Pfff: Parsing PHP

Programmer's Manual and Implementation

Yoann Padioleau
yoann.padioleau@facebook.com

February 23, 2010

Short Contents

Contents

Chapter 1

Introduction

1.1 Why another PHP parser ?

pfff (PHP Frontend For Fun) is mainly an OCaml API to write static analysis or style-preserving source-to-source transformations such as refactorings on PHP source code. It is inspired by a similar tool for C called Coccinelle [11, 12]. [1]

The goal of pfff is to parse the code as-is, and to represent it internally as-is. We thus maintain in the Abstract Syntax Tree (AST) as much information as possible so that one can transform this AST and unparse it in a new file while preserving the coding style of the original file. pfff preserves the whitespaces, newlines, indentation, and comments from the original file. The pfff abstract syntax tree is thus in fact more a Concrete Syntax Tree (cf `parsing_php/ast_php.mli` and Chapter 4).

There are already multiple parsers for PHP:

- The parser included in the official Zend PHP distribution. This includes a PHP tokenizer that is accessible through PHP, see http://www.php.net/manual/en/tokenizer.examples.php. [2]
- The parser in HPHP source code, derived mostly from the previous parser.
- The parser in PHC source code.
- The parser in Lex-pass, a PHP refactoring tool by Daniel Corson.
- Partial parser hacks (ab)using the PHP tokenizer. [3]

Most of those parsers are written in C/C++ using Lex and Yacc (actually Flex/Bison). The one in Lex-pass is written in Haskell using parser combinators.

[1] FACEBOOK: and maybe one day HPHP2 . . .

[2] FACEBOOK: This tokenizer is used by Mark Slee `www/flib/_bin/checkModule` PHP script.

[3] FACEBOOK: For instance `www/scripts/php_parser/`, written by Lucas Nealan.

I decided to write yet another PHP parser, in OCaml, because I think OCaml is a better language to write compilers or static analysis tools (for bugs finding, refactoring assistance, type inference, etc) and that writing a PHP parser is the first step in developing such tools for PHP.

Note that as there is a Lex and Yacc for OCaml (called `ocamllex` and `ocamlyacc`), I was able to copy-paste most of the PHP Lex and Yacc specifications from the official PHP parser (see `pfff/docs/official-grammar/`). It took me about a week-end to write the first version of pfff.

1.2 Features

Here is a list of the main features provided by pfff:

- A full-featured PHP AST using OCaml powerful Algebric Data Types (see http://en.wikipedia.org/wiki/Algebraic_data_type)
- Position information for all tokens, in the leaves of the AST
- Visitors genertor
- Pretty printing of the AST data structures
- Support for calling PHP preprocessors (e.g. XHP)
- Partial support of XHP extensions directly into the AST (by not calling the XHP preprocessor but parsing as-is XHP files) [4]

Note that this manual documents only the parser frontend part of pfff (the `pfff/parsing_php/` directory). Another manual describes the static analysis features of pfff (the `pfff/analysis_php/` directory) including support for control-flow and data-flow graphs, caller/callee graphs, module dependencies, type inference, source-to-source transformations, PHP code pattern matching, etc.

1.3 Copyright

The source code of pfff is governed by the following copyright:

9 ⟨*Facebook copyright* 9⟩≡

```
(* Yoann Padioleau
 *
 * Copyright (C) 2009-2010 Facebook
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License (GPL)
 * version 2 as published by the Free Software Foundation.
```

[4] FACEBOOK: really partial for the moment

```
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * file license.txt for more details.
 *)
```


1.4 Getting started

1.4.1 Requirements

pfff is an OCaml library so you need obviously to install both the runtime and the development libraries for OCaml. Here is the list of packages needed by pfff:

- OCaml (see http://caml.inria.fr/download.en.html)
- GNU make (see http://www.gnu.org/software/make/)

Those packages are usually available on most Linux distributions. For instance on CentOS simply do:

```
$ sudo yum install ocaml
$ sudo yum install make
```

[5]

1.4.2 Compiling

The source of pfff are available at http://padator.org/software/project-pfff/. [6]

To compile pfff, see the instructions in `install.txt`. It should mainly consists in doing:

```
$ cd <pfff_src_directory>
$ ./configure
$ make depend
$ make
```

If you want to embed the parsing library in your own OCaml application, you have just to copy the `parsing_php/` and `commons/` directories in your own project directory, add a recursive make that goes in those directories, and then link your application with the `parsing_php/parsing_php.cma` and `commons/commons.cma` library files (see also `pfff/demos/Makefile`).

[5] FACEBOOK: OCaml is also already installed in `/home/pad/packages/bin` so you just have to `source env.sh` from the pfff source directory

[6] FACEBOOK: The source of pfff are currently managed by git. to git it just do `git clone /home/engshare/git/projects/pfff`

1.4.3 Quick example of use

Once the source are compiled, you can test pfff with:

```
$ cd demos/
$ ocamlc -I ../commons/ -I ../parsing_php/ \
    ../commons/commons.cma ../parsing_php/parsing_php.cma \
    show_function_calls1.ml -o show_function_calls
$ ./show_function_calls foo.php
```

You should then see on stdout some information on the function calls in `foo.php` according to the code in `show_function_calls1.ml` (see Section 2.1.3 for a step-by-step explanation of this program).

1.4.4 The `pfff` command-line tool

The compilation process, in addition to building the `parsing_php.cma` library, also builds a binary program called `pfff` that can let you evaluate among other things how good the pfff parser is. For instance, to test the parser on the PhpBB (`http://www.phpbb.com/`, a popular internet forum package written in PHP) source code, just do:

```
$ cd /tmp
$ wget http://d10xg45o6p6dbl.cloudfront.net/projects/p/phpbb/phpBB-3.0.6.tar.bz2
$ tar xvfj phpBB-3.0.6.tar.bz2
$ cd <pfff_src_directory>
$ ./pfff -parse_php /tmp/phpBB3/
```

The `pfff` program should then iterate over all PHP source code files (`.php` files), and run the parser on each of those files. At the end, `pfff` will output some statistics showing what pfff was not able to handle. On the PhpBB source code the messages are:

```
PARSING: /tmp/phpBB3/posting.php
PARSING: /tmp/phpBB3/cron.php
...
---------------------------------------------------------------
NB total files = 265; perfect = 265; =========> 100%
nb good = 183197,  nb bad = 0 =========> 100.000000%
...
```

meaning pfff was able to parse 100% of the code. [7]

[7] FACEBOOK: For the moment pfff parse 97% of the code in `www`. The remaining errors are in files using XHP extensions that the parser does not yet handle.

1.5 Source organization

Table 1.1 presents a short description of the modules in the `parsing_php/` directory of the pfff source distribution as well as the corresponding chapters the module is discussed.

Function	Chapter	Modules
Parser entry point	3	`parse_php.mli`
Abstract Syntax Tree	4 4.10	`ast_php.mli` `type_php.mli`, `scope_php.mli`
Visitor	5	`visitor_php.mli`

Unparsing	6.1
	6.3
	6.5
		`sexp_ast_php.mli`
		`json_ast_php.mli`
		`unparse_php.mli`

Other services	7.1
	7.2
	7.3
		`lib_parsing_php.mli`
		`flag_parsing_php.mli`
		`test_parsing_php.mli`

Parser code	8
	9
	9.9
	10
	10.13
		`parse_php.ml`
		`lexer_php.mll` (Lex specification))
		`token_helpers_php.ml`
		`parser_php.mly` (Yacc specification)
		`parser_php_mly_helper.ml`

Table 1.1: Chapters and modules

1.6 API organization

Figure 1.1 presents the graph of dependencies between `.mli` files.

1.7 Plan

Part 1 explains the interface of pfff, that is mainly the `.mli` files. Part 2 explains the code, the `.ml` files.

1.8 About this document

This document is a literate program [1]. It is generated from a set of files that can be processed by tools (Noweb [2] and syncweb [3]) to generate either this manual or the actual source code of the program. So, the code and its documentation are strongly connected.

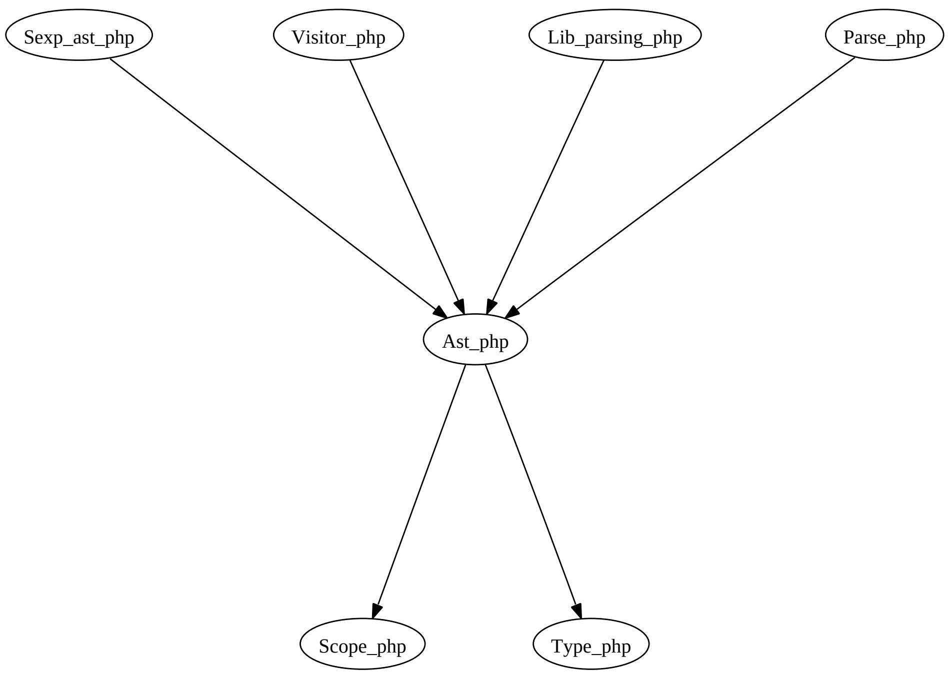

Figure 1.1: API dependency graph between `mli` files

Part I

Using pfff

Chapter 2

Examples of Use

This chapter describes how to write OCaml programs, to be linked with the `parsing_php.cma` library, to perform some simple PHP analysis.

2.1 Function calls statistics

The goal of our first example using the pfff API is to print some information about function calls in a PHP program.

2.1.1 Basic version

Here is the toplevel structure of `pfff/demos/show_function_calls1.ml`:

15 ⟨*show_function_calls1.ml* 15⟩≡

```
⟨basic pfff modules open 16a⟩

⟨show_function_calls v1 16b⟩

let main =
  show_function_calls Sys.argv.(1)
```

To compile and test do:

```
$ cd demos/
$ ocamlc -I ../commons/ -I ../parsing_php/ \
    ../commons/commons.cma ../parsing_php/parsing_php.cma \
    show_function_calls1.ml -o show_function_calls
$ ./show_function_calls foo.php
```

You should then see on stdout some information on the function calls in `foo.php` (binded to `Sys.argv.(1)` in the previous code):

```
Call to foo  at line 11
Call to foo2 at line 12
```

We now describe gradually the different parts of this program. We first open some modules:

16a ⟨*basic pfff modules open* 16a⟩≡
```
open Common
open Ast_php
```

Normally you should avoid the use of `open` directives in your program, as it makes the program more complicated to understand, except for very common libraries, or when your program predominantely uses a single module defining lots of types (which is the case here with `Ast_php` as you will see later).

The `Common` module is not part of the standard OCaml library. It is a library I have developed (see [7] for its full documentation) in the last 10 years or so. It defines many functions not provided by default in the standard OCaml library but are standard in other programming languages (e.g. Haskell, Scheme, F#).

16b ⟨*show_function_calls v1* 16b⟩≡
```
let show_function_calls file =
  let (asts2, _stat) = Parse_php.parse file in
  let asts = Parse_php.program_of_program2 asts2 in

  ⟨iter on asts manually 16c⟩
```

The `Parse_php.parse` function returns in addition to the AST some statistics and extra information attached to each toplevel construct in the program (see Chapter 3). The `Parse_php.program_of_program2` function trims down those extra information to get just the AST.

We are now ready to visit the AST:

16c ⟨*iter on asts manually* 16c⟩≡
```
  asts |> List.iter (fun toplevel ->

    match toplevel with
    | StmtList stmts ->
        ⟨iter on stmts 17a⟩

    | (FuncDef _|ClassDef _|InterfaceDef _|Halt _
        |NotParsedCorrectly _| FinalDef _)
      -> ()
  )
```

The `show_function_calls1.ml` program will just process the toplevel statements in a PHP file, here represented by the AST constructor `StmtList` (see Section 4.7), and will ignore other constructions such as function definitions (`FuncDef`), classes (`ClassDef`), etc. The next section will present a better algorithm processing (visiting) all constructions.

The `|>` operator is not a standard operator. It's part of `Common`. Its semantic is: `data |> f` ≡ `f data`, which allows to see first the data and then the function that will operate on the data. This is useful when the function is a long anonymous block of code. For instance in the previous code, `asts |> List.iter (fun ...)` ≡ `List.iter (fun ...) asts`. It is somehow reminescent of object oriented style.

We will now go deeper into the AST to process all toplevel function calls:

17a ⟨*iter on stmts* 17a⟩≡
```
stmts |> List.iter (fun stmt ->
  (match stmt with
  | ExprStmt (e, _semicolon) ->

      (match Ast_php.untype e with
      | ExprVar var ->

          (match Ast_php.untype var with
          | FunCallSimple (qu_opt, funcname, args) ->
              ⟨print funcname 17b⟩
          | _ -> ()
          )
      | _ -> ()
      )
  | _ -> ()
  )
)
```

The `Ast_php.untype` function is an "extractor" used to abstract away the type information attached to parts of the AST (expressions and variables, see Section 4.10.1 and Section 4.15). The `ExprStmt`, `ExprVar` and `FunCallSimple` are constructors explained respectively in Section 4.4.1, 4.2, and 4.3.3.

Now that we have matched the function call site, we can finally print information about it:

17b ⟨*print funcname* 17b⟩≡
```
let s = Ast_php.name funcname in
let info = Ast_php.info_of_name funcname in
let line = Ast_php.line_of_info info in
pr2 (spf "Call to %s at line %d" s line);
```

The type of the `funcname` variable is not `string` but `name`. This is because we want not only the content of an identifier, but also its position in the source file (see Section 4.8 and 4.9). The `Ast_php.name`, `Ast_php.info_of_name` and `Ast_php.line_of_info` functions are extractors, to get respectively the content, some position information, and the line position of the identifier.

The function `pr2` is also part of `Common`. It's for printing on stderr (stderr is usually bound to file descriptor 2, hence `pr2`). `spf` is an alias for `Printf.sprintf`.

2.1.2 Using a visitor

The previous program was printing information only about function calls at the toplevel. For instance on this program

18a ⟨*foo2.php* 18a⟩≡
```
<?php
function foo($a) {
  bar($a);
}
function bar($a) {
  echo $a;
}
foo("hello world");
?>
```
the output will be:

```
$ ./show_function_calls1 foo2.php
Call to foo at line 8
```

which does not include the call to `bar` nested in the function definition of `foo`.

Processing `StmtList` is not enough. Nevertheless manually specifying all the cases is really tedious, especially as `Ast_php` defines more than 100 constructors, spreaded over more than 5 types. A common solution to this kinds of a problem is to use the Visitor design pattern (see `http://en.wikipedia.org/wiki/Visitor_pattern` and [9, 10]) that we have adapted for pfff in OCaml in the `Visitor_php` module (see Chapter 5).

Here is the new `pfff/demos/show_function_calls2.ml` program:

18b ⟨*show_function_calls2.ml* 18b⟩≡
```

⟨basic pfff modules open 16a⟩
module V = Visitor_php

⟨show_function_calls v2 18c⟩

let main =
  show_function_calls Sys.argv.(1)
```
The module aliasing of `V` allows to not use the evil `open` while still avoiding to repeat long names in the code.

As before a first step is to get the ASTs:

18c ⟨*show_function_calls v2* 18c⟩≡
```
let show_function_calls file =
  let (asts2, _stat) = Parse_php.parse file in
  let asts = Parse_php.program_of_program2 asts2 in

  ⟨create visitor 19a⟩
  ⟨iter on asts using visitor 19c⟩
```

We are now ready to visit:

19a ⟨*create visitor* 19a⟩≡
```
let visitor = V.mk_visitor
 { V.default_visitor with
   V.klvalue = (fun (k, _) var ->

    match Ast_php.untype var with
    | FunCallSimple (qu_opt, funcname, args) ->
        ⟨print funcname 17b⟩

    | _ ->
        ⟨visitor recurse using k 19b⟩
  );
}
in
```

The previous code may look a little bit cryptic. For more discussions about visitors and visitors in OCaml see Chapter 5. The trick is to first specify *hooks* on certain constructions, here the `klvalue` hook that will be called at each lvalue site, and to specify a default behavior for the rest (the `V.default_visitor`). Note that in the PHP terminology, function calls are part of the `lvalue` type which is a restricted form of expressions (see Section4.3.3), hence the use of `klvalue` and not `kexpr`. One can also use the `kstmt`, `kinfo`, and `ktoplevel` hooks (and more).

The use of the prefix `k` is a convention used in Scheme to represent continations (see http://en.wikipedia.org/wiki/Continuation) which is somehow what the `Visitor_php` module provides. Indeed, every hooks (here `klvalue`) get passed as a parameter a function (`k`) which can be called to "continue" visiting the AST or not.

So, for the other constructors of the `lvalue` type (the `| _ ->` pattern in the code above), we do:

19b ⟨*visitor recurse using k* 19b⟩≡
```
 k var
```

Finally, once the visitor is created, we can use it to process the AST:

19c ⟨*iter on asts using visitor* 19c⟩≡
```
asts |> List.iter visitor.V.vtop
```

Here the `asts` variable contains toplevel elements, hence the use of `vtop` (for visiting top). One can also use `vstmt`, `vexpr` (and more) to process respectively statements or expressions.

The output on `foo2.php` should now be:

```
$ ./show_function_calls2 foo2.php
Call to bar at line 3
Call to foo at line 8
```

2.1.3 Arity statistics

20a ⟨*show_function_calls3.ml* 20a⟩≡

```
⟨basic pfff modules open 16a⟩
module V = Visitor_php

⟨show_function_calls v3 20b⟩

let main =
  show_function_calls Sys.argv.(1)
```

20b ⟨*show_function_calls v3* 20b⟩≡

```
let show_function_calls file =
  let (asts2, _stat) = Parse_php.parse file in
  let asts = Parse_php.program_of_program2 asts2 in

  ⟨initialize hfuncs 20c⟩

  ⟨iter on asts using visitor, updating hfuncs 20d⟩

  ⟨display hfuncs to user 21c⟩
```

20c ⟨*initialize hfuncs* 20c⟩≡

```
  let hfuncs = Common.hash_with_default (fun () ->
    Common.hash_with_default (fun () -> 0)
  )
  in
```

20d ⟨*iter on asts using visitor, updating hfuncs* 20d⟩≡

```
  let visitor = V.mk_visitor
   { V.default_visitor with
     V.klvalue = (fun (k, _) var ->
      match Ast_php.untype var with
      | FunCallSimple (qu_opt, funcname, args) ->

          ⟨print funcname and nbargs 21a⟩

          ⟨update hfuncs for name with nbargs 21b⟩

      | _ ->
          k var
    );
  }
  in
  asts |> List.iter visitor.V.vtop;
```

21a ⟨*print funcname and nbargs* 21a⟩≡

```
let f = Ast_php.name funcname in
let nbargs = List.length (Ast_php.unparen args) in
pr2 (spf "Call to %s with %d arguments" f nbargs);
```

21b ⟨*update hfuncs for name with nbargs* 21b⟩≡

```
(* hfuncs[f][nbargs]++ *)
hfuncs#update f (fun hcount ->
  hcount#update nbargs (fun x -> x + 1);
  hcount
)
```

21c ⟨*display hfuncs to user* 21c⟩≡

```
  (* printing statistics *)
  hfuncs#to_list |> List.iter (fun (f, hcount) ->
    pr2 (spf "statistics for %s" f);
    hcount#to_list |> Common.sort_by_key_highfirst
      |> List.iter (fun (nbargs, nbcalls_at_nbargs) ->
        pr2 (spf " when # of args is %d: found %d call sites"
                nbargs nbcalls_at_nbargs)
      )
  )
```

2.1.4 Object statistics

21d ⟨*justin.php* 21d⟩≡

```
<?php
function dashboard_getNews($uid, $appId, $news_ids = null) {
  return prep(new DashboardAppData($uid, $appId))->getNews($news_ids);
}
?>
```

```
$ /home/pad/c-pfff/demos/justin.byte  /home/pad/c-pfff/tests/justin.php
((dashboard_getNews
 ((line: 3)
  (parameters:
   ((uid ()) (appId ())
    (news_ids ((StaticConstant (CName (Name ('null' ""))))))))
  (function_calls: (prep)) (method_calls: (getNews))
  (instantiations: (DashboardAppData)))))
```

2.2 Code matching, `phpgrep`

2.3 A PHP transducer

2.4 `flib` module dependencies

In this section we will port the PHP implementation of a program to print dependencies between files (`flib/_bin/dumpDependencyTree.php` by Justin Bishop). This will help relate different approaches to the same problem, one using PHP and one using OCaml. Note that on this example, the PHP approach is shorter.

Here is the original PHP program:

22 ⟨*dumpDependencyTree.php* 22⟩≡

```
#!/usr/bin/env php
<?php

$_SERVER['PHP_ROOT'] = realpath(dirname(__FILE__).'/../..');
$GLOBALS['THRIFT_ROOT'] = $_SERVER['PHP_ROOT'].'/lib/thrift';

⟨require_xxx redefinitions 23a⟩

function _require($require_type, $dependency) {
  global $current_module, $module_dependencies;
  if (!isset($module_dependencies[$current_module][$require_type])) {
    $module_dependencies[$current_module][$require_type] = array();
  }
  $module_dependencies[$current_module][$require_type][] = $dependency;
}

⟨function add_all_modules 23c⟩
⟨function is_module 23d⟩
⟨function is_test_module 23e⟩

$all_modules = array();
add_all_modules('', $all_modules);

$module_dependencies = array();
$current_module = null;
foreach ($all_modules as $module) {
  $current_module = $module;
  $module_dependencies[$module] = array();
  // @style-override allow flib include
  require_once $_SERVER['PHP_ROOT'].'/flib/'.$module.'/__init__.php';
}

echo json_encode($module_dependencies);
```

23a ⟨*require_xxx redefinitions* 23a⟩≡
```
function require_module($module) {
  _require('module', $module);
}
function require_thrift($file='thrift') {
  _require('thrift', $file);
}
function require_thrift_package($package, $component=null) {
  if (isset($component)) {
    _require('thrift_package', $package.'/'.$component);
  } else {
    _require('thrift_package', $package);
  }
}
function require_thrift_component($component, $name) {
  _require('thrift_component', $component.'/'.$name);
}
```

23b ⟨*require_xxx redefinitions* 23a⟩+≡
```
function require_test($path, $public=true) {}
function require_conf($path) {}
function require_source($path, $public=true) {}
function require_external_source($path) {}
```

23c ⟨*function add_all_modules* 23c⟩≡
```
function add_all_modules($root, &$modules) {
  $path = $_SERVER['PHP_ROOT'].'/flib/'.$root;
  foreach (scandir($path) as $file) {
    if (($file[0] != '.') && is_dir($path.'/'.$file)) {
      $mod = $root.$file;
      if (is_module($path.'/'.$file) &&
          !is_test_module($path.'/'.$file)) {
        $modules[$mod] = $mod;
      }
      add_all_modules($mod.'/', $modules);
    }
  }
}
```

23d ⟨*function is_module* 23d⟩≡
```
function is_module($path) {
  return file_exists($path.'/__init__.php');
}
```

23e ⟨*function is_test_module* 23e⟩≡
```
function is_test_module($module) {
  return in_array('__tests__', explode('/', $module));
}
```

The whole program is remarquably short and makes very good use of PHP ability to dynamically load code and redefine functions (notably with the `require_once` line). In some sense it is using the builtin PHP parser in the PHP interpreter. With pfff things will be different and we will need to process ASTs more manually.

24 ⟨*dump_dependency_tree.ml* 24⟩≡

```
TODO ocaml version
do CFC and maybe remove some graph transitivities, to get less arrows,
(using ocamlgraph/)
```

Chapter 3

Parsing Services

We now switch to a more systematic presentation of the pfff API starting with its first entry point, the parser.

3.1 The main entry point of pfff, `Parse_php.parse`

The `parse_php.mli` file defines the main function to parse a PHP file:

25a ⟨*parse_php.mli* 25a⟩≡

```
⟨type parsing_stat 26c⟩
⟨type program2 25b⟩

(* This is the main function *)
val parse : ?pp:string option -> Common.filename -> (program2 * parsing_stat)

val expr_of_string: string -> Ast_php.expr

val xdebug_expr_of_string: string -> Ast_php.expr
```

The parser does not just return the AST of the file (normally a `Ast_php.program` type, which is an alias for `Ast_php.toplevel list`) but also the tokens associated with each toplevel elements and its string representation (the `program2` type below), as well as parsing statistics (the `parsing_stat` type defined in the next section).

25b ⟨*type program2* 25b⟩≡

```
type program2 = toplevel2 list
  and toplevel2 =
    Ast_php.toplevel (* NotParsedCorrectly if parse error *) * info_item
     (* the token list contains also the comment-tokens *)
     and info_item = (string * Parser_php.token list)
```

[1]

Returning also the tokens is useful as the AST itself by default does not contain the comment or whitespace tokens (except when one call the `comment_annotate_php` function in `pfff/analyzis_php/`) but some later processing phases may need such information. For instance the pfff semantic code visualizer (`pfff_browser` in `pfff/gui/`) need those information to colorize not only the code but also the comments.

If one does not care about those extra information, the `program_of_program2` function helps getting only the “raw” AST:

26a ⟨*parse_php.mli* 25a⟩+≡

```
val program_of_program2 : program2 -> Ast_php.program
```

See the definition of `Ast_php.program` in the next chapter.

The `parse_php.mli` defines also a PHP tokenizer, a subpart of the parser that may be useful on its own.

26b ⟨*parse_php.mli* 25a⟩+≡

```
val tokens: Common.filename -> Parser_php.token list
```

3.2 Parsing statistics

26c ⟨*type parsing_stat* 26c⟩≡

```
type parsing_stat = {
  filename: Common.filename;
  mutable correct: int;
  mutable bad: int;
}
```

26d ⟨*parse_php.mli* 25a⟩+≡

```
val print_parsing_stat_list: parsing_stat list -> unit
```

3.3 pfff -parse_php

26e ⟨*test_parsing_php actions* 26e⟩≡

```
    "-parse_php", "   <file or dir>",
    Common.mk_action_n_arg test_parse_php;
```

[1] The previous snippet contains a note about the `NotParsedCorrectly` constructor which was originally used to provide error recovery in the parser. This is not used any more but it may be back in the futur.

27a ⟨*test_parse_php* 27a⟩≡
```
let test_parse_php xs  =
  let ext = ".*\\.\\(php\\|phpt\\)$" in

  let fullxs = Common.files_of_dir_or_files_no_vcs_post_filter ext xs in

  let stat_list = ref [] in
  ⟨initialize -parse_php regression testing hash 27b⟩

  Common.check_stack_nbfiles (List.length fullxs);

  fullxs +> List.iter (fun file ->
    pr2 ("PARSING: " ^ file);

    let (xs, stat) = Parse_php.parse file in

    Common.push2 stat stat_list;
    ⟨add stat for regression testing in hash 27c⟩
  );

  Parse_php.print_parsing_stat_list !stat_list;
  ⟨print regression testing results 27d⟩
```

27b ⟨*initialize -parse_php regression testing hash* 27b⟩≡
```
let newscore  = Common.empty_score () in
```

27c ⟨*add stat for regression testing in hash* 27c⟩≡
```
    let s = sprintf "bad = %d" stat.Parse_php.bad in
    if stat.Parse_php.bad = 0
    then Hashtbl.add newscore file (Common.Ok)
    else Hashtbl.add newscore file (Common.Pb s)
    ;
```

27d ⟨*print regression testing results* 27d⟩≡
```
  let dirname_opt =
    match xs with
    | [x] when is_directory x -> Some x
    | _ -> None
  in
  let score_path = "/home/pad/c-pfff/tmp" in
  dirname_opt +> Common.do_option (fun dirname ->
    pr2 "--------------------------------";
    pr2 "regression testing  information";
    pr2 "--------------------------------";
    let str = Str.global_replace (Str.regexp "/") "__" dirname in
    Common.regression_testing newscore
```

```
      (Filename.concat score_path
       ("score_parsing__" ^str ^ ext ^ ".marshalled"))
  );
  ()
```

3.4 Preprocessing support, `pfff -pp`

It is not uncommon for programmers to extend their programming language by using preprocessing tools such as `cpp` or `m4`. pfff by default will probably not be able to parse such files as they may contain constructs which are not proper PHP constructs (but `cpp` or `m4` constructs). A solution is to first call your preprocessor on your file and feed the result to pfff. For A small help is provided by pfff

In particular, one can use the `-pp` flag as a first way to handle PHP files using XHP extensions.

