Semgrep: The Program
version 0.3

Yoann Padioleau
pad@returntocorp.com

with code from
Yoann Padioleau

September 8, 2021

Contents

Chapter 1

Introduction

The goal of this document is to explain with full details the source code of Semgrep (and especially the core of Semgrep).

1.1 Motivations

As R2C recruits more and more program analysis people, and as more and more developers want to contribute to the Semgrep codebase, it becomes useful to document the internals of Semgrep so people can quickly get up to speed. The alternative is me (the author), having to explain again and again the same things to many people.

Another motivation is the bus[1]. I wrote most of the code in Semgrep, without much code reviews, so this is potentially dangerous for R2C. The fact that most of the code is written in OCaml[2] makes things even worse, because very few people at R2C are familiar with this language. Thus, I think it is necessary to put extra effort in documenting Semgrep internals.

A final motivation, probably a bit selfish, is that I love *literate programming* [Knu92]. I use this technique extensively for my *Principia Softwarica* [Pad16] book series. Thus, this book is a great way to get feedback on the approach, so that I can in turn improve later Principia Softwarica.

1.2 Getting started

The first thing you need to contribute to Semgrep is to get the code of Semgrep[3].

I will not repeat the installation and build instructions contained in those repositories, but essentially you need first to install:

- OCaml, the wonderful always-amazing programming language
- OPAM, the OCaml package manager
- Dune, the OCaml build system

You can then use your favourite editor to edit the code (e.g., Emacs, or even better Efuns[4]). However, if you are an OCaml beginner, I strongly suggest to install VSCode and its OCaml Platform plugin. See `semgrep/CONTRIBUTING.md` for more information.

[1] https://en.wikipedia.org/wiki/Bus_factor
[2] https://ocaml.org/
[3] https://github.com/returntocorp/semgrep
[4] https://github.com/aryx/fork-efuns

1.3 Requirements

To understand Semgrep, you obviously need to know OCaml [LDF+16]. However, this document does not contain an OCaml tutorial and I assume you already have a basic knowledge of the language. Here are a few pointers to learn OCaml if you need to:

- *Real World OCaml* [?][5], the modern reference on the language and its practical use
- *Developing Applications With Objective Caml* [?][6], a bit outdated (it still uses the old "Objective Caml" name) but still a solid reference
- *OCaml Tutorials*[7], a good list of tutorials on different aspects of the language

Moreover, even though I do not assume you have an extensive program analysis knowledge, I assume you already know what is a lexer, a parser, an abstract syntax tree, or what is the difference between an expression and a statement. To learn more about basic program analysis, read for example *Modern Compiler in ML* [App04].

1.4 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.5 Copyright

Most of the code in this document is licensed under the GNU Lesser General Public License version 2.1 as published by the Free Software Foundation.

⟨*pad/r2c copyright* 11⟩≡ (459b 458 454d 453b 451c 447b 442 428b 427b 426b 418b 416b 411 407f 406 405b 393b 392b 391 390 389 387 38

```
(* Yoann Padioleau
 *
 * Copyright (C) 2019-2021 r2c
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * version 2.1 as published by the Free Software Foundation, with the
 * special exception on linking described in file license.txt.
 *
 * This library 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.
 *)
```

1.6 Acknowledgments

Many of the ideas in Semgrep originated in the Coccinelle[8] project, which I worked on many years ago with Julia Lawall and Gilles Muller. However, the goal of Semgrep is quite different from Coccinelle: Semgrep is focused

[5] https://realworldocaml.org/
[6] https://caml.inria.fr/pub/docs/oreilly-book/html/index.html
[7] https://ocaml.org/learn/tutorials/
[8] http://coccinelle.lip6.fr/

on finding bugs in code written in many different programming languages (e.g., Python, Javascript, Java, Go, C) while Coccinelle is focused on program transformations on C code (and mainly in the Linux kernel). This has led to many different design decisions on the shape of the Semgrep pattern language.

Moreover, many of the recent features of Semgrep owe a lot to its early adopters: the R2C developers (e.g., Isaac Evans, Drew Dennison, Grayson Hardaway, Ulziibayar Otgonbaatar), as well as early users (e.g., Vasilii Ermilove, Ajin Abraham) whose feedbacks and feature requests greatly improved the Semgrep user interface.

Another incredible contribution to the development of Semgrep is its web interface, http://semgrep.dev/editor, by Drew Dennison, which accelerated the adoption of Semgrep and greatly facilitated bug reports and interactions with users.

Chapter 2

Overview

Before showing the source code of Semgrep in the following chapters, I first give an overview in this chapter of the main features of Semgrep. I also quickly describe its command-line interface and the format of its configuration file. Finally, I define terms, explain how the code is organized, and more generally give the background necessary to understand the code I will show later.

2.1 Semgrep main features

The goal of this section is just to give a quick reminder of Semgrep main features, so that I can define terms that I will reference very often later. This section is not a tutorial on Semgrep; see instead https://semgrep.dev/learn for a real tutorial.

Semgrep is based mainly on six principles/features, as explained in the six following sections.

2.1.1 Concrete code syntax matching

Semgrep allows you to find code by specifying *patterns*. Those patterns are then *matched* on some code, to find occurences of the pattern in this code. I call those occurences *matching results*. For example, let's say you want to find all calls to `foo()`. The beauty of Semgrep is that you do not need to know how a function call is represented internally in memory (like with most other SAST tools) to specify this pattern. You can just use the syntax you already know, as shown in the example below:

```
$ semgrep -e 'foo()' foo.py
foo.py:10: return foo(  )
```

Semgrep operates internally on the *Abstract Syntax Tree* (AST), but it lets you use the *concrete code syntax* to specify the pattern. Note that in the previous example, even though there was no space between the parenthesis in the `foo()` pattern, the pattern still matches code with extra spaces between the parenthesis because the engine does the matching on the AST, not on the sequences of characters in the file.

2.1.2 Metavariables

If you only had the ability to specify concrete code in patterns, your pattern would be very specific, and would likely match only a few code sites. *Metavariables* let you abstract away certain parts to generalize your pattern. For example, here is a query to find all calls to `foo` with two arguments (whatever they are):

```
$ semgrep -e 'foo($X, $Y)' foo.py
foo.py:20: return foo(1,2)
```

Note that if you use multiple times the same metavariable in a pattern, Semgrep will ensure that the different places where the metavariable matches contain all the same code. For example, here is a query to find all calls to `foo` where the two arguments are equal:

```
$ semgrep -e 'foo($X, $X)' foo.py
foo.py:30: foo(a+b, a + b)
```

Note again that even though the second argument to `foo` abovehas extra spaces around the addition, it is still considered equal to the first argument because the equality test is also performed on the AST, not on the sequence of characters.

2.1.3 Ellipsis

To abstract away some code, especially sequences, without even giving it a name, Semgrep provides the *ellipsis operator*: "`...`" (sometimes called also the three-dots operator). This operator is quite versatile and can be used in different context to match different kinds of sequences (arguments, parameters, statements, but also characters), as shown in the examples below:

```
$ semgrep -e 'foo(1, 2, ...)' foo.py
foo.py:50: return foo(1, 2, 3, 4 , 5)
$ semgrep -e 'foo("...")' foo.py
foo.py:40: return foo("a long string")
```

2.1.4 Unspecified matches more

Another core principle of Semgrep, that is hard to give a name to, is that even if your pattern does not specify certain things, for example the attributes of a method, or the type of a parameter, it will still match code where those things are specified. More concretely, the pattern `String $METHOD() { ... }` will match the two methods below, even though one of the methods contains an attribute (`@Test`) that was not mentioned in the pattern.

⟨*Foo.java* 14⟩≡

```
class Foo extends Bar {

  @Test
  String foo() { return 1 == 1; }

  String bar() { return False; }
}
```

In the same way, `class $CLASS { ... }` will also match the class above, even though the pattern does not specify the inheritance relation to `Bar`. This is convenient because it allows to write simple patterns that can accomodate many code variations.

Patterns where certain things are left unspecified will match code not containing those things, *and* code containing those things. In short, *unspecified matches more*. However, this also means that there is no way with a single pattern to look for the absence of things. For example, you can not look specifically for a method that does not have attributes (e.g., `bar` above). However, Semgrep provides also boolean logic operations on patterns allowing to combine multiple patterns to perform powerful search, as explained below in Section 2.1.6. Thanks to conjunctions and negations, it is possible to look for the absence of things by using multiple patterns.

2.1.5 Code equivalences and semantic matching

The last three features can be seen as a mean to generalize patterns so that they can accomodate certain code variations. However, there are many code variations that those features can not handle. For example, in Python you can call the same function with keyword arguments in different orders as shown below.

⟨*Foo.py* 15⟩≡
```
foo(kwd1 = 1, kwd2 = 2)
foo(kwd2 = 2, kwd1 = 1)
```

Both lines are *equivalent*, but a pattern like `foo(kwd1 = $X, kwd2 = $Y)` would naively just match the first line if Semgrep was just doing syntactical matching.

In the same way, many programming-language constructs are just syntactic sugar on top of other constructs. For example `a != a` is really the same than `!(a == a)` in most languages[1]. However, a pattern like `$X == $X` would naively not match code like `a != a` if Semgrep was just doing syntactical matching, but it should.

In fact, in programming "there is more than one way to do it" (abbreviated as TMTOWTDI[2] in Perl). However, we do not want to specify all those different ways when writing a pattern. This is why one of the core principle of Semgrep is to accomodate as many code variations as possible by leveraging *code equivalences*. Semgrep does not perform just syntactical matching but also *semantic matching* (hence the name semantic grep) because it knows about the semantic of the underlying programming languages.

Semgrep supports two kinds of code equivalences, as explained in the following sections.

Builtin equivalences

The first set of code equivalences are built into the Semgrep engine, for example the equivalence about keyword arguments presented above. There are many *builtin* equivalences, including sophisticated ones such as the one using a constant propagation analysis as explained in Section 9.3, or the one resolving import aliases as explained in Section 9.4.

User-defined equivalences

The other set of code equivalences can actually be specified by the user in a special file called the *equivalence file*. For example, by writing `$X + $Y <=> $Y + $X` in the equivalence file, we teach the Semgrep engine the commutativity of the addition so that searching for the pattern `1 + $X` would also match code written as `a + 1`. See Chapter 8 for more information on user-defined equivalences.

2.1.6 Boolean logic operations on patterns

2.2 Semgrep command-line interface

2.3 A simple Semgrep pattern

2.4 A simple Semgrep configuration file

2.5 Code organization

The code of Semgrep is split in two GitHub repositories: `pfff`[3] and `semgrep`[4]. `pfff` is now a submodule of `semgrep`. Table 2.1 presents short descriptions of the main source files in `pfff` and `semgrep` and the correspond-

[1] This is actually not always true for Python.
[2] https://en.wikipedia.org/wiki/TMTOWTDI
[3] https://github.com/returntocorp/pfff
[4] https://github.com/returntocorp/semgrep

ing chapters in this document in which the code contained in the file is primarily discussed.

`pfff` contains mainly parsers for different programming languages (e.g., PHP, Python, Javascript, Go, Java, C). Pfff stands for PHP Frontend For Fun. The reason is that I started Pfff while working at Facebook and its website was mostly written in PHP. In 2009, I wanted to statically find bugs in Facebook codebase and I first needed to parse this PHP codebase, hence the name. Later on, I was asked to also analyze our Java code, as well as our Javascript code. Thus, gradually Pfff evolved in a set of parsers for different programming languages, but the original name sticked.

`pfff` now also contains the *generic Abstract Syntax Tree* (generic AST) in `pfff/h_program-lang/AST_generic.ml` which is a uniform representation of a program that all the parsers are converting into (see Section 3.3 for more information). That way, once we write a matching algorithm on the generic AST, it can automatically be used to match code in any of the programming languages supported by Pfff.

`semgrep`, and especially `semgrep/semgrep-core`, contains the core engine behind Semgrep. In this document, I will just use the prefix path `semgrep/` to refer to files actually under `semgrep/semgrep-core/`, and the prefix `pfff/` for files under `semgrep/semgrep-core/pfff`.

2.6 Software architecture

Figure 2.1 describes the main data flow of Semgrep, whereas Figure 2.2 describes the main control flow of Semgrep. Both figures can be decomposed in three parts:

1. *Parsing source code* on the left, where the goal is to translate input programs in generic ASTs programs (`AST_generic.program`[32e])
2. *Parsing rules* containing patterns on the right, where patterns are ultimately translated in generic AST fragments (`AST_generic.any`[32i]) in which certain nodes correspond to special Semgrep features (e.g., ellipsis, metavariables)
3. *Visit* the generic AST programs while *matching* the generic AST fragments on it and return matching results, at the bottom of both figures

Starting from the top of Figure 2.2, the function `Main_semgrep_core.main()`[40c], after some basic command-line processing, calls `Main_semgrep_core.semgrep_with_rules()`[44a] with the name of the file containing the rules (e.g., `semgrep_rules.yaml` in Figure 2.1), a list of files or directories to process (e.g., `foo.py` in Figure 2.1), and the programming language to use (e.g., Python). `Main_semgrep_core.semgrep_with_rules()` first calls `Parse_rules.parse()`[56b] to parse the YAML configuration file containing the rules, and calls optionally `Parse_equivalences.parse()`[58d] to parse a YAML equivalence file. Both functions internally call `Parse_generic.parse_pattern()`[55c] to parse the patterns contained in the configuration or equivalence files. Depending on the language of the pattern (either passed via the `-lang` command-line parameter for interactive `-e` patterns, or selected automatically as the first language in the `languages` field in a rule), `Parse_generic.parse_patt` dispatches the appropriate parsing function in `pfff` (e.g., `Parse_python.any_of_string()`[139c] if it's a Python pattern) and converts the programming-language specific AST fragments (e.g., `AST_python.any`[132h]) in a generic AST fragment (`AST_generic.any`) by calling the associated conversion function (e.g., `Python_to_generic.any()`[161c]).

At this point, `Main_semgrep_core.semgrep_with_rules()` has a list of parsed rules containing parsed patterns (generic AST fragments). It can now switch to processing the source files. First, it calls `Lang.files_of_dirs_or` to get the final list of files to process. Indeed, `semgrep` can accept directories as arguments, in which case it recursively traverses those directories and their subdirectories to discover new files to process (note that `Lang.files_of_dirs_or_files()` internally calls the external program `find` to traverse directories). Once we get a flat list of files, `Main_semgrep_core.semgrep_with_rules()` iterates over all those files and calls `Parse_generic.parse_program()`[54h] on each of those files. Depending on the language, this function dispatches the appropriate parsing function in `pfff` (e.g., `Parse_python.parse_program()`[137b] if the source code is

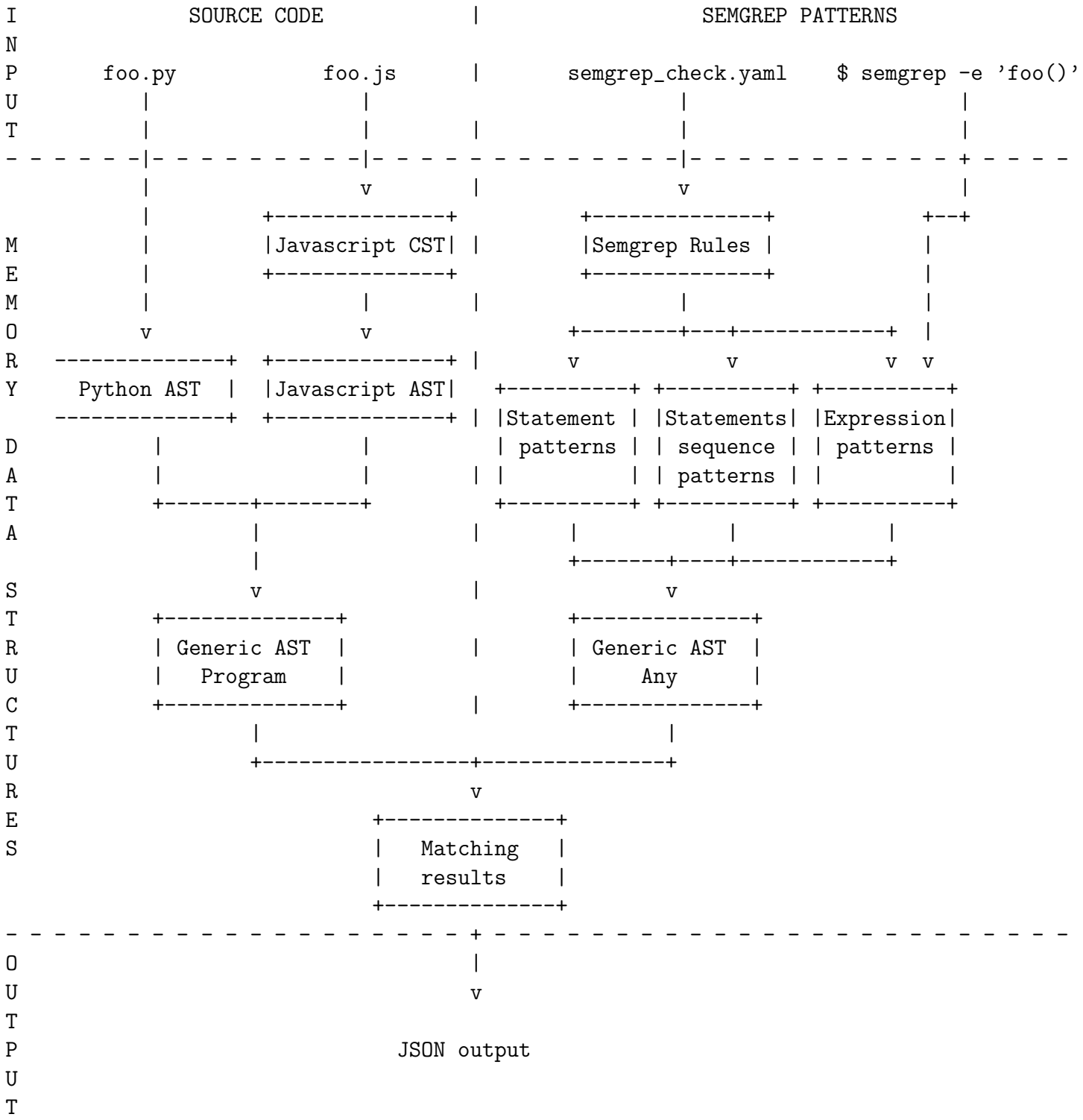

Figure 2.1: Data flow diagram of Semgrep

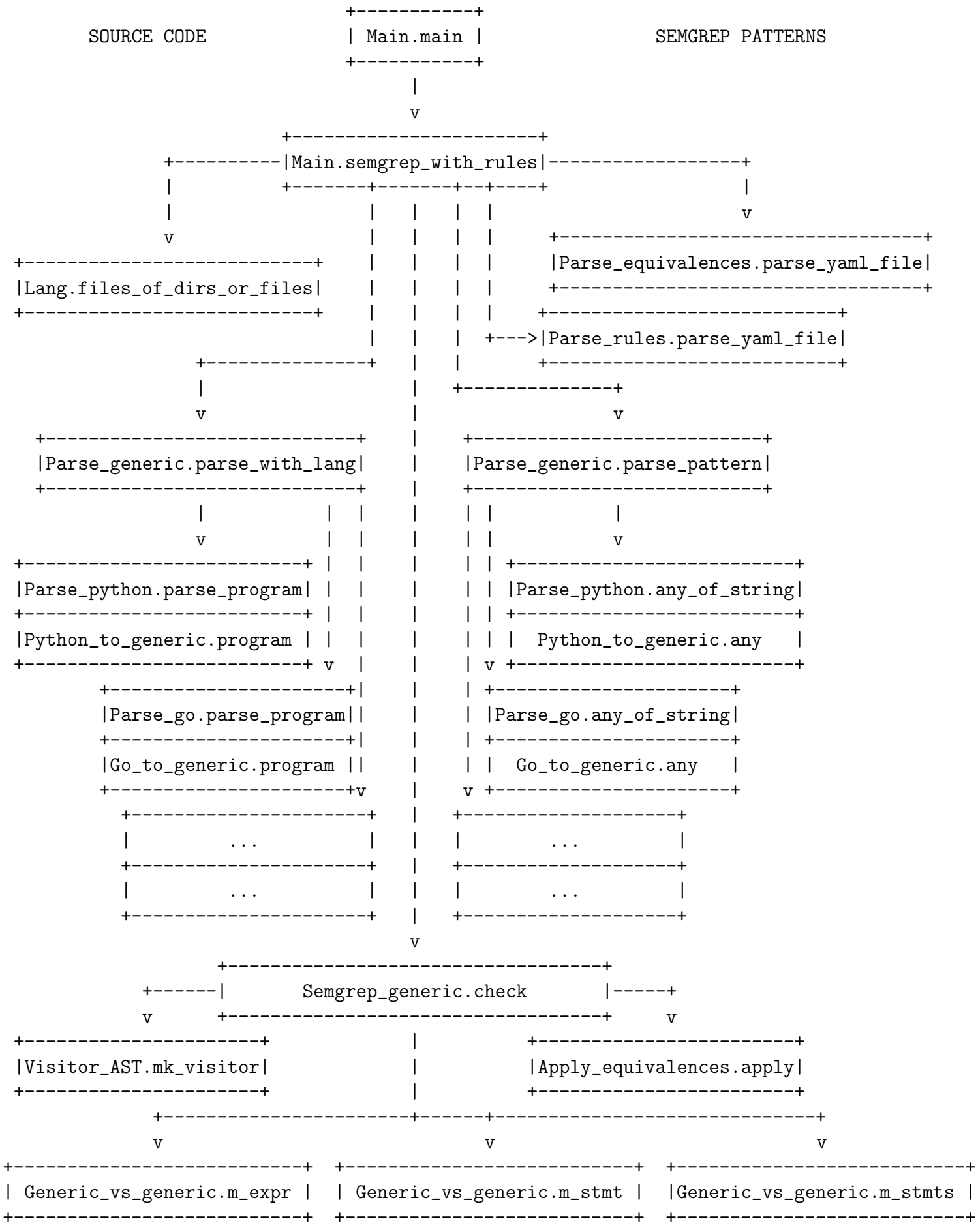

Figure 2.2: Control flow diagram of Semgrep

in Python), and converts the programming-language specific AST programs (e.g., `AST_python.program`[132g]) in a generic AST program (`AST_generic.program`) by calling the associated conversion function (e.g., `Python_to_generi`

