by default, indicates if the returned position if after the final blank or not. Note that this function should not be used in the defi- nition of a grammar using the `black_box` function.

A functor providing support for using and `Input` preprocessor.

7.6 Debuging and flags

```
val debug_lvl : int Stdlib.ref
```

`debug_lvl` is a flag that can be set for `Earley` to display debug data on `stderr`. The default value is `0`, and bigger numbers acti- vate more and more debuging informations.

```
val warn_merge : bool Stdlib.ref
```

`warn_merge` is a flag that is used to choose whether warnings are displayed or not when an ambiguity is encountered while parsing. The default value is `true`.

`keep_all_names` is false by default and allow for inlining grammar with a name to optimise parsing. When debugging, it is possible to set it to true (before all grammar constructions) for more accurate messages.

7.7 Greedy combinator

```
val greedy : 'a grammar -> 'a grammar
```

`greedy g` parses g in a greedy way: only the longest match is considered. Still ambigous if the longest match is not unique

7.8 Sequencing combinators

```
val sequence : 'a grammar ->
  'b grammar -> ('a -> 'b -> 'c) -> 'c grammar
```

`sequence g1 g2 f` is a grammar that first parses using `g1`, and then parses using `g2`. The results of the sequence is then obtained by applying `f` to the results of `g1` and `g2`.

`sequence_position g1 g2 f` is a grammar that first parses using `g1`, and then parses using `g2`. The results of the sequence is then obtained by applying `f` to the results of `g1` and `g2`, and to the positions (i.e. buffer and index) of the corresponding parsed input.

Remark: `sequence g1 g2 f` is equivalent to `sequence_position g1 g2 (fun _ _ _ _ -> f)`.

```
val fsequence : 'a grammar -> ('a -> 'b) grammar -> 'b grammar
```

`fsequence g1 g2` is a grammar that first parses using `g1`, and then parses using `g2`. The results of the sequence is then obtained by applying the result of `g1` to the result of `g2`.

Remark: `fsequence g1 g2` is equivalent to `sequence g1 g2 (fun x f -> f x)`.

same as fsequence, but the result of `g2` also receive the position of the result of `g1`.

```
val fsequence_ignore : 'a grammar -> 'b grammar -> 'b grammar
```

same as fsequence, but the result of `g2` receives nothing, meaning we forget the result of `g1`.

```
val sequence3 :
  'a grammar ->
  'b grammar ->
  'c grammar -> ('a -> 'b -> 'c -> 'd) -> 'd grammar
```

`sequence3` is similar to `sequence`, but it composes three grammars into a sequence.

Remark: `sequence3 g1 g2 g3 f` is equivalent to `sequence (sequence g1 g2 f) g3 (fun f x -> f x)`.

```
val simple_dependent_sequence :
  'a grammar -> ('a -> 'b grammar) -> 'b grammar
```

`simple_dependent_sequence g1 g2` is a grammar that first parses using `g1`, which returns a value `a`, and then continues to parse with `g2 a` and return its result.

`dependent_sequence g1 g2` is a grammar that first parses using `g1`, which returns a value `(a,b)`, and then continues to parse with `g2 a` and return its result applied to `b`. compared to the above function, allow memoizing the second grammar

= fun g → dependent_sequence g (fun x → x)

`option v g` tries to parse the input as `g`, and returns `v` in case of failure.

```
val fixpoint : 'a -> ('a -> 'a) grammar -> 'a grammar
```

`fixpoint v g` parses a repetition of one or more times the input parsed by `g`. The value `v` is used as the initial value (i.e. to finish the sequence).

if parsing X with g returns a function gX, parsing X Y Z with fixpoint a g will return gX (gY (gZ a)).

This consumes stack proportinal to the input length ! use revfixpoint ...

as `fixpoint` but parses at leat once with the given grammar

`listN g sep` parses sequences of `g` separated by `sep` of length at least `N`, for `N=0,1` or `2`.

```
val alternatives : 'a grammar list -> 'a grammar
```

alternatives [g1;...;gn] tries to parse using all the grammars [g1;...;gn] and keeps only the first success.

```
val apply : ('a -> 'b) -> 'a grammar -> 'b grammar
```

apply f g applies function f to the value returned by the grammar g.

apply_position f g applies function f to the value returned by the grammar g and the positions at the beginning and at the end of the input parsed input.

position g tranforms the grammar g to add information about the position of the parsed text.

test c f perform a test f on the input buffer. Do not parse anything (position are unchanged). The charset c should contains all character accepted as at the position given to f

blank_test c f same as above except that f is applied to buf' pos' buf pos where (buf', pos') is the position before the blank. The charset c should contains all character accepted as at the position (buf,pos). This allow to test the presence of blank or even to read the blank and return some information

a test that fails if there is no blank

a test that fails if there are some blank

```
val grammar_family :
  ?param_to_string:('a -> string) ->
  string -> ('a -> 'b grammar) * (('a -> 'b grammar) -> unit)
```

grammar_family to_str name returns a pair (gs, set_gs), where gs is a finite family of grammars parametrized by a value of type 'a. A name name is to be provided for the family, and an optional function to_str can be provided to print the parameter and display better error messages.

```
(* Declare the grammar family *)
let (gr, set_gr) = grammar_family to_str name in

... code using grammars of gr to define mutually recursive grammars ...
... the grammars in gr cannot be used in "left position" ...
... (same restriction as for declare_grammar ...

(* Define the grammar family *)
let _ = set_gr the_grammars

... now the new family can be used ...
```

```
val grammar_prio :
  ?param_to_string:('b -> string) ->
  string ->
  ('b -> 'c grammar) *
  ((('b -> bool) * 'c grammar) list * ('b -> 'c grammar list) ->
   unit)
```

Similar to the previous one, with an optimization. `grammar_prio to_str name` returns a pair `(gs, set_gs)`, where `gs` is a finite family of grammars parametrized by a value of type `'a`. `set_gs` requires two lists of grammars to set the value of the grammar:

- the first list are grammar that can only be activated by the parameter (if the given function return true)
- the second list is used as for grammar family

```
val grammar_prio_family :
  ?param_to_string:('a * 'b -> string) ->
  string ->
  ('a -> 'b -> 'c grammar) *
  (('a ->
    (('b -> bool) * 'c grammar) list * ('b -> 'c grammar list)) ->
   unit)
```

A mixture of the two above

```
val accept_empty : 'a grammar -> bool
```

`accept_empty g` returns `true` if the grammar `g` accepts the empty input and `false` otherwise.

```
val grammar_info : 'a grammar -> bool * Charset.t
val give_name : string -> 'a grammar -> 'a grammar
```

give a name to the grammar. Usefull for debugging.