Note that this is only a partial solution to properly handling XHP or other extensions. Indeed, in a refactoring context, one would prefer to have in the AST a direct representation of the actual source file. So, pfff also supports certain extensions directly in the AST as explained in Section 4.14.

3.5 `pfff -parse_xhp`

Chapter 4

The AST

4.1 Overview

4.1.1 ast_php.mli structure

The `Ast_php` module defines all the types and constructors used to represent PHP code (the Abstract Syntax Tree of PHP). Any user of pfff must thus understand and know those types as any code using the pfff API will probably need to do some pattern matching over those types.

Here is the toplevel structure of the `Ast_php` module:

29 ⟨*ast_php.mli* 29⟩≡

```
open Common

(******************************************************************************)
(* The AST related types *)
(******************************************************************************)
(* -------------------------------------------------------------------------- *)
(* Token/info *)
(* -------------------------------------------------------------------------- *)
⟨AST info 52b⟩
(* -------------------------------------------------------------------------- *)
(* Name *)
(* -------------------------------------------------------------------------- *)
⟨AST name 51e⟩
(* -------------------------------------------------------------------------- *)
(* Type *)
(* -------------------------------------------------------------------------- *)
⟨AST type 50c⟩
(* -------------------------------------------------------------------------- *)
(* Expression *)
(* -------------------------------------------------------------------------- *)
⟨AST expression 35⟩
```

```
(* -------------------------------------------------------------------------- *)
(* Expression bis, lvalue *)
(* -------------------------------------------------------------------------- *)
⟨AST lvalue 41e⟩
(* -------------------------------------------------------------------------- *)
(* Statement *)
(* -------------------------------------------------------------------------- *)
⟨AST statement 43d⟩
(* -------------------------------------------------------------------------- *)
(* Function definition *)
(* -------------------------------------------------------------------------- *)
⟨AST function definition 47g⟩
⟨AST lambda definition 40g⟩
(* -------------------------------------------------------------------------- *)
(* Class definition *)
(* -------------------------------------------------------------------------- *)
⟨AST class definition 48d⟩
(* -------------------------------------------------------------------------- *)
(* Other declarations *)
(* -------------------------------------------------------------------------- *)
⟨AST other declaration 45g⟩
(* -------------------------------------------------------------------------- *)
(* Stmt bis *)
(* -------------------------------------------------------------------------- *)
⟨AST statement bis 51d⟩
(* -------------------------------------------------------------------------- *)
(* phpext: *)
(* -------------------------------------------------------------------------- *)
⟨AST phpext 58d⟩
(* -------------------------------------------------------------------------- *)
(* The toplevels elements *)
(* -------------------------------------------------------------------------- *)
⟨AST toplevel 50d⟩
(******************************************************************************)
(* AST helpers *)
(******************************************************************************)
⟨AST helpers interface 58e⟩
```

4.1.2 AST example

Before explaining in details each of those AST types, we will first see how look the full AST of a simple PHP program:

30 ⟨*foo1.php* 30⟩≡

```
<?php
function foo($a) {
```

```
    echo $a;
  }
  foo("hello world");
  ?>
```

One way to see the AST of this program is to use the OCaml interpreter and its builtin support for pretty printing OCaml values. First we need to build a custom interpreter `pfff.top` (using `ocamlmktop`) containing all the necessary modules:

```
$ make pfff.top
```

Once `pfff.top` is built, you can run it. You should get an OCaml prompt (the #, not to confuse with the shell prompt $):

```
$ ./pfff.top -I commons -I parsing_php
        Objective Caml version 3.11.1
#
```

You can now call any pfff functions (or any OCaml functions) directly. For instance to parse `demos/foo1.php` type:

```
# Parse_php.parse "demos/foo1.php";;
```

Here is what the interpreter should display (some repetitive parts have been ellided):

```
- : Parse_php.program2 * Parse_php.parsing_stat =
([(Ast_php.FuncDef
    {Ast_php.f_tok =
      {Ast_php.pinfo =
        Ast_php.OriginTok
         {Common.str = "function"; Common.charpos = 6; Common.line = 2;
          Common.column = 0; Common.file = "demos/foo1.php"};
       Ast_php.comments = ()};
     Ast_php.f_ref = None;
     Ast_php.f_name =
      Ast_php.Name
       ("foo",
        {Ast_php.pinfo =
          Ast_php.OriginTok
           {Common.str = "foo"; Common.charpos = 15; Common.line = 2;
            Common.column = 9; Common.file = "demos/foo1.php"};
         Ast_php.comments = ()});
     Ast_php.f_params =
      ({Ast_php.pinfo =
         Ast_php.OriginTok
          {Common.str = "("; Common.charpos = 18; Common.line = 2;
```

```
          Common.column = 12; Common.file = "demos/foo1.php"};
    ...
   ("<?php\nfunction foo($a) {\n  echo $a;\n}",

    [Parser_php.T_OPEN_TAG
      {Ast_php.pinfo =
        Ast_php.OriginTok
         {Common.str = "<?php\n"; Common.charpos = 0; Common.line = 1;
          Common.column = 0; Common.file = "demos/foo1.php"};
       Ast_php.comments = ()};
     Parser_php.T_FUNCTION
      {Ast_php.pinfo =
        Ast_php.OriginTok
         {Common.str = "function"; Common.charpos = 6; Common.line = 2;
          Common.column = 0; Common.file = "demos/foo1.php"};
       Ast_php.comments = ()};
     Parser_php.T_WHITESPACE
      {Ast_php.pinfo =
        Ast_php.OriginTok
         {Common.str = " "; Common.charpos = 14; Common.line = 2;
          Common.column = 8; Common.file = "demos/foo1.php"};
       Ast_php.comments = ()};
     ...]));
  ...],
 ...)
```

We can see on the first line the infered type (`Parse_php.program2 * Parse_php.parsing_stat`) mentionned in the previous chapter. Then there is one of the raw AST element (`FuncDef ...`), its string representation, and the tokens it was made of (`T_OPEN_TAG ...`). As mentionned earlier, the AST contains the full information about the program, including the position of its different elements. This leads to all those `OriginTok {... Common.line = ...}` elements. To see a more compact representation of the AST, one can use the `program_of_program2` function mentionned in the previous chapter, as well as the `abstract_position_info_program` function that replaces all the `OriginTok` elements by another constructor (`Ab` for abstract). See section 4.9 for more information.

Here are the magic incantations:

```
# open Ast_php;;
# let (prog2, _stat) = Parse_php.parse "demos/foo1.php";;
val prog2 : Parse_php.program2 =
...
# let prog = Parse_php.program_of_program2 prog2;;
...
# Lib_parsing_php.abstract_position_info_program prog;;
```

The OCaml interpreter should now display the following:

```
- : Ast_php.program =
[FuncDef
  {f_tok = {pinfo = Ab; comments = ()}; f_ref = None;
   f_name = Name ("foo", {pinfo = Ab; comments = ()});
   f_params =
    ({pinfo = Ab; comments = ()},
     [{p_type = None; p_ref = None;
       p_name = DName ("a", {pinfo = Ab; comments = ()}); p_default = None}],
     {pinfo = Ab; comments = ()});
   f_body =
    ({pinfo = Ab; comments = ()},
     [Stmt
       (Echo ({pinfo = Ab; comments = ()},
         [(ExprVar
            (Var (DName ("a", {pinfo = Ab; comments = ()}),
              {contents = Scope_php.NoScope}),
             {tvar = [Type_php.Unknown]}),
           {t = [Type_php.Unknown]})],
         {pinfo = Ab; comments = ()}))],
     {pinfo = Ab; comments = ()});
   f_type = Type_php.Function ([Type_php.Unknown], [])};
 StmtList
  [ExprStmt
    ((ExprVar
       (FunCallSimple (None, Name ("foo", {pinfo = Ab; comments = ()}),
         ({pinfo = Ab; comments = ()},
          [Arg
            (Scalar
              (Constant (String ("hello world", {pinfo = Ab; comments = ()}))),
             {t = [Type_php.Unknown]})],
          {pinfo = Ab; comments = ()})),
        {tvar = [Type_php.Unknown]}),
      {t = [Type_php.Unknown]}),
    {pinfo = Ab; comments = ()})];
 FinalDef {pinfo = Ab; comments = ()}]
```

Another way to display the AST of a PHP program is to call the custom PHP AST pretty printer defined in `sexp_ast_php.ml` (see Chapter 6) which can be accessed via the `-dump_ast` command line flag as in:

```
$ ./pfff -dump_ast demos/foo1.php
```

This is arguably easier than using `pfff.top` which requires a little bit of gymnastic. Here is the output of the previous command:

```
((FuncDef
  ((f_tok "") (f_ref ()) (f_name (Name ('foo' "")))
   (f_params
    ("" (((p_type ()) (p_ref ()) (p_name (DName ('a' ""))) (p_default ())))
     ""))
   (f_body
    (""
     ((Stmt
       (Echo ""
        (((ExprVar ((Var (DName ('a' "")) "") ((tvar (Unknown)))))
          ((t (Unknown)))))
        ""))))
     ""))
   (f_type (Function (Unknown) ()))))
 (StmtList
  ((ExprStmt
    ((ExprVar
      ((FunCallSimple () (Name ('foo' ""))
        (""
         ((Arg
           ((Scalar (Constant (String ("'hello world'" ""))))
            ((t (Unknown))))))
         ""))
       ((tvar (Unknown)))))
     ((t (Unknown))))
    ""))))
 (FinalDef ""))
```

The ability to easily see the internal representation of PHP programs in pfff is very useful for beginners who may not be familiar with the more than 100 constructors defined in `ast_php.mli` (and detailed in the next sections). Indeed, a common way to write a pfff analysis is to write a few test PHP programs, see the corresponding constructors with the help of the `pfff -dump_ast` command, copy paste parts of the output in your code, and finally write the algorithm to handle those different constructors.

4.1.3 Conventions

In the AST definitions below I sometimes use the tag `(* semantic: *)` in comments which means that such information is not computed at parsing time but may be added later in some post processing stage (by code in `pfff/analyze_php/`).

What follows is the full definition of the abstract syntax tree of PHP 5.2. Right now we keep all the information in this AST, such as the tokens, the parenthesis, keywords, etc, with the `tok` (a.k.a `info`) type used in many constructions (see Section 4.9). This makes it easier to pretty print back this AST and to do source-to-source transformations. So it's actually more a Concrete

Syntax Tree (CST) than an Abstract Syntax Tree (AST) [1] [2]. I sometimes annotate this `tok` type with a comment indicating to what concrete symbol the token corresponds to in the parsed file. For instance for this constructor `| AssignRef of variable * tok (* = *) * tok (* & *) * variable`, the first `tok` will contain information regarding the '=' symbol in the parsed file, and the second `tok` information regarding '&'. If at some point you want to give an error message regarding a certain token, then use the helper functions on `tok` (or `info`) described in Section 4.15.

4.2 Expressions

35 ⟨*AST expression* 35⟩≡

```
type expr = exprbis * exp_info
  ⟨type exp_info 54a⟩
  and exprbis =
  | Lvalue of lvalue

  (* start of expr_without_variable *)
  | Scalar of scalar

  ⟨exprbis other constructors 38b⟩
  ⟨type exprbis hook 56e⟩

  ⟨type scalar and constant and encaps 36a⟩

  ⟨AST expression operators 38c⟩

  ⟨AST expression rest 39d⟩
```

The `ExprVar` constructor is explained later. It corresponds essentially to lvalue expressions (variables, but also function calls). Scalars are described in the next section, followed by the description of the remaining expression constructions (e.g. additions).
[3]

---

[1] Maybe one day we will have a `real_ast_php.ml` (`mini_php/ast_mini_php.ml` can partly play this role to experiment with new algorithms for now)

[2] This is not either completely a CST. It does not follow exactly the grammar; there is not one constructor per grammar rule. Some grammar rules exist because of the limitations of the LALR algorithm; the CST does not have to suffer from this. Moreover a few things were simplified, for instance compare the `variable` type and the `variable` grammar rule.

[3] The `expr_without_variable` grammar element is merged with `expr` in the AST as most of the time in the grammar they use both a case for `expr_without_variable` and a case for `variable`. The only difference is in `Foreach` so it's not worthwhile to complicate things just for `Foreach`.

4.2.1 Scalars, constants, encapsulated strings

36a ⟨*type scalar and constant and encaps* 36a⟩≡
```
 and scalar =
   | Constant of constant
   | ClassConstant of (qualifier * name)

   | Guil    of tok (* '"' *) * encaps list * tok (* '"' *)
   | HereDoc of tok (* < < < EOF *) * encaps list * tok  (* EOF; *)
   (* | StringVarName??? *)
```
⟨*type constant* 36b⟩
⟨*type encaps* 37e⟩

Constants

36b ⟨*type constant* 36b⟩≡
```
  and constant =
```
⟨*constant constructors* 36c⟩
⟨*type constant hook* 58a⟩
⟨*constant rest* 37c⟩

Here are the basic constants, numbers:

36c ⟨*constant constructors* 36c⟩≡
```
| Int of string wrap
| Double of string wrap
```
I put `string` for `Int` (and `Double`) because `int` would not be enough as OCaml ints are only 31 bits. So it is simpler to use strings.

Note that `-2` is not a constant; it is the unary operator `-` (`Unary (UnMinus ...)`) applied to the constant 2. So the string in `Int` must represent a positive integer only.

Strings in PHP comes in two forms: constant strings and dynamic strings (aka interpolated or encapsulated strings). In this section we are concerned only with the former.

36d ⟨*constant constructors* 36c⟩+≡
```
| String of string wrap
```
The `string` part does not include the enclosing guillemet `'"'` or quote `'`. The info itself (in `wrap`) will usually contain it, but not always! Indeed if the constant we build is part of a bigger encapsulated strings as in `echo "$x[foo]"` then the `foo` will be parsed as a `String`, even if in the text it appears as a name. [4]

---
[4] If at some point you want to do some program transformation, you may have to normalize this `string wrap` before moving it in another context !!!

Some identifiers have special meaning in PHP such as `true`, `false`, `null`. They are parsed as `CName`:

37a ⟨*constant constructors* 36c⟩+≡
```
| CName of name (* true, false, null,  or defined constant *)
```

PHP also supports `__FILE__` and other directives inspired by the C preprocessor `cpp`:

37b ⟨*constant constructors* 36c⟩+≡
```
| PreProcess of cpp_directive wrap
```

37c ⟨*constant rest* 37c⟩≡
⟨*type cpp_directive* 37d⟩

37d ⟨*type cpp_directive* 37d⟩≡
```
 and cpp_directive =
     | Line  | File
     | ClassC  | MethodC  | FunctionC
```

Encapsulated strings

Strings interpolation in PHP is complicated and documented here: http://php.net/manual/en/language.types.string.php in the "variable parsing" section.

37e ⟨*type encaps* 37e⟩≡
```
 and encaps =
```
⟨*encaps constructors* 37f⟩

37f ⟨*encaps constructors* 37f⟩≡
```
| EncapsString of string wrap

 (* for "xx $beer". I put EncapsVar variable, but if you look
  * at the grammar it's  actually a subset of variable, but I didn't
  * want to duplicate subparts of variable here.
  *)
```

37g ⟨*encaps constructors* 37f⟩+≡
```
 | EncapsVar of lvalue
```

37h ⟨*encaps constructors* 37f⟩+≡
```
(* for "xx {$beer}s" *)
| EncapsCurly of tok * lvalue * tok
```

37i ⟨*encaps constructors* 37f⟩+≡
```
(* for "xx ${beer}s" *)
| EncapsDollarCurly of tok (* '${' *) * lvalue * tok
```

38a ⟨*encaps constructors* 37f⟩+≡
```
| EncapsExpr of tok * expr * tok
```

4.2.2 Basic expressions

PHP supports the usual arithmetic (`+`, `-`, etc) and logic expressions inherited from C:

38b ⟨*exprbis other constructors* 38b⟩≡
```
  | Binary  of expr * binaryOp wrap * expr
  | Unary   of unaryOp wrap * expr
```

38c ⟨*AST expression operators* 38c⟩≡
```

and fixOp    = Dec | Inc
and binaryOp    = Arith of arithOp | Logical of logicalOp
  ⟨php concat operator 38d⟩
      and arithOp   =
        | Plus | Minus | Mul | Div | Mod
        | DecLeft | DecRight
        | And | Or | Xor

      and logicalOp =
        | Inf | Sup | InfEq | SupEq
        | Eq | NotEq
        ⟨php identity operators 38f⟩
        | AndLog | OrLog | XorLog
        | AndBool | OrBool (* diff with AndLog ? *)
and assignOp = AssignOpArith of arithOp
 ⟨php assign concat operator 38e⟩
and unaryOp =
  | UnPlus | UnMinus
  | UnBang | UnTilde
```

It also defines new operators for string concatenation

38d ⟨*php concat operator* 38d⟩≡
```
| BinaryConcat (* . *)
```

38e ⟨*php assign concat operator* 38e⟩≡
```
| AssignConcat (* .= *)
```

and object comparisons:

38f ⟨*php identity operators* 38f⟩≡
```
  | Identical (* === *) | NotIdentical (* !== *)
```

It also inherits the +=, ++ and other side effect expression (that really should not be expression):

39a ⟨*exprbis other constructors* 38b⟩+≡
```
(* should be a statement ... *)
| Assign    of lvalue * tok (* = *) * expr
| AssignOp  of lvalue * assignOp wrap * expr
| Postfix of rw_variable   * fixOp wrap
| Infix   of fixOp wrap    * rw_variable
```

The ugly conditional ternary operator:

39b ⟨*exprbis other constructors* 38b⟩+≡
```
| CondExpr of expr * tok (* ? *) * expr * tok (* : *) * expr
```

4.2.3 Value constructions

39c ⟨*exprbis other constructors* 38b⟩+≡
```
| ConsList  of tok * list_assign comma_list paren * tok * expr
| ConsArray of tok * array_pair  comma_list paren
```

39d ⟨*AST expression rest* 39d⟩≡
```
and list_assign =
  | ListVar of lvalue
  | ListList of tok * list_assign comma_list paren
  | ListEmpty
```

39e ⟨*AST expression rest* 39d⟩+≡
```
and array_pair =
  | ArrayExpr of expr
  | ArrayRef of tok (* & *) * lvalue
  | ArrayArrowExpr of expr * tok (* => *) * expr
  | ArrayArrowRef of expr * tok (* => *) * tok (* & *) * lvalue
```

4.2.4 Object constructions

39f ⟨*exprbis other constructors* 38b⟩+≡
```
| New of tok * class_name_reference * argument comma_list paren option
| Clone of tok * expr
```

39g ⟨*exprbis other constructors* 38b⟩+≡
```
| AssignRef of lvalue * tok (* = *) * tok (* & *) * lvalue
| AssignNew of lvalue * tok (* = *) * tok (* & *) * tok (* new *) *
    class_name_reference *
    argument comma_list paren option
```

40a ⟨*AST expression rest* 39d⟩+≡
```
and class_name_reference =
  | ClassNameRefStatic of name
  | ClassNameRefDynamic of (lvalue * obj_prop_access list)
  and obj_prop_access = tok (* -> *) * obj_property
```

4.2.5 Cast

40b ⟨*exprbis other constructors* 38b⟩+≡
```
| Cast of castOp wrap * expr
| CastUnset of tok * expr (* ??? *)
```

40c ⟨*AST expression operators* 38c⟩+≡
```
and castOp = ptype
```

40d ⟨*exprbis other constructors* 38b⟩+≡
```
| InstanceOf of expr * tok * class_name_reference
```

4.2.6 Eval

40e ⟨*exprbis other constructors* 38b⟩+≡
```
(* !The evil eval! *)
| Eval of tok * expr paren
```

4.2.7 Anonymous functions (PHP 5.3)

40f ⟨*exprbis other constructors* 38b⟩+≡
```
| Lambda of lambda_def
```

40g ⟨*AST lambda definition* 40g⟩≡
```
and lambda_def = {
 l_tok: tok; (* function *)
 l_ref: is_ref;
 (* no l_name, anonymous *)
 l_params: parameter comma_list paren;
 l_use:  lexical_vars option;
 l_body: stmt_and_def list brace;
}
  and lexical_vars = tok (* use *) * lexical_var comma_list paren
  and lexical_var =
    | LexicalVar of is_ref * dname
```

4.2.8 Misc

41a ⟨*exprbis other constructors* 38b⟩+≡
```
    (* should be a statement ... *)
    | Exit of tok * (expr option paren) option
    | At of tok (* @ *) * expr
    | Print of tok * expr
```

41b ⟨*exprbis other constructors* 38b⟩+≡
```
    | BackQuote of tok * encaps list * tok
```

41c ⟨*exprbis other constructors* 38b⟩+≡
```
    (* should be at toplevel *)
    | Include     of tok * expr
    | IncludeOnce of tok * expr
    | Require     of tok * expr
    | RequireOnce of tok * expr
```

41d ⟨*exprbis other constructors* 38b⟩+≡
```
    | Empty of tok * lvalue paren
    | Isset of tok * lvalue comma_list paren
```

4.3 Lvalue expressions

The `lvalue` type below allows a superset of what the PHP grammar actually permits. See the `variable2` type in `parser_php.mly` for a more precise, but far less convenient type to use. [5]

41e ⟨*AST lvalue* 41e⟩≡
```
  and lvalue = lvaluebis * lvalue_info
    ⟨type lvalue_info 54b⟩
    and lvaluebis =
    ⟨lvaluebis constructors 42a⟩

    ⟨type lvalue aux 42e⟩

  (* semantic ? *)
  and rw_variable = lvalue
  and r_variable = lvalue
  and w_variable = lvalue
```

[5] Note that with XHP, we are less a superset because XHP also relaxed some constraints.

4.3.1 Basic variables

Here is the constructor for simple variables, as in `$foo`:

42a ⟨*lvaluebis constructors* 42a⟩≡
```
| Var of dname *
 (* TODO add a constructor for This ? *)
 ⟨scope_php annotation 56c⟩
```

The 'd' in `dname` stands for dollar (dollar name).

42b ⟨*lvaluebis constructors* 42a⟩+≡
```
(* xhp: normally we can not have a FunCall in the lvalue of VArrayAccess,
 * but with xhp we can.
 *
 * TODO? a VArrayAccessSimple with Constant string in expr ?
 *)
| VArrayAccess of lvalue * expr option bracket
```

4.3.2 Indirect variables

42c ⟨*lvaluebis constructors* 42a⟩+≡
```
| VBrace       of tok      * expr brace
| VBraceAccess of lvalue * expr brace
```

42d ⟨*lvaluebis constructors* 42a⟩+≡
```
(* on the left of var *)
| Indirect  of lvalue * indirect
```

42e ⟨*type lvalue aux* 42e⟩≡
```
and indirect = Dollar of tok
```

42f ⟨*lvaluebis constructors* 42a⟩+≡
```
| VQualifier of qualifier * lvalue
```

4.3.3 Function calls

Function calls are considered as part of the `lvalue` category in the original PHP grammar. This is probably because functions can return reference to variables (whereas additions can't).

42g ⟨*lvaluebis constructors* 42a⟩+≡
```
| FunCallSimple of qualifier option * name     * argument comma_list paren
| FunCallVar    of qualifier option * lvalue * argument comma_list paren
```

42h ⟨*type lvalue aux* 42e⟩+≡
```
and argument =
  | Arg    of expr
  | ArgRef of tok * w_variable
```

A few constructs have `Simple` as a suffix. They just correspond to inlined version of other constructs that were put in their own constructor because they occur very often or are conceptually important and deserve their own constructor (for instance `FunCallSimple` which otherwise would force the programmer to match over more nested constructors to check if a `Funcall` has a static name). On one hand it makes it easier to match specific construct, on the other hand when you write an algorithm it forces you to do a little duplication. But usually I first write the algorithm to handle the easy cases anyway and I end up not coding the complex one so ...

4.3.4 Method and object accesses

```
(* TODO go further by having a dname for the variable ? or make a
 * type simple_dvar = dname * Scope_php.phpscope ref and
 * put here a simple_dvar ?
 *)
```

43a ⟨*lvaluebis constructors* 42a⟩+≡

```
  | MethodCallSimple of lvalue * tok * name * argument comma_list paren
```

43b ⟨*lvaluebis constructors* 42a⟩+≡

```
  | ObjAccessSimple of lvalue * tok (* -> *) * name
  | ObjAccess of lvalue * obj_access
```

43c ⟨*type lvalue aux* 42e⟩+≡

```
  and obj_access = tok (* -> *) * obj_property * argument comma_list paren option

  and obj_property =
    | ObjProp of obj_dim
    | ObjPropVar of lvalue (* was originally var_without_obj *)

    (* I would like to remove OName from here, as I inline most of them
     * in the MethodCallSimple and ObjAccessSimple above, but they
     * can also be mentionned in OArrayAccess in the obj_dim, so
     * I keep it
     *)
    and obj_dim =
      | OName of name
      | OBrace of expr brace
      | OArrayAccess of obj_dim * expr option bracket
      | OBraceAccess of obj_dim * expr brace
```

4.4 Statements

43d ⟨*AST statement* 43d⟩≡

```
(* by introducing lambda, expr and stmt are now mutually recursive *)
and stmt =
  ⟨stmt constructors 44a⟩

  ⟨AST statement rest 44d⟩
```

4.4.1 Basic statements

44a ⟨*stmt constructors* 44a⟩≡

```
    | ExprStmt of expr * tok (* ; *)
    | EmptyStmt of tok  (* ; *)
```

44b ⟨*stmt constructors* 44a⟩+≡

```
    | Block of stmt_and_def list brace
```

44c ⟨*stmt constructors* 44a⟩+≡

```
    | If      of tok * expr paren * stmt *
        (* elseif *) (tok * expr paren * stmt) list *
        (* else *) (tok * stmt) option
    ⟨ifcolon 47e⟩
    | While of tok * expr paren * colon_stmt
    | Do of tok * stmt * tok * expr paren * tok
    | For of tok * tok *
        for_expr * tok *
        for_expr * tok *
        for_expr *
        tok *
        colon_stmt
    | Switch of tok * expr paren * switch_case_list
```

44d ⟨*AST statement rest* 44d⟩≡

```
    and switch_case_list =
      | CaseList      of tok * tok option * case list * tok
      | CaseColonList of tok * tok option * case list * tok * tok
      and case =
        | Case    of tok * expr * tok * stmt_and_def list
        | Default of tok * tok * stmt_and_def list
```

44e ⟨*stmt constructors* 44a⟩+≡

```
    (* if it's a expr_without_variable, the second arg must be a Right variable,
     * otherwise if it's a variable then it must be a foreach_variable
     *)
    | Foreach of tok * tok * expr * tok *
        (foreach_variable, lvalue) Common.either * foreach_arrow option * tok *
        colon_stmt
```

45a ⟨*AST statement rest* 44d⟩+≡
```
    and for_expr = expr list (* can be empty *)
    and foreach_arrow = tok * foreach_variable
    and foreach_variable = is_ref * lvalue
```

45b ⟨*stmt constructors* 44a⟩+≡
```
    | Break    of tok * expr option * tok
    | Continue of tok * expr option * tok
    | Return of tok * expr option * tok
```

45c ⟨*stmt constructors* 44a⟩+≡
```
    | Throw of tok * expr * tok
    | Try of tok * stmt_and_def list brace * catch * catch list
```

45d ⟨*AST statement rest* 44d⟩+≡
```
    and catch =
      tok * (fully_qualified_class_name * dname) paren * stmt_and_def list brace
```

45e ⟨*stmt constructors* 44a⟩+≡
```
    | Echo of tok * expr list * tok
```

4.4.2 Globals and static

45f ⟨*stmt constructors* 44a⟩+≡
```
    | Globals    of tok * global_var list * tok
    | StaticVars of tok * static_var list * tok
```

45g ⟨*AST other declaration* 45g⟩≡
```
  and global_var =
    | GlobalVar of dname
    | GlobalDollar of tok * r_variable
    | GlobalDollarExpr of tok * expr brace
```

45h ⟨*AST other declaration* 45g⟩+≡
```
  and static_var = dname * static_scalar_affect option
```

45i ⟨*AST other declaration* 45g⟩+≡
```
    and static_scalar =
      | StaticConstant of constant
      | StaticClassConstant of (qualifier * name) (* semantic ? *)

      | StaticPlus  of tok * static_scalar
      | StaticMinus of tok * static_scalar

      | StaticArray of tok * static_array_pair comma_list paren
    ⟨type static_scalar hook 58b⟩
```

So PHP offers some support for compile-time constant expressions evaluation, but it is very limited (to additions and substractions).

46a ⟨*AST other declaration* 45g⟩+≡
```
and static_scalar_affect = tok (* = *) * static_scalar
```

46b ⟨*AST other declaration* 45g⟩+≡
```
and static_array_pair =
  | StaticArraySingle of static_scalar
  | StaticArrayArrow  of static_scalar * tok (* => *) * static_scalar
```

4.4.3 Inline HTML

PHP allows to freely mix PHP and HTML code in the same file. This was arguably what made PHP successful, providing a smooth transition from static HTML to partially dynamic HTML. In practice, using inline HTML is probably not the best approach for website development as it intermixes business and display in the same file. It is usually better to separate concerns, for instance by using template technology. XHP could be seen as going back to this inline style, while avoiding some of its disadvantages.