At this point, we are not dealing any more with filenames (source code or Semgrep configuration files) but with generic ASTs data structures in memory. Now, `Main_semgrep_core.semgrep_with_rules()` can call the most important function in Semgrep, `Semgrep_generic.check()`[61a], at the bottom of Figure 2.2. This function matches a list of patterns (contained in rules) against a single generic AST program. `Semgrep_generic.check()` first calls `Apply_equivalences.apply()`[103e] to find if it can apply user-defined equivalences on the patterns. This results in a new set of patterns containing alternative choices when an equivalence was found (see Chapter 8 for more information). Then, it calls `Visitor_AST.mk_visitor()`[157b] to build a visitor[5] to explore the generic AST (see Section 17.2 for more information on how to write visitors with Pfff). At this point, the set of patterns are split in three different categories as shown on the right in Figure 2.1: *expression* patterns (e.g., `foo($X) + $Y`), *statement* patterns (e.g., `if($X) return $Y;`), and *sequences of statements* patterns (e.g., `foo($X); ... bar($X);`). For each category, and for each node in the generic AST program, `Semgrep_generic.check()` calls the appropriate visitor (e.g., `Visitor_AST.visitor_in.kexpr()`[239b] for expression patterns), and the appropriate matching function to call (e.g., `Generic_vs_generic.m_expr()`[70b]). This matching function will compare the constructions in the generic AST at the visited node (e.g., an expression), with all the patterns in the appropriate category (e.g., all expression patterns). If one of those patterns matches the current node, a matching result is recorded at this node and the visitor continues to visit the children of this node. Once the visitor finished, `Semgrep_generic.check()` returns to the caller `Main_semgrep_core.semgrep_with_rules()` the list of matching results. This last function can finally reports those results, for example in a JSON format on the standard output.

2.7 Book structure

You now have enough background to understand the source code of Semgrep. The rest of the book is divided in two parts.

In the first part, I will explain the essential code to understand the Semgrep engine. This part is mostly programming-language independent (most of the code deals with the generic AST). I will focus on the program analysis that are really specific to Semgrep (e.g., the generic AST, metavariable matching).

Semgrep relies also on many classic program analysis components: lexers, parsers, ASTs, AST visitors, scope, type, and even dataflow analysis. All of those components are explained later in the second part. However, there are too many lexers, parsers, or ASTs in `pfff` to present (e.g., for Python, Go, Java). Moreover, they have a lot in common, so it would be useless to present them all. It is useful though, to make things concrete, to at least see how one programming-language is handled by Semgrep and Pfff end to end, and especially how Semgrep converts this language to the generic AST. This is why a few chapters in the second part will use a concrete programming language. I chose Python because it is one of the simplest language to explain (it has the smallest grammar and AST definition) and because it is one of the most well-known programming languages among developers (especially inside R2C).

[5] a well known design pattern [**?**], very often used in linter programs (e.g., Flake8, ESLint).

Function	Chapter	File or Directory	LOC
tokens and positions	3.1	`pfff/h_program-lang/Parse_info.ml`	
programming language list	3.2	`pfff/lang_GENERIC/.../Lang.ml`	
generic AST (essentials)	3.3	`pfff/h_program-lang/AST_generic.ml`	
semgrep pattern	3.4	`semgrep/core/Pattern.ml`	
semgrep rule data structure	3.5	`semgrep/core/Rule.ml`	
user-defined code equivalences	3.6	`semgrep/core/Equivalence.ml`	
matching results data structure	3.7	`semgrep/core/Matching_result.ml`	
Pfff entry point	4.1	`pfff/cli/Main.ml`	
Semgrep entry point	4.2	`semgrep/bin/Main.ml`	
parsing (e.g., YAML files)	5	`semgrep/parsing/`	
main algorithm	6.1	`semgrep/matching/Semgrep_generic.ml`	
matching combinators	6.3	`semgrep/matching/Matching_generic.ml`	
matching AST versus AST	7	`semgrep/matching/Generic_vs_generic.ml`	
applying equivalences	8	`semgrep/matching/Apply_equivalences.ml`	
reporting matches	10	`semgrep/reporting/`	
Python AST	14	`pfff/lang_python/.../AST_python.ml`	
Python (ocamllex) lexer	15.1	`pfff/lang_python/.../Lexer_python.mll`	
Python (ocamlyacc) grammar	15.3	`pfff/lang_python/.../Parser_python.mly`	
Python parser	15.5	`pfff/lang_python/.../Parse_python.ml`	
generic AST (full list)	16	`pfff/h_program-lang/AST_generic.ml`	
generic AST dumper	17.1	`pfff/lang_GENERIC/.../Meta_AST.ml`	
generic AST visitor	17.2	`pfff/lang_GENERIC/.../Visitor_AST.ml`	
Python AST to generic AST	18	`pfff/lang_python/.../Python_to_generic.ml`	
managing names and scopes	19	`pfff/lang_GENERIC/.../Naming_AST.ml`	
managing types	20	`semgrep/typing/`	
intermediate language (IL)	21	`pfff/lang_GENERIC/.../IL.ml`	
converting generic AST to IL	22	`pfff/lang_GENERIC/.../AST_to_IL.ml`	
control flow graph (CFG)	23	`pfff/lang_GENERIC/.../CFG.ml`	
converting IL to CFG	24	`pfff/lang_GENERIC/.../CFG_build.ml`	
dataflow analysis library	25	`pfff/lang_GENERIC/.../Dataflow.ml`	
core tainting analysis	26	`pfff/lang_GENERIC/.../Dataflow_tainting.ml`	
tainting rules data structure	27.1	`semgrep/core/Tainting_rules.ml`	
tainting check in semgrep	27.2	`semgrep/tainting/`	
utility functions	A.1	`pfff/commons/Common.mli`	
dumper library	B.2	`pfff/commons/OCaml.mli`	
semgrep regression testing	C.5	`semgrep/tests/Test.ml`	
Total			15000

Table 2.1: Chapters and source files of Semgrep.

Part I

Semgrep Essentials

Chapter 3

Core Data Structures

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Fred Brooks

3.1 The tokens

⟨*type* `Parse_info.token_location` 22a⟩≡ (252k)
```
(* to report errors, regular position information *)
type token_location = {
  str: string; (* the content of the "token" *)
  charpos: int; (* byte position, 0-based *)
  line: int; (* 1-based *)
  column: int; (* 0-based *)
  file: Common.filename;
}
```

⟨*type* `Parse_info.t` 22b⟩≡ (252k)
```
(* Shortcut.
 * Technically speaking this is not a token, because we do not have
 * the kind of the token (e.g., PLUS | IDENT | IF | ...).
 * It's just a lexeme, but the word lexeme is not as known as token.
*)
type t = token_mutable
```

⟨*signature* `Parse_info.token_location_of_info` 22c⟩≡ (252k)
```
val token_location_of_info: t -> token_location
```

⟨*signature* `Parse_info.str_of_info` 22d⟩≡ (252k)
```
val str_of_info   : t -> string
```

⟨*signature* `Parse_info.line_of_info` 22e⟩≡ (252k)
```
val line_of_info  : t -> int
```

⟨*signature* `Parse_info.col_of_info` 22f⟩≡ (252k)
```
val col_of_info   : t -> int
```

⟨*signature* `Parse_info.pos_of_info` 22g⟩≡ (252k)
```
val pos_of_info   : t -> int
```

⟨*signature* `Parse_info.file_of_info` 22h⟩≡ (252k)
```
val file_of_info  : t -> Common.filename
```

⟨*type* Parse_info.info_ 23a⟩≡ (252k)
```
(* deprecated *)
type info_ = t
```

3.2 The programming-language specific Abstract Syntax Trees (ASTs)

⟨*type* Lang.t 23b⟩≡ (260c 257e)
```
type t =
  | Python
  ⟨Lang.t extra Python cases 141b⟩
  (* system *)
  | C
  | Cplusplus
  | Rust
  (* mainstream with Gc *)
  | Javascript
  | Typescript
  | Java
  | Kotlin
  | Csharp
  | Go
  (* functional *)
  | OCaml
  (* scripting (Python is above) *)
  | Ruby
  | PHP
  | Lua
  (* data science *)
  | R
  (* config files *)
  | JSON
  | Yaml
```

⟨*signature* Lang.lang_of_string_opt 23c⟩≡ (257e)
```
val lang_of_string_opt : string -> t option
```

⟨*signature* Lang.files_of_dirs_or_files 23d⟩≡ (257e)
```
val files_of_dirs_or_files : t -> Common.path list -> Common.filename list
```

⟨*signature* Lang.langs_of_filename 23e⟩≡ (257e)
```
val langs_of_filename : Common.filename -> t list
```

3.3 The generic AST

3.3.1 Positions and tokens

⟨*type* AST_generic.tok 23f⟩≡ (263c)
```
(* Contains among other things the position of the token through
 * the Parse_info.token_location embedded inside it, as well as the
 * transformation field that makes possible spatch on the code.
 *)
type tok = Parse_info.t [@@deriving show]
```

⟨*type* AST_generic.wrap 24a⟩≡ (263c)
```
(* a shortcut to annotate some information with position information *)
type 'a wrap = 'a * tok
```

3.3.2 Identifiers

⟨*type* AST_generic.ident 24b⟩≡ (263c)
```
type ident = string wrap
```

⟨*constant* AST_generic.str_of_ident 24c⟩≡ (274)
```
let str_of_ident = fst
```

3.3.3 Expressions

⟨*type* AST_generic.expr 24d⟩≡ (263c)
```
(* todo? we could do like for stmt and have 'expr' and 'expr_kind', which
 * would allow us to store more semantic information at each expr node,
 * e.g., type information, or constant evaluation, or range, but it
 * would be a bigger refactoring than for stmt.
 *)
and expr =
  (* basic (atomic) values *)
  | L of literal
  (* composite values *)
  | Container of container_operator * expr list bracket
  ⟨AST_generic.expr other composite cases 25c⟩
  | N of name
  ⟨AST_generic.expr other identifier cases 143e⟩

  (* operators and function application *)
  | Call of expr * arguments bracket (* can be fake '()' for OCaml/Ruby *)
  ⟨AST_generic.expr other call cases 143h⟩

  (* The left part should be an lvalue (Id, DotAccess, ArrayAccess, Deref)
   * but it can also be a pattern (Tuple, Container, even Record), but
   * you should really use LetPattern for that.
   * Assign can also be abused to declare new variables, but you should use
   * variable_definition for that.
   * less: should be in stmt, but most languages allow this at expr level :(
   * todo: see IL.ml where we normalize this AST with expr/instr/stmt
   * update: should even be in a separate simple_stmt, as in Go
   *)
  | Assign of
      expr * tok (* '=', '<-' in OCaml. ':=' Go is AssignOp (Eq) *) * expr
  ⟨AST_generic.expr other assign cases 143g⟩

  (* can be used for Record, Class, or Module access depending on expr.
   * In the last case it should be rewritten as a (N IdQualified) with a
   * qualifier though.
   *)
  | DotAccess of expr * tok (* ., ::, ->, # *) * name_or_dynamic
  ⟨AST_generic.expr array access cases 26d⟩
  ⟨AST_generic.expr anonymous entity cases 143d⟩
  ⟨AST_generic.expr other cases 144a⟩

  ⟨AST_generic.expr semgrep extensions cases 33b⟩
  (* for ellipsis in method chaining *)
```

```
    | DotAccessEllipsis of expr * tok (* '...' *)
    ⟨AST_generic.expr OtherXxx case 147b⟩
```

⟨*type* AST_generic.literal 25a⟩≡ (263c)
```
  and literal =
    | Bool of bool wrap
    (* the numbers are an option because OCaml numbers (e.g., 63bits int)
     * may not be able to represent all numbers.
     *)
    | Int of int option wrap
    | Float of float option wrap
    | Char of string wrap
    | String of string wrap
    | Regexp of string wrap
    | Unit of tok
    (* a.k.a Void *)
    | Null of tok
    | Undefined of tok (* JS *)
    | Imag of string wrap
    (* Go, Python *)
    | Ratio of string wrap (* Ruby *)
    | Atom of string wrap

  (* Ruby *)
```

⟨*type* AST_generic.container_operator 25b⟩≡ (263c)
```
  and container_operator =
    (* Tuple was lifted up *)
    | Array (* todo? designator? use ArrayAccess for designator? *)
    | List
    | Set
    (* TODO? merge with Record *)
    | Dict

  (* a.k.a Hash or Map (combine with Tuple to get Key/value pair) *)
```

⟨AST_generic.expr *other composite cases* 25c⟩≡ (24d) 143b ▷
```
  (* special case of Container, at least 2 elements (except for Python where
   * you can actually have 1-uple, e.g., '(1,)' *)
  | Tuple of expr list bracket
```

⟨*type* AST_generic.bracket 25d⟩≡ (263c)
```
  (* Use for round(), square[], curly{}, and angle<> brackets.
   * note: in theory we should not care about those tokens in an AST,
   * but they are useful to report correct ranges in sgrep when we match
   * something that can just be those brackets (e.g., an empty container).
   *)
  type 'a bracket = tok * 'a * tok
```

⟨*function* AST_generic.unbracket 25e⟩≡ (263c)
```
  let unbracket (_, x, _) = x
```

⟨*type* AST_generic.arguments 25f⟩≡ (263c)
```
  and arguments = argument list
```

⟨*type* AST_generic.argument 26a⟩≡ (263c)
```
and argument =
  (* regular argument *)
  | Arg of expr (* can be Call (IdSpecial Spread, Id foo) *)
  ⟨AST_generic.argument other cases 26b⟩
  ⟨AST_generic.argument OtherXxx case 146j⟩
```

⟨AST_generic.argument *other cases* 26b⟩≡ (26a) 146i▷
```
(* keyword argument *)
| ArgKwd of ident * expr
```

⟨*type* AST_generic.field_ident 26c⟩≡ (263c)
```
⟨AST_generic.field_ident other cases 146c⟩
```

⟨AST_generic.expr *array access cases* 26d⟩≡ (24d) 143i▷
```
(* in Js ArrayAccess is also abused to perform DotAccess (..., FDynamic) *)
| ArrayAccess of expr * expr bracket
```

3.3.4 Statements

⟨*type* AST_generic.stmt 26e⟩≡ (263c)
```
and stmt = {
  s : stmt_kind;
      [@equal AST_utils.equal_stmt_field_s equal_stmt_kind] [@hash.ignore]
  (* this can be used to compare and hash more efficiently stmts,
     or in semgrep to quickly know if a stmt is a children of another stmt.
  *)
  s_id : AST_utils.Node_ID.t; [@equal AST_utils.equal_stmt_field_s_id]
  (* todo? we could store a range: (tok * tok) to delimit the range of a stmt
   * which would allow us to remove some of the extra 'tok' in stmt_kind.
   * Indeed, the main use of those 'tok' is to accurately report a match range
   * in semgrep.
   *)
  mutable s_use_cache : bool; [@equal fun _a _b -> true] [@hash.ignore]
  (* whether this is a strategic point for match result caching.
     This field is relevant for patterns only.

     This applies to the caching optimization, in which the results of
     matching lists of statements can be cached. A list of statements
     is identified by its leading node. In the current implementation,
     the fields 's_id', 's_use_caching', and 's_backrefs' are treated as
     properties of a (non-empty) list of statements, rather than of individual
     statements. A cleaner implementation would consist of a custom
     list type in which each list has these properties, including the
     empty list.
  *)
  mutable s_backrefs : AST_utils.String_set.t option;
      [@equal fun _a _b -> true] [@hash.ignore]
  (* set of metavariables referenced in the "rest of the pattern", as
     determined by matching order.
     This field is relevant for patterns only.

     This is used to determine which of the bound
     metavariables should be added to the cache key for this node.
     This field is set on pattern ASTs only, in a pass right after parsing
     and before matching.
  *)
```

```
    (* used in semgrep to skip some AST matching *)
    mutable s_bf : Bloom_filter.t option; [@equal fun _a _b -> true] [@hash.ignore]
  }

  and stmt_kind =
    (* See also IL.ml where Call/Assign/Seq are not in expr and where there are
     * separate expr, instr, and stmt types *)
    | ExprStmt of expr * sc (* fake tok in Python, but also in JS/Go with ASI *)
    (* newscope: in C++/Java/Go *)
    | Block of stmt list bracket (* can be fake {} in Python where use layout *)
    (* EmptyStmt = Block [], or separate so can not be matched by $S? $ *)

    (* newscope: for vardef in expr in C++/Go/... *)
    | If of tok (* 'if' or 'elif' *) * expr * stmt * stmt option
    | While of tok * expr * stmt
    | Return of tok * expr option * sc
    ⟨AST_generic.stmt other cases 148a⟩
    ⟨AST_generic.stmt toplevel and nested construct cases 32g⟩
    ⟨AST_generic.stmt semgrep extensions cases 102f⟩
    ⟨AST_generic.stmt OtherXxx case 150b⟩
```

3.3.5 Definitions

⟨*type* AST_generic.definition 27a⟩≡ (263c)
```
  and definition = entity * definition_kind
```

⟨*type* AST_generic.entity 27b⟩≡ (263c)
```
  and entity = {
    (* In Ruby you can define a class with a qualified name as in
     * class A::B::C, and even dynamically.
     * In C++ you can define a method with a class qualifier outside a class,
     * hence the use of name_or_dynamic below and not just ident.
     *)
    name : name_or_dynamic;
    ⟨AST_generic.entity attribute field 30d⟩
    ⟨AST_generic.entity id info field 27c⟩
    ⟨AST_generic.entity other fields 152c⟩
    tparams : type_parameter list;
  }
```

⟨AST_generic.entity *id info field* 27c⟩≡ (27b)

⟨*function* AST_generic.basic_entity 27d⟩≡ (263c)
```
  let basic_entity id attrs =
    let idinfo = empty_id_info () in
    { name = EN (Id (id, idinfo)); attrs; tparams = [] }
```

⟨*type* AST_generic.definition_kind 27e⟩≡ (263c)
```
  and definition_kind =
    (* newvar: can be used also for methods, nested functions, lambdas.
     * note: can have empty "body" when the def is actually a declaration
     * in a header file (called a prototype in C).
     *)
    | FuncDef of function_definition
    (* newvar: can be used also for constants.
     * can contain special_multivardef_pattern ident in which case vinit
```

```
     * is the pattern assignment.
     *)
    | VarDef of variable_definition
    (* FieldDefColon can be used only inside a record (in a FieldStmt).
     * This used to be merged with VarDef, but in semgrep we don't want
     * a VarDef to match a field definition for certain languages
     * (e.g., JS, OCaml), and we definitely don't want the
     * vardef_to_assign equivalence to be used on FieldDefColon.
     * TODO? maybe merge back with VarDef but add a field in
     *  variable_definition saying whether it's using a colon syntax?
     * TODO? merge instead JS objects with Containers?
     *
     * Note that we could have used a FieldVar in the field type instead
     * of this FieldDef here, which would be more precise, but
     * this complicates things in semgrep where it's convenient to have
     * a uniform FieldStmt(DefStmt) that covers field and methods
     * (see m_list__m_field in semgrep).
     * Note that FieldDefColon where vinit is a Lambda instead be converted
     * in a FuncDef!
     *)
    | FieldDefColon of (* todo: tok (*':'*) * *) variable_definition
    | ClassDef of class_definition
    ⟨AST_generic.definition_kind other cases 152d⟩
  and other_def_operator = OD_Todo
```