From the point of view of the parser, HTML snippets are always viewed as embeded in a PHP code, and not the way around, and are represented by the following construct:

46c ⟨*stmt constructors* 44a⟩+≡
```
  | InlineHtml of string wrap
```

So, on this PHP file:

46d ⟨*tests/inline_html.php* 46d⟩≡
```
<html>
<?php
echo "foo";
?>
</html>
```

this is what `pfff -dump_ast` will output:

```
((StmtList
  ((InlineHtml ("'<html>\n'" ""))
   (Echo "" (((Scalar (Constant (String ('foo' "")))) ((t (Unknown))))) "")
   (InlineHtml ("'</html>\n'" ""))))
 (FinalDef ""))
```

In fact we could go one step further and internally transforms all those `InlineHtml` into `Echo` statements, so further analysis does not need to be aware of this *syntactic sugar* provided by PHP. Nevertheless in a refactoring context, it is useful to represent internally exactly as-is the PHP program, so I prefer to keep `InlineHtml`.

4.4.4 Misc statements

47a ⟨*stmt constructors* 44a⟩+≡
```
  | Use of tok * use_filename * tok
  | Unset of tok * lvalue comma_list paren * tok
  | Declare of tok * declare comma_list paren * colon_stmt
```

47b ⟨*AST statement rest* 44d⟩+≡
```
  and use_filename =
    | UseDirect of string wrap
    | UseParen  of string wrap paren
```

47c ⟨*AST statement rest* 44d⟩+≡
```
  and declare = name * static_scalar_affect
```

4.4.5 Colon statement syntax

PHP allows two different forms for sequence of statements. The regular one and the one using a colon : (see http://php.net/manual/en/control-structures.alternative-syntax.php):

47d ⟨*AST statement rest* 44d⟩+≡
```
  and colon_stmt =
    | SingleStmt of stmt
    | ColonStmt of tok (* : *) * stmt_and_def list * tok (* endxxx *) * tok (* ; *)
```

47e ⟨*ifcolon* 47e⟩≡
```
  | IfColon of tok * expr paren *
     tok * stmt_and_def list * new_elseif list * new_else option *
     tok * tok
```

47f ⟨*AST statement rest* 44d⟩+≡
```
  and new_elseif = tok * expr paren * tok * stmt_and_def list
  and new_else = tok * tok * stmt_and_def list
```

4.5 Function and class definitions

4.5.1 Function definition

47g ⟨*AST function definition* 47g⟩≡
```
and func_def = {
  f_tok: tok; (* function *)
  f_ref: is_ref;
  f_name: name;
  f_params: parameter comma_list paren;
  f_body: stmt_and_def list brace;
  ⟨f_type mutable field 55c⟩
```

```
  }
    ⟨AST function definition rest 48a⟩
```

48a ⟨*AST function definition rest* 48a⟩≡

```
    and parameter = {
      p_type: hint_type option;
      p_ref: is_ref;
      p_name: dname;
      p_default: static_scalar_affect option;
    }
```

48b ⟨*AST function definition rest* 48a⟩+≡

```
      and hint_type =
        | Hint of name
        | HintArray  of tok
```

[6]

48c ⟨*AST function definition rest* 48a⟩+≡

```
    and is_ref = tok (* bool wrap ? *) option
```

4.5.2 Class definition

48d ⟨*AST class definition* 48d⟩≡

```
  and class_def = {
    c_type: class_type;
    c_name: name;
    c_extends: extend option;
    c_implements: interface option;
    c_body: class_stmt list brace;
  }
    ⟨type class_type 48e⟩
    ⟨type extend 48f⟩
    ⟨type interface 49a⟩
```

48e ⟨*type class_type* 48e⟩≡

```
    and class_type =
      | ClassRegular  of tok (* class *)
      | ClassFinal    of tok * tok (* final class *)
      | ClassAbstract of tok * tok (* abstract class *)
```

PHP supports only single inheritance, hence the single name below:

48f ⟨*type extend* 48f⟩≡

```
    and extend =    tok * fully_qualified_class_name
```

[6]FACEBOOK: plug here for a better type system for HPHP, with more complex annotation. Right now type annotation in PHP works only for classes, not for basic types. The parser can parse `function foo(int x) {}` but nothing will be enforced I believe.

PHP nevertheless supports multiple interfaces, hence the `list` below:

49a ⟨*type interface* 49a⟩≡
```
and interface = tok * fully_qualified_class_name list
```

4.5.3 Interface definition

49b ⟨*AST class definition* 48d⟩+≡
```
and interface_def = {
  i_tok: tok; (* interface *)
  i_name: name;
  i_extends: interface option;
  i_body: class_stmt list brace;
}
```

4.5.4 Class variables and constants

49c ⟨*AST class definition* 48d⟩+≡
```
and class_stmt =
  | ClassConstants of tok * class_constant list * tok
  | ClassVariables of class_var_modifier * class_variable list * tok
  | Method of method_def
```

⟨*class_stmt types* 49d⟩

49d ⟨*class_stmt types* 49d⟩≡
```
and class_constant = name * static_scalar_affect
```

49e ⟨*class_stmt types* 49d⟩+≡
```
and class_variable = dname * static_scalar_affect option
```

49f ⟨*class_stmt types* 49d⟩+≡
```
and class_var_modifier =
  | NoModifiers of tok (* 'var' *)
  | VModifiers of modifier wrap list
```

4.5.5 Method definitions

49g ⟨*class_stmt types* 49d⟩+≡
```
and method_def = {
  m_modifiers: modifier wrap list;
  m_tok: tok; (* function *)
  m_ref: is_ref;
  m_name: name;
  m_params: parameter comma_list paren;
  m_body: method_body;
}
```

50a ⟨*class_stmt types* 49d⟩+≡
```
        and modifier =
          | Public  | Private | Protected
          | Static  | Abstract | Final
```

50b ⟨*class_stmt types* 49d⟩+≡
```
        and method_body =
          | AbstractMethod of tok
          | MethodBody of stmt_and_def list brace
```

4.6 Types (or the lack of them)

The following type is used only for the cast operations (as in `echo (int) $x`).

50c ⟨*AST type* 50c⟩≡
```
  type ptype =
    | BoolTy
    | IntTy
    | DoubleTy (* float *)

    | StringTy

    | ArrayTy
    | ObjectTy
```

   ⟨*tarzan annotation* 66b⟩

For a real type analysis, see `type_php.ml` and the type annotations on expressions and variables in Section 4.10.1, as well as the type inference algorithm in `pfff/analysis_php`.

4.7 Toplevel constructions

50d ⟨*AST toplevel* 50d⟩≡
```
  and toplevel =
    ⟨toplevel constructors 50e⟩

   and program = toplevel list

   ⟨tarzan annotation 66b⟩
```

50e ⟨*toplevel constructors* 50e⟩≡
```
    | StmtList of stmt list
    | FuncDef of func_def
    | ClassDef of class_def
    | InterfaceDef of interface_def
```

51a ⟨*toplevel constructors* 50e⟩+≡
```
    | Halt of tok * unit paren * tok (* __halt__ ; *)
```

51b ⟨*toplevel constructors* 50e⟩+≡
```
    | NotParsedCorrectly of info list
```

51c ⟨*toplevel constructors* 50e⟩+≡
```
    | FinalDef of info (* EOF *)
```

51d ⟨*AST statement bis* 51d⟩≡
```
  (* Was originally called toplevel, but for parsing reasons and estet I think
   * it's better to differentiate nested func and top func. Also better to
   * group the toplevel statements together (StmtList below), so that
   * in the database later they share the same id.
   *)
  and stmt_and_def =
    | Stmt of stmt
    | FuncDefNested of func_def
    | ClassDefNested of class_def
    | InterfaceDefNested of interface_def
```

4.8 Names

51e ⟨*AST name* 51e⟩≡
```
   ⟨type name 51f⟩

   ⟨type dname 51g⟩

   ⟨qualifiers 52a⟩

   ⟨tarzan annotation 66b⟩
```

51f ⟨*type name* 51f⟩≡
```
  (* T_STRING, which are really just LABEL, see the lexer. *)
  type name =
     | Name of string wrap
     ⟨type name hook 58c⟩
```

51g ⟨*type dname* 51g⟩≡
```
  (* T_VARIABLE. D for dollar. The string does not contain the '$'.
   * The info itself will usually contain it, but not
   * always! Indeed if the variable we build comes from an encapsulated
   * strings as in  echo "${x[foo]}" then the 'x' will be parsed
   * as a T_STRING_VARNAME, and eventually lead to a DName, even if in
   * the text it appears as a name.
```

```
 * So this token is kind of a FakeTok sometimes.
 *
 * So if at some point you want to do some program transformation,
 * you may have to normalize this string wrap before moving it
 * in another context !!!
 *)
and dname =
  | DName of string wrap
```

52a ⟨*qualifiers* 52a⟩≡

```
and qualifier =
  | Qualifier of fully_qualified_class_name * tok (* :: *)
  (* TODO? have a Self | Parent also ? can have self without a :: ? *)

and fully_qualified_class_name = name
```

4.9 Tokens, `info` and `unwrap`

52b ⟨*AST info* 52b⟩≡

```
⟨type pinfo 53a⟩

type info = {
  (* contains among other things the position of the token through
   * the Common.parse_info embedded inside the pinfo type.
   *)
  mutable pinfo : pinfo;

  ⟨type info hook 53e⟩
  }
and tok = info
```

52c ⟨*AST info* 52b⟩+≡

```
(* a shortcut to annotate some information with token/position information *)
and 'a wrap = 'a * info
```

52d ⟨*AST info* 52b⟩+≡

```
and 'a paren   = tok * 'a * tok
and 'a brace   = tok * 'a * tok
and 'a bracket = tok * 'a * tok
and 'a comma_list = 'a list
```

52e ⟨*AST info* 52b⟩+≡

```
  ⟨tarzan annotation 66b⟩
```

53a ⟨*type pinfo* 53a⟩≡
```
type pinfo =
  ⟨pinfo constructors 53b⟩
  ⟨tarzan annotation 66b⟩
```

53b ⟨*pinfo constructors* 53b⟩≡
```
  (* Present both in the AST and list of tokens *)
  | OriginTok  of Common.parse_info
```

For rerefence, here is the definition of `Common.parse_info`:

```
type parse_info = {
    str: string;
    charpos: int;

    line: int;
    column: int;
    file: filename;
  }
```

53c ⟨*pinfo constructors* 53b⟩+≡
```
  (* Present only in the AST and generated after parsing. Can be used
   * when building some extra AST elements. *)
  | FakeTokStr of string (* to help the generic pretty printer *)
```

53d ⟨*pinfo constructors* 53b⟩+≡
```
  (* The Ab constructor is (ab)used to call '=' to compare
   * big AST portions. Indeed as we keep the token information in the AST,
   * if we have an expression in the code like "1+1" and want to test if
   * it's equal to another code like "1+1" located elsewhere, then
   * the Pervasives.'=' of OCaml will not return true because
   * when it recursively goes down to compare the leaf of the AST, that is
   * the parse_info, there will be some differences of positions. If instead
   * all leaves use Ab, then there is no position information and we can
   * use '='. See also the 'al_info' function below.
   *
   * Ab means AbstractLineTok. Use a short name to not
   * polluate in debug mode.
   *)
  | Ab
```

53e ⟨*type info hook* 53e⟩≡
```
  (*TODO*)
  comments: unit;
```

4.10 Semantic annotations

4.10.1 Type annotations

54a ⟨*type exp_info* 54a⟩≡
```
(* semantic: *)
and exp_info = {
   mutable t: Type_php.phptype;
}
```

54b ⟨*type lvalue_info* 54b⟩≡
```
  (* semantic: *)
  and lvalue_info = {
    mutable tlval: Type_php.phptype;
  }
```

```
(*
 * PHP 'pad' type system. Inspired by union types, soft typing, etc.
 *
 * history: I Moved the Union out of phptype, to make phptype a phtypebis list
 * with the intuition that it's so important that it should be "builtin"
 * and be really part of every type definitions.
 *
 * Example of a phptype: [Object "A", Null].
 * The list is sorted to make is easier for unify_type to work
 * efficiently.
 *
 * Add null to phptype ? I think yes, so that can do some null
 * analysis at the same time.
 *
 * Add Ref of phptype ?? Should ref be part of the type system ?
 * I think no. In fact there was some paper about that.
 *
 *)
```

54c ⟨*type_php.mli* 54c⟩≡

  ⟨*type phptype* 54d⟩

  ⟨*type phpfunction_type* 56a⟩

54d ⟨*type phptype* 54d⟩≡
```
type phptype = phptypebis list  (* sorted list, cf typing_php.ml *)

  and phptypebis =
    | Basic       of basictype
```

```
| ArrayFamily of arraytype
(* duck typing style, dont care too much about the name of the class
 * TODO qualified name ?
 * TODO phpmethod_type list * string list
 *)
| Object      of string (* class name *) option

| Resource (* opened file or mysql connection *)

(* PHP 5.3 has closure *)
| Function of phptype * phptype option (* when have default value *) list


| Null

(* TypeVar is used by the type inference and unifier algorithn.
 * It should use a counter for fresh typevariables but it's
 * better to use a string so can give readable type variable like
 * x_1 for the typevar of the $x parameter.
 *)
| TypeVar of string
(* old: | Union of phptype list *)

| Unknown
| Top (* Top aka Variant, but should never be used *)
```

55a ⟨*type phptype* 54d⟩+≡

```
and basictype =
  | Bool
  | Int
  | Float
  | String

  | Unit (* in PHP certain expressions are really more statements *)
```

55b ⟨*type phptype* 54d⟩+≡

```
and arraytype =
  | Array  of phptype
  | Hash   of phptype
  (* duck typing style, ordered list by fieldname *)
  | Record of (string * phptype) list
```

⟨*tarzan annotation* 66b⟩

55c ⟨*f_type mutable field* 55c⟩≡

```
  (* semantic: *)
  mutable f_type: Type_php.phptype;
```

56a ⟨*type phpfunction_type* 56a⟩≡
    ⟨*tarzan annotation* 66b⟩

56b ⟨*type_php.mli* 54c⟩+≡
```
  val string_of_phptype: phptype -> string
```

4.10.2 Scope annotations

56c ⟨*scope_php annotation* 56c⟩≡
```
  Scope_php.phpscope ref
```

56d ⟨*scope_php.mli* 56d⟩≡
```
  type phpscope =
    | Global
    | Local
    | Param
    (* | Class ? *)

    | NoScope
```
    ⟨*tarzan annotation* 66b⟩

4.11 Support for syntactical/semantic grep

56e ⟨*type exprbis hook* 56e⟩≡
```
    | EDots of info
```

4.12 Support for source-to-source transformations

As explained earlier, we want to keep in the AST as much information as possible, and be as faithful as possible to the original PHP constructions, so one can modify this AST and pretty print back while still preserving the style (indentation, comments) of the original file. The approach generally used in compilers is on the opposite to get an AST that is a simplification of the original program (hence the A for "abstract" in AST) by removing syntactic sugar, or by transforming at parsing-time certain constructions into simpler one, for instance by replacing all `while`, `do`, `switch`, `if`, or `foreach` into series of `goto` statements. This makes some further analysis simpler because they have to deal with a smaller set of constructions (only `goto`s), but it makes it hard to do source-to-source style-preserving transformations. Indeed, having done the transformation on the `goto`s, one would still need to back-propagate such transformation in the original file, which contains the `while`, `do`, etc. One can not generate a file with `goto`s because a programmer would not like to further work on such file.

So to builting tools like refactorers using pfff, we need to be faithful to the original file. This led to all those `tok` types embeded in the AST to store information about the tokens with their precise location in the original file. This also forces us to retain in the AST the tokens forming the parenthesis in expressions (which in typical frontends are removed as the tree data structures of the AST already encodes the priority of elements), hence the following extension to the `exprbis` type:

57a ⟨*type exprbis hook* 56e⟩+≡

```
(* unparser: *)
| ParenExpr of expr paren
```

57b ⟨*type info hook* 53e⟩+≡

```
(* transformation: transformation *)
```

4.13 Support for Xdebug

Xdebug is a great debugger/profiler/tracer for PHP. It can among other things generate function call traces of running code, including types and concrete values of parameters. There are many things you can do using such information, such as trivial type inference feedback in a IDE, or type-based bug checking. Here is an example of a trace file:

```
TRACE START [2010-02-08 00:24:28]
    0.0009      99800   -> {main}() /home/pad/mobile/project-facebook/pfff/tests/xdebug/basi
    0.0009      99800     -> main() /home/pad/mobile/project-facebook/pfff/tests/xdebug/basi
    0.0009      99968       -> foo_int(4) /home/pad/mobile/project-facebook/pfff/tests/xdebu
                             >=> 8
    0.0010     100160       -> foo_string('ici') /home/pad/mobile/project-facebook/pfff/test
                             >=> 'icifoo_string'
    0.0010     100320       -> foo_array(array ()) /home/pad/mobile/project-facebook/pfff/te
                             >=> array ('foo_array' => 'foo')
    0.0011     100632       -> foo_nested_array() /home/pad/mobile/project-facebook/pfff/tes
                             >=> array ('key1' => 1, 'key2' => TRUE, 'key3' => 'astring', 'k
                           >=> NULL
                         >=> 1
    0.0012      41208
TRACE END   [2010-02-08 00:24:28]
```

As you can see, those traces contain regular PHP function calls and expressions and so can be parsed by the pfff expression parser.

Xdebug traces also sometimes contain certain constructs that are not regular PHP constructs. For instance ... is sometimes used in arrays arguments to indicate that the value was too big to be included in the trace. Resources such as file handler are also displayed in a non traditional way, as well as objects. So to parse such traces, it is quite simple to extend the grammar and AST to include such extensions:

58a ⟨*type constant hook* 58a⟩≡
```
  | XdebugClass of name * class_stmt list
  | XdebugResource (* TODO *)
```

58b ⟨*type static_scalar hook* 58b⟩≡
```
   | XdebugStaticDots
```

4.14 XHP extensions

58c ⟨*type name hook* 58c⟩≡
```
 (* xhp: *)
 | XhpName of string wrap
```

58d ⟨*AST phpext* 58d⟩≡

4.15 AST accessors, extractors, wrappers

58e ⟨*AST helpers interface* 58e⟩≡
```
 val parse_info_of_info : info -> Common.parse_info
```

58f ⟨*AST helpers interface* 58e⟩+≡
```
 val pinfo_of_info : info -> pinfo
```

58g ⟨*AST helpers interface* 58e⟩+≡
```
 val pos_of_info : info -> int
 val str_of_info : info -> string
 val file_of_info : info -> Common.filename
 val line_of_info : info -> int
 val col_of_info : info -> int
```

58h ⟨*AST helpers interface* 58e⟩+≡
```
 val string_of_info : info -> string
```

58i ⟨*AST helpers interface* 58e⟩+≡
```
 val name : name -> string
 val dname : dname -> string
```

58j ⟨*AST helpers interface* 58e⟩+≡
```
 val info_of_name : name -> info
 val info_of_dname : dname -> info
```

59a ⟨*AST helpers interface* 58e⟩+≡
```
val unwrap : 'a wrap -> 'a
```

59b ⟨*AST helpers interface* 58e⟩+≡
```
val unparen : tok * 'a * tok -> 'a
val unbrace : tok * 'a * tok -> 'a
val unbracket : tok * 'a * tok -> 'a
```

59c ⟨*AST helpers interface* 58e⟩+≡
```
val untype : 'a * 'b -> 'a
```

59d ⟨*AST helpers interface* 58e⟩+≡
```
val get_type : expr -> Type_php.phptype
val set_type : expr -> Type_php.phptype -> unit
```

59e ⟨*AST helpers interface* 58e⟩+≡
```
val rewrap_str : string -> info -> info
val is_origintok : info -> bool
val al_info : 'a -> 'b
val compare_pos : info -> info -> int
```

59f ⟨*AST helpers interface* 58e⟩+≡
```
val noType : unit -> exp_info
val noTypeVar : unit -> lvalue_info
val noScope : unit -> Scope_php.phpscope ref
val noFtype : unit -> Type_php.phptype
```

Chapter 5

The Visitor Interface

5.1 Motivations

Why this module ? The problem is that one often needs to write analysis that needs only to specify actions for a few specific cases, such as the function call case, and recurse for the other cases, but writing the recursion code of those other cases is actually what can take the most time. It is mostly boilerplate code, but it still takes time to write it (and to not make typo).

Here is a simplification of an AST (of C, but the motivations are the same for PHP) to illustrate the problem:

```
type ctype =
 | Basetype of ...
 | Pointer of ctype
 | Array of expression option * ctype
 | ...
and expression =
 | Ident of string
 | FunCall of expression * expression list
 | Postfix of ...
 | RecordAccess of ..
 | ...
and statement =
 ...
and declaration =
 ...
and program =
 ...
```

What we want is really write code like

```
 let my_analysis program =
```

```
analyze_all_expressions program (fun expr ->
  match expr with
  | FunCall (e, es) -> do_something()
  | _ -> <find_a_way_to_recurse_for_all_the_other_cases>
  )
```

The problem is how to write `analyze_all_expressions` and `find_a_way_to_recurse_for_all_the_other`
? Our solution is to mix the ideas of visitor, pattern matching, and continuation. Here is how it looks like using our hybrid technique:

```
let my_analysis program =
 Visitor.visit_iter program {
   Visitor.kexpr = (fun k e ->
    match e with
    | FunCall (e, es) -> do_something()
    | _ -> k e
   );
  }
```

You can of course also give action *hooks* for `kstatement`, `ktype`, etc, but we don't overuse visitors and so it would be stupid to provide `kfunction_call`, `kident`, `kpostfix` hooks as one can just use pattern matching with `kexpr` to achieve the same effect.

5.2 Quick glance at the implementation

It's quite tricky to implement the `visit_xxx` functions. The control flow can gets quite complicated with continuations. Here is an old but simpler version that will allow us to understand more easily the final version:

```
let (iter_expr:((expr -> unit) -> expr -> unit) -> expr -> unit)
 = fun f expr ->
  let rec k e =
    match e with
    | Constant c -> ()
    | FunCall  (e, es)         ->  f k e; List.iter (f k) es
    | CondExpr (e1, e2, e3)    -> f k e1; f k e2; f k e3
    | Sequence (e1, e2)        -> f k e1; f k e2;
    | Assignment (e1, op, e2)  -> f k e1; f k e2;

    | Postfix  (e, op) -> f k e
    | Infix    (e, op) -> f k e
    | Unary    (e, op) -> f k e
    | Binary   (e1, op, e2) -> f k e1; f k  e2;

    | ArrayAccess    (e1, e2) -> f k e1; f k e2;
```

```
    | RecordAccess   (e, s) -> f k e
    | RecordPtAccess (e, s) -> f k e

    | SizeOfExpr  e -> f k e
    | SizeOfType  t -> ()

  in f k expr
```

We first define a default continuation function `k` and pass it to the `f` function passed itself as a parameter to the visitor `iter_expr` function. Here is how to use our visitor generator:

```
let ex1 = Sequence (Sequence (Constant (Ident "1"), Constant (Ident "2")),
                             Constant (Ident "4"))

let test_visit =
  iter_expr (fun k e ->  match e with
  | Constant (Ident x) -> Common.pr2 x
  | rest -> k rest
  ) ex1
```

The output should be

```
1
2
4
```

That is with only 4 lines of code (the code of `test_visit`), we were able to visit any ASTs and most of the boilerplate handling code for recursing on the appropriate constructors is managed for us.

The preceding code works fine for visiting one type, but usually an AST is a set of mutually recursive types (statements, expressions, definitions). So we need a way to define mutliple hooks, hence the use of a record with one field per type: `kexpr`, `kstatement`, etc. We must then define multiple continuations functions `k` that take care to call each other. See the implementation code for more details.

5.3 Iterator visitor

Here is the high level structure of `visitor_php.mli`:

62 ⟨*visitor_php.mli* 62⟩≡

```
  open Ast_php

  ⟨type visitor_in 63a⟩
  ⟨type visitor_out 63d⟩

  ⟨visitor functions 63b⟩
```

63a ⟨*type visitor_in* 63a⟩≡
```
(* the hooks *)
type visitor_in = {
  kexpr: (expr  -> unit) * visitor_out -> expr  -> unit;
  kstmt: (stmt  -> unit) * visitor_out -> stmt  -> unit;
  ktop: (toplevel -> unit) * visitor_out -> toplevel  -> unit;
  klvalue: (lvalue -> unit) * visitor_out -> lvalue  -> unit;
  kconstant: (constant -> unit) * visitor_out -> constant  -> unit;
  kstmt_and_def: (stmt_and_def -> unit) * visitor_out -> stmt_and_def  -> unit;
  kencaps: (encaps -> unit) * visitor_out -> encaps -> unit;
  kclass_stmt: (class_stmt -> unit) * visitor_out -> class_stmt -> unit;
  kparameter: (parameter -> unit) * visitor_out -> parameter -> unit;

  kfully_qualified_class_name:
    (fully_qualified_class_name -> unit) * visitor_out ->
    fully_qualified_class_name -> unit;
  kclass_name_reference:
    (class_name_reference -> unit) * visitor_out ->
    class_name_reference -> unit;
  khint_type: (hint_type -> unit) * visitor_out -> hint_type -> unit;

  kcomma: (unit -> unit) * visitor_out -> unit -> unit;

  kinfo: (info -> unit)  * visitor_out -> info  -> unit;
}
```

63b ⟨*visitor functions* 63b⟩≡
```
val default_visitor : visitor_in
```

63c ⟨*visitor functions* 63b⟩+≡
```
val mk_visitor: visitor_in -> visitor_out
```

63d ⟨*type visitor_out* 63d⟩≡
```
and visitor_out = {
  vexpr: expr  -> unit;
  vstmt: stmt -> unit;
  vtop: toplevel -> unit;
  vstmt_and_def: stmt_and_def -> unit;
  vlvalue: lvalue -> unit;
  vargument: argument -> unit;
  vclass_stmt: class_stmt -> unit;
  vinfo: info -> unit;
  vprogram: program -> unit;
}
```

64a ⟨*visitor functions* 63b⟩+≡
```
val do_visit_with_ref:
  ('a list ref -> visitor_in) ->
  (visitor_out -> unit) -> 'a list
```

5.4 pfff -visit_php

64b ⟨*test_parsing_php actions* 26e⟩+≡
```
  "-visit_php", "   <file>",
    Common.mk_action_1_arg test_visit_php;
```

64c ⟨*test_visit_php* 64c⟩≡
```
let test_visit_php file =
  let (ast2,_stat) = Parse_php.parse file in
  let ast = Parse_php.program_of_program2 ast2 in

  let hooks = { Visitor_php.default_visitor with
    Visitor_php.kinfo = (fun (k, vx) info ->
      let s = Ast_php.str_of_info info in
      pr2 s;
    );

    Visitor_php.kexpr = (fun (k, vx) e ->
      match fst e with
      | Ast_php.Scalar x ->
          pr2 "scalar";
          k e
      | _ -> k e
    );
  } in
  let visitor = Visitor_php.mk_visitor hooks in
  ast +> List.iter visitor.Visitor_php.vtop
```

Chapter 6

Unparsing Services

6.1 Raw AST printing

We have already mentionned in Sections 4.1.2 and 4.4.3 the use of the PHP AST pretty printer, callable through `pfff -dump_ast`. Here is a reminder:

```
$ ./pfff -dump_ast tests/inline_html.php
((StmtList
  ((InlineHtml ("'<html>\n'" ""))
   (Echo "" (((Scalar (Constant (String ('foo' "")))) ((t (Unknown))))) "")
   (InlineHtml ("'</html>\n'" ""))))
 (FinalDef ""))
```

One can also use `pfff.top` to leverage the builtin pretty printer of OCaml (Section 4.1.2).

The actual functions used by `-dump_ast` are in the `sexp_ast_php.mli` file. The word sexp is for s-expression (see `http://en.wikipedia.org/wiki/S-expression`), which is the way LISP code and data are usually encoded[1], which is also a convenient and compact way to print complex hierarchical structures (and a better way than the very verbose XML).

Here are the functions:

65 ⟨*sexp_ast_php.mli* 65⟩≡

```
  ⟨sexp_ast_php flags 66a⟩

  val string_of_program:  Ast_php.program  -> string
  val string_of_toplevel: Ast_php.toplevel -> string
  val string_of_expr:     Ast_php.expr     -> string
  val string_of_phptype:     Type_php.phptype  -> string

  ⟨sexp_ast_php raw sexp 66c⟩
```

[1] s-expressions are the ASTs of LISP, if that was not confusing enough already

The pretty printer can be configured through global variables:

66a ⟨*sexp_ast_php flags* 66a⟩≡
```
val show_info:      bool ref
val show_expr_info: bool ref
val show_annot: bool ref
```

to show or hide certain information. For instance `-dump_ast` by default does not show the concrete position information of the tokens and so set `show_info` to false before calling `string_of_program`.