Functions

⟨*type* AST_generic.function_definition 28a⟩≡ (263c)

```
  and function_definition = {
    fkind : function_kind wrap;
    fparams : parameters;
    frettype : type_ option;
    (* return type *)
    (* newscope:
     * note: can be empty statement for methods in interfaces.
     * update: can also be empty when used in a Partial.
     * can be simple expr too for JS lambdas, so maybe fbody type?
     * FExpr | FNothing | FBlock ?
     * use stmt list bracket instead?
     *)
    fbody : stmt;
  }
```

⟨*type* AST_generic.parameters 28b⟩≡ (263c)

```
  and parameters = parameter list
```

⟨*type* AST_generic.parameter 28c⟩≡ (263c)

```
  (* newvar: *)
  and parameter =
    | ParamClassic of parameter_classic
    ⟨AST_generic.parameter other cases 151f⟩
    (* Both those ParamXxx used to be handled as a ParamClassic with special
     * VariadicXxx attribute in p_attr, but they are used in so many
     * languages that it's better to move then in a separate type.
     * We could do a ParamXxx of tok * ident, but some of those params
     * may have attribute, they need a id_info, so simpler to reuse
```

```
     * parameter_classic, but pname is always a Some (except for Ruby).
     * ParamRest could be called ParamSpread.
     * alt: move that in ParamPattern instead.
     *)
    | ParamRest of tok (* '...' in JS, '*' in Python *) * parameter_classic
    | ParamHashSplat of tok (* '**' in Python *) * parameter_classic
    ⟨AST_generic.parameter semgrep extension cases 33c⟩
    ⟨AST_generic.parameter OtherXxx case 153d⟩
```

⟨*type* AST_generic.parameter_classic 29a⟩≡ (263c)

```
  and parameter_classic = {
    (* alt: use a 'ParamNoIdent of type_' when pname is None instead? *)
    pname : ident option;
    ptype : type_ option;
    pdefault : expr option;
    ⟨AST_generic.parameter_classic attribute field 30e⟩
    ⟨AST_generic.parameter_classic id info field 29b⟩
  }
```

⟨AST_generic.parameter_classic *id info field* 29b⟩≡ (29a)

```
  (* naming *)
  pinfo : id_info;
      (* Always Param *)
```

Variables

⟨*type* AST_generic.variable_definition 29c⟩≡ (263c)

```
  and variable_definition = {
    (* todo? should remove vinit and transform a VarDef with init with a VarDef
     * followed by an Assign (possibly to Null). See vardef_to_assign().
     *)
    vinit : expr option;
    (* less: (tok * expr) option? *)
    vtype : type_ option;
  }
```

Classes

⟨*type* AST_generic.class_definition 29d⟩≡ (263c)

```
  and class_definition = {
    ckind : class_kind wrap;
    (* usually just one parent, and type_ should be a TyApply *)
    cextends : type_ list;
    (* class_kind in type_ must be Interface *)
    cimplements : type_ list;
    (* class_kind in type_ is usually a Trait *)
    cmixins : type_ list;
    (* PHP 'uses' *)

    (* for Java Record or Scala Classes; we could transpile them into fields *)
    cparams : parameters;
    (* newscope:
     * note: this can be an empty fake bracket when used in Partial.
     * TODO? use an option here?
     *)
    cbody : field list bracket;
  }
```

⟨*type* AST_generic.class_kind 30a⟩≡ (263c)
```
  (* invariant: this must remain a simple enum; Map_AST relies on it *)
  and class_kind =
    | Class
    | Interface
    | Trait
    (* Kotlin, Scala *)
    | Object
    (* Java 'record', Scala 'case class' *)
    | RecordClass
    (* java: *)
    | AtInterface

  (* @interface, a.k.a annotation type declaration *)
```

⟨*type* AST_generic.field 30b⟩≡ (263c)
```
  and field =
    | FieldStmt of stmt
    ⟨AST_generic.field other cases 154c⟩
```

3.3.6 Types

⟨*type* AST_generic.type_ 30c⟩≡ (263c)
```
  and type_ =
    (* todo? a type_builtin = TInt | TBool | ...? see Literal.
     * or just delete and use (TyN Id) instead?
     *)
    | TyBuiltin of string wrap (* int, bool, etc. could be TApply with no args *)
    (* old: was 'type_ list * type*' , but languages such as C and
     * Go allow also to name those parameters, and Go even allow ParamRest
     * parameters so we need at least 'type_ * attributes', at which point
     * it's better to just use parameter.
     *)
    | TyFun of parameter list * type_ (* return type *)
    (* a special case of TApply, also a special case of TPointer *)
    | TyArray of (* const_expr *) expr option bracket * type_
    | TyTuple of type_ list bracket
    ⟨AST_generic.type_ other cases 150c⟩
    ⟨AST_generic.type_ OtherXxx case 151a⟩
```

3.3.7 Attributes

⟨AST_generic.entity *attribute field* 30d⟩≡ (27b)
```
  attrs : attribute list;
```

⟨AST_generic.parameter_classic *attribute field* 30e⟩≡ (29a)
```
  pattrs : attribute list;
```

⟨*type* AST_generic.attribute 30f⟩≡ (263c)
```
  and attribute =
    | KeywordAttr of keyword_attribute wrap
    (* for general @annotations. *)
    | NamedAttr of tok (* @ *) * name * arguments bracket
    ⟨AST_generic.attribute OtherXxx case 154i⟩
```

⟨*type* AST_generic.keyword_attribute 31a⟩≡ (263c)
```
  and keyword_attribute =
    | Static
    | Volatile
    | Extern
    (* for class fields *)
    | Public
    | Private
    | Protected
    | Abstract
    | Final
    | Override (* override *)
    (* for vars (JS) *)
    | Var
    | Let
    (* for fields (kinda types) *)
    | Mutable
    | Const (* a.k.a 'readonly' in Typescript *)
    (* less: should be part of the type *)
    | Optional
    (* Typescript '?' *)
    | NotNull (* Typescript '!' *)
    (* for functions *)
    | Generator
    (* '*' in JS *)
    | Async
    | Recursive
    | MutuallyRecursive
    | Inline
    (* for methods *)
    | Ctor
    | Dtor
    | Getter
    | Setter
    (* Rust *)
    | Unsafe
    | DefaultImpl

  (* unstable, RFC 1210 *)
```

3.3.8 Directives

⟨*type* AST_generic.directive 31b⟩≡ (263c)
```
  (* It is tempting to simplify all those ImportXxx in a simpler
   * 'Import of dotted_ident * ...', but module_name is not always a DottedName
   * so it is better to clearly separate what is module_name/namespace from an
   * entity (in this module/namespace) even though some languages such as Python
   * blurs the difference.
   *)
  and directive =
    (* newvar: *)
    | ImportFrom of
        tok (* 'import'/'from' for Python, 'include' for C *)
        * module_name
        * ident
        * alias option (* as name alias *)
    ⟨AST_generic.directive other imports 32a⟩
    ⟨AST_generic.directive package cases 155b⟩
```

```
    | Pragma of ident * any list
    ⟨AST_generic.directive OtherXxx cases 155c⟩
```

⟨AST_generic.directive *other imports* 32a⟩≡ (31b)
```
  | ImportAs of tok * module_name * alias option (* as name *)
  (* bad practice! hard to resolve name locally *)
  | ImportAll of tok * module_name * tok (* '.' in Go, '*' in Java/Python *)
```

⟨*type* AST_generic.module_name 32b⟩≡ (263c)
```
  (* module_name can also be used for a package name or a namespace *)
  type module_name =
    | DottedName of dotted_ident (* ex: Python *)
    (* in FileName the '/' is similar to the '.' in DottedName *)
    | FileName of string wrap (* ex: Js import, C #include, Go import *)
```

⟨*type* AST_generic.dotted_ident 32c⟩≡ (263c)
```
  (* usually separated by a '.', but can be used also with '::' separators *)
  type dotted_ident = ident list (* at least 1 element *)
```

⟨*type* AST_generic.alias 32d⟩≡ (263c)
```
  (* ... as name *)
  and alias = ident * id_info
```

3.3.9 The final program

⟨*type* AST_generic.program 32e⟩≡ (263c)
```
  and program = item list
```

⟨*type* AST_generic.item 32f⟩≡ (263c)
```
  and item = stmt
```

⟨AST_generic.stmt *toplevel and nested construct cases* 32g⟩≡ (26e) 32h ▷
```
  | DefStmt of definition
```

⟨AST_generic.stmt *toplevel and nested construct cases* 32h⟩+≡ (26e) ◁32g
```
  | DirectiveStmt of directive
```

3.3.10 The any type

⟨*type* AST_generic.any 32i⟩≡ (263c)
```
  and any =
    ⟨AST_generic.any semgrep cases 34a⟩
    (* also used for semgrep *)
    | T of type_
    | P of pattern
    | At of attribute
    | Fld of field
    | Args of argument list
    | Partial of partial
    (* misc *)
    | I of ident
    | Str of string wrap
    | Def of definition
    | Dir of directive
    | Pr of program
    ⟨AST_generic.any other cases 226a⟩
```

3.4 The Semgrep pattern AST

⟨*type* Pattern.t 33a⟩≡ (387c)
```
(* right now Expr/Stmt/Stmts/Types/Patterns/Partial and probably
 * more are supported *)
type t = AST_generic.any [@@deriving show, eq]
```

3.4.1 Ellipsis

⟨AST_generic.expr *semgrep extensions cases* 33b⟩≡ (24d) 90a ▷
```
(* sgrep: ... in expressions, args, stmts, items, and fields
 * (and unfortunately also in types in Python) *)
| Ellipsis of tok (* '...' *)
```

⟨AST_generic.parameter *semgrep extension cases* 33c⟩≡ (28c)
```
(* sgrep: ... in parameters
 * note: foo(...x) of Js/Go is using the ParamRest, not this *)
| ParamEllipsis of tok
```

3.4.2 Metavariables

⟨*type* Metavars_generic.mvar 33d⟩≡ (385)
```
type mvar = string [@@deriving show, eq, hash]
```

⟨*constant* Metavars_generic.metavar_regexp_string 33e⟩≡ (385)
```
(* ex: $X, $FAIL, $VAR2, $_
 * Note that some languages such as PHP or Javascript allows '$' in identifier
 * names, so forcing metavariables to have uppercase letters at least allow
 * us to match specifically also identifiers in lower case (e.g., $foo will
 * only match the $foo identifiers in some concrete code; this is not a
 * metavariable).
 * We allow _ as a prefix to disable the unused-metavar check (we use
 * the same convention than OCaml).
 * However this conflicts with PHP superglobals, hence the special
 * cases below in is_metavar_name.
 * coupling: AST_generic.is_metavar_name
 *)
let metavar_regexp_string = "^\\(\\$[A-Z_][A-Z_0-9]*\\)$"
```

⟨*function* Metavars_generic.is_metavar_name 33f⟩≡ (385)
```
(*
 * Hacks abusing existing constructs to encode extra constructions.
 * One day we will have a pattern_ast.ml that mimics mostly
 * AST.ml and extends it with special sgrep constructs.
 *)
let is_metavar_name s =
  match s with
  (* ugly: we should probably pass the language to is_metavar_name, but
   * that would require to thread it through lots of functions, so for
   * now we have this special case for PHP superglobals.
   * ref: https://www.php.net/manual/en/language.variables.superglobals.php
   *)
  | "$_SERVER" | "$_GET" | "$_POST" | "$_FILES" | "$_COOKIE" | "$_SESSION"
  | "$_REQUEST" | "$_ENV"
  (* todo: there's also "$GLOBALS" but this may interface with existing rules*)
```

```
      ->
        false
    | _ -> s =~ metavar_regexp_string
```

3.4.3 The any Semgrep cases

⟨AST_generic.any *semgrep cases* 34a⟩≡ (32i)
```
  | E of expr
  | S of stmt
  | Ss of stmt list
```

3.5 The Semgrep rule

⟨*type* Rule.rule 34b⟩≡ (388)
```
  type rule = {
    id : string;
    pattern : Pattern.t;
    message : string;
    severity : severity;
    languages : Lang.t list;
    (* at least one element *)

    (* Useful for debugging, to report bad rules. We could rule.id to
     * report those bad rules, but semgrep-python uses a weird encoding
     * when flattening the pattern-xxx (-and, -either, -not, etc.)
     * patterns in the original rule yaml file, which makes it hard
     * to know what what was the corresponding pattern.
     *)
    pattern_string : string;
  }
```

⟨*type* Rule.pattern 34c⟩≡ (388)

⟨*type* Rule.rules 34d⟩≡ (388)
```
  and rules = rule list
```

⟨*type* Rule.severity 34e⟩≡ (388)
```
  and severity = Error | Warning | Info
```

⟨*type* Rule.t 34f⟩≡ (388)
```
  (* alias *)
  type t = rule [@@deriving show]
```

3.6 User-defined code equivalences

⟨*type* Equivalence.equivalence 34g⟩≡ (389b)
```
  type equivalence = {
    id : string;
    (* useful? to be able to disable some selectively by name? *)
    left : pattern;
    right : pattern;
```

```
    op : equivalence_kind;
    languages : Lang.t list; (* at least one element *)
  }
```

⟨*type* Equivalence.equivalence_kind 35a⟩≡ (389b)
```
  type equivalence_kind =
    | Equiv
    (* <==> *)
    | Imply

  (* ==> *)
```

⟨*type* Equivalence.pattern 35b⟩≡ (389b)
```
  (* right now only Expr is supported *)
  type pattern = Pattern.t
```

⟨*type* Equivalence.equivalences 35c⟩≡ (389b)
```
  and equivalences = equivalence list
```

⟨*type* Equivalence.t 35d⟩≡ (389b)
```
  (* alias *)
  type t = equivalence
```

3.7 Matching results

⟨*type* Match_result.t 35e⟩≡ (390)
```
  type t = {
    (* rule (or mini rule) responsible for the pattern match found *)
    rule_id : rule_id; [@equal fun a b -> a.id = b.id]
    (* location information *)
    file : Common.filename;
    (* less: redundant with location? *)
    (* note that the two token_location can be equal *)
    range_loc : Parse_info.token_location * Parse_info.token_location;
    (* less: do we need to be lazy? *)
    tokens : Parse_info.t list Lazy.t; [@equal fun _a _b -> true]
    (* metavars for the pattern match *)
    env : Metavariable.bindings;
  }

  (* This is currently a record, but really only the rule id should matter.
   *
   * We could derive information in the other fields from the id, but that
   * would require to pass around the list of rules to get back the
   * information. Instead by embedding the information in the pattern match,
   * some functions are simpler (we use the same trick with Parse_info.t
   * where for example we embed the filename in it, not just a position).
   * alt: reuse Mini_rule.t
   *)
  and rule_id = {
    (* This id is usually a string like 'check-double-equal'.
     * It can be the id of a rule or mini rule.
     *
```

```
    * Note that when we process a full rule, this id can temporarily
    * contain a Rule.pattern_id.
    *)
   id : string;
   (* other parts of a rule (or mini_rule) used in JSON_report.ml *)
   message : string;
   (* used for debugging (could be removed at some point) *)
   pattern_string : string;
 }
 [@@deriving show, eq]
```

⟨*type* Metavars_generic.metavars_binding 36⟩≡ (385)

```
 (* note that the mvalue acts as the value of the metavar and also
    as its concrete code "witness". You can get position information from it,
    it is not Parse_info.Ab(stractPos)

    TODO: ensure that ["$A", Foo; "$B", Bar] and ["$B", Bar; "$A", Foo]
    are equivalent for the equal and hash functions.
    The current implementation is incorrect in general but should work in the
    context of memoizing pattern matching.
 *)
 type bindings = (mvar * mvalue) list (* = Common.assoc *)
 [@@deriving show, eq, hash]
```

Chapter 4

Entry points

⟨*signature* Semgrep_generic.check 37a⟩≡ (447c)
```
val check :
  hook:(Metavariable.bindings -> Parse_info.t list Lazy.t -> unit) ->
  Config_semgrep.t ->
  Mini_rule.rules ->
  Equivalence.equivalences ->
  Common.filename * Lang.t * Target.t ->
  Pattern_match.t list
```

4.1 pfff/cli/Main.ml

⟨*function* Main.main 37b⟩≡ (240)
```
let main () =
  ⟨Main.main() tune the GC 117d⟩

  let usage_msg =
    "Usage: " ^ Filename.basename Sys.argv.(0) ^
    " [options] <file or dir> " ^ "\n" ^ "Options are:"
  in
  (* does side effect on many global flags *)
  let args = Common.parse_options (options()) usage_msg Sys.argv in

  (* must be done after Arg.parse, because Common.profile is set by it *)
  Common.profile_code "Main total" (fun () ->

    (match args with
     ⟨Main.main() match args actions 38i⟩

     (* ------------------------------------------------------ *)
     (* main entry *)
     (* ------------------------------------------------------ *)
     | x::xs ->
         main_action (x::xs)

     (* ------------------------------------------------------ *)
     (* empty entry *)
     (* ------------------------------------------------------ *)
     | [] ->
         Common.usage usage_msg (options());
         failwith "too few arguments"
    )
  )
```

⟨*toplevel* `Main._1` 38a⟩≡ (240)
```
let _ =
  Common.main_boilerplate (fun () ->
    main ();
  )
```

4.1.1 Parsing files: `pfff -lang <lang> <dir>`

⟨*constant* `Main.lang` 38b⟩≡ (240)
```
let lang = ref "c"
```

⟨*function* `Main.main_action` 38c⟩≡ (240)
```
let main_action _xs =
  raise Todo
```

4.1.2 Command-line options: `pfff -<flag>`

⟨*constant* `Main.verbose` 38d⟩≡ (240)
```
let verbose = ref false
```

⟨*function* `Main.options` 38e⟩≡ (240)
```
let options () = [
  "-verbose", Arg.Set verbose,
  " ";
  ⟨Main.options main cases 38f⟩
] @
  ⟨Main.options concatenated flags 225a⟩
  ⟨Main.options concatenated actions 38h⟩
  [
    "-version",   Arg.Unit (fun () ->
      pr2 (spf "pfff version: %s" Config_pfff.version);
      exit 0;
    ), "  guess what";
  ]
```

⟨Main.options *main cases* 38f⟩≡ (38e) 54i▷
```
"-lang", Arg.String (fun s ->
  lang := s;
  ⟨Main.options in -lang callback 50f⟩
), (spf " <str> choose language (default = %s)" !lang);
```

4.1.3 Command-line actions: `pfff -<command>`

⟨*constant* `Main.action` 38g⟩≡ (240)
```
(* action mode *)
let action = ref ""
```

⟨Main.options *concatenated actions* 38h⟩≡ (38e)
```
Common.options_of_actions action (all_actions()) @
```

⟨Main.main() *match* `args` *actions* 38i⟩≡ (37b)
```
(* ------------------------------------------------------ *)
(* actions, useful to debug subpart *)
(* ------------------------------------------------------ *)
| xs when List.mem !action (Common.action_list (all_actions())) ->
    Common.do_action !action xs (all_actions())

| _ when not (Common.null_string !action) ->
    failwith ("unrecognized action or wrong params: " ^ !action)
```

⟨*function* `Main.all_actions` 39a⟩≡ (240)
```
let all_actions () =
  pfff_extra_actions() @
  ⟨Main.all_actions concatenated actions 39e⟩
  []
```

⟨*function* `Main.pfff_extra_actions` 39b⟩≡ (240)
```
let pfff_extra_actions () = [
  "-dump_json", " <file>",
  Common.mk_action_1_arg test_json_pretty_printer;
  ⟨Main.pfff_extra_actions other cases 39d⟩
]
```

⟨*function* `Main.test_json_pretty_printer` 39c⟩≡ (240)
```
let test_json_pretty_printer file =
  let json = J.load_json file in
  let s = J.string_of_json json in
  pr s
```

⟨`Main.pfff_extra_actions` *other cases* 39d⟩≡ (39b) 124a ▷
```
"-json_pp", " <file>",
Common.mk_action_1_arg test_json_pretty_printer;
```

4.1.4 Dumping the generic AST: `pfff -dump_ast <file>`

⟨`Main.all_actions` *concatenated actions* 39e⟩≡ (39a) 225b ▷
```
(* Test_parsing_generic.actions() @ *)
```

⟨*signature* `Test_parsing_generic.actions` 39f⟩≡ (280a)
```
val actions: unit -> Common.cmdline_actions
```

⟨*function* `Test_parsing_generic.actions` 39g⟩≡ (280b)
```
let actions () = [
  "-parse_generic", " <dirs_or_files>",
  Common.mk_action_n_arg test_parse_generic;
  ⟨Test_parsing_generic.actions other cases 39i⟩
]
```

⟨*function* `Test_parsing_generic.test_parse_generic` 39h⟩≡ (280b)
```
let test_parse_generic xs =
  let xs = List.map Common.fullpath xs in
  let files = Common.files_of_dir_or_files_no_vcs_nofilter xs in
  files |> List.iter (fun file ->
    match Lang.langs_of_filename file with
    | [] -> pr2 (spf "skipping %s" file)
    | _x::_xs ->
        Error_code.try_with_print_exn_and_reraise file (fun () ->
          let _ast = Parse_generic.parse_program file in
          ()
        )
  )
```