Note that the code in `sexp_ast_php.ml` is mostly auto-generated from `ast_php.mli`. Indeed it is very tedious to manually write such code. I have written a small program called `ocamltarzan` (see [8]) to auto generate the code (which then uses a library called `sexplib`, included in `commons/`). `ocamltarzan` assumes the presence of special marks in type definitions[2], hence the use of the following snippet in diffent places in the code:

66b ⟨*tarzan annotation* 66b⟩≡
```
  (* with tarzan *)
```

As the generated code is included in the source, you don't have to install `ocamltarzan` to compile pfff. You may need it only if you modify `ast_php.mli` in a complex way and you want to refresh the pretty printer code. If the change is small, you can usually hack directly the generated code and extend it.

66c ⟨*sexp_ast_php raw sexp* 66c⟩≡
```
(* used by json_ast_php *)
val sexp_of_program: Ast_php.program -> Sexp.t
(* used by demos/justin.ml *)
val sexp_of_static_scalar: Ast_php.static_scalar -> Sexp.t
```

6.2 pfff -dump_ast

66d ⟨*test_parsing_php actions* 26e⟩+≡
```
    (* an alias for -sexp_php *)
    "-dump_ast", "   <file>",
      Common.mk_action_1_arg test_sexp_php;
```

66e ⟨*test_parsing_php actions* 26e⟩+≡
```
    "-sexp_php", "   <file>",
      Common.mk_action_1_arg test_sexp_php;
```

[2]For those familiar with Haskell, this is similar to the use of the `deriving` keyword

67a ⟨*test_sexp_php* 67a⟩≡

```
let test_sexp_php file =
  let (ast2,_stat) = Parse_php.parse file in
  let ast = Parse_php.program_of_program2 ast2 in
  (* let _ast = Type_annoter.annotate_program !Type_annoter.initial_env ast *)

  Sexp_ast_php.show_info := false;
  let s = Sexp_ast_php.string_of_program ast in
  pr2 s;
  ()
```

67b ⟨*test_parsing_php actions* 26e⟩+≡

```
  (* an alias for -sexp_php *)
  "-dump_full_ast", "   <file>",
    Common.mk_action_1_arg test_sexp_full_php;
```

67c ⟨*test_sexp_php* 67a⟩+≡

```
let test_sexp_full_php file =
  let (ast2,_stat) = Parse_php.parse file in
  let ast = Parse_php.program_of_program2 ast2 in

  Sexp_ast_php.show_info := true;
  let s = Sexp_ast_php.string_of_program ast in
  pr2 s;
  ()
```

6.3 Exporting JSON data

pfff can also export the JSON representation of a PHP AST, programmatically via `json_ast_php.ml` or interactively via `pfff -json`. One can then import this data in other languages with JSON support such as Python (or PHP). Here is an excerpt of the exported JSON of `demos/foo1.php`:

```
$ ./pfff -json demos/foo1.php
[
  [
    "FuncDef",
    {
      "f_tok": {
        "pinfo": [
          "OriginTok",
          {
            "str": "function",
            "charpos": 6,
            "line": 2,
```

```
            "column": 0,
            "file": "demos/foo1.php"
          }
        ],
        "comments": []
      },
      "f_ref": [],
      "f_name": [
        "Name",
        [
          "'foo'",
...
```


The JSON pretty printer is automatically generated from `ast_php.mli` so there is an exact correspondance between the constructor names in the OCaml types and the strings or fields in the JSON data. One can thus use the types documentation in this manual to translate that into JSON. For instance here is a port of `show_function_calls.ml` seen in Section 2.1 in Python:

68a ⟨*show_function_calls.py* 68a⟩≡

```
TODO basic version. Search for nodes with FunCallSimple
and extract position information from children.
Is there a visitor library for JSON data in Python or PHP ?
Is there XPATH for JSON ?
```

While pfff makes it possible to analyze PHP code in other languages, thanks to JSON, we strongly discourage coding complex static analysis or transformations in other languages. The big advantage of OCaml (or Haskell) and so of pfff is its strong pattern matching capability and type checking which are ideal for such tasks. Moreover pfff provides more than just an AST manipulation library. Indeed `pfff/analyzis_php` gives access to more services such as control-flow graphs, caller/callee analysis (inluding for virtual methods using object aliasing analysis), etc.

Here are the functions defined by `json_ast_php.mli`:

68b ⟨*json_ast_php.mli* 68b⟩≡

```

⟨json_ast_php flags 68c⟩

val string_of_program:  Ast_php.program  -> string
val string_of_toplevel: Ast_php.toplevel -> string
val string_of_expr:     Ast_php.expr     -> string
```

68c ⟨*json_ast_php flags* 68c⟩≡

6.4 pfff -json

69a ⟨*test_parsing_php actions* 26e⟩+≡

```
(* an alias for -sexp_php *)
"-json", "   <file> export the AST of file into JSON",
  Common.mk_action_1_arg test_json_php;
```

69b ⟨*test_json_php* 69b⟩≡

```
let test_json_php file =
  let (ast2,_stat) = Parse_php.parse file in
  let ast = Parse_php.program_of_program2 ast2 in

  let s = Json_ast_php.string_of_program ast in
  pr2 s;
  ()
```

6.5 Style preserving unparsing

69c ⟨*unparse_php.mli* 69c⟩≡

```

val string_of_program2: Parse_php.program2 -> string

val string_of_toplevel: Ast_php.toplevel -> string
```

Chapter 7

Other Services

This chapter describes the other services provided by files in `parsing_php/`. For the static analysis services of pfff (control-flow and data-flow graphs, caller/callee graphs, module dependencies, type inference, source-to-source transformations, PHP code pattern matching, etc), see the `Analysis_php.pdf` manual. For explanations about the semantic PHP source code visualizer and explorer `pfff_browser`, see the `Gui_php.pdf` manual.

7.1 Extra accessors, extractors, wrappers

70a ⟨*lib_parsing_php.mli* 70a⟩≡

```
val ii_of_toplevel: Ast_php.toplevel -> Ast_php.info list
val ii_of_expr: Ast_php.expr -> Ast_php.info list
val ii_of_stmt: Ast_php.stmt -> Ast_php.info list
val ii_of_argument: Ast_php.argument -> Ast_php.info list
val ii_of_lvalue: Ast_php.lvalue -> Ast_php.info list
```

70b ⟨*lib_parsing_php.mli* 70a⟩+≡

```
(* do via side effects *)
val abstract_position_info_toplevel: Ast_php.toplevel -> Ast_php.toplevel
val abstract_position_info_expr: Ast_php.expr -> Ast_php.expr
val abstract_position_info_program: Ast_php.program -> Ast_php.program
```

70c ⟨*lib_parsing_php.mli* 70a⟩+≡

```
val range_of_origin_ii: Ast_php.info list -> (int * int) option
val min_max_ii_by_pos: Ast_php.info list -> Ast_php.info * Ast_php.info
```

70d ⟨*lib_parsing_php.mli* 70a⟩+≡

```
val get_all_funcalls_in_body: Ast_php.stmt_and_def list -> string list
val get_all_funcalls_ast: Ast_php.toplevel -> string list
val get_all_constant_strings_ast: Ast_php.toplevel -> string list
val get_all_funcvars_ast: Ast_php.toplevel -> string (* dname *) list
```

7.2 Debugging pfff, `pfff -<flags>`

71a ⟨*flag_parsing_php.ml* 71a⟩≡
```
let verbose_parsing = ref true
let verbose_lexing = ref true
let verbose_visit = ref true
```

71b ⟨*flag_parsing_php.ml* 71a⟩+≡
```
let cmdline_flags_verbose () = [
  "-no_verbose_parsing", Arg.Clear verbose_parsing , "  ";
  "-no_verbose_lexing", Arg.Clear verbose_lexing , "  ";
  "-no_verbose_visit", Arg.Clear verbose_visit , "  ";
]
```

71c ⟨*flag_parsing_php.ml* 71a⟩+≡
```
let debug_lexer   = ref false
```

71d ⟨*flag_parsing_php.ml* 71a⟩+≡
```
let cmdline_flags_debugging () = [
  "-debug_lexer",        Arg.Set  debug_lexer , " ";
]
```

71e ⟨*flag_parsing_php.ml* 71a⟩+≡
```
let show_parsing_error = ref true
```

71f ⟨*flag_parsing_php.ml* 71a⟩+≡
```
let short_open_tag = ref true

let verbose_pp = ref false
let xhp_command = "xhpize"
(* in facebook context, we want -xhp to be the default *)
let pp_default = ref (Some xhp_command: string option)

let cmdline_flags_pp () = [
  "-pp", Arg.String (fun s -> pp_default := Some s),
  " <cmd> optional preprocessor (e.g. xhpize)";
  "-xhp", Arg.Unit (fun () -> pp_default := Some xhp_command),
  " using xhpize as a preprocessor (default)";
  "-no_xhp", Arg.Unit (fun () -> pp_default := None),
  " ";
  "-verbose_pp", Arg.Set verbose_pp,
  " ";
]
```

7.3 Testing pfff components

72a ⟨*test_parsing_php.mli* 72a⟩≡
```
val test_parse_php  : Common.filename list -> unit
```

72b ⟨*test_parsing_php.mli* 72a⟩+≡
```
val test_tokens_php : Common.filename -> unit
val test_sexp_php   : Common.filename -> unit
val test_json_php   : Common.filename -> unit
val test_visit_php  : Common.filename -> unit
```

72c ⟨*test_parsing_php.mli* 72a⟩+≡
```
val actions : unit -> (string * string * Common.action_func) list
```

7.4 `pfff.top`

7.5 Interoperability (JSON and thrift)

We have already described in Section 6.3 that pfff can export the JSON or sexp of an AST. This makes it possible to somehow interoperate with other programming languages.

TODO thrift so better typed interoperability

See also `pfff/ffi/`.

Part II

pfff Internals

Chapter 8

Implementation Overview

8.1 Introduction

The goal of this document is not to explain how a compiler frontend works, or how to use Lex and Yacc, but just how the pfff parser is concretely implemented. We assume a basic knowledge of the literature on compilers such as [5] or [6].

8.2 Code organization

Figure 8.1 presents the graph of dependencies between ml files.

8.3 `parse_php.ml`

The code of the parser is quite straightforward as it mostly consists of Lex and Yacc specifications. The few subtelities are:

- the need for contextual lexing and state management in the lexer to cope with the fact that one can embed HTML in PHP code and vice versa which in principle requires two different lexers and parsers. In practice our HTML lexer is very simple and just returns a RAW string for the whole HTML snippet (no tree) and we have slightly hacked around `ocamllex` to makes the two lexers work together. In fact the need for interpolated strings and HereDocs (`<<<EOF` constructs) also imposes some constraints on the lexer.
- this free mixing of HTML and PHP should normally also have consequences on the grammar and the AST, with the need for mutually recursive rules and types. In practice the parser internally transforms HTML snippets in sort of `echo` statements so that the AST is almost oblivious to this PHP syntactic sugar.

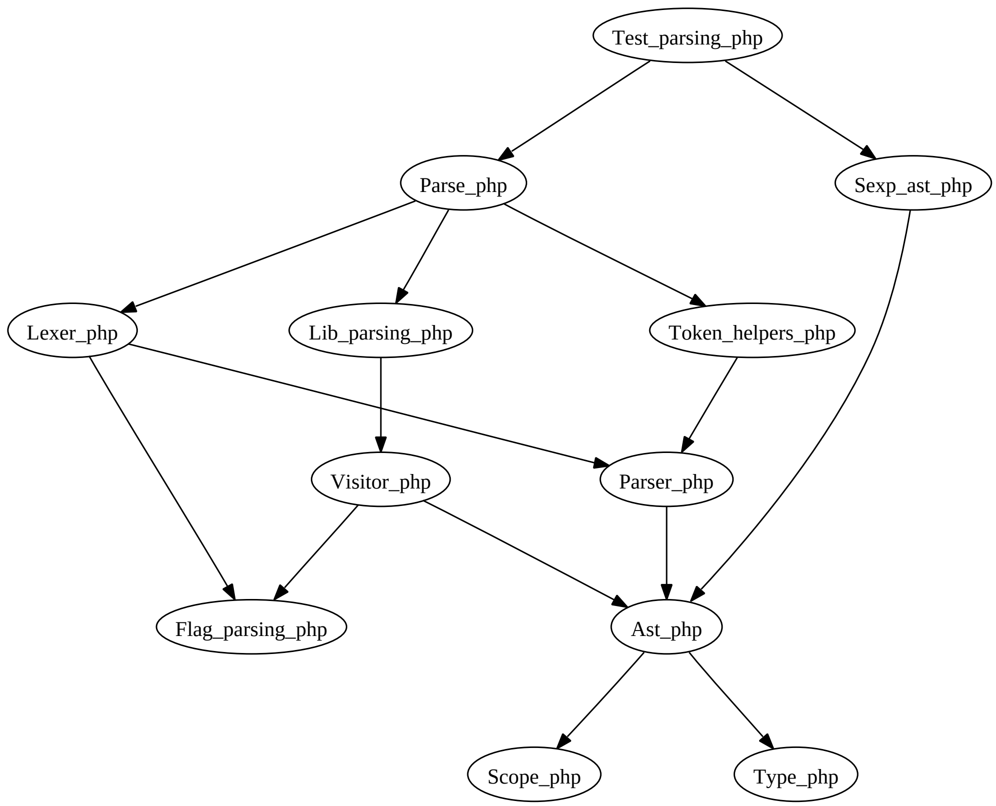

Figure 8.1: API dependency graph between `ml` files

- the need to remember the position information (line and column numbrers) of the different PHP elements in the AST imposed another small hack around `ocamllex` which by default offer very few support for that.
- managing XHP is not yet done

In the following chapters we describe almost the full code of the pfff parser. To avoid some repetitions, and because some code are really boring, we sometimes use the literate programming prefix `repetitive` in chunk names to mean code that mostly follow the structure of the code you just seen but handle other similar constructs. .

Here is the high-level structure of `parse_php.ml`:

76 ⟨*parse_php.ml* 76⟩≡

```
⟨Facebook copyright 9⟩

open Common

⟨parse_php module aliases 139a⟩

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

⟨type program2 25b⟩

⟨function program_of_program2 139b⟩

(*****************************************************************************)
(* Wrappers *)
(*****************************************************************************)
let pr2_err, pr2_once = Common.mk_pr2_wrappers Flag.verbose_parsing

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨parse_php helpers 139c⟩

(*****************************************************************************)
(* Error diagnostic  *)
(*****************************************************************************)
⟨parse_php error diagnostic 140b⟩

(*****************************************************************************)
(* Stat *)
(*****************************************************************************)
⟨type parsing_stat 26c⟩
```

```
⟨parse_php stat function 141a⟩

(******************************************************************************)
(* Lexing only *)
(******************************************************************************)
⟨function tokens 85c⟩

(******************************************************************************)
(* Helper for main entry point *)
(******************************************************************************)
⟨parse tokens_state helper 87a⟩

(******************************************************************************)
(* Main entry point *)
(******************************************************************************)
⟨Parse_php.parse 78⟩

(******************************************************************************)

let (expr_of_string: string -> Ast_php.expr) = fun s ->
  let tmpfile = Common.new_temp_file "pff_expr_of_s" "php" in
  Common.write_file tmpfile ("<?php \n" ^ s ^ ";\n");

  let (ast2, _stat) = parse tmpfile in
  let ast = program_of_program2 ast2 in

  let res =
    (match ast with
    | [Ast.StmtList [Ast.ExprStmt (e, _tok)];Ast.FinalDef _] -> e
  | _ -> failwith "only expr pattern are supported for now"
  )
  in
  Common.erase_this_temp_file tmpfile;
  res


let (xdebug_expr_of_string: string -> Ast_php.expr) = fun s ->
  let lexbuf = Lexing.from_string s in
  let rec mylex lexbuf =
    let tok = Lexer_php.st_in_scripting lexbuf in
    if TH.is_comment tok
    then mylex lexbuf
    else tok

  in
```

```
  let expr = Parser_php.expr mylex lexbuf in
  expr
```

Here is the skeleton of the main entry point:

78 ⟨*Parse_php.parse* 78⟩≡

```
let parse2 ?(pp=(!Flag.pp_default)) filename =

  let orig_filename = filename in
  let filename =
    match pp with
    | None -> orig_filename
    | Some cmd ->
        Common.profile_code "Parse_php.pp" (fun () ->

          let pp_flag = if !Flag.verbose_pp then "-v" else "" in

          (* The following requires the preprocessor command to
           * support the -q command line flag.
           *
           * Maybe a little bit specific to XHP and xhpize ... But
           * because I use as a convention that 0 means no_need_pp, if
           * the preprocessor does not support -q, it should return an
           * error code, in which case we will fall back to the regular
           * case. *)
          let cmd_need_pp =
            spf "%s -q %s %s" cmd pp_flag filename in
          if !Flag.verbose_pp then pr2 (spf "executing %s" cmd_need_pp);
          let ret = Sys.command cmd_need_pp in
          if ret = 0
          then orig_filename
          else begin

            let tmpfile = Common.new_temp_file "pp" ".pphp" in
            let fullcmd =
              spf "%s %s %s > %s" cmd pp_flag filename tmpfile in
            if !Flag.verbose_pp then pr2 (spf "executing %s" fullcmd);
            let ret = Sys.command fullcmd in
            if ret <> 0
            then failwith "The preprocessor command returned an error code";
            tmpfile
          end
        )
  in
```

```
let stat = default_stat filename in
let filelines = Common.cat_array filename in

let toks = tokens filename in

(* The preprocessor command will generate a file in /tmp which means
 * errors or further analysis will report position information
 * on this tmp file. This can be inconvenient. If the
 * preprocessor maintain line positions (which is the case for instance
 * with xhp), at least we can slightly improve the situation by
 * changing the .file field in parse_info.
 *
 * TODO: certain preprocessor such as xhp also remove comments.
 * It could be useful to merge the original comments in the original
 * files with the tokens in the expanded file.
 *)
let toks = toks +> List.rev_map (fun tok ->
  tok +> TH.visitor_info_of_tok (fun ii ->
    let pinfo = Ast.pinfo_of_info ii in
    { ii with Ast.pinfo =
        match pinfo with
        | Ast.OriginTok pi ->
            Ast.OriginTok { pi with
              Common.file = orig_filename;
            }
        | Ast.FakeTokStr _
        | Ast.Ab
          -> pinfo
    })
)  +> List.rev (* ugly, but need tail-call rev_map and so this rev *)
in

let tr = mk_tokens_state toks in

let checkpoint = TH.line_of_tok tr.current in

let lexbuf_fake = Lexing.from_function (fun buf n -> raise Impossible) in
let elems =
  try (
    (* -------------------------------------------------- *)
    (* Call parser *)
    (* -------------------------------------------------- *)
    Left
      (Common.profile_code "Parser_php.main" (fun () ->
        (Parser_php.main (lexer_function tr) lexbuf_fake)
      ))
```

```
  ) with e ->

    let line_error = TH.line_of_tok tr.current in

    let _passed_before_error = tr.passed in
    let current = tr.current in

    (* no error recovery, the whole file is discarded *)
    tr.passed <- List.rev toks;

    let info_of_bads = Common.map_eff_rev TH.info_of_tok tr.passed in

    Right (info_of_bads, line_error, current, e)
in

match elems with
| Left xs ->
    stat.correct <- (Common.cat filename +> List.length);

    distribute_info_items_toplevel xs toks filename,
    stat
| Right (info_of_bads, line_error, cur, exn) ->

    (match exn with
    | Lexer_php.Lexical _
    | Parsing.Parse_error
        (*| Semantic_c.Semantic _  *)
      -> ()
    | e -> raise e
    );

    if !Flag.show_parsing_error
    then
      (match exn with
      (* Lexical is not anymore launched I think *)
      | Lexer_php.Lexical s ->
          pr2 ("lexical error " ^s^ "\n =" ^ error_msg_tok cur)
      | Parsing.Parse_error ->
          pr2 ("parse error \n = " ^ error_msg_tok cur)
            (* | Semantic_java.Semantic (s, i) ->
                 pr2 ("semantic error " ^s^ "\n ="^ error_msg_tok tr.current)
        *)
      | e -> raise Impossible
      );
    let checkpoint2 = Common.cat filename +> List.length in
```

```
    if !Flag.show_parsing_error
    then print_bad line_error (checkpoint, checkpoint2) filelines;

    stat.bad     <- Common.cat filename +> List.length;

    let info_item = mk_info_item filename (List.rev tr.passed) in
    [Ast.NotParsedCorrectly info_of_bads, info_item],
    stat
```

81 ⟨*Parse_php.parse* 78⟩+≡

```
let parse ?pp a =
  Common.profile_code "Parse_php.parse" (fun () -> parse2 ?pp a)
```

The important parts are the calls to `tokens`, a wrapper around the `ocamllex` lexer, and to `Parser_php.main`, the toplevel grammar rule automatically generated by `ocamlyacc`. This last function takes as parameters a function providing a stream of tokens and a lexing buffer. Because we had to hack around `ocamllex`, the streaming function and buffer do not come directly from a call to `Lexing.from_channel` coupled with an `ocamllex` rule specified in `lexer_php.mll`, which is how things are usually done. Instead we pass a custom build steaming function `lexer_function` and a fake buffer. Both `tokens` and `lexer_function` will be explained in Chapter 9 while `Parser_php.main` will be explained in 10. The remaining code used in the code above will be finally described in Chapter 11.

Chapter 9

Lexer

9.1 Overview

The code in `lexer_php.mll` is mostly a copy paste of the Flex scanner in the PHP Zend source code (included in `pfff/docs/official-grammar/5.2.11/zend_language_scanner.l`) adapted for `ocamllex`:

82 ⟨*lexer_php.mll* 82⟩≡

```
{
⟨Facebook copyright 9⟩

open Common

⟨basic pfff module open and aliases 158⟩

open Parser_php

(*****************************************************************************)
(* Wrappers *)
(*****************************************************************************)
let pr2, pr2_once = Common.mk_pr2_wrappers Flag.verbose_lexing

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
exception Lexical of string

(* ---------------------------------------------------------------------- *)
⟨lexer helpers 88e⟩

(* ---------------------------------------------------------------------- *)
⟨keywords_table hash 94d⟩
```

```
(* ---------------------------------------------------------------------- *)
⟨type state_mode 84b⟩

⟨lexer state trick helpers 84c⟩

}

(*****************************************************************************)
⟨regexp aliases 84a⟩


(*****************************************************************************)
⟨rule st_in_scripting 90a⟩

(*****************************************************************************)
⟨rule initial 89⟩

(*****************************************************************************)
⟨rule st_looking_for_property 102⟩

(*****************************************************************************)
⟨rule st_looking_for_varname 103a⟩

(*****************************************************************************)
⟨rule st_var_offset 103b⟩

(*****************************************************************************)
⟨rule st_double_quotes 100a⟩

(* ----------------------------------------------------------------------- *)
⟨rule st_backquote 101a⟩

(* ----------------------------------------------------------------------- *)
⟨rule st_start_heredoc 101b⟩

(*****************************************************************************)
⟨rule st_comment 91c⟩

⟨rule st_one_line_comment 92a⟩
```

The file defines mainly the functions `Lexer_php.st_initial` and `Lexer_php.st_scripting`, auto generated by `ocamllex`, to respectively lex a file in HTML mode (the default initial mode) and PHP mode (aka scripting mode). As usual with Lex and Yacc the tokens are actually specified in the Yacc file (see Section 10.14), hence the `open Parser_php` at the beginning of the file.

84a ⟨*regexp aliases* 84a⟩≡
```
let ANY_CHAR = (_ | ['\n'] )
```

9.2 Lex states and other `ocamllex` hacks

9.2.1 Contextual lexing

The lexer needs a contextual capability. This is because PHP allows to embed HTML snippets directly into the code, where tokens have a different meaning. This is also because some tokens like `if` mean something in one context (a statement keyword) and something else in another (they are allowed as name of properties for instance). Also, like in Perl, PHP allows HereDoc, and a few other tricks that makes the job of the lexer slightly more complicated than in other programming languages.

Contextual lexing is available in Flex but not really in `ocamllex`. So the lexing logic is splitted into this file and into a small function in `parse_php.ml` that handles some state machine. See also the `state_mode` type below.

84b ⟨*type state_mode* 84b⟩≡
```
type state_mode =
  | INITIAL
  | ST_IN_SCRIPTING
  (* handled by using ocamllex ability to define multiple lexers
   * | ST_COMMENT
   * | ST_DOC_COMMENT
   * | ST_ONE_LINE_COMMENT
   *)
  | ST_DOUBLE_QUOTES
  | ST_BACKQUOTE
  | ST_LOOKING_FOR_PROPERTY
  | ST_LOOKING_FOR_VARNAME
  | ST_VAR_OFFSET
  | ST_START_HEREDOC of string
```

84c ⟨*lexer state trick helpers* 84c⟩≡
```
⟨lexer state global variables 84d⟩
⟨lexer state global reinitializer 85a⟩

⟨lexer state function hepers 85b⟩
```

84d ⟨*lexer state global variables* 84d⟩≡
```
let default_state = INITIAL

let _mode_stack =
  ref [default_state]
```

85a ⟨*lexer state global reinitializer* 85a⟩≡
```
let reset () =
  _mode_stack := [default_state];
  ⟨auxillary reset lexing actions 88b⟩
  ()
```

85b ⟨*lexer state function hepers* 85b⟩≡
```
let rec current_mode () =
  try
    Common.top !_mode_stack
  with Failure("hd") ->
    pr2("LEXER: mode_stack is empty, defaulting to INITIAL");
    reset();
    current_mode ()
```

85c ⟨*function tokens* 85c⟩≡
```
let tokens2 file =
  let table     = Common.full_charpos_to_pos_large file in

  Common.with_open_infile file (fun chan ->
    let lexbuf = Lexing.from_channel chan in

    Lexer_php.reset();
    try
      ⟨function phptoken 86a⟩

      let rec tokens_aux acc =
        let tok = phptoken lexbuf in

        if !Flag.debug_lexer then Common.pr2_gen tok;

        ⟨fill in the line and col information for tok 86c⟩

        if TH.is_eof tok
        then List.rev (tok::acc)
        else tokens_aux (tok::acc)
    in
    tokens_aux []
  with
  | Lexer_php.Lexical s ->
      failwith ("lexical error " ^ s ^ "\n =" ^
                   (Common.error_message file (lexbuf_to_strpos lexbuf)))
  | e -> raise e
 )
```

85d ⟨*function tokens* 85c⟩+≡

```
let tokens a =
  Common.profile_code "Parse_php.tokens" (fun () -> tokens2 a)
```

86a ⟨*function phptoken* 86a⟩≡

```
let phptoken lexbuf =
  ⟨yyless trick in phptoken 88d⟩
      (match Lexer_php.current_mode () with
      | Lexer_php.INITIAL ->
          Lexer_php.initial lexbuf
      | Lexer_php.ST_IN_SCRIPTING ->
          Lexer_php.st_in_scripting lexbuf
      | Lexer_php.ST_DOUBLE_QUOTES ->
          Lexer_php.st_double_quotes lexbuf
      | Lexer_php.ST_BACKQUOTE ->
          Lexer_php.st_backquote lexbuf
      | Lexer_php.ST_LOOKING_FOR_PROPERTY ->
          Lexer_php.st_looking_for_property lexbuf
      | Lexer_php.ST_LOOKING_FOR_VARNAME ->
          Lexer_php.st_looking_for_varname lexbuf
      | Lexer_php.ST_VAR_OFFSET ->
          Lexer_php.st_var_offset lexbuf
      | Lexer_php.ST_START_HEREDOC s ->
          Lexer_php.st_start_heredoc s lexbuf
      )
in
```

86b ⟨*lexer state function hepers* 85b⟩+≡

```
let push_mode mode = Common.push2 mode _mode_stack
let pop_mode () = ignore(Common.pop2 _mode_stack)

(* What is the semantic of BEGIN() in flex ? start from scratch with empty
 * stack ?
 *)
let set_mode mode =
  pop_mode();
  push_mode mode;
  ()
```

9.2.2 Position information

86c ⟨*fill in the line and col information for tok* 86c⟩≡

```
let tok = tok +> TH.visitor_info_of_tok (fun ii ->
{ ii with Ast.pinfo=
  (* could assert pinfo.filename = file ? *)
       match Ast.pinfo_of_info ii with
       | Ast.OriginTok pi ->
```

```
                    Ast.OriginTok
                      (Common.complete_parse_info_large file table pi)
          | Ast.FakeTokStr _
          | Ast.Ab
                  -> raise Impossible
             })
  in
```

9.2.3 Filtering comments

Below you will see that we use a special lexing scheme. Why use this lexing scheme ? Why not classically give a regular lexer func to the parser ? Because we keep the comments in the lexer. Could just do a simple wrapper that when comment asks again for a token, but probably simpler to use the `cur_tok` technique.

87a ⟨*parse tokens_state helper* 87a⟩≡

```
  type tokens_state = {
    mutable rest :         Parser_php.token list;
    mutable current :      Parser_php.token;
    (* it's passed since last "checkpoint", not passed from the beginning *)
    mutable passed :       Parser_php.token list;
    (* if want to do some lalr(k) hacking ... cf yacfe.
     * mutable passed_clean : Parser_php_c.token list;
     * mutable rest_clean :   Parser_php_c.token list;
     *)
  }
```

87b ⟨*parse tokens_state helper* 87a⟩+≡

```
  let mk_tokens_state toks =
    {
      rest       = toks;
      current    = (List.hd toks);
      passed = [];
      (* passed_clean = [];
       * rest_clean = (toks +> List.filter TH.is_not_comment);
       *)
    }
```

87c ⟨*parse tokens_state helper* 87a⟩+≡

```
  (* Hacked lex. This function use refs passed by parse.
   * 'tr' means 'token refs'.
   *)
  let rec lexer_function tr = fun lexbuf ->
    match tr.rest with
    | [] -> (pr2 "LEXER: ALREADY AT END"; tr.current)
```

```
| v::xs ->
    tr.rest <- xs;
    tr.current <- v;
    tr.passed <- v::tr.passed;

    if TH.is_comment v ||
      (* TODO a little bit specific to FB ? *)
      (match v with
      | Parser_php.T_OPEN_TAG _ -> true
      | Parser_php.T_CLOSE_TAG _ -> true
      | Parser_php.T_OPEN_TAG_WITH_ECHO _ -> true
      | _ -> false
      )
    then lexer_function (*~pass*) tr lexbuf
    else v
```

9.2.4 Other hacks

88a ⟨*lexer state global variables* 84d⟩+≡

```
(* because ocamllex does not have the yyless feature, have to cheat *)
let _pending_tokens =
  ref ([]: Parser_php.token list)
```

88b ⟨*auxillary reset lexing actions* 88b⟩≡

```
  _pending_tokens := [];
```

88c ⟨*lexer state function hepers* 85b⟩+≡

```
let push_token tok =
  _pending_tokens := tok::!_pending_tokens
```

88d ⟨*yyless trick in phptoken* 88d⟩≡

```
  (* for yyless emulation *)
  match !Lexer_php._pending_tokens with
  | x::xs -> Lexer_php._pending_tokens := xs; x
  | [] ->
```

88e ⟨*lexer helpers* 88e⟩≡

```
(* pad: hack around ocamllex to emulate the yylesss of flex. It seems
 * to work.
 *)
let yyless n lexbuf =
  lexbuf.Lexing.lex_curr_pos <- lexbuf.Lexing.lex_curr_pos - n;
  let currp = lexbuf.Lexing.lex_curr_p in
  lexbuf.Lexing.lex_curr_p <- { currp with
    Lexing.pos_cnum = currp.Lexing.pos_cnum - n;
  }
```

9.3 Initial state (HTML mode)

89 ⟨*rule initial* 89⟩≡

```
and initial = parse

  | "<?php"([' ''\t']|NEWLINE)
      { set_mode ST_IN_SCRIPTING;
        T_OPEN_TAG(tokinfo lexbuf)
      }

  | "<?PHP"([' ''\t']|NEWLINE)
      {
        pr2 "BAD USE OF <PHP at initial state, replace by <?php";
        set_mode ST_IN_SCRIPTING;
        T_OPEN_TAG(tokinfo lexbuf)
      }

  | (([^'<']|"<"[^'?''%''s''<'])+(*{1,400}*))|"<s"|"<" {
      (* more? cf orinal lexer  *)
    T_INLINE_HTML(tok lexbuf, tokinfo lexbuf)
    }

  | "<?=" {
      (* XXX if short_tags normally, otherwise T_INLINE_HTML *)
      set_mode ST_IN_SCRIPTING;
      (* note that T_OPEN_TAG_WITH_ECHO is not mentionned in the grammar.
       * TODO It comes from an intermediate lexing phases ?
       *)
      T_OPEN_TAG_WITH_ECHO(tokinfo lexbuf);
    }

  | "<?" | "<script" WHITESPACE+ "language" WHITESPACE* "=" WHITESPACE *
           ("php"|"\"php\""|"\'php\'") WHITESPACE*">"
     {
       (* XXX if short_tags normally otherwise T_INLINE_HTML *)
       pr2 "BAD USE OF <? at initial state, replace by <?php";
       set_mode ST_IN_SCRIPTING;
       T_OPEN_TAG(tokinfo lexbuf);
     }

  (*------------------------------------------------------------------------ *)

  | eof { EOF (tokinfo lexbuf +> Ast.rewrap_str "") }
  | _ (* ANY_CHAR *) {
      if !Flag.verbose_lexing
      then pr2_once ("LEXER:unrecognised symbol, in token rule:"^tok lexbuf);
```

```
    TUnknown (tokinfo lexbuf)
  }
```

9.4 Script state (PHP mode)

90a ⟨*rule st_in_scripting* 90a⟩≡

```
rule st_in_scripting = parse

  (* ----------------------------------------------------------------------- *)
  (* spacing/comments *)
  (* ----------------------------------------------------------------------- *)
  ⟨comments rules 91a⟩

  (* ----------------------------------------------------------------------- *)
  (* Symbols *)
  (* ----------------------------------------------------------------------- *)
  ⟨symbol rules 92b⟩

  (* ----------------------------------------------------------------------- *)
  (* Keywords and ident *)
  (* ----------------------------------------------------------------------- *)
  ⟨keyword and ident rules 94b⟩

  (* ----------------------------------------------------------------------- *)
  (* Constant *)
  (* ----------------------------------------------------------------------- *)
  ⟨constant rules 95a⟩

  (* ----------------------------------------------------------------------- *)
  (* Strings *)
  (* ----------------------------------------------------------------------- *)
  ⟨strings rules 96a⟩

  (* ----------------------------------------------------------------------- *)
  (* Misc *)
  (* ----------------------------------------------------------------------- *)
  ⟨misc rules 99a⟩

  (* ----------------------------------------------------------------------- *)
  ⟨semi repetitive st_in_scripting rules for eof and error handling 90b⟩
```

90b ⟨*semi repetitive st_in_scripting rules for eof and error handling* 90b⟩≡

```
  | eof { EOF (tokinfo lexbuf +> Ast.rewrap_str "") }
  | _ {
      if !Flag.verbose_lexing
      then pr2_once ("LEXER:unrecognised symbol, in token rule:"^tok lexbuf);
      TUnknown (tokinfo lexbuf)
    }
```

9.4.1 Comments

This lexer generate tokens for comments which is very unusual for a compiler. Usually a compiler frontend will just drops everything that is not relevant to generate code. But in some contexts (refactoring, source code visualization) it is useful to keep those comments somehow in the AST. So one can not give this lexer as-is to the parsing function. The caller must preprocess it, e.g. by using techniques like `cur_tok` ref in `parse_php.ml` as described in Section 9.2.3.

91a ⟨*comments rules* 91a⟩≡

```
  | "/*" {
      let info = tokinfo lexbuf in
      let com = st_comment lexbuf in
      T_COMMENT(info +> tok_add_s com)
    }

  | "/**" { (* RESET_DOC_COMMENT(); *)
      let info = tokinfo lexbuf in
      let com = st_comment lexbuf in
      T_DOC_COMMENT(info +> tok_add_s com)
    }
  | "#"|"//" {
      let info = tokinfo lexbuf in
      let com = st_one_line_comment lexbuf in
      T_COMMENT(info +> tok_add_s com)
    }

  | WHITESPACE { T_WHITESPACE(tokinfo lexbuf) }
```

91b ⟨*regexp aliases* 84a⟩+≡

```
(* \x7f-\xff ???*)
let WHITESPACE = [' ' '\n' '\r' '\t']+
let TABS_AND_SPACES = [' ''\t']*
let NEWLINE = ("\r"|"\n"|"\r\n")
let WHITESPACEOPT = [' ' '\n' '\r' '\t']*
```

91c ⟨*rule st_comment* 91c⟩≡

```
and st_comment = parse
```

```
  | "*/" { tok lexbuf }

  (* noteopti: *)
  | [^'*']+ { let s = tok lexbuf in s ^ st_comment lexbuf }
  | "*"     { let s = tok lexbuf in s ^ st_comment lexbuf }
```

⟨*repetitive st_comment rules for error handling ??*⟩

92a ⟨*rule st_one_line_comment 92a*⟩≡

```
and st_one_line_comment = parse
  | "?"|"%"|">" { let s = tok lexbuf in s ^ st_one_line_comment lexbuf }
  | [^'\n' '\r' '?''%''>']* (ANY_CHAR as x)
      {
        (* what about yyless ??? *)
        let s = tok lexbuf in
        (match x with
        | '?' | '%' | '>' ->
            s ^ st_one_line_comment lexbuf
        | '\n' -> s
        | _ -> s
        )
      }
  | NEWLINE { tok lexbuf }
  | "?>"|"%>" {
      raise Todo
    }
```

⟨*repetitive st_one_line_comment rules for error handling ??*⟩

9.4.2 Symbols

92b ⟨*symbol rules 92b*⟩≡

```
  | '+' { TPLUS(tokinfo lexbuf) }      | '-' { TMINUS(tokinfo lexbuf) }
  | '*' { TMUL(tokinfo lexbuf) }       | '/' { TDIV(tokinfo lexbuf) }
  | '%' { TMOD(tokinfo lexbuf) }

  | "++" { T_INC(tokinfo lexbuf) }   | "--" { T_DEC(tokinfo lexbuf) }

  | "="  { TEQ(tokinfo lexbuf) }
```

⟨*repetitive symbol rules ??*⟩

92c ⟨*symbol rules 92b*⟩+≡

```
 (* Flex/Bison allow to use single characters directly as-is in the grammar
  * by adding this in the lexer:
  *
```

```
*       <ST_IN_SCRIPTING>{TOKENS} { return yytext[0];}
*
* We don't, so we have transformed all those tokens in proper tokens with
* a name in the parser, and return them in the lexer.
*)

| '.'  { TDOT(tokinfo lexbuf) }
| ','  { TCOMMA(tokinfo lexbuf) }
| '@'  { T__AT(tokinfo lexbuf) }
| "=>" { T_DOUBLE_ARROW(tokinfo lexbuf) }
| "~"  { TTILDE(tokinfo lexbuf) }
| ";"  { TSEMICOLON(tokinfo lexbuf) }
| "!"  { TBANG(tokinfo lexbuf) }
| "::" { TCOLCOL (tokinfo lexbuf) } (* was called T_PAAMAYIM_NEKUDOTAYIM *)

| '(' { TOPAR(tokinfo lexbuf) }  | ')' { TCPAR(tokinfo lexbuf) }
| '[' { TOBRA(tokinfo lexbuf) }  | ']' { TCBRA(tokinfo lexbuf) }

| ":" { TCOLON(tokinfo lexbuf) }
| "?" { TQUESTION(tokinfo lexbuf) }
(* semantic grep *)
| "..." { TDOTS(tokinfo lexbuf) }
```

93a ⟨*symbol rules* 92b⟩+≡

```
(* we may come from a st_looking_for_xxx context, like in string
 * interpolation, so seeing a } we pop_mode!
 *)
| '}' {
    pop_mode ();
    (* RESET_DOC_COMMENT(); ??? *)
    TCBRACE(tokinfo lexbuf)
  }
| '{' {
    push_mode ST_IN_SCRIPTING;
    TOBRACE(tokinfo lexbuf)
  }
```

93b ⟨*symbol rules* 92b⟩+≡

```
| ("->" as sym) (WHITESPACEOPT as _white) (LABEL as label) {
    (* TODO: The ST_LOOKING_FOR_PROPERTY state does not work for now because
     * it requires a yyless(1) which is not available in ocamllex (or is it ?)
     * So have to cheat and use instead the pending_token with push_token.
     *
     * buggy: push_mode ST_LOOKING_FOR_PROPERTY;
     *
```

```
     * TODO: also generate token for WHITESPACEOPT
     *)
    let info = tokinfo lexbuf in
    let syminfo = rewrap_str sym info in
    let lblinfo = rewrap_str label info (* TODO line number ? col ? *) in

    push_token (T_STRING (label, lblinfo));
    T_OBJECT_OPERATOR(syminfo)
  }
| "->" {
    T_OBJECT_OPERATOR(tokinfo lexbuf)
  }
```

94a ⟨*symbol rules* 92b⟩+≡

```
(* see also T_VARIABLE below. lex use longest matching strings so this
 * rule is used only in a last resort, for code such as $$x, ${, etc
 *)
| "$" { TDOLLAR(tokinfo lexbuf) }
```

9.4.3 Keywords and idents

94b ⟨*keyword and ident rules* 94b⟩≡

```
| LABEL
    { let info = tokinfo lexbuf in
      let s = tok lexbuf in
      match Common.optionise (fun () ->
        Hashtbl.find keyword_table (String.lowercase s))
      with
      | Some f -> f info
      | None -> T_STRING (s, info)
    }

| "$" (LABEL as s) { T_VARIABLE(s, tokinfo lexbuf) }
```

94c ⟨*regexp aliases* 84a⟩+≡

```
let LABEL =     ['a'-'z''A'-'Z''_']['a'-'z''A'-'Z''0'-'9''_']*
```

94d ⟨*keywords_table hash* 94d⟩≡

```
(* opti: less convenient, but using a hash is faster than using a match *)
let keyword_table = Common.hash_of_list [

  "while",          (fun ii -> T_WHILE ii);
  "endwhile",       (fun ii -> T_ENDWHILE ii);
  "do",             (fun ii -> T_DO ii);
```

```
  "for",             (fun ii -> T_FOR ii);
  "endfor",          (fun ii -> T_ENDFOR ii);
  "foreach",         (fun ii -> T_FOREACH ii);
  "endforeach",      (fun ii -> T_ENDFOREACH ii);

  "class_xdebug",           (fun ii -> T_CLASS_XDEBUG ii);
  "resource_xdebug",           (fun ii -> T_RESOURCE_XDEBUG ii);

⟨repetitive keywords table ??⟩

  "__halt_compiler", (fun ii -> T_HALT_COMPILER ii);

  "__CLASS__",       (fun ii -> T_CLASS_C ii);
  "__FUNCTION__",    (fun ii -> T_FUNC_C ii);
  "__METHOD__",      (fun ii -> T_METHOD_C ii);
  "__LINE__",        (fun ii -> T_LINE ii);
  "__FILE__",        (fun ii -> T_FILE ii);
]
```

9.4.4 Constants

95a ⟨*constant rules* 95a⟩≡

```
  | LNUM
      {
        (* more? cf original lexer *)
        let s = tok lexbuf in
        let ii = tokinfo lexbuf in
        try
          let _ = int_of_string s in
          T_LNUMBER(s, ii)
        with Failure _ ->
          T_DNUMBER(s, (*float_of_string s,*) ii)

      }
  | HNUM
      {
        (* more? cf orginal lexer *)
        T_DNUMBER(tok lexbuf, tokinfo lexbuf)

      }

  | DNUM|EXPONENT_DNUM { T_DNUMBER(tok lexbuf, tokinfo lexbuf) }
```

95b ⟨*regexp aliases* 84a⟩+≡

```
let LNUM =      ['0'-'9']+
```

```
let DNUM =      (['0'-'9']*['.']['0'-'9']+) | (['0'-'9']+['.']['0'-'9']* )

let EXPONENT_DNUM =     ((LNUM|DNUM)['e''E']['+''-']?LNUM)
let HNUM =      "0x"['0'-'9''a'-'f''A'-'F']+
```

9.4.5 Strings

96a ⟨*strings rules* 96a⟩≡

```
(*
 * The original PHP lexer does a few things to make the
 * difference at parsing time between static strings (which do not
 * contain any interpolation) and dynamic strings. So some regexps
 * below are quite hard to understand ... but apparently it works.
 * When the lexer thinks it's a dynamic strings, it let the grammar
 * do most of the hard work. See the rules using TGUIL in the grammar
 * (and here in the lexer).
 *
 *
 * /*
 *   ("{"*|"$"* ) handles { or $ at the end of a string (or the entire
 *  contents)
 *
 * what is this  'b?' at the beginning ?
 *
 * int bprefix = (yytext[0] != '"') ? 1 : 0;
 * zend_scan_escape_string(zendlval, yytext+bprefix+1, yyleng-bprefix-2, '"' TSRMLS_CC);
 */
 *)

(* static strings *)
| (['"'] ((DOUBLE_QUOTES_CHARS* ("{"*|"$"* )) as s) ['"'])
    { T_CONSTANT_ENCAPSED_STRING(s, tokinfo lexbuf) }

(* b? *)
| (['\''] (([^'\'' '\\']|('\\' ANY_CHAR))* as s)  ['\''])
    {
      (* more? cf original lexer *)
      T_CONSTANT_ENCAPSED_STRING(s, tokinfo lexbuf)
    }
```

96b ⟨*strings rules* 96a⟩+≡

```
(* dynamic strings *)
| ['"'] {
    push_mode ST_DOUBLE_QUOTES;
    TGUIL(tokinfo lexbuf)
  }
```

```
| ['`'] {
    push_mode ST_BACKQUOTE;
    TBACKQUOTE(tokinfo lexbuf)
  }
```

97a ⟨*strings rules* 96a⟩+≡

```
(* b? *)
| "<<<" TABS_AND_SPACES (LABEL as s) NEWLINE {
    set_mode (ST_START_HEREDOC s);
    T_START_HEREDOC (tokinfo lexbuf)
  }
```

97b ⟨*regexp aliases* 84a⟩+≡

```
(*/*
 * LITERAL_DOLLAR matches unescaped $ that aren't followed by a label character
 * or a { and therefore will be taken literally. The case of literal $ before
 * a variable or "${" is handled in a rule for each string type
 *
 * TODO: \x7f-\xff
 */
 *)
let DOUBLE_QUOTES_LITERAL_DOLLAR =
  ("$"+([^'a'-'z''A'-'Z''_''$''"''\\' '{']|('\\' ANY_CHAR)))
let BACKQUOTE_LITERAL_DOLLAR =
  ("$"+([^'a'-'z''A'-'Z''_''$''`''\\' '{']|('\\' ANY_CHAR)))
```

97c ⟨*regexp aliases* 84a⟩+≡

```
(*/*
 * CHARS matches everything up to a variable or "{$"
 * {'s are matched as long as they aren't followed by a $
 * The case of { before "{$" is handled in a rule for each string type
 *
 * For heredocs, matching continues across/after newlines if/when it's known
 * that the next line doesn't contain a possible ending label
 */
 *)
let DOUBLE_QUOTES_CHARS =
  ("{"*([^'$''"''\\''{']|
    ("\\" ANY_CHAR))| DOUBLE_QUOTES_LITERAL_DOLLAR)
let BACKQUOTE_CHARS =
  ("{"*([^'$' '`' '\\' '{']|('\\' ANY_CHAR))| BACKQUOTE_LITERAL_DOLLAR)
```

97d ⟨*regexp aliases* 84a⟩+≡

```
let HEREDOC_LITERAL_DOLLAR =
  ("$"+([^'a'-'z''A'-'Z''_''$''\n' '\r' '\\' '{' ]|('\\'[^'\n' '\r' ])))
```

```
(*/*
 * Usually, HEREDOC_NEWLINE will just function like a simple NEWLINE, but some
 * special cases need to be handled. HEREDOC_CHARS doesn't allow a line to
 * match when { or $, and/or \ is at the end. (("{"*|"$"* )"\\"?) handles that,
 * along with cases where { or $, and/or \ is the ONLY thing on a line
 *
 * The other case is when a line contains a label, followed by ONLY
 * { or $, and/or \  Handled by ({LABEL}";"?((("{"+|"$"+)"\\"?)|"\\"))
 */
 *)
let HEREDOC_NEWLINE =
  (((LABEL";"?((("{"+|"$"+)'\\'?)|'\\'))|(("{"*|"$"*)'\\'?))NEWLINE)


(*/*
 * This pattern is just used in the next 2 for matching { or literal $, and/or
 * \ escape sequence immediately at the beginning of a line or after a label
 */
 *)
let HEREDOC_CURLY_OR_ESCAPE_OR_DOLLAR =
  (("{"+[^'$' '\n' '\r' '\\' '{'])|("{"*'\\'[^'\n' '\r'])|
   HEREDOC_LITERAL_DOLLAR)

(*/*
 * These 2 label-related patterns allow HEREDOC_CHARS to continue "regular"
 * matching after a newline that starts with either a non-label character or a
 * label that isn't followed by a newline. Like HEREDOC_CHARS, they won't match
 * a variable or "{$"  Matching a newline, and possibly label, up TO a variable
 * or "{$", is handled in the heredoc rules
 *
 * The HEREDOC_LABEL_NO_NEWLINE pattern (";"[^$\n\r\\{]) handles cases where ;
 * follows a label. [^a-zA-Z0-9_\x7f-\xff;$\n\r\\{] is needed to prevent a label
 * character or ; from matching on a possible (real) ending label
 */*)
let HEREDOC_NON_LABEL =
  ([^'a'-'z''A'-'Z''_' '$' '\n''\r''\\' '{']|HEREDOC_CURLY_OR_ESCAPE_OR_DOLLAR)
let HEREDOC_LABEL_NO_NEWLINE =
 (LABEL([^'a'-'z''A'-'Z''0'-'9''_'';''$''\n' '\r' '\\' '{']|
   (";"[^'$' '\n' '\r' '\\' '{' ])|(";"? HEREDOC_CURLY_OR_ESCAPE_OR_DOLLAR)))
```

98 ⟨*regexp aliases* 84a⟩+≡

```
let HEREDOC_CHARS =
  ("{"*([^'$' '\n' '\r' '\\' '{']|('\\'[^'\n' '\r']))|
   HEREDOC_LITERAL_DOLLAR|(HEREDOC_NEWLINE+(HEREDOC_NON_LABEL|HEREDOC_LABEL_NO_NEWLINE)))
```

Note: I don't understand some of those regexps. I just copy pasted them from the original lexer and pray that I would never have to modify them.

9.4.6 Misc

99a ⟨*misc rules* 99a⟩≡

```
(* ugly, they could have done that in the grammar ... or maybe it was
 * because it could lead to some ambiguities ?
 *)
| "(" TABS_AND_SPACES ("int"|"integer") TABS_AND_SPACES ")"
    { T_INT_CAST(tokinfo lexbuf) }

| "(" TABS_AND_SPACES ("real"|"double"|"float") TABS_AND_SPACES ")"
    { T_DOUBLE_CAST(tokinfo lexbuf) }

| "(" TABS_AND_SPACES "string" TABS_AND_SPACES ")"
    { T_STRING_CAST(tokinfo lexbuf); }

| "(" TABS_AND_SPACES "binary" TABS_AND_SPACES ")"
    { T_STRING_CAST(tokinfo lexbuf); }

| "(" TABS_AND_SPACES "array" TABS_AND_SPACES ")"
    { T_ARRAY_CAST(tokinfo lexbuf); }

| "(" TABS_AND_SPACES "object" TABS_AND_SPACES ")"
    { T_OBJECT_CAST(tokinfo lexbuf); }

| "(" TABS_AND_SPACES ("bool"|"boolean") TABS_AND_SPACES ")"
    { T_BOOL_CAST(tokinfo lexbuf); }

| "(" TABS_AND_SPACES ("unset") TABS_AND_SPACES ")"
    { T_UNSET_CAST(tokinfo lexbuf); }
```

99b ⟨*misc rules* 99a⟩+≡

```
| ("?>" | "</script"WHITESPACE*">")NEWLINE?
    {
      set_mode INITIAL;
      T_CLOSE_TAG(tokinfo lexbuf) (*/* implicit ';' at php-end tag */*)
    }
```

9.5 Interpolated strings states

9.5.1 Double quotes

Some of the rules defined in `st_double_quotes` are duplicated in other `st_xxx functions`. In the orignal lexer, they could factorize them because Flex have this feature, but not `ocamllex`. Fortunately the use of literate programming on the `ocamllex` file gives us back this feature for free.

100a ⟨*rule st_double_quotes* 100a⟩≡

```
  and st_double_quotes = parse

    | DOUBLE_QUOTES_CHARS+ {
        T_ENCAPSED_AND_WHITESPACE(tok lexbuf, tokinfo lexbuf)
      }

    ⟨encapsulated dollar stuff rules 100b⟩

    | ['"'] {
        set_mode ST_IN_SCRIPTING;
        TGUIL(tokinfo lexbuf)
      }
   ⟨repetitive st_double_quotes rules for error handling ??⟩
```

100b ⟨*encapsulated dollar stuff rules* 100b⟩≡

```
    | "$" (LABEL as s)     { T_VARIABLE(s, tokinfo lexbuf) }
```

100c ⟨*encapsulated dollar stuff rules* 100b⟩+≡

```
    | "$" (LABEL as s) "[" {
        push_token (TOBRA (tokinfo lexbuf)); (* TODO wrong info *)
        push_mode ST_VAR_OFFSET;
        T_VARIABLE(s, tokinfo lexbuf)
      }
```

100d ⟨*encapsulated dollar stuff rules* 100b⟩+≡

```
    | "{$" {
        yyless 1 lexbuf;
        push_mode ST_IN_SCRIPTING;
        T_CURLY_OPEN(tokinfo lexbuf);
      }
```

100e ⟨*encapsulated dollar stuff rules* 100b⟩+≡

```
    | "${" {
        push_mode ST_LOOKING_FOR_VARNAME;
        T_DOLLAR_OPEN_CURLY_BRACES(tokinfo lexbuf);
      }
```

9.5.2 Backquotes

101a ⟨*rule st_backquote* 101a⟩≡
```
(* mostly copy paste of st_double_quotes; just the end regexp is different *)
and st_backquote = parse

  | BACKQUOTE_CHARS+ {
      T_ENCAPSED_AND_WHITESPACE(tok lexbuf, tokinfo lexbuf)
    }

  ⟨encapsulated dollar stuff rules 100b⟩

  | ['‘'] {
      set_mode ST_IN_SCRIPTING;
      TBACKQUOTE(tokinfo lexbuf)
    }

  ⟨repetitive st_backquote rules for error handling ??⟩
```

9.5.3 Here docs (<<<EOF)

101b ⟨*rule st_start_heredoc* 101b⟩≡
```
(* as heredoc have some of the semantic of double quote strings, again some
 * rules from st_double_quotes are copy pasted here.
 *)
and st_start_heredoc stopdoc = parse

  | (LABEL as s) (";"? as semi) ['\n' '\r'] {
      if s = stopdoc
      then begin
        set_mode ST_IN_SCRIPTING;
        if semi = ";"
        then push_token (TSEMICOLON (tokinfo lexbuf)); (* TODO wrong info *)

        T_END_HEREDOC(tokinfo lexbuf)
      end else
        T_ENCAPSED_AND_WHITESPACE(tok lexbuf, tokinfo lexbuf)
    }
  (* | ANY_CHAR { set_mode ST_HERE_DOC; yymore() ??? } *)

  ⟨encapsulated dollar stuff rules 100b⟩


 (*/* Match everything up to and including a possible ending label, so if the label
  * doesn't match, it's kept with the rest of the string
  *
```

```
 * {HEREDOC_NEWLINE}+ handles the case of more than one newline sequence that
 * couldn't be matched with HEREDOC_CHARS, because of the following label
 */
 *)
  | ((HEREDOC_CHARS* HEREDOC_NEWLINE+) as str)
       (LABEL as s)
       (";"? as semi)['\n' '\r'] {
      if s = stopdoc
      then begin
        set_mode ST_IN_SCRIPTING;
        if semi = ";"
        then push_token (TSEMICOLON (tokinfo lexbuf)); (* TODO Wrong info *)
        push_token (T_END_HEREDOC(tokinfo lexbuf)); (* TODO wrong info *)
        T_ENCAPSED_AND_WHITESPACE(str, tokinfo lexbuf) (* TODO wrong info *)
      end
      else begin
        T_ENCAPSED_AND_WHITESPACE (tok lexbuf, tokinfo lexbuf)
      end
    }

(*/* ({HEREDOC_NEWLINE}+({LABEL}";"?)?)? handles the possible case of newline
 * sequences, possibly followed by a label, that couldn't be matched with
 * HEREDOC_CHARS because of a following variable or "{$"
 *
 * This doesn't affect real ending labels, as they are followed by a newline,
 * which will result in a longer match for the correct rule if present
 */
 *)
  | HEREDOC_CHARS*(HEREDOC_NEWLINE+(LABEL";"?)?)? {
      T_ENCAPSED_AND_WHITESPACE(tok lexbuf, tokinfo lexbuf)
    }
```

⟨*repetitive st_start_heredoc rules for error handling ??*⟩

9.6 Other states

102 ⟨*rule st_looking_for_property 102*⟩≡

```
(* TODO not used for now *)
and st_looking_for_property = parse
  | "->" { T_OBJECT_OPERATOR(tokinfo lexbuf) }
  | LABEL {
      pop_mode();
      T_STRING(tok lexbuf, tokinfo lexbuf)
```

```
    }
(*
  | ANY_CHAR {
      (* XXX yyless(0) ?? *)
      pop_mode();
    }
*)
```

103a ⟨*rule st_looking_for_varname* 103a⟩≡

```
and st_looking_for_varname = parse
  | LABEL {
      set_mode ST_IN_SCRIPTING;
      T_STRING_VARNAME(tok lexbuf, tokinfo lexbuf)
    }
(*
  | ANY_CHAR {
      (* XXX yyless(0) ?? *)
      pop_mode();
      push_mode ST_IN_SCRIPTING
    }
*)
```

103b ⟨*rule st_var_offset* 103b⟩≡

```
and st_var_offset = parse

  | LNUM | HNUM { (* /* Offset must be treated as a string */ *)
      T_NUM_STRING (tok lexbuf, tokinfo lexbuf)
    }

  | "$" (LABEL as s) { T_VARIABLE(s, tokinfo lexbuf) }
  | LABEL            { T_STRING(tok lexbuf, tokinfo lexbuf)  }

  | "]" {
      pop_mode();
      TCBRA(tokinfo lexbuf);
    }
   ⟨repetitive st_var_offset rules for error handling ??⟩
```