⟨`Test_parsing_generic.actions` *other cases* 39i⟩≡ (39g) 40b ▷
```
"-show_ast", " <file>",
Common.mk_action_1_arg test_show_generic;
(* now also in sgrep *)
"-dump_ast", " <file>",
Common.mk_action_1_arg test_dump_generic;
```

⟨*function* Test_parsing_generic.test_dump_generic 40a⟩≡ (280b)
```
let test_dump_generic file =
  let x = Parse_generic.parse_program file in

  let v = Meta_AST.vof_any (AST_generic.Pr x) in
  let s = OCaml.string_of_v v in
  pr2 s
```

⟨Test_parsing_generic.actions *other cases* 40b⟩+≡ (39g) ◁39i 50c▷
```
"-dump_generic", " <file>",
Common.mk_action_1_arg test_dump_generic;
```

4.2 semgrep/bin/Main.ml

⟨*function* Main_semgrep_core.main 40c⟩≡ (369)
```
let main () =
  profile_start := Unix.gettimeofday ();

  let usage_msg =
    spf
      "Usage: %s [options] -lang <str> [-e|-f|-rules_file|-config] <pattern> \
       <files_or_dirs> \n\
       Options:"
      (Filename.basename Sys.argv.(0))
  in

  (* ---------------------------------------------------------- *)
  (* Setting up debugging/profiling *)
  (* ---------------------------------------------------------- *)
  let argv =
    Array.to_list Sys.argv
    @ (if Sys.getenv_opt env_debug <> None then [ "-debug" ] else [])
    @ (if Sys.getenv_opt env_profile <> None then [ "-profile" ] else [])
    @
    match Sys.getenv_opt env_extra with
    | Some s -> Common.split "[ \t]+" s
    | None -> []
  in

  (* does side effect on many global flags *)
  let args = Common.parse_options (options ()) usage_msg (Array.of_list argv) in
  if !Flag.gc_tuning then set_gc ();
  let args = if !target_file = "" then args else Common.cat !target_file in

  if Sys.file_exists !log_config_file then (
    Logging.load_config_file !log_config_file;
    logger#info "loaded %s" !log_config_file );
  if !debug then (
    let open Easy_logging in
    let h = Handlers.make (CliErr Debug) in
    logger#add_handler h;
    logger#set_level Debug;
    () );

  logger#info "Executed as: %s" (Sys.argv |> Array.to_list |> String.concat " ");
  logger#info "Version: %s" version;
  if !profile then (
    logger#info "Profile mode On";
```

```
      logger#info "disabling -j when in profiling mode";
      ncores := 1 );

    (* must be done after Arg.parse, because Common.profile is set by it *)
    Common.profile_code "Main total" (fun () ->
        match args with
        ⟨Main_semgrep_core.main() match args actions 48c⟩

        (* ---------------------------------------------------------- *)
        (* main entry *)
        (* ---------------------------------------------------------- *)
        | x :: xs -> (
            match () with
            | _ when !config_file <> "" ->
                semgrep_with_real_rules_file !config_file (x :: xs)
            ⟨Main_semgrep_core.main() main entry match cases 43l⟩
            ⟨Main_semgrep_core.main() main entry match cases default case 42a⟩
            )
        (* ---------------------------------------------------------- *)
        (* empty entry *)
        (* ---------------------------------------------------------- *)
        (* TODO: should not need that, semgrep should not call us when there
         * are no files to process. *)
        | [] when !target_file <> "" && !config_file <> "" ->
            semgrep_with_real_rules_file !config_file []
        | [] -> Common.usage usage_msg (options ()))
```

⟨*toplevel* Main_semgrep_core._1 41a⟩≡ (369)

```
  let _ =
    Common.main_boilerplate (fun () ->
        Common.finalize
          (fun () -> main ())
          (fun () -> !Hooks.exit |> List.iter (fun f -> f ())))
```

⟨*constant* Main_semgrep_core.lang 41b⟩≡ (369)

```
  let lang = ref "unset"
```

⟨*constant* Main_semgrep_core.supported_langs 41c⟩≡ (369)

```
  let supported_langs : string = String.concat ", " keys
```

⟨*constant* Main_semgrep_core.keys 41d⟩≡ (369)

```
  let keys = Common2.hkeys Lang.lang_of_string_map
```

⟨*function* Main_semgrep_core.unsupported_language_message 41e⟩≡ (369)

```
  let unsupported_language_message lang =
    if lang = "unset" then "no language specified; use -lang"
    else
      spf "unsupported language: %s; supported language tags are: %s" lang
        supported_langs
```

4.2.1 Pattern string: `semgrep -e <pattern>`

⟨Main_semgrep_core.main() *main entry match cases default case* 42a⟩≡ (40c)
```
| _ ->
    let lang = lang_of_string !lang in
    semgrep_with_one_pattern lang (x :: xs)
```

⟨*constant* Main_semgrep_core.pattern_string 42b⟩≡ (369)
```
(* -e *)
let pattern_string = ref ""
```

⟨*function* Main_semgrep_core.semgrep_with_one_pattern 42c⟩≡ (369)
```
(* simpler code path compared to semgrep_with_rules *)
let semgrep_with_one_pattern lang xs =
  (* old: let xs = List.map Common.fullpath xs in
   * better no fullpath here, not our responsability.
   *)
  let pattern, pattern_string =
    match (!pattern_file, !pattern_string) with
    ⟨Main_semgrep_core.semgrep_with_one_pattern() sanity check cases 43j⟩
    ⟨Main_semgrep_core.semgrep_with_one_pattern() pattern file case 43i⟩
    (* this is for Emma, who often confuses -e with -f :) *)
    | _, s when s =~ ".*\\.sgrep$" ->
        failwith "you probably want -f with a .sgrep file, not -e"
    | _, s when s <> "" -> (parse_pattern lang s, s)
    | _ -> raise Impossible
  in
  let rule, rule_parse_time =
    Common.with_time (fun () -> [ rule_of_pattern lang pattern_string pattern ])
  in

  match !output_format with
  | Json ->
      (* closer to -rules_file, but no incremental match output *)
      semgrep_with_rules lang (rule, rule_parse_time) xs
  | Text ->
      (* simpler code path than in semgrep_with_rules *)
      let files = Lang.files_of_dirs_or_files lang xs in
      ⟨Main_semgrep_core.semgrep_with_one_pattern() no lang specified 43g⟩
      ⟨Main_semgrep_core.semgrep_with_one_pattern() filter files 119b⟩
      files
      |> List.iter (fun file ->
             ⟨Main_semgrep_core.semgrep_with_one_pattern() if verbose 43e⟩
             let process file =
               timeout_function file (fun () ->
                   let ast, errors = parse_generic lang file in
                   if errors <> [] then
                     pr2 (spf "WARNING: fail to fully parse %s" file);
                   Semgrep_generic.check
                     ~hook:(fun env matched_tokens ->
                       let xs = Lazy.force matched_tokens in
                       print_match !mvars env Metavariable.ii_of_mval xs)
                     Config_semgrep.default_config rule (parse_equivalences ())
                     (file, lang, ast)
                   |> ignore)
             in

             if not !error_recovery then
               E.try_with_print_exn_and_reraise file (fun () -> process file)
```

```
        else E.try_with_exn_to_error file (fun () -> process file));

    ⟨Main_semgrep_core.semgrep_with_one_pattern() display error count 43f⟩
    ⟨Main_semgrep_core.semgrep_with_one_pattern() optional layer generation 123k⟩
```

⟨*type* Main_semgrep_core.ast 43a⟩≡ (369)
```
  ⟨Main_semgrep_core.ast other cases 123e⟩
```

⟨*type* Main_semgrep_core.pattern 43b⟩≡ (369)
```
  ⟨Main_semgrep_core.pattern other cases 123f⟩
```

⟨*function* Main_semgrep_core.sgrep_ast 43c⟩≡ (369)
```
  ⟨Main_semgrep_core.sgrep_ast() hook argument to check 43d⟩
  ⟨Main_semgrep_core.sgrep_ast() match pattern and any_ast other cases 123g⟩
```

⟨Main_semgrep_core.sgrep_ast() hook *argument to* check 43d⟩≡ (43c)

⟨Main_semgrep_core.semgrep_with_one_pattern() *if* verbose 43e⟩≡ (42c)
```
  logger#info "processing: %s" file;
```

⟨Main_semgrep_core.semgrep_with_one_pattern() *display error count* 43f⟩≡ (42c)
```
  let n = List.length !E.g_errors in
  if n > 0 then pr2 (spf "error count: %d" n);
```

⟨Main_semgrep_core.semgrep_with_one_pattern() *no* lang *specified* 43g⟩≡ (42c)

4.2.2 Pattern file: semgrep -f <file>

⟨*constant* Main_semgrep_core.pattern_file 43h⟩≡ (369)
```
  (* -f *)
  let pattern_file = ref ""
```

⟨Main_semgrep_core.semgrep_with_one_pattern() *pattern file case* 43i⟩≡ (42c)
```
  | file, _ when file <> "" ->
      let s = Common.read_file file in
      (parse_pattern lang s, s)
```

⟨Main_semgrep_core.semgrep_with_one_pattern() *sanity check cases* 43j⟩≡ (42c)
```
  | "", "" -> failwith "I need a pattern; use -f or -e"
  | s1, s2 when s1 <> "" && s2 <> "" ->
      failwith "I need just one pattern; use -f OR -e (not both)"
```

4.2.3 Patterns configuration file: semgrep -rules_file <file>

⟨*constant* Main_semgrep_core.rules_file 43k⟩≡ (369)
```
  (* -rules_file (mini rules) *)
  let rules_file = ref ""
```

⟨Main_semgrep_core.main() *main entry match cases* 43l⟩≡ (40c) 193c ▷
```
  | _ when !rules_file <> "" ->
      let lang = lang_of_string !lang in
      semgrep_with_rules_file lang !rules_file (x :: xs)
```

⟨*function* Main_semgrep_core.semgrep_with_rules 44a⟩≡ (369)

```
let semgrep_with_rules lang (rules, rule_parse_time) files_or_dirs =
  let files = get_final_files lang files_or_dirs in
  logger#info "processing %d files" (List.length files);
  let file_results =
    files
    |> iter_files_and_get_matches_and_exn_to_errors (fun file ->
           let (ast, errors), parse_time =
             Common.with_time (fun () -> parse_generic lang file)
           in
           let (matches, errors), match_time =
             Common.with_time (fun () ->
                 let rules =
                   rules |> List.filter (fun r -> List.mem lang r.MR.languages)
                 in
                 ( Semgrep_generic.check
                     ~hook:(fun _ _ -> ())
                     Config_semgrep.default_config rules (parse_equivalences ())
                     (file, lang, ast),
                   errors ))
           in
           { RP.matches; errors; profiling = { file; parse_time; match_time } })
  in
  let res = RP.make_rule_result file_results !report_time rule_parse_time in
  logger#info "found %d matches and %d errors"
    (List.length res.RP.matches)
    (List.length res.RP.errors);
  let matches, new_errors =
    filter_files_with_too_many_matches_and_transform_as_timeout res.RP.matches
  in
  let errors = new_errors @ res.RP.errors in
  let res = { RP.matches; errors; rule_profiling = res.RP.rule_profiling } in
  (* note: uncomment the following and use semgrep-core -stat_matches
   * to debug too-many-matches issues.
   * Common2.write_value matches "/tmp/debug_matches";
   *)
  let flds = JSON_report.json_fields_of_matches_and_errors files res in
  let flds =
    if !profile then (
      let json = JSON_report.json_of_profile_info !profile_start in
      (* so we don't get also the profile output of Common.main_boilerplate*)
      Common.profile := Common.ProfNone;
      flds @ [ ("profiling", json) ] )
    else flds
  in
  (*
     Not pretty-printing the json output (Yojson.Safe.prettify)
     because it kills performance, adding an extra 50% time on our
     calculate_ci_perf.py benchmarks.
     User should use an external tool like jq or ydump (latter comes with
     yojson) for pretty-printing json.
  *)
  let s = J.string_of_json (J.Object flds) in
  logger#info "size of returned JSON string: %d" (String.length s);
  pr s
```

⟨*function* Main_semgrep_core.iter_generic_ast_of_files_and_get_matches_and_exn_to_errors 44b⟩≡ (369)

```
let iter_files_and_get_matches_and_exn_to_errors f files =
  files
```

```
|> map (fun file ->
       logger#info "Analyzing %s" file;
       let res, run_time =
         Common.with_time (fun () ->
             try
               run_with_memory_limit !max_memory (fun () ->
                   timeout_function file (fun () ->
                       f file |> fun v ->
                       (* This is just to test -max_memory, to give a chance
                        * to Gc.create_alarm to run even if the program does
                        * not even need to run the Gc. However, this has a slow
                        * perf penality on small programs, which is why it's
                        * better to keep guarded when you're
                        * not testing -max_memory.
                        *)
                       if !test then Gc.full_major ();
                       logger#info "done with %s" file;
                       v))
             with
             (* note that Error_code.exn_to_error already handles Timeout
              * and would generate a TimeoutError code for it, but we intercept
              * Timeout here to give a better diagnostic.
              *)
             | (Timeout | Out_of_memory) as exn ->
                 let str_opt =
                   match !Semgrep_generic.last_matched_rule with
                   | None -> None
                   | Some rule ->
                       logger#info "critical exn while matching ruleid %s"
                         rule.MR.id;
                       logger#info "full pattern is: %s"
                         rule.MR.pattern_string;
                       Some (spf " with ruleid %s" rule.MR.id)
                 in
                 let loc = Parse_info.first_loc_of_file file in
                 {
                   RP.matches = [];
                   errors =
                     [
                       Error_code.mk_error_loc loc
                         ( match exn with
                         | Timeout ->
                             logger#info "Timeout on %s" file;
                             Error_code.Timeout str_opt
                         | Out_of_memory ->
                             logger#info "OutOfMemory on %s" file;
                             Error_code.OutOfMemory str_opt
                         | _ -> raise Impossible );
                     ];
                   profiling = RP.empty_partial_profiling file;
                 }
             | exn when not !fail_fast ->
                 {
                   RP.matches = [];
                   errors = [ Error_code.exn_to_error file exn ];
                   profiling = RP.empty_partial_profiling file;
                 })
       in
       RP.add_run_time run_time res)
```

⟨Main_semgrep_core.semgrep_with_rules() *if* verbose 46a⟩≡ (369)
```
logger#info "Parsing %s" rules_file;
```

4.2.4 The main algorithm

4.2.5 Command-line options: `semgrep -<flag>`

⟨*constant* Main_semgrep_core.verbose 46b⟩≡ (369)

⟨*constant* Flag_semgrep.verbose 46c⟩≡ (384)

⟨*function* Main_semgrep_core.options 46d⟩≡ (369)
```
let options () =
  [
    ("-e", Arg.Set_string pattern_string, " <str> use the string as the pattern");
    ( "-f",
      Arg.Set_string pattern_file,
      " <file> use the file content as the pattern" );
    ( "-rules_file",
      Arg.Set_string rules_file,
      " <file> obtain a list of patterns from YAML file (implies -json)" );
    ( "-config",
      Arg.Set_string config_file,
      " <file> obtain formula of patterns from YAML/JSON/Jsonnet file" );
    ( "-lang",
      Arg.Set_string lang,
      spf " <str> choose language (valid choices:\n     %s)" supported_langs );
    ( "-target_file",
      Arg.Set_string target_file,
      " <file> obtain list of targets to run patterns on" );
    ⟨Main_semgrep_core.options user-defined equivalences case 102c⟩
    ⟨Main_semgrep_core.options file filters cases 119a⟩
    ⟨Main_semgrep_core.options -j case 117b⟩
    ( "-opt_cache",
      Arg.Set Flag.with_opt_cache,
      " enable caching optimization during matching" );
    ( "-no_opt_cache",
      Arg.Clear Flag.with_opt_cache,
      " disable caching optimization during matching" );
    ( "-opt_max_cache",
      Arg.Unit
        (fun () ->
          Flag.with_opt_cache := true;
          Flag.max_cache := true),
      " cache matches more aggressively; implies -opt_cache (experimental)" );
    ( "-no_gc_tuning",
      Arg.Clear Flag.gc_tuning,
      " use OCaml's default garbage collector settings" );
    ⟨Main_semgrep_core.options report match mode cases 111b⟩
    ⟨Main_semgrep_core.options other cases 115b⟩
    ( "-use_parsing_cache",
      Arg.Set_string use_parsing_cache,
      " <dir> save and use parsed ASTs in a cache at given directory.\n\
      \    It is the caller's responsiblity to clear the cache" );
    ( "-filter_irrelevant_patterns",
      Arg.Set Flag.filter_irrelevant_patterns,
      " filter patterns not containing any strings in target file" );
    ( "-no_filter_irrelevant_patterns",
      Arg.Clear Flag.filter_irrelevant_patterns,
```

```
    " do not filter patterns" );
  ( "-filter_irrelevant_rules",
    Arg.Set Flag.filter_irrelevant_rules,
    " filter rules not containing any strings in target file" );
  ( "-no_filter_irrelevant_rules",
    Arg.Clear Flag.filter_irrelevant_rules,
    " do not filter rules" );
  ( "-fast",
    Arg.Set Flag.filter_irrelevant_rules,
    " filter rules not containing any strings in target file" );
  ( "-bloom_filter",
    Arg.Set Flag.use_bloom_filter,
    " use a bloom filter to only attempt matches when strings in the pattern \
     are in the target" );
  ( "-no_bloom_filter",
    Arg.Clear Flag.use_bloom_filter,
    " do not use bloom filter" );
  ( "-set_filter",
    Arg.Set Flag.set_instead_of_bloom_filter,
    "use a set instead of bloom filters" );
  ( "-tree_sitter_only",
    Arg.Set Flag.tree_sitter_only,
    " only use tree-sitter-based parsers" );
  ( "-timeout",
    Arg.Set_float timeout,
    " <float> time limit to process one input program (in seconds); 0 \
     disables timeouts (default is 0)" );
  ( "-max_memory",
    Arg.Set_int max_memory,
    " <int> maximum memory available (in MB); allows for clean termination\n\
    \     when running out of memory. This value should be less than the \
     actual\n\
    \     memory available because the limit will be exceeded before it gets\n\
    \     detected. Try 5% less or 15000 if you have 16 GB." );
  ( "-max_match_per_file",
    Arg.Set_int max_match_per_file,
    " <int> maximum numbers of match per file" );
  ("-debug", Arg.Set debug, " output debugging information");
  ( "-debug_matching",
    Arg.Set Flag.debug_matching,
    " raise an exception at the first match failure" );
  ( "-log_config_file",
    Arg.Set_string log_config_file,
    " <file> logging configuration file" );
  ( "-log_to_file",
    Arg.String
      (fun file ->
        let open Easy_logging in
        let h = Handlers.make (File (file, Debug)) in
        logger#add_handler h;
        logger#set_level Debug),
    " <file> log debugging info to file" );
  ("-test", Arg.Set test, " (internal) set test context");
  ( "-lsp",
    Arg.Unit (fun () -> LSP_client.init ()),
    " connect to LSP lang server to get type information" );
]
⟨Main_semgrep_core.options concatenated flags 199b⟩
⟨Main_semgrep_core.options concatenated actions 48b⟩
@ [
```

```
    ( "-version",
      Arg.Unit
        (fun () ->
          pr2 version;
          exit 0),
      "  guess what" );
  ]
```

4.2.6 Command-line actions: `semgrep -<command>`

⟨*constant* `Main_semgrep_core.action` 48a⟩≡ (369)
```
(* action mode *)
let action = ref ""
```

⟨`Main_semgrep_core.options` *concatenated actions* 48b⟩≡ (46d)
```
@ Common.options_of_actions action (all_actions ())
```

⟨`Main_semgrep_core.main()` *match* `args` *actions* 48c⟩≡ (40c)
```
(* ------------------------------------------------------- *)
(* actions, useful to debug subpart *)
(* ------------------------------------------------------- *)
| xs when List.mem !action (Common.action_list (all_actions ())) ->
    Common.do_action !action xs (all_actions ())
| _ when not (Common.null_string !action) ->
    failwith ("unrecognized action or wrong params: " ^ !action)
```

⟨*function* `Main_semgrep_core.all_actions` 48d⟩≡ (369)
```
let all_actions () =
  [
    ⟨Main_semgrep_core.all_actions dumper cases 49a⟩
    ("-dump_rule", " <file>", Common.mk_action_1_arg dump_rule);
    ( "-dump_tree_sitter_cst",
      " <file>",
      Common.mk_action_1_arg Test_parsing.dump_tree_sitter_cst );
    ( "-dump_tree_sitter_pattern_cst",
      " <file>",
      Common.mk_action_1_arg (fun file ->
          Parse_pattern.dump_tree_sitter_pattern_cst (lang_of_string !lang) file)
    );
    ( "-dump_ast_pfff",
      " <file>",
      Common.mk_action_1_arg Test_parsing.dump_ast_pfff );
    ("-dump_il", " <file>", Common.mk_action_1_arg Datalog_experiment.dump_il);
    ( "-diff_pfff_tree_sitter",
      " <file>",
      Common.mk_action_n_arg Test_parsing.diff_pfff_tree_sitter );
    ⟨Main_semgrep_core.all_actions other cases 123a⟩
    ( "-expr_at_range",
      " <l:c-l:c> <file>",
      Common.mk_action_2_arg Test_synthesizing.expr_at_range );
    ( "-synthesize_patterns",
      " <l:c-l:c> <file>",
      Common.mk_action_2_arg Test_synthesizing.synthesize_patterns );
    ( "-generate_patterns",
      " <l:c-l:c>+ <file>",
      Common.mk_action_n_arg Test_synthesizing.generate_pattern_choices );
    ( "-stat_matches",
```

```
      " <marshalled file>",
      Common.mk_action_1_arg Experiments.stat_matches );
    ( "-ebnf_to_menhir",
      " <ebnf file>",
      Common.mk_action_1_arg Experiments.ebnf_to_menhir );
    ( "-parsing_stats",
      " <files or dirs> generate parsing statistics (use -json for JSON output)",
      Common.mk_action_n_arg (fun xs ->
          Test_parsing.parsing_stats (lang_of_string !lang)
            (!output_format = Json) xs) );
    ( "-parsing_regressions",
      " <files or dirs> look for parsing regressions",
      Common.mk_action_n_arg (fun xs ->
          Test_parsing.parsing_regressions (lang_of_string !lang) xs) );
    ( "-test_parse_tree_sitter",
      " <files or dirs> test tree-sitter parser on target files",
      Common.mk_action_n_arg (fun xs ->
          Test_parsing.test_parse_tree_sitter (lang_of_string !lang) xs) );
    ( "-check_rules",
      " <files or dirs>",
      Common.mk_action_n_arg (Check_rule.check_files Parse_rule.parse) );
    ( "-stat_rules",
      " <files or dirs>",
      Common.mk_action_n_arg (Check_rule.stat_files Parse_rule.parse) );
    ( "-test_rules",
      " <files or dirs>",
      Common.mk_action_n_arg Test_engine.test_rules );
    ( "-parse_rules",
      " <files or dirs>",
      Common.mk_action_n_arg Test_parsing.test_parse_rules );
    ( "-datalog_experiment",
      " <file> <dir>",
      Common.mk_action_2_arg Datalog_experiment.gen_facts );
    ("-postmortem", " <log file", Common.mk_action_1_arg Statistics_report.stat);
    ( "-test_comby",
      " <pattern> <file>",
      Common.mk_action_2_arg Test_comby.test_comby );
    ("-eval", " <JSON file>", Common.mk_action_1_arg Eval_generic.eval_json_file);
    ("-test_eval", " <JSON file>", Common.mk_action_1_arg Eval_generic.test_eval);
  ]
  @ Test_analyze_generic.actions ()
```

⟨Main_semgrep_core.all_actions *dumper cases* 49a⟩≡ (48d) 50a ▷

```
  ( "-dump_extensions",
    " print file extension to language mapping",
    Common.mk_action_0_arg dump_ext_of_lang );
```

⟨*function* Main_semgrep_core.dump_ext_of_lang 49b⟩≡ (369)

```
  let dump_ext_of_lang () =
    let lang_to_exts =
      keys
      |> List.map (fun lang_str ->
             match Lang.lang_of_string_opt lang_str with
             | Some lang ->
                 lang_str ^ "->" ^ String.concat ", " (Lang.ext_of_lang lang)
             | None -> "")
    in
    pr2
      (spf "Language to supported file extension mappings:\n %s"
```

```
        (String.concat "\n" lang_to_exts))
```

4.2.7 Dumping the pattern AST: `semgrep -lang <lang> -dump_pattern <file>`

⟨Main_semgrep_core.all_actions *dumper cases* 50a⟩+≡ (48d) ◁49a 50h▷
```
  ("-dump_pattern", " <file>", Common.mk_action_1_arg dump_pattern);
```

⟨*function* Main_semgrep_core.dump_pattern 50b⟩≡ (369)
```
  (* works with -lang *)
  let dump_pattern (file : Common.filename) =
    let s = Common.read_file file in
    (* mostly copy-paste of parse_pattern above, but with better error report *)
    let lang = lang_of_string !lang in
    E.try_with_print_exn_and_reraise file (fun () ->
        let any = Parse_pattern.parse_pattern lang ~print_errors:true s in
        let v = Meta_AST.vof_any any in
        let s = dump_v_to_format v in
        pr s)
```

⟨Test_parsing_generic.actions *other cases* 50c⟩+≡ (39g) ◁40b
```
  "-dump_pattern", " <file>",
  Common.mk_action_1_arg test_dump_pattern_generic;
```

⟨*signature* Test_parsing_generic.lang 50d⟩≡ (280a)
```
  val lang: string ref
```

⟨*constant* Test_parsing_generic.lang 50e⟩≡ (280b)
```
  let lang = ref ""
```

⟨Main.options *in* -lang *callback* 50f⟩≡ (38f)
```
  (* a big ugly *)
  (* Test_parsing_generic.lang := s; *)
```

⟨*function* Test_parsing_generic.test_dump_pattern_generic 50g⟩≡ (280b)
```
  let test_dump_pattern_generic file =
    match Lang.lang_of_string_opt !lang with
    | _ when !lang = "" -> failwith "use -lang"
    | Some lang ->
        let s = Common.read_file file in
        Error_code.try_with_print_exn_and_reraise file (fun () ->
          let any = Parse_generic.parse_pattern lang s in
          let s = AST_generic.show_any any in
          pr2 s
        )
    | None -> failwith (spf "unsupported language: %s" !lang)
```

4.2.8 Dumping the generic AST: `semgrep -lang <lang> -dump_ast <file>`

⟨Main_semgrep_core.all_actions *dumper cases* 50h⟩+≡ (48d) ◁50a 105a▷
```
  ("-dump_ast", " <file>", Common.mk_action_1_arg (dump_ast ~naming:false));
  ( "-dump_named_ast",
    " <file>",
    Common.mk_action_1_arg (dump_ast ~naming:true) );
  ("-dump_v1_json", " <file>", Common.mk_action_1_arg dump_v1_json);
```

⟨*function* `Main_semgrep_core.dump_ast` 51a⟩≡ (369)

```
let dump_ast ?(naming = false) file =
  match Lang.langs_of_filename file with
  | lang :: _ ->
      E.try_with_print_exn_and_reraise file (fun () ->
          let { Parse_target.ast; errors; _ } =
            if naming then
              Parse_target.parse_and_resolve_name_use_pfff_or_treesitter lang
                file
            else Parse_target.just_parse_with_lang lang file
          in
          let v = Meta_AST.vof_any (AST_generic.Pr ast) in
          let s = dump_v_to_format v in
          pr s;
          if errors <> [] then pr2 (spf "WARNING: fail to fully parse %s" file))
  | [] -> failwith (spf "unsupported language for %s" file)
```

⟨*function* `Main_semgrep_core.dump_v_to_format` 51b⟩≡ (369)

```
let dump_v_to_format (v : OCaml.v) =
  match !output_format with
  | Text -> OCaml.string_of_v v
  | Json -> J.string_of_json (json_of_v v)
```

Chapter 5

Parsing

5.1 Parsing code

⟨*function* Main_semgrep_core.create_ast 52a⟩≡ (369)
  ⟨Main_semgrep_core.create_ast() *when not a supported language* 123h⟩

⟨*function* Main_semgrep_core.parse_generic 52b⟩≡ (369)
```
(* It should really be just a call to Parse_target.parse_and_resolve...
 * but the semgrep python wrapper calls semgrep-core separately for each
 * rule, so we need to cache parsed AST to avoid extra work.
 *)
let parse_generic lang file =
  ⟨Main_semgrep_core.parse_generic() use standard macros if parsing C 199a⟩
  let v =
    cache_computation file
      (fun file ->
        (* we may use different parsers for the same file (e.g., in Python3 or
         * Python2 mode), so put the lang as part of the cache "dependency".
         * We also add ast_version here so bumping the version will not
         * try to use the old cache file (which should generate an exception).
         *)
        let full_filename =
          spf "%s__%s__%s" file (Lang.string_of_lang lang) Version.version
        in
        cache_file_of_file full_filename)
      (fun () ->
        try
          logger#info "parsing %s" file;
          (* finally calling the actual function *)
          let { Parse_target.ast; errors; _ } =
            Parse_target.parse_and_resolve_name_use_pfff_or_treesitter lang file
          in
          ⟨Main_semgrep_core.parse_generic() resolve names in the AST 173a⟩
          Left (ast, errors)
          (* This is a bit subtle, but we now store in the cache whether we had
           * an exception on this file, especially Timeout. Indeed, semgrep now calls
           * semgrep-core per rule, and if one file timeout during parsing, it would
           * timeout for each rule, but we don't want to wait each time 5sec for each
           * rule. So here we store the exn in the cache, and below we reraise it
           * after we got it back from the cache.
           *
           * TODO: right now we just capture Timeout, but we should capture any exn.
           *  However this introduces some weird regressions in CI so we focus on
           *  just Timeout for now.
           *)
```

```
      with Timeout -> Right Timeout)
  in
  match v with Left x -> x | Right exn -> raise exn
  [@@profiling]
```

⟨*signature* Parse_generic.parse_with_lang 53a⟩≡ (279d)
```
val parse_with_lang: Lang.t -> Common.filename ->
  AST_generic.program * Parse_info.parsing_stat
```

⟨*function* Parse_generic.parse_with_lang 53b⟩≡ (279e)
```
let parse_with_lang lang file =
  match lang with
  | Lang.Python | Lang.Python2 | Lang.Python3 ->
      let parsing_mode = lang_to_python_parsing_mode lang in
      let {Parse_info. ast; stat; _} = Parse_python.parse ~parsing_mode file in
      (* old: Resolve_python.resolve ast;
       * switched to call Naming_AST.ml in sgrep to correct def and use tagger
      *)
      Python_to_generic.program ast, stat
  | Lang.Typescript (* abusing Js parser for now here *)
  | Lang.Javascript ->
      let {Parse_info. ast; stat; _} = Parse_js.parse file in
      Js_to_generic.program ast, stat
  | Lang.JSON ->
      let ast = Parse_json.parse_program file in
      Json_to_generic.program ast, Parse_info.correct_stat file
  | Lang.C ->
      (* this internally uses the CST for c++ *)
      let {Parse_info. ast; stat; _} = Parse_c.parse file in
      C_to_generic.program ast, stat
  | Lang.Cplusplus ->
      failwith "TODO"
  | Lang.Java ->
      let {Parse_info. ast; stat; _} = Parse_java.parse file in
      Java_to_generic.program ast, stat
  | Lang.Go ->
      let {Parse_info. ast; stat; _} = Parse_go.parse file in
      (* old: Resolve_go.resolve ast;
       * switched to call Naming_AST.ml in sgrep to correct def and use tagger
      *)
      Go_to_generic.program ast, stat
  | Lang.OCaml ->
      let {Parse_info. ast; stat; _} = Parse_ml.parse file in
      Ml_to_generic.program ast, stat
  | Lang.Ruby ->
      let {Parse_info. ast; stat; _} = Parse_ruby.parse file in
      Ruby_to_generic.program ast, stat
  | Lang.PHP ->
      let {Parse_info. ast = cst; stat; _} = Parse_php.parse file in
      let ast = Ast_php_build.program cst in
      Php_to_generic.program ast, stat
  | Lang.Csharp ->
      failwith "No C# parser in pfff; use the one in tree-sitter"
  | Lang.Kotlin ->
      failwith "No Kotlin parser in pfff; use the one in tree-sitter"
  | Lang.Lua ->
      failwith "No Lua parser in pfff; use the one in tree-sitter"
  | Lang.Rust ->
      failwith "No Rust parser in pfff; use the one in tree-sitter"
```

```
    | Lang.R ->
        failwith "No R parser in pfff; use the one in tree-sitter"
    | Lang.Yaml ->
        failwith "No Yaml parser yet, parsed only for semgrep in Parse_rule.ml"
```

⟨*exception* Parse_info.Lexical_error 54a⟩≡ (252k)
```
  (* see also Parsing.Parse_error and Failure "empty token" raised by Lexing *)
  exception Lexical_error of string * t
```

⟨*exception* Parse_info.Parsing_error 54b⟩≡ (252k)
```
  (* better than Parsing.Parse_error, which does not have location information *)
  exception Parsing_error of t
```

⟨*exception* Parse_info.Ast_builder_error 54c⟩≡ (252k)
```
  (* when convert from CST to AST *)
  exception Ast_builder_error of string * t
```

⟨*exception* Parse_info.Other_error 54d⟩≡ (252k)
```
  (* other stuff *)
  exception Other_error of string * t
```

⟨*exception* AST_generic.Error 54e⟩≡ (263c)
```
  (* this can be used in the xxx_to_generic.ml file to signal limitations,
   * and can be captured in Error_code.exn_to_error to pinpoint the error
   * location.
   *)
  exception Error of string * Parse_info.t
```

⟨*function* AST_generic.error 54f⟩≡ (263c)
```
  let error tok msg = raise (Error (msg, tok))
```

⟨*signature* Parse_generic.parse_program 54g⟩≡ (279d)
```
  (* infer the language from the filename *)
  val parse_program: Common.filename -> AST_generic.program
```

⟨*function* Parse_generic.parse_program 54h⟩≡ (279e)
```
  let parse_program file =
    match Lang.langs_of_filename file with
    | [x] -> parse_with_lang x file |> fst
    | x::_xs ->
        (* print a warning? that we default to one? *)
        parse_with_lang x file |> fst
    | [] -> failwith (spf "unsupported file for AST generic: %s" file)
```

5.2 Parsing a Semgrep simple pattern

⟨Main.options *main cases* 54i⟩+≡ (38e) ◁38f
```
  "-sgrep_mode", Arg.Set Flag_parsing.sgrep_mode,
  " enable sgrep mode parsing (to debug)";
```

⟨*function* Main_semgrep_core.parse_pattern 55a⟩≡ (369)

```
let parse_pattern lang_pattern str =
  try
    Common.save_excursion Flag_parsing.sgrep_mode true (fun () ->
        let res =
          Parse_pattern.parse_pattern lang_pattern ~print_errors:false str
        in
        ⟨Main_semgrep_core.parse_pattern() when not a supported language 123i⟩
        res)
  with exn ->
    raise
      (Parse_mini_rule.InvalidPatternException
         ("no-id", str, !lang, Common.exn_to_s exn))
  [@@profiling]
```

⟨*signature* Parse_generic.parse_pattern 55b⟩≡ (279d)

```
(* for sgrep *)
val parse_pattern: Lang.t -> string -> AST_generic.any
```

⟨*function* Parse_generic.parse_pattern 55c⟩≡ (279e)

```
let parse_pattern lang str =
  match lang with
  | Lang.Python | Lang.Python2 | Lang.Python3 ->
      let parsing_mode = lang_to_python_parsing_mode lang in
      let any = Parse_python.any_of_string ~parsing_mode str in
      Python_to_generic.any any
  (* abusing JS parser so no need extend tree-sitter grammar*)
  | Lang.Typescript
  | Lang.Javascript ->
      let any = Parse_js.any_of_string str in
      Js_to_generic.any any
  | Lang.JSON ->
      let any = Parse_json.any_of_string str in
      Json_to_generic.any any
  | Lang.C ->
      let any = Parse_c.any_of_string str in
      C_to_generic.any any
  | Lang.Java ->
      let any = Parse_java.any_of_string str in
      Java_to_generic.any any
  | Lang.Go ->
      let any = Parse_go.any_of_string str in
      Go_to_generic.any any
  | Lang.OCaml ->
      let any = Parse_ml.any_of_string str in
      Ml_to_generic.any any
  | Lang.Ruby ->
      let any = Parse_ruby.any_of_string str in
      Ruby_to_generic.any any
  | Lang.PHP ->
      let any_cst = Parse_php.any_of_string str in
      let any = Ast_php_build.any any_cst in
      Php_to_generic.any any
  | Lang.Csharp ->
      failwith "No C# parser in pfff; use the one in tree-sitter"
  | Lang.Kotlin ->
      failwith "No Kotlin parser in pfff; use the one in tree-sitter"
  | Lang.Cplusplus ->
      failwith "No C++ generic parser in pfff; use the one in tree-sitter"
```

```
  | Lang.Lua ->
      failwith "No Lua generic parser in pfff; use the one in tree-sitter"
  | Lang.Rust ->
      failwith "No Rust generic parser in pfff; use the one in tree-sitter"
  | Lang.R ->
      failwith "No R generic parser in pfff; use the one in tree-sitter"
  | Lang.Yaml ->
      failwith "No Yaml parser yet, parsed only for semgrep in Parse_rule.ml"
```

5.3 Parsing a Semgrep configuration file (rules)

⟨*signature* Parse_rules.parse 56a⟩≡ (425a)

```
val parse : Common.filename -> Mini_rule.rules
```

⟨*function* Parse_rules.parse 56b⟩≡ (425b)

```
let parse file =
  let str = Common.read_file file in
  let yaml_res = Yaml.of_string str in
  match yaml_res with
  | Result.Ok v -> (
      match v with
      | `O [ ("rules", `A xs) ] ->
          xs
          |> List.map (fun v ->
                 match v with
                 | `O xs -> (
                     match Common.sort_by_key_lowfirst xs with
                     | [
                      ("id", `String id);
                      ("languages", `A langs);
                      ("message", `String message);
                      ("pattern", `String pattern_string);
                      ("severity", `String sev);
                     ] ->
                         let languages, lang = parse_languages ~id langs in
                         let pattern = parse_pattern ~id ~lang pattern_string in
                         let severity = parse_severity ~id sev in
                         {
                           R.id;
                           pattern;
                           message;
                           languages;
                           severity;
                           pattern_string;
                         }
                     | x ->
                         pr2_gen x;
                         raise (InvalidYamlException "wrong rule fields") )
                 | x ->
                     pr2_gen x;
                     raise (InvalidYamlException "wrong rule fields"))
      | _ -> raise (InvalidYamlException "missing rules entry as top-level key")
      )
  | Result.Error (`Msg s) -> raise (UnparsableYamlException s)
```

⟨*exception* Parse_rules.InvalidRuleException 57a⟩≡ (425)
```
exception InvalidRuleException of string * string
```

⟨*exception* Parse_rules.InvalidLanguageException 57b⟩≡ (425)
```
exception InvalidLanguageException of string * string
```

⟨*exception* Parse_rules.InvalidPatternException 57c⟩≡ (425)
```
exception InvalidPatternException of string * string * string * string
```

⟨*exception* Parse_rules.UnparsableYamlException 57d⟩≡ (425)
```
exception UnparsableYamlException of string
```

⟨*exception* Parse_rules.InvalidYamlException 57e⟩≡ (425)
```
exception InvalidYamlException of string
```

⟨*function* Main_semgrep_core.format_output_exception 57f⟩≡ (369)

⟨*signature* Parse_rules.parse_languages 57g⟩≡ (425a)
```
val parse_languages : id:string -> Yaml.value list -> Lang.t list * Lang.t
```

⟨*signature* Parse_rules.parse_severity 57h⟩≡ (425a)
```
val parse_severity : id:string -> string -> Mini_rule.severity
```

⟨*signature* Parse_rules.parse_pattern 57i⟩≡ (425a)
```
val parse_pattern : id:string -> lang:Lang.t -> string -> Pattern.t
```

⟨*function* Parse_rules.parse_severity 57j⟩≡ (425b)
```
(* also used in Parse_tainting_rules.ml *)
let parse_severity ~id s =
  match s with
  | "ERROR" -> R.Error
  | "WARNING" -> R.Warning
  | "INFO" -> R.Info
  | s ->
      raise
        (InvalidRuleException
           (id, spf "Bad severity: %s (expected ERROR, WARNING or INFO)" s))
```

⟨*function* Parse_rules.parse_pattern 57k⟩≡ (425b)
```
let parse_pattern ~id ~lang pattern =
  (* todo? call Normalize_ast.normalize here? *)
  try Parse_pattern.parse_pattern lang ~print_errors:false pattern with
  | Timeout -> raise Timeout
  | UnixExit n -> raise (UnixExit n)
  | exn ->
      raise
        (InvalidPatternException
           (id, pattern, Lang.string_of_lang lang, Common.exn_to_s exn))
```

⟨*function* Parse_rules.parse_languages 58a⟩≡ (425b)
```
let parse_languages ~id langs =
  let languages =
    langs
    |> List.map (function
         | `String s -> (
             match Lang.lang_of_string_opt s with
             | None ->
                 raise
                   (InvalidLanguageException
                      (id, spf "unsupported language: %s" s))
             | Some l -> l )
         | _ ->
             raise
               (InvalidRuleException (id, spf "expecting a string for languages")))
  in
  let lang =
    match languages with
    | [] -> raise (InvalidRuleException (id, "we need at least one language"))
    | x :: _xs -> x
  in
  (languages, lang)
```