9.7 XHP extensions

103c ⟨*symbol rules* 92b⟩+≡

```
  (* xhp: TODO should perhaps split ":" to have better info *)
 (* PB, it is legal to do e?1:null; in PHP
  | ":" XHPLABEL (":" XHPLABEL)* { TXHPCOLONID (tok lexbuf, tokinfo lexbuf) }
 *)
```

104a ⟨*regexp aliases* 84a⟩+≡
```
let XHPLABEL =  ['a'-'z''A'-'Z''_']['a'-'z''A'-'Z''0'-'9''_''-']*
```

9.8 Misc

104b ⟨*lexer helpers* 88e⟩+≡
```
let tok     lexbuf  = Lexing.lexeme lexbuf

let tokinfo lexbuf  =
  {
    Ast.pinfo = Ast.OriginTok {
      Common.charpos = Lexing.lexeme_start lexbuf;
      Common.str     = Lexing.lexeme lexbuf;

      (* info filled in a post-lexing phase, cf Parse_php.tokens *)
      Common.line = -1;
      Common.column = -1;
      Common.file = "";
    };
    comments = ();
  }
```

104c ⟨*lexer helpers* 88e⟩+≡
```
let tok_add_s s ii  =
  Ast.rewrap_str ((Ast.str_of_info ii) ^ s) ii
```

9.9 Token Helpers

104d ⟨*token_helpers_php.mli* 104d⟩≡
```
val is_eof          : Parser_php.token -> bool
val is_comment      : Parser_php.token -> bool
val is_just_comment : Parser_php.token -> bool
```

104e ⟨*token_helpers_php.mli* 104d⟩+≡
```
val info_of_tok :
  Parser_php.token -> Ast_php.info
val visitor_info_of_tok :
  (Ast_php.info -> Ast_php.info) -> Parser_php.token -> Parser_php.token
```

104f ⟨*token_helpers_php.mli* 104d⟩+≡
```
val line_of_tok  : Parser_php.token -> int
val str_of_tok   : Parser_php.token -> string
val file_of_tok  : Parser_php.token -> Common.filename
val pos_of_tok   : Parser_php.token -> int
```

105 ⟨*token_helpers_php.mli* 104d⟩+≡

```
val pinfo_of_tok   : Parser_php.token -> Ast_php.pinfo
val is_origin : Parser_php.token -> bool
```

Chapter 10

Grammar

10.1 Overview

The code in `parser_php.mly` is mostly a copy paste of the Yacc parser in the PHP source code (in `pfff/docs/official-grammar/5.2.11/zend_language_parser.y`) adapted for `ocamlyacc`. Here is the toplevel structure of `parser_php.mly`:

106a ⟨*parser_php.mly* 106a⟩≡

```
⟨Facebook copyright2 ??⟩

⟨GRAMMAR prelude 128e⟩

/*(*************************************************************************)*/
/*(* Tokens *)*/
/*(*************************************************************************)*/
⟨GRAMMAR tokens declaration 133a⟩

⟨GRAMMAR tokens priorities 136⟩

/*(*************************************************************************)*/
/*(* Rules type declaration *)*/
/*(*************************************************************************)*/
%start main expr
⟨GRAMMAR type of main rule 106b⟩

%%

⟨GRAMMAR long set of rules 107⟩
```

106b ⟨*GRAMMAR type of main rule* 106b⟩≡

```
%type <Ast_php.toplevel list> main
%type <Ast_php.expr> expr
```

107 ⟨*GRAMMAR long set of rules* 107⟩≡

```
/*(**************************************************************************)*/
/*(* toplevel *)*/
/*(**************************************************************************)*/
⟨GRAMMAR toplevel 108a⟩


/*(**************************************************************************)*/
/*(* statement *)*/
/*(**************************************************************************)*/
⟨GRAMMAR statement 108c⟩

/*(**************************************************************************)*/
/*(* function declaration *)*/
/*(**************************************************************************)*/
⟨GRAMMAR function declaration 119d⟩

/*(**************************************************************************)*/
/*(* class declaration *)*/
/*(**************************************************************************)*/
⟨GRAMMAR class declaration 121a⟩

/*(**************************************************************************)*/
/*(* expr and variable *)*/
/*(**************************************************************************)*/
⟨GRAMMAR expression 112a⟩

/*(**************************************************************************)*/
/*(* namespace *)*/
/*(**************************************************************************)*/
⟨GRAMMAR namespace 124b⟩


/*(**************************************************************************)*/
/*(* class bis *)*/
/*(**************************************************************************)*/
⟨GRAMMAR class bis 123b⟩

/*(**************************************************************************)*/
/*(* Encaps *)*/
/*(**************************************************************************)*/
⟨GRAMMAR encaps 125⟩

/*(**************************************************************************)*/
/*(* xxx_list, xxx_opt *)*/
/*(**************************************************************************)*/
```

⟨*GRAMMAR xxxlist or xxxopt* 137⟩

10.2 Toplevel

108a ⟨*GRAMMAR toplevel* 108a⟩≡

```
main: start EOF { top_statements_to_toplevels $1 $2 }

start: top_statement_list { $1 }
```

108b ⟨*GRAMMAR toplevel* 108a⟩+≡

```
top_statement:
 | statement                            { Stmt $1 }
 | function_declaration_statement       { FuncDefNested $1 }
 | class_declaration_statement          {
     match $1 with
     | Left x -> ClassDefNested x
     | Right x -> InterfaceDefNested x
   }
```

10.3 Statement

108c ⟨*GRAMMAR statement* 108c⟩≡

```
inner_statement: top_statement { $1 }
statement: unticked_statement { $1 }
```

108d ⟨*GRAMMAR statement* 108c⟩+≡

```
unticked_statement:
 | expr            TSEMICOLON              { ExprStmt($1,$2) }
 | /*(* empty*)*/ TSEMICOLON              { EmptyStmt($1) }
 | TOBRACE inner_statement_list TCBRACE   { Block($1,$2,$3) }

 | T_IF TOPAR expr TCPAR statement elseif_list else_single
     { If($1,($2,$3,$4),$5,$6,$7) }
 | T_IF TOPAR expr TCPAR TCOLON
     inner_statement_list new_elseif_list new_else_single
     T_ENDIF TSEMICOLON
     { IfColon($1,($2,$3,$4),$5,$6,$7,$8,$9,$10)  }

 | T_WHILE TOPAR expr  TCPAR while_statement
     { While($1,($2,$3,$4),$5) }
 | T_DO statement T_WHILE TOPAR expr TCPAR TSEMICOLON
     { Do($1,$2,$3,($4,$5,$6),$7) }
 | T_FOR TOPAR
```

```
    for_expr TSEMICOLON
    for_expr TSEMICOLON
    for_expr
    TCPAR
    for_statement
    { For($1,$2,$3,$4,$5,$6,$7,$8,$9) }

| T_SWITCH TOPAR expr TCPAR    switch_case_list
    { Switch($1,($2,$3,$4),$5) }

| T_FOREACH TOPAR variable T_AS
    foreach_variable foreach_optional_arg TCPAR
    foreach_statement
    { Foreach($1,$2,mk_e (Lvalue $3),$4,Left $5,$6,$7,$8) }
| T_FOREACH TOPAR expr_without_variable T_AS
    variable foreach_optional_arg TCPAR
    foreach_statement
    { Foreach($1,$2,$3,$4,Right $5,$6,$7,$8)  }

| T_BREAK TSEMICOLON           { Break($1,None,$2) }
| T_BREAK expr TSEMICOLON      { Break($1,Some $2, $3) }
| T_CONTINUE TSEMICOLON        { Continue($1,None,$2) }
| T_CONTINUE expr TSEMICOLON   { Continue($1,Some $2, $3) }

| T_RETURN TSEMICOLON                           { Return ($1,None, $2) }
| T_RETURN expr_without_variable TSEMICOLON  { Return ($1,Some ($2), $3)}
| T_RETURN variable TSEMICOLON                 { Return ($1,Some (mk_e (Lvalue $2)), $3)}

| T_TRY
    TOBRACE inner_statement_list TCBRACE
  T_CATCH TOPAR fully_qualified_class_name  T_VARIABLE TCPAR
    TOBRACE inner_statement_list TCBRACE
    additional_catches
    {
      let try_block = ($2,$3,$4) in
      let catch_block = ($10, $11, $12) in
      let catch = ($5, ($6, ($7, DName $8), $9), catch_block) in
      Try($1, try_block, catch, $13)
    }
| T_THROW expr TSEMICOLON { Throw($1,$2,$3) }

| T_ECHO echo_expr_list TSEMICOLON     { Echo($1,$2,$3) }
| T_INLINE_HTML                        { InlineHtml($1) }

| T_GLOBAL global_var_list TSEMICOLON { Globals($1,$2,$3) }
| T_STATIC static_var_list TSEMICOLON { StaticVars($1,$2,$3) }
```

```
 | T_UNSET TOPAR unset_variables TCPAR TSEMICOLON { Unset($1,($2,$3,$4),$5) }

 | T_USE use_filename TSEMICOLON                  { Use($1,$2,$3) }
 | T_DECLARE  TOPAR declare_list TCPAR declare_statement
     { Declare($1,($2,$3,$4),$5) }
```

110 ⟨*GRAMMAR statement* 108c⟩+≡

```
/*(*----------------------------*)*/
/*(* auxillary statements *)*/
/*(*----------------------------*)*/

for_expr:
 | /*(*empty*)*/        { [] }
 | non_empty_for_expr   { $1 }

foreach_optional_arg:
  | /*(*empty*)*/                        { None }
  | T_DOUBLE_ARROW foreach_variable     { Some($1,$2) }

foreach_variable: is_reference variable { ($1, $2) }


switch_case_list:
 | TOBRACE            case_list TCBRACE  { CaseList($1,None,$2,$3) }
 | TOBRACE TSEMICOLON case_list TCBRACE  { CaseList($1, Some $2, $3, $4) }
 | TCOLON             case_list T_ENDSWITCH TSEMICOLON
     { CaseColonList($1,None,$2, $3, $4) }
 | TCOLON TSEMICOLON  case_list T_ENDSWITCH TSEMICOLON
     { CaseColonList($1, Some $2, $3, $4, $5) }

case_list:
 | /*(*empty*)*/        { [] }
 | case_list    T_CASE expr case_separator inner_statement_list
     { $1 ++ [Case($2,$3,$4,$5) ] }
 | case_list    T_DEFAULT   case_separator inner_statement_list
     { $1 ++ [Default($2,$3,$4) ] }

case_separator:
 | TCOLON     { $1 }
 | TSEMICOLON { $1 }



while_statement:
 | statement                                        { SingleStmt $1 }
```

```
 | TCOLON inner_statement_list T_ENDWHILE TSEMICOLON { ColonStmt($1,$2,$3,$4) }

for_statement:
 | statement                                      { SingleStmt $1 }
 | TCOLON inner_statement_list T_ENDFOR TSEMICOLON { ColonStmt($1,$2,$3,$4) }

foreach_statement:
 | statement                                          { SingleStmt $1 }
 | TCOLON inner_statement_list T_ENDFOREACH TSEMICOLON { ColonStmt($1,$2,$3,$4)}

declare_statement:
 | statement                                          { SingleStmt $1 }
 | TCOLON inner_statement_list T_ENDDECLARE TSEMICOLON { ColonStmt($1,$2,$3,$4)}



elseif_list:
 | /*(*empty*)*/ { [] }
 | elseif_list  T_ELSEIF TOPAR expr TCPAR statement { $1 ++ [$2,($3,$4,$5),$6] }

new_elseif_list:
 | /*(*empty*)*/ { [] }
 | new_elseif_list    T_ELSEIF TOPAR expr TCPAR TCOLON inner_statement_list
     { $1 ++ [$2,($3,$4,$5),$6,$7] }


else_single:
 | /*(*empty*)*/    { None }
 | T_ELSE statement { Some($1,$2) }

new_else_single:
 | /*(*empty*)*/                     { None }
 | T_ELSE TCOLON inner_statement_list { Some($1,$2,$3) }


additional_catch:
 | T_CATCH
     TOPAR fully_qualified_class_name T_VARIABLE TCPAR
     TOBRACE inner_statement_list TCBRACE
     {
       let catch_block = ($6, $7, $8) in
       let catch = ($1, ($2, ($3, DName $4), $5), catch_block) in
       catch
     }
```

111 ⟨*GRAMMAR statement* 108c⟩+≡

```
/*(*----------------------------*)*/
/*(* auxillary bis *)*/
/*(*----------------------------*)*/

declare: T_STRING   TEQ static_scalar { Name $1, ($2, $3) }

global_var:
 | T_VARIABLE                   { GlobalVar (DName $1) }
 | TDOLLAR r_variable           { GlobalDollar ($1, $2) }
 | TDOLLAR TOBRACE expr TCBRACE { GlobalDollarExpr ($1, ($2, $3, $4)) }

/*(* can not factorize, otherwise shift/reduce conflict *)*/
static_var_list:
 | T_VARIABLE                   { [DName $1, None] }
 | T_VARIABLE TEQ static_scalar { [DName $1, Some ($2, $3) ] }
 | static_var_list TCOMMA   T_VARIABLE
     { $1 ++ [DName $3, None] }
 | static_var_list TCOMMA   T_VARIABLE TEQ static_scalar
     { $1 ++ [DName $3, Some ($4, $5) ] }

unset_variable: variable        { $1 }

use_filename:
 |       T_CONSTANT_ENCAPSED_STRING             { UseDirect $1 }
 | TOPAR T_CONSTANT_ENCAPSED_STRING TCPAR       { UseParen ($1, $2, $3) }
```

10.4 Expression

112a ⟨*GRAMMAR expression* 112a⟩≡

```
/*(* a little coupling with non_empty_function_call_parameter_list *)*/
expr:
 | r_variable                           { mk_e (Lvalue $1) }
 | expr_without_variable                { $1 }

expr_without_variable: expr_without_variable_bis { mk_e $1 }
```

112b ⟨*GRAMMAR expression* 112a⟩+≡

```
expr_without_variable_bis:
 | scalar                               { Scalar $1 }

 | TOPAR expr TCPAR     { ParenExpr($1,$2,$3) }

 | variable TEQ expr            { Assign($1,$2,$3) }
 | variable TEQ TAND variable   { AssignRef($1,$2,$3,$4) }
 | variable TEQ TAND T_NEW class_name_reference ctor_arguments
```

```
    { AssignNew($1,$2,$3,$4,$5,$6) }

| variable T_PLUS_EQUAL   expr { AssignOp($1,(AssignOpArith Plus,$2),$3) }
| variable T_MINUS_EQUAL  expr { AssignOp($1,(AssignOpArith Minus,$2),$3) }
| variable T_MUL_EQUAL    expr { AssignOp($1,(AssignOpArith Mul,$2),$3) }
| variable T_DIV_EQUAL    expr { AssignOp($1,(AssignOpArith Div,$2),$3) }
| variable T_MOD_EQUAL    expr { AssignOp($1,(AssignOpArith Mod,$2),$3) }
| variable T_AND_EQUAL    expr { AssignOp($1,(AssignOpArith And,$2),$3) }
| variable T_OR_EQUAL     expr { AssignOp($1,(AssignOpArith Or,$2),$3) }
| variable T_XOR_EQUAL    expr { AssignOp($1,(AssignOpArith Xor,$2),$3) }
| variable T_SL_EQUAL     expr { AssignOp($1,(AssignOpArith DecLeft,$2),$3) }
| variable T_SR_EQUAL     expr { AssignOp($1,(AssignOpArith DecRight,$2),$3) }

| variable T_CONCAT_EQUAL expr { AssignOp($1,(AssignConcat,$2),$3) }

| rw_variable T_INC { Postfix($1, (Inc, $2)) }
| rw_variable T_DEC { Postfix($1, (Dec, $2)) }
| T_INC rw_variable { Infix((Inc, $1),$2) }
| T_DEC rw_variable { Infix((Dec, $1),$2) }

| expr T_BOOLEAN_OR expr   { Binary($1,(Logical OrBool ,$2),$3) }
| expr T_BOOLEAN_AND  expr { Binary($1,(Logical AndBool,$2),$3) }
| expr T_LOGICAL_OR  expr  { Binary($1,(Logical OrLog,  $2),$3) }
| expr T_LOGICAL_AND  expr { Binary($1,(Logical AndLog, $2),$3) }
| expr T_LOGICAL_XOR expr  { Binary($1,(Logical XorLog, $2),$3) }

| expr TPLUS expr      { Binary($1,(Arith Plus ,$2),$3) }
| expr TMINUS expr     { Binary($1,(Arith Minus,$2),$3) }
| expr TMUL expr       { Binary($1,(Arith Mul,$2),$3) }
| expr TDIV expr       { Binary($1,(Arith Div,$2),$3) }
| expr TMOD expr       { Binary($1,(Arith Mod,$2),$3) }
| expr TAND expr       { Binary($1,(Arith And,$2),$3) }
| expr TOR expr        { Binary($1,(Arith Or,$2),$3) }
| expr TXOR expr       { Binary($1,(Arith Xor,$2),$3) }
| expr T_SL expr       { Binary($1,(Arith DecLeft,$2),$3) }
| expr T_SR expr       { Binary($1,(Arith DecRight,$2),$3) }

| expr TDOT expr       { Binary($1,(BinaryConcat,$2),$3) }

| expr T_IS_IDENTICAL        expr { Binary($1,(Logical Identical,$2),$3) }
| expr T_IS_NOT_IDENTICAL    expr { Binary($1,(Logical NotIdentical,$2),$3) }
| expr T_IS_EQUAL            expr { Binary($1,(Logical Eq,$2),$3) }
| expr T_IS_NOT_EQUAL        expr { Binary($1,(Logical NotEq,$2),$3) }
| expr TSMALLER              expr { Binary($1,(Logical Inf,$2),$3) }
| expr T_IS_SMALLER_OR_EQUAL expr { Binary($1,(Logical InfEq,$2),$3) }
```

```
| expr TGREATER              expr { Binary($1,(Logical Sup,$2),$3) }
| expr T_IS_GREATER_OR_EQUAL expr { Binary($1,(Logical SupEq,$2),$3) }

| TPLUS  expr    %prec T_INC           { Unary((UnPlus,$1),$2) }
| TMINUS expr    %prec T_INC           { Unary((UnMinus,$1),$2) }
| TBANG  expr                          { Unary((UnBang,$1),$2) }
| TTILDE expr                          { Unary((UnTilde,$1),$2) }


| T_LIST TOPAR assignment_list TCPAR TEQ expr
    { ConsList($1,($2,$3,$4),$5,$6) }
| T_ARRAY TOPAR array_pair_list TCPAR
    { ConsArray($1,($2,$3,$4)) }

| T_NEW class_name_reference ctor_arguments
    { New($1,$2,$3) }
| T_CLONE expr { Clone($1,$2) }
| expr T_INSTANCEOF class_name_reference
    { InstanceOf($1,$2,$3) }

| expr TQUESTION  expr TCOLON  expr     { CondExpr($1,$2,$3,$4,$5) }

| T_BOOL_CAST   expr   { Cast((BoolTy,$1),$2) }
| T_INT_CAST    expr   { Cast((IntTy,$1),$2) }
| T_DOUBLE_CAST expr   { Cast((DoubleTy,$1),$2) }
| T_STRING_CAST expr   { Cast((StringTy,$1),$2) }
| T_ARRAY_CAST  expr   { Cast((ArrayTy,$1),$2) }
| T_OBJECT_CAST expr   { Cast((ObjectTy,$1),$2) }


| T_UNSET_CAST  expr   { CastUnset($1,$2) }

| T_EXIT exit_expr     { Exit($1,$2) }
| T__AT expr           { At($1,$2) }
| T_PRINT expr  { Print($1,$2) }

| TBACKQUOTE encaps_list TBACKQUOTE   { BackQuote($1,$2,$3) }
/*(* php 5.3 only *)*/
| T_FUNCTION is_reference TOPAR parameter_list TCPAR lexical_vars
  TOBRACE inner_statement_list TCBRACE
    {
      let params = ($3, $4, $5) in
      let body = ($7, $8, $9) in

      let ldef = {
        l_tok = $1;
```

```
          l_ref = $2;
          l_params = params;
          l_use = $6;
          l_body = body;
        }
        in
        Lambda ldef
      }

  | internal_functions_in_yacc { $1 }


  ⟨exprbis grammar rule hook 127b⟩
```

115a ⟨*GRAMMAR expression* 112a⟩+≡

```
 /*(*pad: why this name ? *)*/
 internal_functions_in_yacc:
  | T_INCLUDE      expr                  { Include($1,$2) }
  | T_INCLUDE_ONCE expr                  { IncludeOnce($1,$2) }
  | T_REQUIRE      expr                  { Require($1,$2) }
  | T_REQUIRE_ONCE expr                  { RequireOnce($1,$2) }

  | T_ISSET TOPAR isset_variables TCPAR { Isset($1,($2,$3,$4)) }
  | T_EMPTY TOPAR variable TCPAR        { Empty($1,($2,$3,$4)) }

  | T_EVAL TOPAR expr TCPAR             { Eval($1,($2,$3,$4)) }
```

115b ⟨*GRAMMAR expression* 112a⟩+≡

```
 /*(*----------------------------*)*/
 /*(* scalar *)*/
 /*(*----------------------------*)*/

 ⟨GRAMMAR scalar 115c⟩

 /*(*----------------------------*)*/
 /*(* variable *)*/
 /*(*----------------------------*)*/

 ⟨GRAMMAR variable 117b⟩
```

10.4.1 Scalar

115c ⟨*GRAMMAR scalar* 115c⟩≡

```
 scalar:
  | common_scalar              { Constant $1 }
  | T_STRING                   { Constant (CName (Name $1)) }
```

```
 | class_constant              { ClassConstant $1 }

 | TGUIL encaps_list TGUIL
     { Guil ($1, $2, $3)}
 | T_START_HEREDOC encaps_list T_END_HEREDOC
     { HereDoc ($1, $2, $3) }

 /*(* generated by lexer for special case of ${beer}s. So it's really
    * more a variable than a constant. So I've decided to inline this
    * special case rule in encaps. Maybe this is too restrictive.
    *)*/
  /*(* | T_STRING_VARNAME {  raise Todo } *)*/
```

116 ⟨*GRAMMAR scalar* 115c⟩+≡

```
static_scalar: /* compile-time evaluated scalars */
 | common_scalar        { StaticConstant $1 }
 | T_STRING             { StaticConstant (CName (Name $1)) }
 | static_class_constant { StaticClassConstant $1 }

 | TPLUS static_scalar   { StaticPlus($1,$2) }
 | TMINUS static_scalar  { StaticMinus($1,$2) }
 | T_ARRAY TOPAR static_array_pair_list TCPAR
     { StaticArray($1, ($2, $3, $4)) }
```

⟨*static_scalar grammar rule hook* 128b⟩

```
common_scalar:
 | T_LNUMBER                   { Int($1) }
 | T_DNUMBER                   { Double($1) }

 | T_CONSTANT_ENCAPSED_STRING  { String($1) }

 | T_LINE                      { PreProcess(Line, $1) }
 | T_FILE                      { PreProcess(File, $1) }
 | T_CLASS_C                   { PreProcess(ClassC, $1) }
 | T_METHOD_C                  { PreProcess(MethodC, $1) }
 | T_FUNC_C                    { PreProcess(FunctionC, $1) }
```

⟨*common_scalar grammar rule hook* 128c⟩

```
class_constant: qualifier T_STRING { $1, (Name $2) }

static_class_constant: class_constant { $1 }
```

117a ⟨*GRAMMAR scalar* 115c⟩+≡
```
/*(* can not factorize, otherwise shift/reduce conflict *)*/
non_empty_static_array_pair_list:
 | static_scalar
     { [StaticArraySingle $1] }
 | static_scalar T_DOUBLE_ARROW static_scalar
     { [StaticArrayArrow ($1,$2,$3)]}
```

⟨*repetitive non_empty_static_array_pair_list* ??⟩

10.4.2 Variable

In the original grammar they use the term `variable` to actually refer to what I think would be best described by the term `lvalue`. Indeed function calls or method calls are part of this category, and it would be confusing for the user to consider such entity as "variables". So I've kept the term `variable` in the grammar, but in the AST I use a `lvalue` type.

117b ⟨*GRAMMAR variable* 117b⟩≡
```
variable: variable2 { variable2_to_lvalue $1 }
```

117c ⟨*GRAMMAR variable* 117b⟩+≡
```
variable2:
 | base_variable_with_function_calls
     { Variable ($1,[]) }
 | base_variable_with_function_calls
     T_OBJECT_OPERATOR object_property method_or_not
     variable_properties
     { Variable ($1, ($2, $3, $4)::$5) }

base_variable_with_function_calls:
 | base_variable {  BaseVar $1 }
 | function_call {  $1 }

base_variable:
 |            variable_without_objects                         { None,    $1 }
 | qualifier  variable_without_objects /*(*static_member*)*/ { Some $1, $2 }


variable_without_objects:
 |                             reference_variable { [], $1 }
 | simple_indirect_reference   reference_variable { $1, $2 }

reference_variable:
 | compound_variable                             { $1 }
 | reference_variable TOBRA dim_offset TCBRA  { VArrayAccess2($1, ($2,$3,$4)) }
```

```
 | reference_variable TOBRACE expr TCBRACE    { VBraceAccess2($1, ($2,$3,$4)) }

compound_variable:
 | T_VARIABLE                     { Var2 (DName $1, Ast_php.noScope()) }
 | TDOLLAR TOBRACE expr TCBRACE { VDollar2 ($1, ($2, $3, $4)) }
```

118a ⟨*GRAMMAR variable* 117b⟩+≡

```
simple_indirect_reference:
 | TDOLLAR                             { [Dollar $1] }
 | simple_indirect_reference TDOLLAR { $1 ++ [Dollar $2] }

dim_offset:
 | /*(*empty*)*/   { None }
 | expr            { Some $1 }
```

118b ⟨*GRAMMAR variable* 117b⟩+≡

```
r_variable: variable { $1 }
w_variable: variable { $1 }
rw_variable: variable { $1 }
```

118c ⟨*GRAMMAR variable* 117b⟩+≡

```
/*(*----------------------------*)*/
/*(* function call *)*/
/*(*----------------------------*)*/
function_call: function_head TOPAR function_call_parameter_list TCPAR
  { FunCall ($1, ($2, $3, $4)) }

 ⟨function_call grammar rule hook 127c⟩

/*(* cant factorize the rule with a qualifier_opt because it leads to
   * many conflicts :( *)*/
function_head:
 | T_STRING                        { FuncName (None, Name $1) }
 | variable_without_objects      { FuncVar  (None, $1) }
 | qualifier    T_STRING                   { FuncName(Some $1, Name $2) }
 | qualifier    variable_without_objects  { FuncVar(Some $1, $2) }
```

118d ⟨*GRAMMAR variable* 117b⟩+≡

```
/*(* can not factorize, otherwise shift/reduce conflict *)*/
non_empty_function_call_parameter_list:
 | variable                      { [Arg (mk_e (Lvalue $1))] }
 | expr_without_variable         { [Arg ($1)] }
 | TAND w_variable               { [ArgRef($1,$2)] }

 ⟨repetitive non_empty_function_call_parameter_list ??⟩
```

```
bra: TOBRA dim_offset TCBRA { ($1, $2, $3) }
```

119a ⟨*GRAMMAR variable* 117b⟩+≡

```
/*(*----------------------------*)*/
/*(* list/array *)*/
/*(*----------------------------*)*/

assignment_list_element:
 | variable                              { ListVar $1 }
 | T_LIST TOPAR assignment_list TCPAR    { ListList ($1, ($2, $3, $4)) }
 | /*(*empty*)*/                         { ListEmpty }
```

119b ⟨*GRAMMAR variable* 117b⟩+≡

```
/*(* can not factorize, otherwise shift/reduce conflict *)*/
non_empty_array_pair_list:
 | expr                                  { [ArrayExpr $1] }
 | TAND w_variable                       { [ArrayRef ($1,$2)] }
 | expr T_DOUBLE_ARROW   expr            { [ArrayArrowExpr($1,$2,$3)] }
 | expr T_DOUBLE_ARROW   TAND w_variable  { [ArrayArrowRef($1,$2,$3,$4)] }
```

⟨*repetitive non_empty_array_pair_list* ??⟩

119c ⟨*GRAMMAR variable* 117b⟩+≡

```
/*(*----------------------------*)*/
/*(* auxillary bis *)*/
/*(*----------------------------*)*/

exit_expr:
 | /*(*empty*)*/         { None }
 | TOPAR TCPAR           { Some($1, None, $2) }
 | TOPAR expr TCPAR      { Some($1, Some $2, $3) }
```

10.5 Function declaration

119d ⟨*GRAMMAR function declaration* 119d⟩≡

```
function_declaration_statement: unticked_function_declaration_statement { $1 }

unticked_function_declaration_statement:
  T_FUNCTION is_reference T_STRING
  TOPAR parameter_list TCPAR
  TOBRACE inner_statement_list TCBRACE
  {
    let params = ($4, $5, $6) in
```

```
    let body = ($7, $8, $9) in
    ({
      f_tok = $1;
      f_ref = $2;
      f_name = Name $3;
      f_params = params;
      f_body = body;
      f_type = Ast_php.noFtype();
    })
  }
```