5.4 Parsing a Semgrep equivalence

⟨*signature* Parse_equivalences.parse 58b⟩≡ (427a)
```
val parse : Common.filename -> Equivalence.equivalences
```

⟨*function* Parse_equivalences.error 58c⟩≡ (427b)
```
let error s = failwith (spf "sgrep_equivalence: wrong format. %s" s)
```

⟨*function* Parse_equivalences.parse 58d⟩≡ (427b)
```
let parse file =
  let str = Common.read_file file in
  let yaml_res = Yaml.of_string str in
  match yaml_res with
  | Result.Ok v -> (
      match v with
      | `O [ ("equivalences", `A xs) ] ->
          xs
          |> List.map (fun v ->
                 match v with
                 | `O xs -> (
                     match Common.sort_by_key_lowfirst xs with
                     | [
                      ("id", `String id);
                      ("languages", `A langs);
                      ("pattern", `String str);
                     ] ->
                         let languages =
                           langs
                           |> List.map (function
                                | `String s -> (
                                    match Lang.lang_of_string_opt s with
                                    | None ->
```

```
                                error (spf "unsupported language: %s" s)
                            | Some l -> l )
                        | _ ->
                            error
                              (spf "expecting a string for languages"))
                    in
                    let lang =
                      match languages with
                      | [] -> error "we need at least one language"
                      | x :: _xs -> x
                    in
                    let left, op, right =
                      let xs =
                        Str.full_split (Str.regexp "<==>\\|==>") str
                      in
                      match xs with
                      | [ Str.Text a; Str.Delim "<==>"; Str.Text b ] ->
                          (a, Eq.Equiv, b)
                      | [ Str.Text a; Str.Delim "==>"; Str.Text b ] ->
                          (a, Eq.Imply, b)
                      | _ ->
                          error
                            (spf "could not parse the equivalence: %s" str)
                    in
                    let left =
                      try Parse_pattern.parse_pattern lang left
                      with exn ->
                        error
                          (spf
                             "could not parse the left pattern: %s (exn = \
                              %s)"
                             left (Common.exn_to_s exn))
                    in
                    let right =
                      try Parse_pattern.parse_pattern lang right
                      with exn ->
                        error
                          (spf
                             "could not parse the right pattern: %s (exn \
                              = %s)"
                             right (Common.exn_to_s exn))
                    in
                    { Eq.id; left; op; right; languages }
                | x ->
                    pr2_gen x;
                    error "wrong equivalence fields" )
            | x ->
                pr2_gen x;
                error "wrong equivalence fields")
    | _ -> error "missing equivalences entry" )
| Result.Error (`Msg s) ->
    failwith (spf "sgrep_equivalence: could not parse %s (error = %s)" file s)
```

Chapter 6

Visiting and Matching

6.1 Semgrep_generic.check()

⟨*function* Semgrep_generic.check2 60⟩≡ (448)

```
let check2 ~hook config rules equivs (file, lang, ast) =
  logger#info "checking %s with %d mini rules" file (List.length rules);

  let rules =
    (* simple opti using regexps; the bloom filter opti might supersede this *)
    if !Flag.filter_irrelevant_patterns then
      Mini_rules_filter.filter_mini_rules_relevant_to_file_using_regexp rules
        lang file
    else rules
  in
  if rules = [] then []
  else
    let matches = ref [] in

    (* old: let prog = Normalize_AST.normalize (Pr ast) lang in
       * we were rewriting code, e.g., A != B was rewritten as !(A == B),
       * which enable some nice semantic matching demo where searching for
       * $X == $X would also find code written as a != a. The problem
       * is that if we don't do the same rewriting on the pattern, then
       * looking for $X != $X would not find anything anymore.
       * In any case, rewriting the source code is less necessary
       * now that we have user-defined code equivalences (see Equivalence.ml)
       * and this will also be less surprising (you can see the set of
       * equivalences in the equivalence file).
    *)
    let prog = Pr ast in

    let expr_rules = ref [] in
    let stmt_rules = ref [] in
    let stmts_rules = ref [] in
    let type_rules = ref [] in
    let pattern_rules = ref [] in
    let attribute_rules = ref [] in
    let fld_rules = ref [] in
    let partial_rules = ref [] in
    ⟨Semgrep_generic.check2() populate expr_rules and other 61b⟩
    let hooks =
      {
        V.default_visitor with
        ⟨Semgrep_generic.check2() visitor fields 62a⟩
        V.ktype_ =
```

```
          (fun (k, _) x ->
            match_rules_and_recurse config (file, hook, matches) !type_rules
              match_t_t k
              (fun x -> T x)
              x);
        V.kpattern =
          (fun (k, _) x ->
            match_rules_and_recurse config (file, hook, matches) !pattern_rules
              match_p_p k
              (fun x -> P x)
              x);
        V.kattr =
          (fun (k, _) x ->
            match_rules_and_recurse config (file, hook, matches)
              !attribute_rules match_at_at k
              (fun x -> At x)
              x);
        V.kfield =
          (fun (k, _) x ->
            match_rules_and_recurse config (file, hook, matches) !fld_rules
              match_fld_fld k
              (fun x -> Fld x)
              x);
        V.kpartial =
          (fun (k, _) x ->
            match_rules_and_recurse config (file, hook, matches) !partial_rules
              match_partial_partial k
              (fun x -> Partial x)
              x);
      }
    in
    let visitor = V.mk_visitor hooks in
    (* later: opti: dont analyze certain ASTs if they do not contain
     * certain constants that interect with the pattern?
     * But this requires to analyze the pattern to extract those
     * constants (name of function, field, etc.).
     *)
    visitor prog;

    !matches |> List.rev
    (* TODO: optimize uniq_by? Too slow? Use a hash?
     * Note that this may not be enough for Semgrep.ml. Indeed, we can have
     * different mini-rules matching the same code with the same metavar,
     * but in Semgrep.ml they get agglomerated under the same rule id, in
     * which case we want to dedup them.
     * old: this uniq_by was introducing regressions in semgrep!
     * See tests/OTHER/rules/regression_uniq_or_ellipsis.go but it's fixed now.
     *)
    |> Common.uniq_by (AST_utils.with_structural_equal Pattern_match.equal)
```

⟨*function* Semgrep_generic.check 61a⟩≡

```
let check ~hook a b c d e =
  Common.profile_code "Semgrep.check" (fun () -> check2 ~hook a b c d e)
```

⟨Semgrep_generic.check2() *populate* expr_rules *and other* 61b⟩≡ (60)

```
rules
|> List.iter (fun rule ->
       (* less: normalize the pattern? *)
       let any = rule.R.pattern in
```

```
⟨Semgrep_generic.check2() apply equivalences to rule pattern any 103d⟩
let cache =
  if !Flag.with_opt_cache then Some (Caching.Cache.create ())
  else None
in
(* Annotate exp, stmt, stmts patterns with the rule strings *)
let push_with_annotation any pattern rules =
  let strs =
    if !Flag.use_bloom_filter then
      Bloom_annotation.list_of_pattern_strings any
    else []
  in
  Common.push (pattern, strs, rule, cache) rules
in
match any with
| E pattern -> push_with_annotation any pattern expr_rules
| S pattern -> push_with_annotation any pattern stmt_rules
| Ss pattern -> push_with_annotation any pattern stmts_rules
| T pattern -> Common.push (pattern, rule, cache) type_rules
| P pattern -> Common.push (pattern, rule, cache) pattern_rules
| At pattern -> Common.push (pattern, rule, cache) attribute_rules
| Fld pattern -> Common.push (pattern, rule, cache) fld_rules
| Partial pattern -> Common.push (pattern, rule, cache) partial_rules
| _ ->
    failwith
      "only expr/stmt/stmts/type/pattern/annotation/field/partial \
       patterns are supported");
```

6.1.1 Visiting and matching expressions

⟨Semgrep_generic.check2() *visitor fields* 62a⟩≡ (60) 63g▷

```
V.kexpr =
  (fun (k, _) x ->
    (* this could be quite slow ... we match many sgrep patterns
     * against an expression recursively
     *)
    !expr_rules
    |> List.iter (fun (pattern, _bf, rule, cache) ->
           let env = MG.empty_environment cache config in
           let matches_with_env = match_e_e rule pattern x env in
           if matches_with_env <> [] then
             (* Found a match *)
             matches_with_env
             |> List.iter (fun (env : MG.tin) ->
                    let env = env.mv.full_env in
                    let range_loc = V.range_of_any (E x) in
                    let tokens = lazy (V.ii_of_any (E x)) in
                    let rule_id = rule_id_of_mini_rule rule in
                    Common.push
                      { PM.rule_id; file; env; range_loc; tokens }
                      matches;
                    hook env tokens));
    (* try the rules on subexpressions *)
    (* this can recurse to find nested matching inside the
     * matched code itself *)
    k x);
```

⟨*signature* Semgrep_generic.match_e_e 62b⟩≡ (447c)

```
  val match_e_e :
    Mini_rule.t -> (AST_generic.expr, AST_generic.expr) Matching_generic.matcher
```

⟨*type* Semgrep_generic.matcher 63a⟩≡ (448 447c)

⟨*function* Semgrep_generic.match_e_e 63b⟩≡ (448)
```
  let match_e_e rule a b env =
    Common.profile_code ("rule:" ^ rule.R.id) (fun () ->
        set_last_matched_rule rule (fun () -> GG.m_expr a b env))
    [@@profiling]
```

⟨*signature* Matching_generic.empty_environment 63c⟩≡ (440)
```
  val empty_environment : tout Caching.Cache.t option -> Config_semgrep.t -> tin
```

⟨*function* Matching_generic.empty_environment 63d⟩≡ (442)
```
  let empty_environment opt_cache config =
    { mv = Env.empty; stmts_match_span = Empty; cache = opt_cache; config }
```

⟨*signature* Semgrep_generic.match_any_any 63e⟩≡ (447c)
```
  (* for unit testing *)
  val match_any_any : (AST_generic.any, AST_generic.any) Matching_generic.matcher
```

⟨*function* Semgrep_generic.match_any_any 63f⟩≡ (448)
```
  (* for unit testing *)
  let match_any_any pattern e env = GG.m_any pattern e env
```

6.1.2 Visiting and matching statements

⟨Semgrep_generic.check2() *visitor fields* 63g⟩+≡ (60) ◁62a 64b▷
```
  (* mostly copy paste of expr code but with the _st functions *)
  V.kstmt =
    (fun (k, _) x ->
      (* old:
       *   match_rules_and_recurse (file, hook, matches)
       *   !stmt_rules match_st_st k (fun x -> S x) x
       * but inlined to handle specially Bloom filter in stmts for now.
       *)
      let visit_stmt () =
        !stmt_rules
        |> List.iter (fun (pattern, _pattern_strs, rule, cache) ->
               let env = MG.empty_environment cache config in
               let matches_with_env = match_st_st rule pattern x env in
               if matches_with_env <> [] then
                 (* Found a match *)
                 matches_with_env
                 |> List.iter (fun (env : MG.tin) ->
                        let env = env.mv.full_env in
                        let range_loc = V.range_of_any (S x) in
                        let tokens = lazy (V.ii_of_any (S x)) in
                        let rule_id = rule_id_of_mini_rule rule in
                        Common.push
                          { PM.rule_id; file; env; range_loc; tokens }
                          matches;
```

```
                    hook env tokens));
      k x
    in
    (* If bloom_filter is not enabled, always visit the statement *)
    (* Otherwise, filter rules first *)
    if not !Flag.use_bloom_filter then visit_stmt ()
    else
      let new_stmt_rules =
        !stmt_rules
        |> List.filter (fun (_, pattern_strs, _, _cache) ->
               must_analyze_statement_bloom_opti_failed pattern_strs x)
      in
      let new_stmts_rules =
        !stmts_rules
        |> List.filter (fun (_, pattern_strs, _, _cache) ->
               must_analyze_statement_bloom_opti_failed pattern_strs x)
      in
      let new_expr_rules =
        !expr_rules
        |> List.filter (fun (_, pattern_strs, _, _cache) ->
               must_analyze_statement_bloom_opti_failed pattern_strs x)
      in
      Common.save_excursion stmt_rules new_stmt_rules (fun () ->
          Common.save_excursion stmts_rules new_stmts_rules (fun () ->
              Common.save_excursion expr_rules new_expr_rules (fun () ->
                  visit_stmt ())));
```

⟨*function* Semgrep_generic.match_st_st 64a⟩≡ (448)

```
  let match_st_st rule a b env =
    Common.profile_code ("rule:" ^ rule.R.id) (fun () ->
        set_last_matched_rule rule (fun () -> GG.m_stmt a b env))
    [@@profiling]
```

⟨Semgrep_generic.check2() *visitor fields* 64b⟩+≡ (60) ◁63g

```
  V.kstmts =
    (fun (k, _) x ->
      (* this is potentially slower than what we did in Coccinelle with
       * CTL. We try every sequences. Hopefully the first statement in
       * the pattern will filter lots of sequences so we need to do
       * the heavy stuff (e.g., handling '...' between statements) rarely.
       *
       * we can't factorize with match_rules_and_recurse because we
       * do things a little bit different with the matched_statements also
       * in matches_with_env here.
       *)
      !stmts_rules
      |> List.iter (fun (pattern, _pattern_strs, rule, cache) ->
             Common.profile_code "Semgrep_generic.kstmts" (fun () ->
                 let env = MG.empty_environment cache config in
                 let matches_with_env =
                   match_sts_sts rule pattern x env
                 in
                 if matches_with_env <> [] then
                   (* Found a match *)
                   matches_with_env
                   |> List.iter (fun (env : MG.tin) ->
                          let span = env.stmts_match_span in
                          match Stmts_match_span.location span with
                          | None -> () (* empty sequence or bug *)
```

```
                        | Some range_loc ->
                            let env = env.mv.full_env in
                            let tokens =
                              lazy
                                (Stmts_match_span.list_original_tokens
                                   span)
                            in
                            let rule_id = rule_id_of_mini_rule rule in
                            Common.push
                              {
                                PM.rule_id;
                                file;
                                env;
                                range_loc;
                                tokens;
                              }
                              matches;
                            hook env tokens)));
      k x);
```

⟨*function* Semgrep_generic.match_sts_sts 65a⟩≡ (448)

```
  let match_sts_sts rule a b env =
    Common.profile_code ("rule:" ^ rule.R.id) (fun () ->
        set_last_matched_rule rule (fun () ->
            (* When matching statements, we need not only to report whether
             * there is match, but also the actual statements that were matched.
             * Indeed, even if we want the implicit '...' at the end of
             * a sequence of statements pattern (AST_generic.Ss) to match all
             * the rest, we don't want to report the whole Ss as a match but just
             * the actually matched subset.
             *
             * TODO? do we need to generate unique key? we don't want
             * nested calls to m_stmts_deep to pollute our metavar? We need
             * to pass the key to m_stmts_deep?
             *)
            let env =
              match b with
              | [] -> env
              | stmt :: _ -> MG.extend_stmts_match_span stmt env
            in
            GG.m_stmts_deep ~less_is_ok:true a b env))
    [@@profiling]
```

6.2 Matcher interface

⟨*signature* Generic_vs_generic.m_expr 65b⟩≡ (428a)

```
  val m_expr : (AST_generic.expr, AST_generic.expr) Matching_generic.matcher
```

⟨*signature* Generic_vs_generic.m_stmt 65c⟩≡ (428a)

```
  val m_stmt : (AST_generic.stmt, AST_generic.stmt) Matching_generic.matcher
```

⟨*type* Matching_generic.matcher 65d⟩≡ (442 440)

```
  (* A matcher is something taking an element A and an element B
   * (for this module A will be the AST of the pattern and B
   * the AST of the program we want to match over), then some environment
```

```
 * information tin, and it will return something (tout) that will
 * represent a match between element A and B.
 *)
(* currently 'a and 'b are usually the same type as we use the
 * same language for the host language and pattern language
 *)
type ('a, 'b) matcher = 'a -> 'b -> tin -> tout
```

⟨*type* Matching_generic.tin 66a⟩≡ (442 440)
```
(* tin is for 'type in' and tout for 'type out' *)
(* incoming environment *)
type tin = {
  mv : Metavariable_capture.t;
  stmts_match_span : Stmts_match_span.t;
  cache : tout Caching.Cache.t option;
  (* TODO: this does not have to be in tout; maybe split tin in 2? *)
  config : Config_semgrep.t;
}
```

⟨*type* Matching_generic.tout 66b⟩≡ (442 440)
```
(* list of possible outcoming matching environments *)
and tout = tin list
```

6.3 Matcher monadic combinators

⟨*signature* Matching_generic.monadic_bind 66c⟩≡ (440)
```
val ( >>= ) : (tin -> tout) -> (unit -> tin -> tout) -> tin -> tout
```

⟨*signature* Matching_generic.return 66d⟩≡ (440)
```
val return : unit -> tin -> tout
```

⟨*signature* Matching_generic.fail 66e⟩≡ (440)
```
val fail : unit -> tin -> tout
```

⟨*function* Matching_generic.return_bis 66f⟩≡ (442)
```
let return () = return
```

⟨*function* Matching_generic.fail_bis 66g⟩≡ (442)
```
let fail () = fail
```

⟨*function* Matching_generic.monadic_bind 66h⟩≡ (442)
```
let (( >>= ) : (tin -> tout) -> (unit -> tin -> tout) -> tin -> tout) =
 fun m1 m2 tin ->
  (* let's get a list of possible environment match (could be
   * the empty list when it didn't match, playing the role None
   * had before)
   *)
  let xs = m1 tin in
  (* try m2 on each possible returned bindings *)
  let xxs = xs |> List.map (fun binding -> m2 () binding) in
  List.flatten xxs
```

⟨*function* Matching_generic.return 67a⟩≡ (442)
```
(* The classical monad combinators *)
let (return : tin -> tout) = fun tin -> [ tin ]
```

⟨*function* Matching_generic.fail 67b⟩≡ (442)
```
let (fail : tin -> tout) =
 fun _tin ->
  if !Flag.debug_matching then failwith "Generic_vs_generic.fail: Match failure";
  []
```

6.4 Advanced combinators

⟨*signature* Matching_generic.TODOOPERATOR2 67c⟩≡ (440)
```
val ( >||> ) : (tin -> tout) -> (tin -> tout) -> tin -> tout
```

⟨*function* Matching_generic.monadic_or 67d⟩≡ (442)
```
(* the disjunctive combinator *)
let (( >||> ) : (tin -> tout) -> (tin -> tout) -> tin -> tout) =
 fun m1 m2 tin ->
  (* CHOICE
        let xs = m1 tin in
        if null xs
        then m2 tin
        else xs
  *)
  (* opti? use set instead of list *)
  m1 tin @ m2 tin
```

⟨*signature* Matching_generic.TODOOPERATOR3 67e⟩≡ (440)
```
val ( >!> ) : (tin -> tout) -> (unit -> tin -> tout) -> tin -> tout
```

⟨*function* Matching_generic.monadic_if_fail 67f⟩≡ (442)
```
(* the if-fail combinator *)
let ( >!> ) m1 else_cont tin =
  match m1 tin with [] -> (else_cont ()) tin | xs -> xs
```

6.5 Matchers for standard types

⟨*signature* Matching_generic.m_bool 67g⟩≡ (440)
```
val m_bool : (bool, bool) matcher
```

⟨*signature* Matching_generic.m_int 67h⟩≡ (440)
```
val m_int : (int, int) matcher
```

⟨*signature* Matching_generic.m_string 67i⟩≡ (440)
```
val m_string : (string, string) matcher
```

⟨*function* Matching_generic.m_bool 68a⟩≡ (442)
```
let m_bool a b = if a = b then return () else fail ()
```

⟨*function* Matching_generic.m_int 68b⟩≡ (442)
```
let m_int a b = if a =|= b then return () else fail ()
```

⟨*function* Matching_generic.m_string 68c⟩≡ (442)
```
let m_string a b = if a =$= b then return () else fail ()
```

⟨*signature* Matching_generic.m_option 68d⟩≡ (440)
```
val m_option : ('a, 'b) matcher -> ('a option, 'b option) matcher
```

⟨*function* Matching_generic.m_option 68e⟩≡ (442)
```
let (m_option : ('a, 'b) matcher -> ('a option, 'b option) matcher) =
 fun f a b ->
  match (a, b) with
  | None, None -> return ()
  | Some xa, Some xb -> f xa xb
  | None, _ | Some _, _ -> fail ()
```

⟨*signature* Matching_generic.m_ref 68f⟩≡ (440)
```
val m_ref : ('a, 'b) matcher -> ('a ref, 'b ref) matcher
```

⟨*function* Matching_generic.m_ref 68g⟩≡ (442)
```
let (m_ref : ('a, 'b) matcher -> ('a ref, 'b ref) matcher) =
 fun f a b ->
  match (a, b) with { contents = xa }, { contents = xb } -> f xa xb
```

⟨*signature* Matching_generic.m_list 68h⟩≡ (440)
```
val m_list : ('a, 'b) matcher -> ('a list, 'b list) matcher
```

⟨*function* Matching_generic.m_list 68i⟩≡ (442)
```
let rec m_list f a b =
  match (a, b) with
  | [], [] -> return ()
  | xa :: aas, xb :: bbs -> f xa xb >>= fun () -> m_list f aas bbs
  | [], _ | _ :: _, _ -> fail ()
```

Chapter 7

Matching a pattern AST with a code AST

7.1 AST-based syntactical matching

7.1.1 Matching tokens

⟨*signature* Matching_generic.m_tok 69a⟩≡ (440)
```
val m_tok : 'a -> 'b -> tin -> tout
```

⟨*function* Matching_generic.m_tok 69b⟩≡ (442)
```
let m_tok a b = m_info a b
```

⟨*signature* Matching_generic.m_info 69c⟩≡ (440)
```
val m_info : 'a -> 'b -> tin -> tout
```

⟨*function* Matching_generic.m_info 69d⟩≡ (442)
```
(* we do not care about position! or differences in space/indent/comment!
 * so we can just  'return ()'
 *)
let m_info _a _b = return ()
```

⟨*signature* Matching_generic.m_wrap 69e⟩≡ (440)
```
val m_wrap : ('a -> 'b -> tin -> tout) -> 'a * 'c -> 'b * 'd -> tin -> tout
```

⟨*function* Matching_generic.m_wrap 69f⟩≡ (442)
```
let m_wrap f a b =
  match (a, b) with
  | (xaa, ainfo), (xbb, binfo) -> f xaa xbb >>= fun () -> m_info ainfo binfo
```

⟨*signature* Matching_generic.m_bracket 69g⟩≡ (440)
```
val m_bracket :
  ('a -> 'b -> tin -> tout) -> 'c * 'a * 'd -> 'e * 'b * 'f -> tin -> tout
```

⟨*function* Matching_generic.m_bracket 69h⟩≡ (442)
```
let m_bracket f (a1, a2, a3) (b1, b2, b3) =
  m_info a1 b1 >>= fun () ->
  f a2 b2 >>= fun () -> m_info a3 b3
```

7.1.2 Matching identifiers

⟨*function* Generic_vs_generic.m_ident 70a⟩≡ (428b)
```
  (* coupling: modify also m_ident_and_id_info *)
  let m_ident a b =
    match (a, b) with
    ⟨Generic_vs_generic.m_ident() metavariable case 77a⟩
    ⟨Generic_vs_generic.m_ident() regexp case 89a⟩

    (* general case *)
    | a, b -> (m_wrap m_string) a b
```

7.1.3 Matching expressions

⟨*function* Generic_vs_generic.m_expr 70b⟩≡ (428b)
```
  and m_expr a b =
    match (a, b) with
    (* the order of the matches matters! take care! *)
    ⟨Generic_vs_generic.m_expr() disjunction case 103a⟩
    ⟨Generic_vs_generic.m_expr() resolving alias case 108d⟩
    ⟨Generic_vs_generic.m_expr() metavariable case 77b⟩
    ⟨Generic_vs_generic.m_expr() typed metavariable case 107b⟩
    ⟨Generic_vs_generic.m_expr() ellipsis cases 84d⟩

    (* must be before constant propagation case below *)
    | A.L a1, B.L b1 -> m_literal a1 b1
    ⟨Generic_vs_generic.m_expr() propagated constant case 107e⟩

    ⟨Generic_vs_generic.m_expr() sequencable container cases 81⟩
    | ( A.Container (((A.Set | A.Dict) as a1), (_, a2, _)),
        B.Container (((B.Set | B.Dict) as b1), (_, b2, _)) ) ->
        m_container_set_or_dict_unordered_elements (a1, a2) (b1, b2)
    ⟨Generic_vs_generic.m_expr() interpolated strings case 85d⟩
    (* The pattern '$X = 1 + 2 + ...' is parsed as '$X = (1 + 2) + ...', but
     * if the concrete code is 'foo = 1 + 2 + 3 + 4', this will naively not
     * match because (1+2)+... will be matched against ((1+2)+3)+4 and fails.
     * The ellipsis operator with binary operators should be more flexible
     * and allows any number of additional arguments, which when translated
     * in Call() means we need to go deeper.
     *)
    | ( A.Call
          ( A.IdSpecial (A.Op aop, _toka),
            (_, [ A.Arg a1; A.Arg (A.Ellipsis _tdots) ], _) ),
        B.Call (B.IdSpecial (B.Op bop, _tokb), (_, [ B.Arg b1; B.Arg _b2 ], _)) )
      ->
        m_arithmetic_operator aop bop >>= fun () ->
        m_expr a1 b1 >!> fun () ->
        (* try again deeper on b1 *)
        m_expr a b1
    (* boilerplate *)
    (* TODO: via m_name! and miss IdQualfied vs IdQualified otherwise *)
    | A.N (A.Id (a1, a2)), B.N (B.Id (b1, b2)) ->
        m_ident a1 b1 >>= fun () -> m_id_info a2 b2
    | A.Call (a1, a2), B.Call (b1, b2) ->
        m_expr a1 b1 >>= fun () -> m_arguments a2 b2
    | A.Assign (a1, at, a2), B.Assign (b1, bt, b2) -> (
        m_expr a1 b1
        >>= (fun () -> m_tok at bt >>= fun () -> m_expr a2 b2)
```

```
      (* If the code has tuples as b1 and b2 and the lengths of
       * the tuples are equal, create a tuple of (variable, value)
       * pairs and try to match the pattern with each entry in the tuple.
       * This should enable multiple assignments if the number of
       * variables and values are equal. *)
      >||>
      match (b1, b2) with
      | B.Tuple (_, vars, _), B.Tuple (_, vals, _)
        when List.length vars = List.length vals ->
          let create_assigns expr1 expr2 = B.Assign (expr1, bt, expr2) in
          let mult_assigns = List.map2 create_assigns vars vals in
          let rec aux xs =
            match xs with [] -> fail () | x :: xs -> m_expr a x >||> aux xs
          in
          aux mult_assigns
      | _, _ -> fail () )
  | A.DotAccess (a1, at, a2), B.DotAccess (b1, bt, b2) ->
      m_expr a1 b1 >>= fun () ->
      m_tok at bt >>= fun () -> m_name_or_dynamic a2 b2
  (* <a1> ... vs o.m1().m2().m3().
   * Remember than o.m1().m2().m3() is parsed as (((o.m1()).m2()).m3())
   *)
  | A.DotAccessEllipsis (a1, _a2), (B.DotAccess _ | B.Call (B.DotAccess _, _))
    ->
      (* => o, [m3();m2();m1() *)
      let obj, ys = obj_and_method_calls_of_expr b in
      (* the method chain ellipsis can match 0 or more of those method calls *)
      let candidates = all_suffix_of_list ys in
      let rec aux xxs =
        match xxs with
        | [] -> fail ()
        | xs :: xxs ->
            let b = expr_of_obj_and_method_calls (obj, xs) in
            m_expr a1 b >||> aux xxs
      in
      aux candidates
  | A.ArrayAccess (a1, a2), B.ArrayAccess (b1, b2) ->
      m_expr a1 b1 >>= fun () -> m_bracket m_expr a2 b2
  ⟨Generic_vs_generic.m_expr() boilerplate cases 227e⟩
```

⟨*function* Generic_vs_generic.m_literal 71⟩≡ (428b)

```
and m_literal a b =
  match (a, b) with
  ⟨Generic_vs_generic.m_literal() ellipsis case 85b⟩
  ⟨Generic_vs_generic.m_literal() regexp case 88b⟩
  | A.Atom (s, tok), B.Atom _b1
    when s =~ "^:\\(.*\\)" && MV.is_metavar_name (Common.matched1 s) ->
      let mvar = Common.matched1 s in
      envf (mvar, tok) (MV.E (B.L b))
  (* boilerplate *)
  | A.Unit a1, B.Unit b1 -> m_tok a1 b1
  | A.Bool a1, B.Bool b1 -> (m_wrap m_bool) a1 b1
  | A.Int a1, B.Int b1 -> m_wrap_m_int_opt a1 b1
  | A.Float a1, B.Float b1 -> m_wrap_m_float_opt a1 b1
  | A.Imag a1, B.Imag b1 -> (m_wrap m_string) a1 b1
  | A.Ratio a1, B.Ratio b1 -> (m_wrap m_string) a1 b1
  | A.Atom a1, B.Atom b1 -> (m_wrap m_string) a1 b1
  | A.Char a1, B.Char b1 -> (m_wrap m_string) a1 b1
  | A.Regexp a1, B.Regexp b1 -> (m_wrap m_string) a1 b1
  | A.Null a1, B.Null b1 -> m_tok a1 b1
```

```
    | A.Undefined a1, B.Undefined b1 -> m_tok a1 b1
    | A.Unit _, _
    | A.Bool _, _
    | A.Int _, _
    | A.Float _, _
    | A.Char _, _
    | A.String _, _
    | A.Regexp _, _
    | A.Null _, _
    | A.Undefined _, _
    | A.Imag _, _
    | A.Ratio _, _
    | A.Atom _, _ ->
        fail ()
```

⟨*function* Generic_vs_generic.m_container_operator 72a⟩≡ (428b)

```
  (* coupling: if you add a constructor in AST_generic.container,
   * you probrably need to update m_container_set_or_dict_unordered_elements
   * and m_expr to handle this new kind of container.
   *)
  and m_container_operator a b =
    match (a, b) with
    (* boilerplate *)
    | A.Array, B.Array -> return ()
    | A.List, B.List -> return ()
    | A.Set, B.Set -> return ()
    | A.Dict, B.Dict -> return ()
    | A.Array, _ | A.List, _ | A.Set, _ | A.Dict, _ -> fail ()
```

⟨*function* Generic_vs_generic.m_argument 72b⟩≡ (428b)

```
  and m_argument a b =
    match (a, b) with
    (* TODO: iso on keyword argument, keyword is optional in pattern *)

    (* boilerplate *)
    | A.Arg a1, B.Arg b1 -> m_expr a1 b1
    ⟨Generic_vs_generic.m_argument() boilerplate cases 228⟩
```

⟨*function* Generic_vs_generic.m_field_ident 72c⟩≡ (428b)

```
  and m_name_or_dynamic a b =
    match (a, b) with
    (* TODO: factorize in m_name *)
    | A.EN (A.Id ((str, tok), a2)), B.EN (B.Id (idb, b2))
      when MV.is_metavar_name str ->
        (* a bit OCaml specific, cos only ml_to_generic tags id_type in pattern *)
        let* () = m_type_option_with_hook idb !(a2.A.id_type) !(b2.B.id_type) in
        let* () = m_id_info a2 b2 in
        envf (str, tok) (MV.Id (idb, Some b2))
    | A.EN (A.Id ((str, tok), _idinfoa)), b when MV.is_metavar_name str ->
        let e = H.name_or_dynamic_to_expr b None in
        envf (str, tok) (MV.E e)
    | A.EN (A.Id (a, idinfoa)), B.EN (B.Id (b, idinfob)) ->
        m_ident_and_id_info (a, idinfoa) (b, idinfob)
    (* boilerplate *)
    ⟨Generic_vs_generic.m_field_ident() boilerplate cases 229c⟩
```

7.1.4 Matching statements

⟨*function* Generic_vs_generic.m_stmt 73a⟩≡ (428b)
```
  and m_stmt a b =
    match (a.s, b.s) with
    (* the order of the matches matters! take care! *)
    ⟨Generic_vs_generic.m_stmt() disjunction case 103b⟩
    ⟨Generic_vs_generic.m_stmt() metavariable case 78c⟩
    ⟨Generic_vs_generic.m_stmt() ellipsis cases 84e⟩
    ⟨Generic_vs_generic.m_stmt() deep matching cases 89b⟩
    ⟨Generic_vs_generic.m_stmt() builtin equivalences cases 96g⟩

    (* boilerplate *)
    | A.If (a0, a1, a2, a3), B.If (b0, b1, b2, b3) ->
        m_tok a0 b0 >>= fun () ->
        (* too many regressions doing m_expr_deep by default; Use DeepEllipsis *)
        m_expr a1 b1 >>= fun () ->
        m_block a2 b2 >>= fun () ->
        (* less-is-more: *)
        m_option_none_can_match_some m_block a3 b3
    | A.While (a0, a1, a2), B.While (b0, b1, b2) ->
        m_tok a0 b0 >>= fun () ->
        m_expr a1 b1 >>= fun () -> m_stmt a2 b2
    ⟨Generic_vs_generic.m_stmt boilerplate cases 73b⟩
```

⟨Generic_vs_generic.m_stmt *boilerplate cases* 73b⟩≡ (73a) 232e▷
```
  | A.DefStmt a1, B.DefStmt b1 -> m_definition a1 b1
  | A.DirectiveStmt a1, B.DirectiveStmt b1 -> m_directive a1 b1
```

7.1.5 Matching definitions

⟨*function* Generic_vs_generic.m_definition 73c⟩≡ (428b)
```
  and m_definition a b =
    match (a, b) with
    | (a1, a2), (b1, b2) ->
        (* subtle: if you change the order here, so that we execute m_entity
         * only when the definition_kind matches, this helps to avoid
         * calls to ocamllsp when you put a type constraint on a function.
         * Indeed, we will call ocamllsp only for FuncDef.
         * This also avoids to call ocamllsp on type definition entities,
         * which can leads to errors in type_of_string.
         *)
        let* () = m_entity a1 b1 in
        let* () = m_definition_kind a2 b2 in
        return ()
```

⟨*function* Generic_vs_generic.m_definition_kind 73d⟩≡ (428b)
```
  and m_definition_kind a b =
    match (a, b) with
    (* boilerplate *)
    | A.FuncDef a1, B.FuncDef b1 -> m_function_definition a1 b1
    | A.VarDef a1, B.VarDef b1 -> m_variable_definition a1 b1
    | A.FieldDefColon a1, B.FieldDefColon b1 -> m_variable_definition a1 b1
    | A.ClassDef a1, B.ClassDef b1 -> m_class_definition a1 b1
    ⟨Generic_vs_generic.m_definition_kind boilerplate cases 235c⟩
```

Matching function definitions

⟨*function* Generic_vs_generic.m_function_definition 74a⟩≡ (428b)
```
  and m_function_definition a b =
    match (a, b) with
    | ( { A.fparams = a1; frettype = a2; fbody = a3; fkind = a4 },
        { B.fparams = b1; frettype = b2; fbody = b3; fkind = b4 } ) ->
        m_parameters a1 b1 >>= fun () ->
        (m_option_none_can_match_some m_type_) a2 b2 >>= fun () ->
        m_stmt a3 b3 >>= fun () -> m_wrap m_function_kind a4 b4
```

⟨*function* Generic_vs_generic.m_parameter 74b⟩≡ (428b)
```
  and m_parameter a b =
    match (a, b) with
    (* boilerplate *)
    | A.ParamClassic a1, B.ParamClassic b1 -> m_parameter_classic a1 b1
    | A.ParamRest (a1, a2), B.ParamRest (b1, b2) ->
        let* () = m_tok a1 b1 in
        m_parameter_classic a2 b2
    | A.ParamHashSplat (a1, a2), B.ParamHashSplat (b1, b2) ->
        let* () = m_tok a1 b1 in
        m_parameter_classic a2 b2
    ⟨Generic_vs_generic.m_parameter boilerplate cases 236c⟩
```

⟨*function* Generic_vs_generic.m_parameter_classic 74c⟩≡ (428b)
```
  and m_parameter_classic a b =
    match (a, b) with
    ⟨Generic_vs_generic.m_parameter_classic metavariable case 77d⟩

    (* boilerplate *)
    | ( { A.pname = a1; pdefault = a2; ptype = a3; pattrs = a4; pinfo = a5 },
        { B.pname = b1; pdefault = b2; ptype = b3; pattrs = b4; pinfo = b5 } ) ->
        (m_option m_ident) a1 b1 >>= fun () ->
        (m_option m_expr) a2 b2 >>= fun () ->
        (m_option_none_can_match_some m_type_) a3 b3 >>= fun () ->
        m_list_in_any_order ~less_is_ok:true m_attribute a4 b4 >>= fun () ->
        m_id_info a5 b5
```

Matching variable definitions

⟨*function* Generic_vs_generic.m_variable_definition 74d⟩≡ (428b)
```
  and m_variable_definition a b =
    match (a, b) with
    (* boilerplate *)
    | { A.vinit = a1; vtype = a2 }, { B.vinit = b1; vtype = b2 } ->
        (m_option m_expr) a1 b1 >>= fun () ->
        (m_option_none_can_match_some m_type_) a2 b2
```

Matching class definitions

⟨*function* Generic_vs_generic.m_class_definition 74e⟩≡ (428b)
```
  and m_class_definition a b =
    match (a, b) with
    | ( {
          A.ckind = a1;
          cextends = a2;
```

```
        cimplements = a3;
        cmixins = a5;
        cbody = a4;
        cparams = a6;
      },
      {
        B.ckind = b1;
        cextends = b2;
        cimplements = b3;
        cmixins = b5;
        cbody = b4;
        cparams = b6;
      } ) ->
      m_class_kind a1 b1 >>= fun () ->
      (* TODO: use also m_list_in_any_order? regressions for python? *)
      m_list__m_type_ a2 b2 >>= fun () ->
      m_list__m_type_any_order a3 b3 >>= fun () ->
      m_list__m_type_any_order a5 b5 >>= fun () ->
      m_parameters a6 b6 >>= fun () -> m_bracket m_fields a4 b4
```

⟨*function* Generic_vs_generic.m_fields 75a⟩≡ (428b)

```
  and m_fields (xsa : A.field list) (xsb : A.field list) =
    (* let's filter the '...' *)
    let xsa =
      xsa
      |> Common.exclude (function
           | A.FieldStmt { s = A.ExprStmt (A.Ellipsis _, _); _ } -> true
           | _ -> false)
    in
    m_list__m_field xsa xsb
```

⟨*function* Generic_vs_generic.m_field 75b⟩≡ (428b)

```
  and m_field a b =
    match (a, b) with
    (* boilerplate *)
    | A.FieldStmt a1, B.FieldStmt b1 -> m_stmt a1 b1
    ⟨Generic_vs_generic.m_field boilerplate cases 236e⟩
```

7.1.6 Matching types

⟨*function* Generic_vs_generic.m_type_ 75c⟩≡ (428b)

```
  and m_type_ a b =
    match (a, b) with
    (* equivalence: name resolving! *)
    (* TODO: factorize in a new m_name? *)
    | ( a,
        B.TyN
          (B.Id
            ( idb,
              {
                B.id_resolved =
                  {
                    contents =
                      Some
                        ( ( B.ImportedEntity dotted
                          | B.ImportedModule (B.DottedName dotted) ),
                          _sid );
```

```
              };
            _;
          } )) ) ->
      m_type_ a (B.TyN (B.Id (idb, B.empty_id_info ())))
      >|> (* try this time a match with the resolved entity *)
      m_type_ a (B.TyN (H.name_of_ids dotted))
  ⟨Generic_vs_generic.m_type_ metavariable case 79a⟩
  (* dots: *)
  | A.TyEllipsis _, _ -> return ()
  (* boilerplate *)
  | A.TyBuiltin a1, B.TyBuiltin b1 -> (m_wrap m_string) a1 b1
  | A.TyFun (a1, a2), B.TyFun (b1, b2) ->
      (m_list m_parameter) a1 b1 >>= fun () -> m_type_ a2 b2
  | A.TyArray (a1, a2), B.TyArray (b1, b2) ->
      m_bracket (m_option m_expr) a1 b1 >>= fun () -> m_type_ a2 b2
  | A.TyTuple a1, B.TyTuple b1 ->
      (*TODO: m_list__m_type_ ? *)
      (m_bracket (m_list m_type_)) a1 b1
  ⟨Generic_vs_generic.m_type_ boilerplate cases 231a⟩
```

7.1.7 Matching Attributes

```
⟨function Generic_vs_generic.m_attribute 76a⟩≡                    (428b)
  and m_attribute a b =
    match (a, b) with
    ⟨Generic_vs_generic.m_attribute resolving alias case 109b⟩

    (* boilerplate *)
    | A.KeywordAttr a1, B.KeywordAttr b1 -> m_wrap m_keyword_attribute a1 b1
    | A.NamedAttr (a0, a1, a2), B.NamedAttr (b0, b1, b2) ->
        m_tok a0 b0 >>= fun () ->
        m_name a1 b1 >>= fun () -> m_bracket m_list__m_argument a2 b2
    | A.OtherAttribute (a1, a2), B.OtherAttribute (b1, b2) ->
        m_other_attribute_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
    | A.KeywordAttr _, _ | A.NamedAttr _, _ | A.OtherAttribute _, _ -> fail ()
```

7.1.8 Matching directives

```
⟨function Generic_vs_generic.m_directive_basic 76b⟩≡              (428b)
  and m_directive_basic a b =
    match (a, b) with
    ⟨Generic_vs_generic.m_directive_basic import cases 95b⟩
    (* boilerplate *)
    ⟨Generic_vs_generic.m_directive_basic boilerplate cases 238d⟩
```