120a ⟨*GRAMMAR function declaration* 119d⟩+≡

```
/*(* can not factorize, otherwise shift/reduce conflict *)*/
non_empty_parameter_list:
 | optional_class_type T_VARIABLE
     { let p = mk_param $1 $2 in [p] }
 | optional_class_type TAND T_VARIABLE
     { let p = mk_param $1 $3 in [{p with p_ref = Some $2}] }
 | optional_class_type T_VARIABLE         TEQ static_scalar
     { let p = mk_param $1 $2 in [{p with p_default = Some ($3,$4)}] }
 | optional_class_type TAND T_VARIABLE    TEQ static_scalar
     { let p = mk_param $1 $3 in
       [{p with p_ref = Some $2; p_default = Some ($4, $5)}]
     }
```

⟨*repetitive non_empty_parameter_list* ??⟩

120b ⟨*GRAMMAR function declaration* 119d⟩+≡

```
optional_class_type:
 | /*(*empty*)*/        { None }
 | T_STRING             { Some (Hint (Name $1)) }
 | T_ARRAY              { Some (HintArray $1) }

is_reference:
 | /*(*empty*)*/  { None }
 | TAND           { Some $1 }

/*(* PHP 5.3 *)*/
lexical_vars:
 | /*(*empty*)*/  { None }
 | T_USE TOPAR lexical_var_list TCPAR {
     Some ($1, ($2, ($3 +> List.map (fun (a,b) -> LexicalVar (a,b))), $4)) }

lexical_var_list:
 | T_VARIABLE                           { [None, DName $1] }
 | TAND T_VARIABLE                      { [Some $1, DName $2] }
```

```
  | lexical_var_list TCOMMA T_VARIABLE       { $1 ++ [None, DName $3]  }
  | lexical_var_list TCOMMA TAND T_VARIABLE  { $1 ++ [Some $3, DName $4] }
```

10.6 Class declaration

121a ⟨*GRAMMAR class declaration* 121a⟩≡

```
 class_declaration_statement: unticked_class_declaration_statement { $1 }

 unticked_class_declaration_statement:
  | class_entry_type class_name
      extends_from implements_list
      TOBRACE  class_statement_list TCBRACE
      { Left {
          c_type = $1;
          c_name = $2;
          c_extends = $3;
          c_implements = $4;
          c_body = $5, $6, $7;
        }
      }
  | interface_entry class_name
      interface_extends_list
      TOBRACE class_statement_list TCBRACE
      { Right {
          i_tok = $1;
          i_name = $2;
          i_extends = $3;
          i_body = $4, $5, $6;
        }
      }
```

121b ⟨*GRAMMAR class declaration* 121a⟩+≡

```
 class_name:
  | T_STRING { Name $1 }
  ⟨class_name grammar rule hook 127e⟩

 class_entry_type:
  | T_CLASS            { ClassRegular $1 }
  | T_ABSTRACT T_CLASS { ClassAbstract ($1, $2) }
  | T_FINAL    T_CLASS { ClassFinal ($1, $2) }

 interface_entry:
  | T_INTERFACE          { $1 }
```

122a ⟨*GRAMMAR class declaration* 121a⟩+≡
```
extends_from:
 | /*(*empty*)*/                         { None }
 | T_EXTENDS fully_qualified_class_name { Some ($1, $2) }

interface_extends_list:
 | /*(*empty*)*/            { None }
 | T_EXTENDS interface_list { Some($1,$2) }

implements_list:
 | /*(*empty*)*/               { None }
 | T_IMPLEMENTS interface_list { Some($1, $2) }
```

122b ⟨*GRAMMAR class declaration* 121a⟩+≡
```
/*(*----------------------------*)*/
/*(* class statement *)*/
/*(*----------------------------*)*/

class_statement:
 | T_CONST class_constant_declaration            TSEMICOLON
     { ClassConstants($1, $2, $3) }
 | variable_modifiers class_variable_declaration TSEMICOLON
     { ClassVariables($1, $2, $3) }
 | method_modifiers T_FUNCTION is_reference T_STRING
     TOPAR parameter_list TCPAR
     method_body
     { Method {
         m_modifiers = $1;
         m_tok = $2;
         m_ref = $3;
         m_name = Name $4;
         m_params = ($5, $6, $7);
         m_body = $8;
       }
     }
```

122c ⟨*GRAMMAR class declaration* 121a⟩+≡
```
class_constant_declaration:
 | T_STRING TEQ static_scalar
     { [(Name $1), ($2, $3)] }
 | class_constant_declaration TCOMMA      T_STRING TEQ static_scalar
     { $1 ++ [(Name $3, ($4, $5))] }


variable_modifiers:
 | T_VAR                                { NoModifiers $1 }
```

```
 | non_empty_member_modifiers          { VModifiers $1 }


/*(* can not factorize, otherwise shift/reduce conflict *)*/
class_variable_declaration:
 | T_VARIABLE                     { [DName $1, None] }
 | T_VARIABLE TEQ static_scalar { [DName $1, Some ($2, $3)] }
```

⟨*repetitive class_variable_declaration with comma* ??⟩

123a ⟨*GRAMMAR class declaration* 121a⟩+≡

```
member_modifier:
 | T_PUBLIC                            { Public,($1) }
 | T_PROTECTED                         { Protected,($1) }
 | T_PRIVATE                           { Private,($1) }

 | T_STATIC                            { Static,($1) }

 | T_ABSTRACT                          { Abstract,($1) }
 | T_FINAL                             { Final,($1) }


method_body:
 | TSEMICOLON                          { AbstractMethod $1 }
 | TOBRACE inner_statement_list TCBRACE { MethodBody ($1, $2, $3) }
```

10.7 Class bis

123b ⟨*GRAMMAR class bis* 123b⟩≡

```
class_name_reference:
 | T_STRING                     { ClassNameRefStatic (Name $1) }
 | dynamic_class_name_reference { ClassNameRefDynamic $1 }


dynamic_class_name_reference:
 | base_variable_bis { ($1, []) }
 | base_variable_bis
     T_OBJECT_OPERATOR object_property
     dynamic_class_name_variable_properties
     { ($1, ($2, $3)::$4) }


base_variable_bis: base_variable { basevar_to_variable $1 }
```

```
method_or_not:
  | TOPAR function_call_parameter_list TCPAR    { Some ($1, $2, $3) }
  | /*(*empty*)*/ { None }

ctor_arguments:
  | TOPAR function_call_parameter_list TCPAR    { Some ($1, $2, $3) }
  | /*(*empty*)*/ { None }
```

124a ⟨*GRAMMAR class bis* 123b⟩+≡

```
/*(*----------------------------*)*/
/*(* object property, variable property *)*/
/*(*----------------------------*)*/

object_property:
 | object_dim_list          { ObjProp $1 }
 | variable_without_objects_bis { ObjPropVar $1 }

variable_without_objects_bis: variable_without_objects
  { vwithoutobj_to_variable $1 }

/*(* quite similar to reference_variable, but without the '$' *)*/
object_dim_list:
 | variable_name { $1 }
 | object_dim_list TOBRA dim_offset TCBRA       { OArrayAccess($1, ($2,$3,$4)) }
 | object_dim_list TOBRACE expr TCBRACE         { OBraceAccess($1, ($2,$3,$4)) }

variable_name:
 | T_STRING             { OName (Name $1) }
 | TOBRACE expr TCBRACE { OBrace ($1,$2,$3) }



variable_property: T_OBJECT_OPERATOR object_property method_or_not
  { $1, $2, $3 }

dynamic_class_name_variable_property: T_OBJECT_OPERATOR object_property
  { $1, $2 }
```

10.8 Namespace

124b ⟨*GRAMMAR namespace* 124b⟩≡

```
qualifier: fully_qualified_class_name TCOLCOL { Qualifier ($1, $2) }

fully_qualified_class_name:
```

```
  | T_STRING { Name $1 }
```
⟨*fully_qualified_class_name grammar rule hook* 128a⟩

10.9 Encaps

125 ⟨*GRAMMAR encaps* 125⟩≡

```
encaps:
 | T_ENCAPSED_AND_WHITESPACE { EncapsString $1 }

 | T_VARIABLE
     {
       let refvar = (Var2 (DName $1, Ast_php.noScope())) in
       let basevar = None, ([], refvar) in
       let basevarbis = BaseVar basevar in
       let var = Variable (basevarbis, []) in
       EncapsVar (variable2_to_lvalue var)
     }
 | T_VARIABLE TOBRA encaps_var_offset TCBRA
     {
       let refvar = (Var2 (DName $1, Ast_php.noScope())) in
       let dimoffset = Some (mk_e $3) in
       let refvar = VArrayAccess2(refvar, ($2, dimoffset, $4)) in

       let basevar = None, ([], refvar) in
       let basevarbis = BaseVar basevar in
       let var = Variable (basevarbis, []) in
       EncapsVar (variable2_to_lvalue var)
     }
 | T_VARIABLE T_OBJECT_OPERATOR T_STRING
     {
       let refvar = (Var2 (DName $1, Ast_php.noScope())) in
       let basevar = None, ([], refvar) in
       let basevarbis = BaseVar basevar in

       let prop_string = ObjProp (OName (Name $1)) in
       let obj_prop = ($2, prop_string, None) in
       let var = Variable (basevarbis, [obj_prop]) in
       EncapsVar (variable2_to_lvalue var)
     }

 /*(* for ${beer}s. Note that this rule does not exist in the original PHP
    * grammer. Instead only the case with a TOBRA after the T_STRING_VARNAME
    * is covered. The case with only a T_STRING_VARNAME is handled
    * originally in the scalar rule, but it does not makes sense to me
```

```
    * as it's really more a variable than a scaler. So for now I have
    * defined this rule. maybe it's too restrictive, we'll see.
    *)*/
  | T_DOLLAR_OPEN_CURLY_BRACES T_STRING_VARNAME TCBRACE
     {
       (* this is not really a T_VARIABLE, bit it's still conceptually
        * a variable so we build it almost like above
        *)
       let refvar = (Var2 (DName $2, Ast_php.noScope())) in
       let basevar = None, ([], refvar) in
       let basevarbis = BaseVar basevar in
       let var = Variable (basevarbis, []) in
       EncapsDollarCurly ($1, variable2_to_lvalue var, $3)
     }

  | T_DOLLAR_OPEN_CURLY_BRACES T_STRING_VARNAME  TOBRA expr TCBRA  TCBRACE
     {
       let refvar = (Var2 (DName $2, Ast_php.noScope())) in
       let dimoffset = Some ($4) in
       let refvar = VArrayAccess2(refvar, ($3, dimoffset, $5)) in

       let basevar = None, ([], refvar) in
       let basevarbis = BaseVar basevar in
       let var = Variable (basevarbis, []) in
       EncapsDollarCurly ($1, variable2_to_lvalue var, $6)
     }

  /*(* for {$beer}s *)*/
  | T_CURLY_OPEN variable TCBRACE
     { EncapsCurly($1, $2, $3) }

  /*(* for ? *)*/
  | T_DOLLAR_OPEN_CURLY_BRACES expr TCBRACE
     { EncapsExpr ($1, $2, $3) }
```

126 ⟨*GRAMMAR encaps* 125⟩+≡

```
 encaps_var_offset:
  | T_STRING     {
     (* It looks like an ident (remember that T_STRING is a faux-ami,
      * it's actually used in the lexer for LABEL),
      * but as we are in encaps_var_offset,
      * php allows array access inside strings to omit the quote
      * around fieldname, so it's actually really a Constant (String)
      * rather than an ident, as we usually do for other T_STRING
      * cases.
      *)
```

```
      let cst = String $1 in (* will not have enclosing "'"  as usual *)
      Scalar (Constant cst)
    }
  | T_VARIABLE   {
        let refvar = (Var2 (DName $1, Ast_php.noScope())) in
        let basevar = None, ([], refvar) in
        let basevarbis = BaseVar basevar in
        let var = Variable (basevarbis, []) in
        Lvalue (variable2_to_lvalue var)
    }
  | T_NUM_STRING {
      (* the original php lexer does not return some numbers for
       * offset of array access inside strings. Not sure why ...
       * TODO?
       *)
      let cst = String $1 in (* will not have enclosing "'"  as usual *)
      Scalar (Constant cst)
    }
```

10.10 Pattern extensions

127a ⟨*GRAMMAR tokens hook* 127a⟩≡

```
%token <Ast_php.info> TDOTS
```

127b ⟨*exprbis grammar rule hook* 127b⟩≡

```
| TDOTS { EDots $1 }
```

10.11 XHP extensions

127c ⟨*function_call grammar rule hook* 127c⟩≡

```
/*(* xhp: in xhp grammar they use
   *   expr_without_variable:  expr '[' dim_offset ']'
   * but this generates 5 s/r conflicts. So better to put it here.
   *) */
| function_head TOPAR function_call_parameter_list TCPAR bra_list
 { FunCallArrayXhp($1, ($2, $3, $4), $5) }
```

127d ⟨*GRAMMAR tokens hook* 127a⟩+≡

```
%token <string * Ast_php.info> TXHPCOLONID
```

127e ⟨*class_name grammar rule hook* 127e⟩≡

```
/*(* xhp: *)*/
| TXHPCOLONID { XhpName $1 }
```

128a ⟨*fully_qualified_class_name grammar rule hook* 128a⟩≡
```
/*(* xhp: *)*/
| TXHPCOLONID { XhpName $1 }
```

10.12 Xdebug extensions

128b ⟨*static_scalar grammar rule hook* 128b⟩≡
```
/* xdebug TODO AST  */
| TDOTS { XdebugStaticDots  }
```

128c ⟨*common_scalar grammar rule hook* 128c⟩≡
```
| T_CLASS_XDEBUG class_name TOBRACE class_statement_list TCBRACE {
    XdebugClass ($2, $4)
  }
| T_CLASS_XDEBUG class_name TOBRACE TDOTS TCBRACE {
    XdebugClass ($2, [])

  }
| T_CLASS_XDEBUG class_name TOBRACE TDOTS TSEMICOLON TCBRACE {
    XdebugClass ($2, [])
  }
| T_RESOURCE_XDEBUG  {
    XdebugResource
  }
```

128d ⟨*GRAMMAR tokens hook* 127a⟩+≡
```
%token <Ast_php.info> T_CLASS_XDEBUG
%token <Ast_php.info> T_RESOURCE_XDEBUG
```

10.13 Prelude

128e ⟨*GRAMMAR prelude* 128e⟩≡
```
%{
(* src: ocamlyaccified from zend_language_parser.y in PHP source code.
 *
 ⟨Zend copyright 129a⟩
 *
 * /* Id: zend_language_parser.y 263383 2008-07-24 11:47:14Z dmitry */
 *
 * LALR shift/reduce conflicts and how they are resolved:
 *
 *  - 2 shift/reduce conflicts due to the dangeling elseif/else ambiguity.
 *  Solved by shift.
 *
```

```
 * %pure_parser
 * %expect 2

 *)
open Common

open Ast_php
open Parser_php_mly_helper

%}
```

129a ⟨*Zend copyright* 129a⟩≡

```
 * +----------------------------------------------------------------------+
 * | Zend Engine                                                          |
 * +----------------------------------------------------------------------+
 * | Copyright (c) 1998-2006 Zend Technologies Ltd. (http://www.zend.com) |
 * +----------------------------------------------------------------------+
 * | This source file is subject to version 2.00 of the Zend license,     |
 * | that is bundled with this package in the file LICENSE, and is        |
 * | available through the world-wide-web at the following url:          |
 * | http://www.zend.com/license/2_00.txt.                                |
 * | If you did not receive a copy of the Zend license and are unable to  |
 * | obtain it through the world-wide-web, please send a note to          |
 * | license@zend.com so we can mail you a copy immediately.              |
 * +----------------------------------------------------------------------+
 * | Authors: Andi Gutmans <andi@zend.com>                                |
 * |          Zeev Suraski <zeev@zend.com>                                |
 * +----------------------------------------------------------------------+
```

129b ⟨*parser_php_mly_helper.ml* 129b⟩≡

```
open Common

open Ast_php

(*****************************************************************************)
(* Parse helpers functions *)
(*****************************************************************************)
⟨function top_statements_to_toplevels 132b⟩

(*****************************************************************************)
(* Variable original type *)
(*****************************************************************************)
⟨type variable2 130a⟩

⟨variable2 to variable functions 130b⟩
```

```
(*****************************************************************************)
(* shortcuts *)
(*****************************************************************************)
⟨AST builder 132a⟩
```

130a ⟨*type variable2* 130a⟩≡

```
(* This type is only used for now during parsing time. It was originally
 * fully part of the PHP AST but it makes some processing like typing
 * harder with all the special cases. This type is more precise
 * than the one currently in the AST but it's not worthwhile the
 * extra complexity.
 *)
type variable2 =
  | Variable of base_var_and_funcall * obj_access list

  and base_var_and_funcall =
    | BaseVar of base_variable
    | FunCall of func_head * argument list paren
    (* xhp: idx trick *)
    | FunCallArrayXhp of func_head * argument list paren *
        expr option bracket list

    and base_variable = qualifier option * var_without_obj
      and var_without_obj = indirect list * ref_variable

      and ref_variable =
        | Var2 of dname * Scope_php.phpscope ref  (* semantic: *)
        | VDollar2 of tok * expr brace
        | VArrayAccess2 of ref_variable * expr option bracket
        | VBraceAccess2 of ref_variable * expr brace


    and func_head =
      (* static function call (or mostly static because in php
       * you can redefine functions ...)  *)
      | FuncName of qualifier option * name
      (* dynamic function call *)
      | FuncVar  of qualifier option * var_without_obj
```

130b ⟨*variable2 to variable functions* 130b⟩≡

```
let mkvar var = var, noTypeVar()

let method_object_simple x =
  match x with
  | ObjAccess(var, (t1, obj, argsopt)) ->
```

```
      (match obj, argsopt with
      | ObjProp (OName name), Some args ->
          (* todo? do special case when var is a Var ? *)
          MethodCallSimple (var, t1, name, args)
      | ObjProp (OName name), None ->
          ObjAccessSimple (var, t1, name)
      | _ -> x
      )
  | _ ->
      raise Impossible

let rec variable2_to_lvalue var =
  match var with
  | Variable (basevar, objs) ->
      let v = basevarfun_to_variable basevar in
      (* TODO left ? right ? *)
      objs +> List.fold_left (fun acc obj ->
        mkvar (method_object_simple (ObjAccess (acc, obj)))
      ) v

and basevarfun_to_variable basevarfun =
  match basevarfun with
  | BaseVar basevar ->
      basevar_to_variable basevar
  | FunCall (head, args) ->
      let v =
      (match head with
      | FuncName (qopt, name) ->
          FunCallSimple (qopt, name, args)
      | FuncVar (qopt, vwithoutobj) ->
          FunCallVar (qopt, vwithoutobj_to_variable vwithoutobj, args)
      )
      in
      mkvar v
  | FunCallArrayXhp (head, args, dims) ->
      let v = basevarfun_to_variable (FunCall(head, args)) in
      (* left is good direction *)
      dims +> List.fold_left (fun acc dim ->
        mkvar (VArrayAccess (acc, dim))
      ) v


and basevar_to_variable basevar =
  let (qu_opt, vwithoutobj) = basevar in
  let v = vwithoutobj_to_variable vwithoutobj in
  (match qu_opt with
```

```
  | None -> v
  | Some qu -> mkvar (VQualifier (qu, v))
  )

and vwithoutobj_to_variable vwithoutobj =
  let (indirects, refvar) = vwithoutobj in
  let v = refvar_to_variable refvar in
  indirects +> List.fold_left (fun acc indirect ->
    mkvar (Indirect (acc, indirect))) v


and refvar_to_variable refvar =
  let v =
    match refvar with
    | Var2 (name, scope) -> Var(name, scope)
    | VDollar2 (tok, exprp) -> VBrace(tok, exprp)
    | VArrayAccess2(refvar, exprb) ->
        let v = refvar_to_variable refvar in
        VArrayAccess(v, exprb)
    | VBraceAccess2(refvar, exprb) ->
        let v = refvar_to_variable refvar in
        VBraceAccess(v, exprb)
  in
  mkvar v
```

132a ⟨*AST builder* 132a⟩≡

```
let mk_param typ s =
  { p_type = typ;
    p_ref = None;
    p_name = DName s;
    p_default = None;
  }

let mk_e e = (e, Ast_php.noType())
```

132b ⟨*function top_statements_to_toplevels* 132b⟩≡

```
(* could have also created some fake Blocks, but simpler to have a
 * dedicated constructor for toplevel statements *)
let rec top_statements_to_toplevels topstatements eofinfo =
  match topstatements with
  | [] -> [FinalDef eofinfo]
  | x::xs ->
      let v, rest =
        (match x with
        | FuncDefNested      def -> FuncDef def,  xs
```

```
      | ClassDefNested     def -> ClassDef def, xs
      | InterfaceDefNested def -> InterfaceDef def, xs
      | Stmt st ->
          let stmts, rest = xs +> Common.span (function
            | Stmt st -> true
            | _ -> false
            ) in
          let stmts' = stmts +> List.map (function
            | Stmt st -> st
            | _ -> raise Impossible
          ) in
          StmtList (st::stmts'), rest
    )
  in
  v::top_statements_to_toplevels rest eofinfo
```

10.14 Tokens declaration and operator priorities

133a ⟨*GRAMMAR tokens declaration* 133a⟩≡

```
/*(*-----------------------------------------*)*/
/*(* the comment tokens *)*/
/*(*-----------------------------------------*)*/
⟨GRAMMAR comment tokens 133b⟩

/*(*-----------------------------------------*)*/
/*(* the normal tokens *)*/
/*(*-----------------------------------------*)*/
⟨GRAMMAR normal tokens 134⟩

/*(*-----------------------------------------*)*/
/*(* extra tokens: *)*/
/*(*-----------------------------------------*)*/
⟨GRAMMAR tokens hook 127a⟩



/*(*-----------------------------------------*)*/
%token <Ast_php.info> TUnknown /*(* unrecognized token *)*/
%token <Ast_php.info> EOF
```

Some tokens are not even used in the grammar file because they are filtered in some intermediate phases. But they still must be declared because `ocamllex` may generate them, or some intermediate phase may also generate them.

133b ⟨*GRAMMAR comment tokens* 133b⟩≡

```
/*(* coupling: Token_helpers.is_real_comment *)*/
%token <Ast_php.info> TCommentSpace TCommentNewline   TComment

/*(* not mentionned in this grammar. preprocessed *)*/
%token <Ast_php.info> T_COMMENT
%token <Ast_php.info> T_DOC_COMMENT
%token <Ast_php.info> T_WHITESPACE
```

134 ⟨*GRAMMAR normal tokens* 134⟩≡

```
%token <string * Ast_php.info> T_LNUMBER
%token <string * Ast_php.info> T_DNUMBER

/*(* T_STRING is regular ident and  T_VARIABLE is a dollar ident *)*/
%token <string * Ast_php.info> T_STRING
%token <string * Ast_php.info> T_VARIABLE

%token <string * Ast_php.info> T_CONSTANT_ENCAPSED_STRING
%token <string * Ast_php.info> T_ENCAPSED_AND_WHITESPACE

/*(* used only for offset of array access inside strings *)*/
%token <string * Ast_php.info> T_NUM_STRING

%token <string * Ast_php.info> T_INLINE_HTML


%token <string * Ast_php.info> T_STRING_VARNAME

%token <Ast_php.info> T_CHARACTER
%token <Ast_php.info> T_BAD_CHARACTER


%token <Ast_php.info> T_ECHO T_PRINT

%token <Ast_php.info> T_IF
%token <Ast_php.info> T_ELSE T_ELSEIF T_ENDIF
%token <Ast_php.info> T_DO
%token <Ast_php.info> T_WHILE   T_ENDWHILE
%token <Ast_php.info> T_FOR     T_ENDFOR
%token <Ast_php.info> T_FOREACH T_ENDFOREACH
%token <Ast_php.info> T_SWITCH  T_ENDSWITCH
%token <Ast_php.info> T_CASE T_DEFAULT    T_BREAK T_CONTINUE
%token <Ast_php.info> T_RETURN
%token <Ast_php.info> T_TRY  T_CATCH T_THROW
%token <Ast_php.info> T_EXIT

%token <Ast_php.info> T_DECLARE T_ENDDECLARE
```

```
%token <Ast_php.info> T_USE
%token <Ast_php.info> T_GLOBAL
%token <Ast_php.info> T_AS
%token <Ast_php.info> T_FUNCTION
%token <Ast_php.info> T_CONST

/*(* pad: was declared via right ... ??? mean token ? *)*/
%token <Ast_php.info> T_STATIC  T_ABSTRACT  T_FINAL
%token <Ast_php.info> T_PRIVATE T_PROTECTED T_PUBLIC
%token <Ast_php.info> T_VAR

%token <Ast_php.info> T_UNSET
%token <Ast_php.info> T_ISSET
%token <Ast_php.info> T_EMPTY

%token <Ast_php.info> T_HALT_COMPILER

%token <Ast_php.info> T_CLASS   T_INTERFACE
%token <Ast_php.info> T_EXTENDS T_IMPLEMENTS
%token <Ast_php.info> T_OBJECT_OPERATOR

%token <Ast_php.info> T_DOUBLE_ARROW

%token <Ast_php.info> T_LIST T_ARRAY

%token <Ast_php.info> T_CLASS_C T_METHOD_C T_FUNC_C
%token <Ast_php.info> T_LINE   T_FILE


%token <Ast_php.info> T_OPEN_TAG  T_CLOSE_TAG
%token <Ast_php.info> T_OPEN_TAG_WITH_ECHO


%token <Ast_php.info> T_START_HEREDOC    T_END_HEREDOC
%token <Ast_php.info> T_DOLLAR_OPEN_CURLY_BRACES
%token <Ast_php.info> T_CURLY_OPEN

%token <Ast_php.info> TCOLCOL

/*(* pad: was declared as left/right, without a token decl in orig gram *)*/
%token <Ast_php.info> TCOLON TCOMMA TDOT TBANG TTILDE TQUESTION

%token <Ast_php.info> TOBRA

%token <Ast_php.info> TPLUS TMINUS TMUL TDIV TMOD
```

```
%token <Ast_php.info> TAND TOR TXOR
%token <Ast_php.info> TEQ
%token <Ast_php.info> TSMALLER TGREATER

%token <Ast_php.info> T_PLUS_EQUAL  T_MINUS_EQUAL  T_MUL_EQUAL  T_DIV_EQUAL
%token <Ast_php.info> T_CONCAT_EQUAL  T_MOD_EQUAL
%token <Ast_php.info> T_AND_EQUAL T_OR_EQUAL T_XOR_EQUAL T_SL_EQUAL T_SR_EQUAL
%token <Ast_php.info> T_INC    T_DEC
%token <Ast_php.info> T_BOOLEAN_OR   T_BOOLEAN_AND
%token <Ast_php.info> T_LOGICAL_OR   T_LOGICAL_AND   T_LOGICAL_XOR
%token <Ast_php.info> T_SL    T_SR
%token <Ast_php.info> T_IS_SMALLER_OR_EQUAL    T_IS_GREATER_OR_EQUAL

%token <Ast_php.info> T_BOOL_CAST  T_INT_CAST   T_DOUBLE_CAST  T_STRING_CAST
%token <Ast_php.info> T_ARRAY_CAST T_OBJECT_CAST
%token <Ast_php.info> T_UNSET_CAST

%token <Ast_php.info> T_IS_IDENTICAL T_IS_NOT_IDENTICAL
%token <Ast_php.info> T_IS_EQUAL     T_IS_NOT_EQUAL


%token <Ast_php.info> T__AT

%token <Ast_php.info> T_NEW T_CLONE T_INSTANCEOF

%token <Ast_php.info> T_INCLUDE T_INCLUDE_ONCE T_REQUIRE T_REQUIRE_ONCE
%token <Ast_php.info> T_EVAL

/*(* was declared implicitely cos was using directly the character *)*/
%token <Ast_php.info> TOPAR TCPAR
%token <Ast_php.info> TOBRACE TCBRACE
%token <Ast_php.info> TCBRA
%token <Ast_php.info> TBACKQUOTE
%token <Ast_php.info> TSEMICOLON
%token <Ast_php.info> TDOLLAR /*(* see also T_VARIABLE *)*/
%token <Ast_php.info> TGUIL
```

136 ⟨*GRAMMAR tokens priorities* 136⟩≡

```
/*(*-----------------------------------------*)*/
/*(* must be at the top so that it has the lowest priority *)*/
%nonassoc SHIFTHERE


%left      T_INCLUDE T_INCLUDE_ONCE T_EVAL T_REQUIRE T_REQUIRE_ONCE
```

```
%left      TCOMMA
%left      T_LOGICAL_OR
%left      T_LOGICAL_XOR
%left      T_LOGICAL_AND
%right     T_PRINT
%left      TEQ T_PLUS_EQUAL T_MINUS_EQUAL T_MUL_EQUAL T_DIV_EQUAL T_CONCAT_EQUAL T_MOD_EQU
%left      TQUESTION TCOLON
%left      T_BOOLEAN_OR
%left      T_BOOLEAN_AND
%left      TOR
%left      TXOR
%left      TAND
%nonassoc  T_IS_EQUAL T_IS_NOT_EQUAL T_IS_IDENTICAL T_IS_NOT_IDENTICAL
%nonassoc  TSMALLER T_IS_SMALLER_OR_EQUAL TGREATER T_IS_GREATER_OR_EQUAL
%left      T_SL T_SR
%left      TPLUS TMINUS TDOT
%left      TMUL TDIV TMOD
%right     TBANG
%nonassoc  T_INSTANCEOF
%right     TTILDE T_INC T_DEC T_INT_CAST T_DOUBLE_CAST T_STRING_CAST T_ARRAY_CAST T_OBJECT
%right     T__AT
%right     TOBRA
%nonassoc  T_NEW T_CLONE
%left      T_ELSEIF
%left      T_ELSE
%left      T_ENDIF
```

10.15 Yacc annoyances (EBNF vs BNF)

137 ⟨*GRAMMAR xxxlist or xxxopt* 137⟩≡

```
top_statement_list:
 | top_statement_list  top_statement { $1 ++ [$2] }
 | /*(*empty*)*/ { [] }
```

⟨*repetitive xxx_list* ??⟩

```
additional_catches:
 | non_empty_additional_catches { $1 }
 | /*(*empty*)*/ { [] }

non_empty_additional_catches:
 | additional_catch                               { [$1] }
 | non_empty_additional_catches additional_catch { $1 ++ [$2] }
```

⟨*repetitive xxx and non_empty_xxx* ??⟩

```
unset_variables:
 | unset_variable { [$1] }
 | unset_variables TCOMMA unset_variable { $1 ++ [$3] }
```

⟨*repetitive xxx_list with TCOMMA* ??⟩

```
bra_list:
 | bra { [$1] }
 | bra_list bra { $1 ++ [$2] }


possible_comma:
 | /*(*empty*)*/ { None }
 | TCOMMA        { Some $1 }

static_array_pair_list:
 | /*(*empty*)*/ {  [] }
 | non_empty_static_array_pair_list possible_comma      { $1 }

array_pair_list:
 | /*(*empty*)*/ { [] }
 | non_empty_array_pair_list possible_comma     { $1 }
```

Chapter 11

Parser glue code

The high-level structure of `parse_php.ml` has already been described in Section 8.3. The previous chapters have also described some of the functions in `parse_php.ml` (for getting a stream of tokens and calling ocamlyacc parser). In this section we will mostly fill in the remaining holes.

139a ⟨*parse_php module aliases* 139a⟩≡

```
module Ast  = Ast_php
module Flag = Flag_parsing_php
module TH   = Token_helpers_php
```

139b ⟨*function program_of_program2* 139b⟩≡

```
let program_of_program2 xs =
  xs +> List.map fst
```

139c ⟨*parse_php helpers* 139c⟩≡

```
let lexbuf_to_strpos lexbuf     =
  (Lexing.lexeme lexbuf, Lexing.lexeme_start lexbuf)

let token_to_strpos tok =
  (TH.str_of_tok tok, TH.pos_of_tok tok)
```

139d ⟨*parse_php helpers* 139c⟩+≡

```
let mk_info_item2 filename toks =
  let buf = Buffer.create 100 in
  let s =
    (* old: get_slice_file filename (line1, line2) *)
    begin
      toks +> List.iter (fun tok ->
        match TH.pinfo_of_tok tok with
        | Ast.OriginTok _ ->
            Buffer.add_string buf (TH.str_of_tok tok)
```

```
        | Ast.Ab _ | Ast.FakeTokStr _ -> raise Impossible
      );
      Buffer.contents buf
    end
  in
  (s, toks)

let mk_info_item a b =
  Common.profile_code "Parsing.mk_info_item"
    (fun () -> mk_info_item2 a b)
```

140a ⟨*parse_php helpers* 139c⟩+≡

```
(* on very huge file, this function was previously segmentation fault
 * in native mode because span was not tail call
 *)
let rec distribute_info_items_toplevel2 xs toks filename =
  match xs with
  | [] -> raise Impossible
  | [Ast_php.FinalDef e] ->
      (* assert (null toks) ??? no cos can have whitespace tokens *)
      let info_item = mk_info_item filename toks in
      [Ast_php.FinalDef e, info_item]
  | ast::xs ->

      let ii = Lib_parsing_php.ii_of_toplevel ast in
      let (min, max) = Lib_parsing_php.min_max_ii_by_pos ii in

      let max = Ast_php.parse_info_of_info max in

      let toks_before_max, toks_after =
        Common.profile_code "spanning tokens" (fun () ->
        toks +> Common.span_tail_call (fun tok ->
          Token_helpers_php.pos_of_tok tok <= max.charpos
        ))
      in
      let info_item = mk_info_item filename toks_before_max in
      (ast, info_item)::distribute_info_items_toplevel2 xs toks_after filename

let distribute_info_items_toplevel a b c =
  Common.profile_code "distribute_info_items" (fun () ->
    distribute_info_items_toplevel2 a b c
  )
```

140b ⟨*parse_php error diagnostic* 140b⟩≡

```
let error_msg_tok tok =
  let file = TH.file_of_tok tok in
```

```
    if !Flag.verbose_parsing
    then Common.error_message file (token_to_strpos tok)
    else ("error in " ^ file  ^ "set verbose_parsing for more info")

  let print_bad line_error (start_line, end_line) filelines  =
    begin
      pr2 ("badcount: " ^ i_to_s (end_line - start_line));

      for i = start_line to end_line do
        let line = filelines.(i) in

        if i =|= line_error
        then  pr2 ("BAD:!!!!!" ^ " " ^ line)
        else  pr2 ("bad:" ^ " " ^      line)
      done
    end
```

141a ⟨*parse_php stat function* 141a⟩≡

```
  let default_stat file =  {
      filename = file;
      correct = 0; bad = 0;
  (*
      have_timeout = false;
      commentized = 0;
      problematic_lines = [];
  *)
    }
```

141b ⟨*parse_php stat function* 141a⟩+≡

```
  let print_parsing_stat_list statxs =
    let total = List.length statxs in
    let perfect =
      statxs
        +> List.filter (function
        | {bad = n} when n = 0 -> true
        | _ -> false)
        +> List.length
    in

    pr "\n\n\n---------------------------------------------------------------";
    pr (
    (spf "NB total files = %d; " total) ^
    (spf "perfect = %d; " perfect) ^
    (spf "=========> %d" ((100 * perfect) / total)) ^ "%"
    );
```

```
let good = statxs +> List.fold_left (fun acc {correct = x} -> acc+x) 0 in
let bad  = statxs +> List.fold_left (fun acc {bad = x} -> acc+x) 0  in

let gf, badf = float_of_int good, float_of_int bad in
pr (
(spf "nb good = %d,  nb bad = %d " good bad) ^
(spf "=========> %f"  (100.0 *. (gf /. (gf +. badf))) ^ "%"
 )
)
```

Chapter 12

Style preserving unparsing

143 ⟨*unparse_php.ml* 143⟩≡
```
open Common

open Ast_php

module V = Visitor_php
module Ast = Ast_php

(* TODO
Want to put this module in parsing_php/
it does not have to be here, but maybe simpler
to put it here so have   basic parser/unparser
together.
*)


let string_of_program2 ast2 =
  Common.with_open_stringbuf (fun (_pr_with_nl, buf) ->
    let pp s =
      Buffer.add_string buf s
    in
    let cur_line = ref 1 in

    pp "<?php";
    pp "\n";
    incr cur_line;

    let hooks = { V.default_visitor with
      V.kinfo = (fun (k, _) info ->
        match info.pinfo with
        | OriginTok p ->
```

```
            let line = p.Common.line in
            if line > !cur_line
            then begin
              (line - !cur_line) +> Common.times (fun () -> pp "\n");
              cur_line := line;
            end;

            let s =  p.Common.str in
            pp s; pp " ";
        | FakeTokStr s ->
            pp s; pp " ";
            if s = ";"
            then begin
              pp "\n";
              incr cur_line;
            end

        | Ab
          ->
            ()
      );
      V.kcomma = (fun (k,_) () ->
        pp ", ";
      );
    }
    in

    ast2 +> List.iter (fun (top, infos) ->
      (V.mk_visitor hooks).V.vtop top
    )


  )


let string_of_toplevel top =
  Common.with_open_stringbuf (fun (_pr_with_nl, buf) ->

    let pp s =
      Buffer.add_string buf s
    in
    let hooks = { V.default_visitor with
      V.kinfo = (fun (k, _) info ->
        match info.pinfo with
        | OriginTok p ->
            let s =  p.Common.str in
```

```
          pp s; pp " ";
      | FakeTokStr s ->
          pp s; pp " ";
          if s = ";" || s = "{" || s = "}"
          then begin
            pp "\n";
          end

      | Ab
        ->
          ()
    );
    V.kcomma = (fun (k,_) () ->
      pp ", ";
    );
  }
  in
  (V.mk_visitor hooks).V.vtop top
)
```

Chapter 13

Auxillary parsing code

13.1 ast_php.ml

146 ⟨*ast_php.ml* 146⟩≡
⟨*Facebook copyright* 9⟩

```
open Common
(*****************************************************************************)
(* The AST related types *)
(*****************************************************************************)
(* ------------------------------------------------------------------------- *)
(* Token/info *)
(* ------------------------------------------------------------------------- *)
⟨AST info 52b⟩
(* ------------------------------------------------------------------------- *)
(* Name. See also analyze_php/namespace_php.ml  *)
(* ------------------------------------------------------------------------- *)
⟨AST name 51e⟩
(* ------------------------------------------------------------------------- *)
(* Type. This is used in Cast, but for type analysis see type_php.ml  *)
(* ------------------------------------------------------------------------- *)
⟨AST type 50c⟩
(* ------------------------------------------------------------------------- *)
(* Expression *)
(* ------------------------------------------------------------------------- *)
⟨AST expression 35⟩
(* ------------------------------------------------------------------------- *)
(* Variable (which in fact also contains function calls) *)
(* ------------------------------------------------------------------------- *)
⟨AST lvalue 41e⟩
(* ------------------------------------------------------------------------- *)
```

```
(* Statement *)
(* ------------------------------------------------------------------------ *)
⟨AST statement 43d⟩
(* ------------------------------------------------------------------------ *)
(* Function definition *)
(* ------------------------------------------------------------------------ *)
⟨AST function definition 47g⟩
⟨AST lambda definition 40g⟩
(* ------------------------------------------------------------------------ *)
(* Class definition *)
(* ------------------------------------------------------------------------ *)
⟨AST class definition 48d⟩
(* ------------------------------------------------------------------------ *)
(* Other declarations *)
(* ------------------------------------------------------------------------ *)
⟨AST other declaration 45g⟩
(* ------------------------------------------------------------------------ *)
(* Stmt bis *)
(* ------------------------------------------------------------------------ *)
⟨AST statement bis 51d⟩
(* ------------------------------------------------------------------------ *)
(* phpext: *)
(* ------------------------------------------------------------------------ *)
⟨AST phpext 58d⟩
(* ------------------------------------------------------------------------ *)
(* The toplevels elements *)
(* ------------------------------------------------------------------------ *)
⟨AST toplevel 50d⟩

(***************************************************************************)
(* Comments *)
(***************************************************************************)
```

147a ⟨*ast_php.ml* 146⟩+≡

```
(***************************************************************************)
(* Some constructors *)
(***************************************************************************)
let noType () = ({ t = [Type_php.Unknown]})
let noTypeVar () = ({ tlval = [Type_php.Unknown]})
let noScope () = ref (Scope_php.NoScope)
let noFtype () = ([Type_php.Unknown])
```

147b ⟨*ast_php.ml* 146⟩+≡

```
(***************************************************************************)
(* Wrappers *)
(***************************************************************************)
```

```
let unwrap = fst

let unparen (a,b,c) = b
let unbrace = unparen
let unbracket = unparen
```

148a ⟨*ast_php.ml* 146⟩+≡

```
let untype (e, xinfo) = e
```

148b ⟨*ast_php.ml* 146⟩+≡

```
let parse_info_of_info ii =
  match ii.pinfo with
  | OriginTok pinfo -> pinfo
  | FakeTokStr _
  | Ab
    -> failwith "parse_info_of_info: no OriginTok"
```

148c ⟨*ast_php.ml* 146⟩+≡

```
let pos_of_info  ii = (parse_info_of_info ii).Common.charpos
let str_of_info  ii = (parse_info_of_info ii).Common.str
let file_of_info ii = (parse_info_of_info ii).Common.file
let line_of_info ii = (parse_info_of_info ii).Common.line
let col_of_info  ii = (parse_info_of_info ii).Common.column
```

148d ⟨*ast_php.ml* 146⟩+≡

```
let pinfo_of_info ii = ii.pinfo
```

148e ⟨*ast_php.ml* 146⟩+≡

```
let rewrap_str s ii =
  {ii with pinfo =
    (match ii.pinfo with
    | OriginTok pi -> OriginTok { pi with Common.str = s;}
    | FakeTokStr s -> FakeTokStr s
    | Ab -> Ab
    )
  }
```

148f ⟨*ast_php.ml* 146⟩+≡

```
(* for error reporting *)
let string_of_info ii =
  Common.string_of_parse_info (parse_info_of_info ii)

let is_origintok ii =
  match ii.pinfo with
  | OriginTok pi -> true
  | FakeTokStr _ | Ab -> false
```

```
let compare_pos ii1 ii2 =
  let get_pos = function
    | OriginTok pi -> (*Real*) pi
    | FakeTokStr _
    | Ab
      -> failwith "Ab or FakeTok"
  in
  let pos1 = get_pos (pinfo_of_info ii1) in
  let pos2 = get_pos (pinfo_of_info ii2) in
  match (pos1,pos2) with
  ((*Real*) p1, (*Real*) p2) ->
    compare p1.Common.charpos p2.Common.charpos
```

149a ⟨*ast_php.ml* 146⟩+≡

```
let get_type (e: expr) = (snd e).t
let set_type (e: expr) (ty: Type_php.phptype) =
  (snd e).t <- ty
```

149b ⟨*ast_php.ml* 146⟩+≡

```
(*****************************************************************************)
(* Abstract line *)
(*****************************************************************************)

(* When we have extended the AST to add some info about the tokens,
 * such as its line number in the file, we can not use anymore the
 * ocaml '=' to compare Ast elements. To overcome this problem, to be
 * able to use again '=', we just have to get rid of all those extra
 * information, to "abstract those line" (al) information.
 *)

let al_info x =
  raise Todo
```

149c ⟨*ast_php.ml* 146⟩+≡

```
(*****************************************************************************)
(* Views *)
(*****************************************************************************)

(* examples:
 * inline more static funcall in expr type or variable type
 *
 *)
```

150a ⟨*ast_php.ml* 146⟩+≡
```
(*****************************************************************************)
(* Helpers, could also be put in lib_parsing.ml instead *)
(*****************************************************************************)
let name e =
  match e with
  | (Name x) -> unwrap x
  | XhpName x -> unwrap x (* TODO ? analyze the string for ':' ? *)

let dname (DName x) = unwrap x
```

150b ⟨*ast_php.ml* 146⟩+≡
```
let info_of_name e =
  match e with
  | (Name (x,y)) -> y
  | (XhpName (x,y)) -> y
let info_of_dname (DName (x,y)) = y
```

13.2 lib_parsing_php.ml

150c ⟨*lib_parsing_php.ml* 150c⟩≡
```
⟨Facebook copyright 9⟩

open Common

⟨basic pfff module open and aliases 158⟩
module V = Visitor_php

(*****************************************************************************)
(* Wrappers *)
(*****************************************************************************)
let pr2, pr2_once = Common.mk_pr2_wrappers Flag.verbose_parsing

(*****************************************************************************)
(* Extract infos *)
(*****************************************************************************)
⟨extract infos 151a⟩

(*****************************************************************************)
(* Abstract position *)
(*****************************************************************************)
⟨abstract infos 151c⟩

(*****************************************************************************)
(* Max min, range *)
```

```
(*****************************************************************************)
⟨max min range 152b⟩

(*****************************************************************************)
(* Ast getters *)
(*****************************************************************************)
⟨ast getters 153a⟩
```

151a ⟨*extract infos* 151a⟩≡

```
let extract_info_visitor recursor =
  let globals = ref [] in
  let hooks = { V.default_visitor with
    V.kinfo = (fun (k, _) i -> Common.push2 i globals)
  } in
  begin
    let vout = V.mk_visitor hooks in
    recursor vout;
    !globals
  end
```

151b ⟨*extract infos* 151a⟩+≡

```
let ii_of_toplevel top =
  extract_info_visitor (fun visitor -> visitor.V.vtop top)

let ii_of_expr e =
  extract_info_visitor (fun visitor -> visitor.V.vexpr e)

let ii_of_stmt e =
  extract_info_visitor (fun visitor -> visitor.V.vstmt e)

let ii_of_argument e =
  extract_info_visitor (fun visitor -> visitor.V.vargument e)

let ii_of_lvalue e =
  extract_info_visitor (fun visitor -> visitor.V.vlvalue e)
```

151c ⟨*abstract infos* 151c⟩≡

```
let abstract_position_visitor recursor =
  let hooks = { V.default_visitor with
    V.kinfo = (fun (k, _) i ->
      i.pinfo <- Ast_php.Ab;
    )
  } in
  begin
    let vout = V.mk_visitor hooks in
```

```
    recursor vout;
  end
```

152a ⟨*abstract infos* 151c⟩+≡

```
let abstract_position_info_program x =
  abstract_position_visitor (fun visitor -> visitor.V.vprogram x; x)
let abstract_position_info_expr x =
  abstract_position_visitor (fun visitor -> visitor.V.vexpr x; x)
let abstract_position_info_toplevel x =
  abstract_position_visitor (fun visitor -> visitor.V.vtop x; x)
```

152b ⟨*max min range* 152b⟩≡

```
let min_max_ii_by_pos xs =
  match xs with
  | [] -> failwith "empty list, max_min_ii_by_pos"
  | [x] -> (x, x)
  | x::xs ->
      let pos_leq p1 p2 = (Ast_php.compare_pos p1 p2) =|= (-1) in
      xs +> List.fold_left (fun (minii,maxii) e ->
        let maxii' = if pos_leq maxii e then e else maxii in
        let minii' = if pos_leq e minii then e else minii in
        minii', maxii'
      ) (x,x)
```

152c ⟨*max min range* 152b⟩+≡

```
let info_to_fixpos ii =
  match Ast_php.pinfo_of_info ii with
  | Ast_php.OriginTok pi ->
      (* Ast_cocci.Real *)
      pi.Common.charpos
  | Ast_php.FakeTokStr _
  | Ast_php.Ab
    -> failwith "unexpected abstract or faketok"

let min_max_by_pos xs =
  let (i1, i2) = min_max_ii_by_pos xs in
  (info_to_fixpos i1, info_to_fixpos i2)

let (range_of_origin_ii: Ast_php.info list -> (int * int) option) =
 fun ii ->
  let ii = List.filter Ast_php.is_origintok ii in
  try
    let (min, max) = min_max_ii_by_pos ii in
    assert(Ast_php.is_origintok max);
    assert(Ast_php.is_origintok min);
    let strmax = Ast_php.str_of_info max in
```

```
      Some
        (Ast_php.pos_of_info min, Ast_php.pos_of_info max + String.length strmax)
    with _ ->
      None
```

153a ⟨*ast getters* 153a⟩≡

```
let get_all_funcalls f =
  let h = Hashtbl.create 101 in

  let hooks = { V.default_visitor with

    (* TODO if nested function ??? still wants to report ? *)
    V.klvalue = (fun (k,vx) x ->
      match untype x with
      | FunCallSimple (qu_opt, callname, args) ->
          let str = Ast_php.name callname in
          Hashtbl.replace h str true;
          k x
      | _ -> k x
    );
  }
  in
  let visitor = V.mk_visitor hooks in
  f visitor;
  Common.hashset_to_list h
```

153b ⟨*ast getters* 153a⟩+≡

```
let get_all_funcalls_ast ast =
  get_all_funcalls (fun visitor ->  visitor.V.vtop ast)

let get_all_funcalls_in_body body =
  get_all_funcalls (fun visitor ->  body +> List.iter visitor.V.vstmt_and_def)
```

153c ⟨*ast getters* 153a⟩+≡

```
let get_all_constant_strings_ast ast =
  let h = Hashtbl.create 101 in

  let hooks = { V.default_visitor with
    V.kconstant = (fun (k,vx) x ->
      match x with
      | String (str,ii) ->
          Hashtbl.replace h str true;
      | _ -> k x
    );
    V.kencaps = (fun (k,vx) x ->
      match x with
```

```
      | EncapsString (str, ii) ->
          Hashtbl.replace h str true;
      | _ -> k x
    );
  }

  in
  (V.mk_visitor hooks).V.vtop ast;
  Common.hashset_to_list h
```

154a ⟨*ast getters* 153a⟩+≡

```
let get_all_funcvars_ast ast =
  let h = Hashtbl.create 101 in

  let hooks = { V.default_visitor with

    V.klvalue = (fun (k,vx) x ->
      match untype x with
      | FunCallVar (qu_opt, var, args) ->

          (* TODO enough ? what about qopt ?
           * and what if not directly a Var ?
           *)
          (match untype var with
          | Var (dname, _scope) ->
              let str = Ast_php.dname dname in
              Hashtbl.replace h str true;
              k x

          | _ -> k x
          )
      | _ ->  k x
    );
  }
  in
  let visitor = V.mk_visitor hooks in
  visitor.V.vtop ast;
  Common.hashset_to_list h
```

13.3 json_ast_php.ml

154b ⟨*json_ast_php.ml* 154b⟩≡

```
open Common

module J = Json_type
```

```
let json_ex =
  J.Object [
    ("fld1", J.Bool true);
    ("fld2", J.Int 2);
  ]

let rec sexp_to_json sexp =
  match sexp with

  | Sexp.List xs ->
      (* try to recognize records to generate some J.Object *)

      (match xs with
      (* assumes the sexp was auto generated via ocamltarzan code which
       * adds those ':' to record fields.
       * See pa_sexp2_conv.ml.
       *)
      | (Sexp.List [(Sexp.Atom s);arg])::_ys when s =~ ".*:" ->
          J.Object (xs +> List.map (function
          | Sexp.List [(Sexp.Atom s);arg] ->
              if s =~ "\\(.*\\):"
              then
                let fld = Common.matched1 s in
                fld, sexp_to_json arg
              else
                failwith "wrong sexp; was it generated via ocamltarzan code ?"
          | _ ->
              failwith "wrong sexp; was it generated via ocamltarzan code ?"
          ))

      | _ ->

          (* default behavior *)
          J.Array (List.map sexp_to_json xs)
      )

  | Sexp.Atom s ->
      (* try to "reverse engineer" the basic types *)
      (try
        let i = int_of_string s in
        J.Int i
      with _ ->

        (try
```

```
            let f = float_of_string s in
            J.Float f
          with _ ->

            (match s with
            | "true" -> J.Bool true
            | "false" -> J.Bool false
            (* | "None"  ??? J.Null *)

            | _ ->
                (* default behavior *)
                J.String s
            )
        )
      )

let json_of_program x =
  Common.save_excursion_and_enable (Sexp_ast_php.show_info) (fun () ->
    let sexp = Sexp_ast_php.sexp_of_program x in
    sexp_to_json sexp
  )

let string_of_program x =
  let json = json_of_program x in
  Json_out.string_of_json json

let string_of_expr x =
  raise Todo

let string_of_toplevel x =
  raise Todo
```

13.4 type_php.ml

156 ⟨*type_php.ml* 156⟩≡
⟨*Facebook copyright* 9⟩

```
open Common
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*
 * It would be more convenient to move this file elsewhere like in analyse_php/
```

```
 * but we want our AST to contain type annotations so it's convenient to
 * have the type definition of PHP types here in parsing_php/.
 * If later we decide to make a 'a expr, 'a stmt, and have a convenient
 * mapper between some 'a expr to 'b expr, then maybe we can move
 * this file to a better place.
 *
 * TODO? have a scalar supertype ? that enclose string/int/bool ?
 * after automatic string interpolation of basic types are useful.
 * Having to do those %s %d in ocaml sometimes sux.
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
⟨type phptype 54d⟩

⟨type phpfunction_type 56a⟩

(*****************************************************************************)
(* String of *)
(*****************************************************************************)

let string_of_phptype t =
  raise Todo
```

13.5 scope_php.ml

157 ⟨*scope_php.ml* 157⟩≡

```
⟨Facebook copyright 9⟩

open Common
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(*
 * It would be more convenient to move this file elsewhere like in analyse_php/
 * but we want our AST to contain scope annotations so it's convenient to
 * have the type definition of PHP scope here in parsing_php/.
 * See also type_php.ml
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
⟨scope_php.mli 56d⟩
```

158 ⟨*basic pfff module open and aliases* 158⟩≡

```
open Ast_php

module Ast = Ast_php
module Flag = Flag_parsing_php
```

Conclusion

Appendix A

Remaining Testing Sample Code

160 ⟨*test_parsing_php.ml* 160⟩≡

```
open Common

(*****************************************************************************)
(* Subsystem testing *)
(*****************************************************************************)
⟨test_tokens_php 161b⟩
(* ------------------------------------------------------------------------ *)
⟨test_parse_php 27a⟩
(* ------------------------------------------------------------------------ *)
⟨test_sexp_php 67a⟩
(* ------------------------------------------------------------------------ *)
⟨test_json_php 69b⟩
(* ------------------------------------------------------------------------ *)
⟨test_visit_php 64c⟩

(* ------------------------------------------------------------------------ *)
let test_unparse_php file =
  let (ast2, stat) = Parse_php.parse file in
  let s = Unparse_php.string_of_program2 ast2 in
  pr2 s;
  ()

(* ------------------------------------------------------------------------ *)
let test_parse_xhp file =
  let pp_cmd = "xhpize" in
  let (ast2, stat) = Parse_php.parse ~pp:(Some pp_cmd) file in
  let ast = Parse_php.program_of_program2 ast2 in
```

```
  Sexp_ast_php.show_info := false;
  let s = Sexp_ast_php.string_of_program ast in
  pr2 s;
  ()

let test_parse_xdebug_expr s =
  let e = Parse_php.xdebug_expr_of_string s in
  Sexp_ast_php.show_info := false;
  let s = Sexp_ast_php.string_of_expr e in
  pr2 s;
  ()

(*****************************************************************************)
(* Main entry for Arg *)
(*****************************************************************************)
let actions () = [
  ⟨test_parsing_php actions 26e⟩

    "-unparse_php", "   <file>",
    Common.mk_action_1_arg test_unparse_php;
    "-parse_xdebug_expr", "   <string>",
    Common.mk_action_1_arg test_parse_xdebug_expr;
    "-parse_xhp", "   <file>",
    Common.mk_action_1_arg test_parse_xhp;
]
```

161a ⟨*test_parsing_php actions* 26e⟩+≡

```
  "-tokens_php", "   <file>",
  Common.mk_action_1_arg test_tokens_php;
```

161b ⟨*test_tokens_php* 161b⟩≡

```
let test_tokens_php file =
  if not (file =~ ".*\\.php")
  then pr2 "warning: seems not a .php file";

  Flag_parsing_php.verbose_lexing := true;
  Flag_parsing_php.verbose_parsing := true;

  let toks = Parse_php.tokens file in
  toks +> List.iter (fun x -> pr2_gen x);
  ()
```

Indexes

Bibliography

[1] Donald Knuth,, *Literate Programming*, http://en.wikipedia.org/wiki/Literate_Program cited page(s) 12

[2] Norman Ramsey, *Noweb*, http://www.cs.tufts.edu/~nr/noweb/ cited page(s) 12

[3] Yoann Padioleau, *Syncweb, literate programming meets unison*, http://padator.org/software/project-syncweb/readme.txt cited page(s) 12

[4] Hannes Magnusson et al, *PHP Manual*, http://php.net/manual/en/index.php cited page(s)

[5] Alfred Aho et al, *Compilers, Principles, Techniques, and tools*, http://en.wikipedia.org/wiki/Dragon_Book_(computer_science) cited page(s) 74

[6] Andrew Appel, *Modern Compilers in ML*, Cambridge University Press cited page(s) 74

[7] Yoann Padioleau, *Commons Pad OCaml Library*, http://padator.org/docs/Commons.pdf cited page(s) 16

[8] Yoann Padioleau, *OCamltarzan, code generation with and without camlp4*, http://padator.org/ocaml/ocamltarzan-0.1.tgz cited page(s) 66

[9] Eric Gamma et al, *Design Patterns*, Addison-Wesley cited page(s) 18

[10] Peter Norvig, *Design Patterns in Dynamic Programming*, http://norvig.com/design-patterns/ cited page(s) 18

[11] Yoann Padioleau, Julia Lawall, Gilles Muller, Rene Rydhof Hansen, *Documenting and Automating Collateral Evolutions in Linux Device Drivers* Eurosys 2008 cited page(s) 8

[12] *Coccinelle: A Program Matching and Transformation Tool for Systems Code*, http://coccinelle.lip6.fr/ cited page(s) 8

[12] George Necula, *CIL*, CC. http://manju.cs.berkeley.edu/cil/ cited page(s)