7.1.9 Matching other constructs

Matching patterns

```
⟨function Generic_vs_generic.m_pattern 76c⟩≡                      (428b)
  and m_pattern a b =
    match (a, b) with
    ⟨Generic_vs_generic.m_pattern() disjunction case 103c⟩
    ⟨Generic_vs_generic.m_pattern() metavariable case 79b⟩

    (* dots: *)
```

```
  | A.PatEllipsis _, _ -> return ()
  (* boilerplate *)
  ⟨Generic_vs_generic.m_pattern boilerplate cases 234h⟩
```

7.2 Metavariables

7.2.1 Identitifer metavariables

⟨Generic_vs_generic.m_ident() *metavariable case* 77a⟩≡ (70a)

```
  (* metavar: *)
  | (str, tok), b when MV.is_metavar_name str ->
      envf (str, tok) (MV.Id (b, None))
```

7.2.2 Expression metavariables

⟨Generic_vs_generic.m_expr() *metavariable case* 77b⟩≡ (70b)

```
  ⟨Generic_vs_generic.m_expr() forbidden metavariable case 77c⟩
  (* TODO: factorize in m_name? *)
  | A.N (A.Id ((str, tok), _id_info)), B.N (B.Id (idb, id_infob))
    when MV.is_metavar_name str ->
      envf (str, tok) (MV.Id (idb, Some id_infob))
  | A.N (A.Id ((str, tok), _id_info)), e2 when MV.is_metavar_name str ->
      envf (str, tok) (MV.E e2)
```

⟨Generic_vs_generic.m_expr() *forbidden metavariable case* 77c⟩≡ (77b)

```
  (* $X should not match an IdSpecial in a call context,
   * otherwise $X(...) would match a+b because this is transformed in a
   * Call(IdSpecial Plus, ...).
   * bugfix: note that we must forbid that only in a Call context; we want
   * $THIS to match IdSpecial (This) for example.
   *)
  | ( A.Call (A.N (A.Id ((str, _tok), _id_info)), _argsa),
      B.Call (B.IdSpecial _, _argsb) )
    when MV.is_metavar_name str ->
      fail ()
  (* metavar: *)
  (* Matching a generic Id metavariable to an IdSpecial will fail as it is missing the token
   * info; instead the Id should match Call(IdSpecial _, _)
   *)
  | A.N (A.Id ((str, _), _)), B.IdSpecial (B.ConcatString _, _)
  | A.N (A.Id ((str, _), _)), B.IdSpecial (B.Instanceof, _)
  | A.N (A.Id ((str, _), _)), B.IdSpecial (B.New, _)
    when MV.is_metavar_name str ->
      fail ()
```

7.2.3 Parameter metavariables

⟨Generic_vs_generic.m_parameter_classic *metavariable case* 77d⟩≡ (74c)

```
  (* bugfix: when we use a metavar to match a parameter, as in foo($X): ...
   * and later we use $X again to match a name, the $X is first an ident and
   * later an expression, which would prevent a match. Instead we need to
   * make $X an expression early on
   *)
  | ( { A.pname = Some a1; pdefault = a2; ptype = a3; pattrs = a4; pinfo = a5 },
      { B.pname = Some b1; pdefault = b2; ptype = b3; pattrs = b4; pinfo = b5 }
    ) ->
```

```
    m_ident_and_id_info (a1, a5) (b1, b5) >>= fun () ->
    (m_option m_expr) a2 b2 >>= fun () ->
    (m_type_option_with_hook b1) a3 b3 >>= fun () ->
    m_list_in_any_order ~less_is_ok:true m_attribute a4 b4
```

⟨*function* `Generic_vs_generic.m_ident_and_id_info_add_in_env_Expr` 78a⟩≡ (428b)

```
and m_ident_and_id_info (a1, a2) (b1, b2) =
  (* metavar: *)
  match (a1, b1) with
  | (str, tok), b when MV.is_metavar_name str ->
      (* a bit OCaml specific, cos only ml_to_generic tags id_type in pattern *)
      m_type_option_with_hook b1 !(a2.A.id_type) !(b2.B.id_type) >>= fun () ->
      m_id_info a2 b2 >>= fun () -> envf (str, tok) (MV.Id (b, Some b2))
  (* same code than for m_ident *)
  ⟨Generic_vs_generic.m_ident() regexp case 89a⟩
  (* general case *)
  (* todo: we should check m_id_info, but anyway this function is currently
   * a nop *)
  | a, b -> (m_wrap m_string) a b
```

7.2.4 Entity metavariables

⟨*function* `Generic_vs_generic.m_entity` 78b⟩≡ (428b)

```
and m_entity a b =
  match (a, b) with
  (* bugfix: when we use a metavar to match an entity, as in $X(...): ...
   * and later we use $X again to match a name, the $X is first an ident and
   * later an expression, which would prevent a match. Instead we need to
   * make $X an expression early on.
   * update: actually better to use a special MV.Id for that.
   *)
  | ( { A.name = a1; attrs = a2; tparams = a4 },
      { B.name = b1; attrs = b2; tparams = b4 } ) ->
      m_name_or_dynamic a1 b1 >>= fun () ->
      m_list_in_any_order ~less_is_ok:true m_attribute a2 b2 >>= fun () ->
      (m_list m_type_parameter) a4 b4
```

7.2.5 Statement metavariables

⟨`Generic_vs_generic.m_stmt()` *metavariable case* 78c⟩≡ (73a)

```
(* metavar: *)
(* Note that we can't consider $S a statement metavariable only if the
 * semicolon is a fake one. Indeed in many places we have patterns
 * like 'if(...) $S;' because 'if(...) $S' would not parse.
 * alt: parse if(...) $S without the ending semicolon?
 * But at least we can try to match $S as a statement metavar
 * _or_ an expression metavar with >||>. below
 *)
| A.ExprStmt ((A.N (A.Id ((str, tok), _id_info)) as suba), sc), _b
  when MV.is_metavar_name str -> (
    envf (str, tok) (MV.S b)
    >||>
    match b.s with
    | B.ExprStmt (subb, _) when not (Parse_info.is_fake sc) ->
        m_expr suba subb
    | _ -> fail () )
```

7.2.6 Type metavariables

⟨Generic_vs_generic.m_type_ *metavariable case* 79a⟩≡ (75c)
```
  | A.TyN (A.Id ((str, tok), _id_info)), B.TyN (B.Id (idb, id_infob))
    when MV.is_metavar_name str ->
      envf (str, tok) (MV.Id (idb, Some id_infob))
  (* TODO: TyId vs TyId => add MV.Id *)
  | A.TyN (A.Id ((str, tok), _id_info)), t2 when MV.is_metavar_name str ->
      envf (str, tok) (MV.T t2)
```

7.2.7 Field name metavariables

7.2.8 Keyword argument metavariables

7.2.9 XML attribute metavariables

7.2.10 Other metavariables

Pattern metavariables

⟨Generic_vs_generic.m_pattern() *metavariable case* 79b⟩≡ (76c)
```
  (* metavar: *)
  (* less: A.PatId vs B.PatId? Use MV.Id then ? *)
  | A.PatId ((str, tok), _id_info), b2 when MV.is_metavar_name str -> (
      try
        let e2 = H.pattern_to_expr b2 in
        envf (str, tok) (MV.E e2)
        (* this can happen with PatAs in exception handler in Python *)
      with H.NotAnExpr -> envf (str, tok) (MV.P b2) )
```

7.3 Metavariables equality

⟨*signature* Matching_generic.envf 79c⟩≡ (440)
```
  val envf : (Metavariable.mvar AST_generic.wrap, Metavariable.mvalue) matcher
```

⟨*function* Matching_generic.envf 79d⟩≡ (442)
```
  let (envf : (MV.mvar AST.wrap, MV.mvalue) matcher) =
   fun (mvar, _imvar) any tin ->
    match check_and_add_metavar_binding (mvar, any) tin with
    | None ->
        ⟨Matching_generic.envf if verbose when fail 79e⟩
        fail tin
    | Some new_binding ->
        ⟨Matching_generic.envf if verbose when success 79f⟩
        return new_binding
```

⟨Matching_generic.envf *if* verbose *when fail* 79e⟩≡ (79d)
```
  logger#ldebug (lazy (spf "envf: fail, %s (%s)" mvar (MV.str_of_mval any)));
```

⟨Matching_generic.envf *if* verbose *when success* 79f⟩≡ (79d)
```
  logger#ldebug
    (lazy (spf "envf: success, %s (%s)" mvar (MV.str_of_mval any)));
```

⟨*signature* Matching_generic.check_and_add_metavar_binding 79g⟩≡ (440)
```
  val check_and_add_metavar_binding :
    Metavariable.mvar * Metavariable.mvalue -> tin -> tin option
```

⟨*function* Matching_generic.check_and_add_metavar_binding 80a⟩≡ (442)

```
let check_and_add_metavar_binding ((mvar : MV.mvar), valu) (tin : tin) =
  match Common2.assoc_opt mvar tin.mv.full_env with
  | Some valu' ->
      (* Should we use generic_vs_generic itself for comparing the code?
       * Hmmm, we can't because it leads to a circular dependencies.
       * Moreover here we know both valu and valu' are regular code,
       * not patterns, so we can just use the generic '=' of OCaml.
       *)
      if equal_ast_binded_code tin.config valu valu' then Some tin
        (* valu remains the metavar witness *)
      else None
  | None ->
      (* 'backrefs' is the set of metavariables that may be referenced later
         in the pattern. It's inherited from the last stmt pattern,
         so it might contain a few extra members.
      *)
      (* first time the metavar is bound, just add it to the environment *)
      Some (add_mv_capture mvar valu tin)
```

⟨*function* Matching_generic.equal_ast_binded_code 80b⟩≡ (442)

```
(* pre: both 'a' and 'b' contains only regular code; there are no
 * metavariables inside them.
 *)
let rec equal_ast_binded_code (config : Config_semgrep.t) (a : MV.mvalue)
    (b : MV.mvalue) : bool =
  let res =
    match (a, b) with
    (* if one of the two IDs is not resolved, then we allow
     * a match, so a pattern like 'self.$FOO = $FOO' matches
     * code like 'self.foo = foo'.
     * Maybe we should not ... but let's try.
     *
     * At least we don't allow a resolved id with a precise sid to match
     * another id with a different sid (same id but in different scope),
     * which we rely on with our deep stmt matching hacks.
     *
     * TODO: relax even more and allow some id_resolved EnclosedVar (a field)
     * to match anything?
     *)
    | ( MV.Id ((s1, _), Some { AST.id_resolved = { contents = None }; _ }),
        MV.Id ((s2, _), _) )
    | ( MV.Id ((s1, _), _),
        MV.Id ((s2, _), Some { AST.id_resolved = { contents = None }; _ }) ) ->
        s1 = s2
    (* A variable occurrence that is known to have a constant value is equal to
     * that same constant value.
     *
     * THINK: We could also equal two different variable occurrences that happen
     * to have the same constant value. *)
    | ( MV.E (A.L a_lit),
        MV.Id (_, Some { B.id_constness = { contents = Some (B.Lit b_lit) }; _ })
      )
    | ( MV.Id (_, Some { A.id_constness = { contents = Some (A.Lit a_lit) }; _ }),
        MV.E (B.L b_lit) )
      when config.constant_propagation ->
        A.equal_literal a_lit b_lit
    (* general case, equality modulo-position-and-constness.
     * TODO: in theory we should use user-defined equivalence to allow
```

```
     * equality modulo-equivalence rewriting!
     *)
    | MV.Id _, MV.Id _
    | MV.E _, MV.E _
    | MV.S _, MV.S _
    | MV.P _, MV.P _
    | MV.T _, MV.T _
    | MV.Args _, MV.Args _ ->
        (* Note that because we want to retain the position information
         * of the matched code in the environment (e.g. for the -pvar
         * sgrep command line argument), we can not just use the
         * generic '=' OCaml operator as 'a' and 'b' may represent
         * the same code but they will contain leaves in their AST
         * with different position information.

         * old: So before doing
         * the comparison we just need to remove/abstract-away
         * the line number information in each ASTs.
         * let a = MV.abstract_position_info_mval a in
         * let b = MV.abstract_position_info_mval b in
         * a =*= b
         *)
        (* This will perform equality but not care about:
         * - position information (see adhoc AST_generic.equal_tok)
         * - id_constness (see the special @equal for id_constness)
         *)
        MV.Structural.equal_mvalue a b
    | MV.Id _, MV.E (A.N (A.Id (b_id, b_id_info))) ->
        (* TODO still needed now that we have the better MV.Id of id_info? *)
        (* TOFIX: regression if remove this code *)
        (* Allow identifier nodes to match pure identifier expressions *)

        (* You should prefer to add metavar as expression (A.E), not id (A.I),
         * (see Generic_vs_generic.m_ident_and_id_info_add_in_env_Expr)
         * but in some cases you have no choice and you need to match an expression
         * metavar with an id metavar.
         * For example, we want the pattern 'const $X = foo.$X' to match 'const bar = foo.bar'
         * (this is useful in the Javascript transpilation context of complex pattern parameter).
         *)
        equal_ast_binded_code config a (MV.Id (b_id, Some b_id_info))
    | _, _ -> false
  in

  if not res then
    logger#ldebug
      (lazy (spf "A = %s\nB = %s\n" (MV.str_of_mval a) (MV.str_of_mval b)));
  res
```

7.4 Ellipsis

7.4.1 Ordered sequences

Ellipsis in Containers

⟨Generic_vs_generic.m_expr() *sequencable container cases* 81⟩≡ (70b)

```
  | A.Container (A.Array, a2), B.Container (B.Array, b2) ->
      (m_bracket m_container_ordered_elements) a2 b2
```

```
  | A.Container (A.List, a2), B.Container (B.List, b2) ->
      (m_bracket m_container_ordered_elements) a2 b2
  | A.Tuple a1, B.Tuple b1 -> (m_container_ordered_elements |> m_bracket) a1 b1
```

⟨*function* Generic_vs_generic.m_container_ordered_elements 82a⟩≡ (428b)

```
  and m_container_ordered_elements a b =
    m_list_with_dots m_expr
      (function A.Ellipsis _ -> true | _ -> false)
      false (* empty list can not match non-empty list *) a b
```

⟨*signature* Matching_generic.m_list_with_dots 82b⟩≡ (440)

```
  val m_list_with_dots :
    ('a, 'b) matcher -> ('a -> bool) -> bool -> ('a list, 'b list) matcher
```

⟨*function* Matching_generic.m_list_with_dots 82c⟩≡ (442)

```
  let rec m_list_with_dots f is_dots less_is_ok xsa xsb =
    match (xsa, xsb) with
    | [], [] -> return ()
    ⟨Matching_generic.m_list_with_dots()¿ empty list vs list case 94a⟩
    ⟨Matching_generic.m_list_with_dots()¿ ellipsis cases 82d⟩
    (* the general case *)
    | xa :: aas, xb :: bbs ->
        f xa xb >>= fun () -> m_list_with_dots f is_dots less_is_ok aas bbs
    | [], _ | _ :: _, _ -> fail ()
```

⟨Matching_generic.m_list_with_dots()¿ *ellipsis cases* 82d⟩≡ (82c)

```
  (* dots: '...', can also match no argument *)
  | [ a ], [] when is_dots a -> return ()
  | a :: xsa, xb :: xsb when is_dots a ->
      (* can match nothing *)
      m_list_with_dots f is_dots less_is_ok xsa (xb :: xsb)
      >||> (* can match more *)
      m_list_with_dots f is_dots less_is_ok (a :: xsa) xsb
```

Ellipsis in parameters

⟨*function* Generic_vs_generic.m_parameters 82e⟩≡ (428b)

```
  and m_parameters a b =
    m_list_with_dots m_parameter
      (function A.ParamEllipsis _ -> true | _ -> false)
      false (* empty list can not match non-empty list *) a b
```

Ellipsis in arguments

⟨*function* Generic_vs_generic.m_arguments 82f⟩≡ (428b)

```
  and m_arguments a b =
    match (a, b) with a, b -> m_bracket m_list__m_argument a b
```

⟨*function* Generic_vs_generic.m_list__m_argument 83a⟩≡ (428b)
```
  and m_list__m_argument (xsa : A.argument list) (xsb : A.argument list) =
    match (xsa, xsb) with
    | [], [] -> return ()
    ⟨Generic_vs_generic.m_list__m_argument() ellipsis cases 83b⟩
    ⟨Generic_vs_generic.m_list__m_argument() ArgKwd pattern case 86b⟩
    (* the general case *)
    | xa :: aas, xb :: bbs ->
        m_argument xa xb >>= fun () -> m_list__m_argument aas bbs
    | [], _ | _ :: _, _ -> fail ()
```

⟨Generic_vs_generic.m_list__m_argument() *ellipsis cases* 83b⟩≡ (83a)
```
  (* dots: ..., can also match no argument *)
  | [ A.Arg (A.Ellipsis _i) ], [] -> return ()
  | A.Arg (A.N (A.Id ((s, tok), _idinfo))) :: xsa, xb :: xsb
    when MV.is_metavar_ellipsis s ->
      (* can match 1 or more arguments (or 0 is ok too?) *)
      let candidates = inits_and_rest_of_list (xb :: xsb) in
      let rec aux xs =
        match xs with
        | [] -> fail ()
        | (inits, rest) :: xs ->
            envf (s, tok) (MV.Args inits)
            >>= (fun () -> m_list__m_argument xsa rest)
            >||> aux xs
      in
      aux candidates
  | A.Arg (A.Ellipsis i) :: xsa, xb :: xsb ->
      (* can match nothing *)
      m_list__m_argument xsa (xb :: xsb)
      >||> (* can match more *)
      m_list__m_argument (A.Arg (A.Ellipsis i) :: xsa) xsb
```

Ellipsis between statements

⟨*function* Generic_vs_generic.m_list__m_stmt 83c⟩≡ (428b)
```
  and m_list__m_stmt_uncached ~list_kind (xsa : A.stmt list) (xsb : A.stmt list) =
    (* TODO: getting this list every time is redundant *)
    match stmts_may_match xsa xsb with
    | No -> fail ()
    | Maybe -> (
        ⟨Generic_vs_generic.m_list__m_stmt if debug 84a⟩
        match (xsa, xsb) with
        | [], [] -> return ()
        ⟨Generic_vs_generic.m_list__m_stmt() empty list vs list case 94c⟩
        ⟨Generic_vs_generic.m_list__m_stmt() ellipsis cases 83d⟩
        (* the general case *)
        | xa :: aas, xb :: bbs ->
            m_stmt xa xb >>= fun () ->
            env_add_matched_stmt xb >>= fun () ->
            m_list__m_stmt ~list_kind aas bbs
        | _ :: _, _ -> fail () )
```

⟨Generic_vs_generic.m_list__m_stmt() *ellipsis cases* 83d⟩≡ (83c)
```
  (* dots: '...', can also match no statement *)
  | [ { s = A.ExprStmt (A.Ellipsis _i, _); _ } ], [] -> return ()
  | ( { s = A.ExprStmt (A.Ellipsis _i, _); _ } :: xsa_tail,
```

```
      (xb :: xsb_tail as xsb) ) ->
      (* can match nothing *)
      m_list__m_stmt ~list_kind xsa_tail xsb
      >||> (* can match more *)
      ( env_add_matched_stmt xb >>= fun () ->
        m_list__m_stmt ~list_kind xsa xsb_tail )
```

⟨Generic_vs_generic.m_list__m_stmt *if* debug 84a⟩≡ (83c)
```
  logger#ldebug (lazy (spf "%d vs %d" (List.length xsa) (List.length xsb)));
```

Ellipsis in fields

⟨*function* Generic_vs_generic.m_list__m_field 84b⟩≡ (428b)
```
  and m_list__m_field (xsa : A.field list) (xsb : A.field list) =
    match (xsa, xsb) with
    | [], [] -> return ()
    ⟨Generic_vs_generic.m_list__m_field() empty list vs list case 94b⟩
    ⟨Generic_vs_generic.m_list__m_field() ellipsis cases 84c⟩
    ⟨Generic_vs_generic.m_list__m_field() DefStmt pattern case 87d⟩
    (* the general case *)
    | xa :: aas, xb :: bbs -> m_field xa xb >>= fun () -> m_list__m_field aas bbs
    | _ :: _, _ -> fail ()
```

⟨Generic_vs_generic.m_list__m_field() *ellipsis cases* 84c⟩≡ (84b)
```
  | A.FieldStmt { s = A.ExprStmt (A.Ellipsis _, _); _ } :: _, _ ->
      raise Impossible
```

7.4.2 Any code matching

⟨Generic_vs_generic.m_expr() *ellipsis cases* 84d⟩≡ (70b) 90b▷
```
  (* dots: should be patterned-match before in arguments, or statements,
   * but this is useful for keyword parameters, as in f(..., foo=..., ...)
   *)
  | A.Ellipsis _a1, _ -> return ()
```

⟨Generic_vs_generic.m_stmt() *ellipsis cases* 84e⟩≡ (73a) 84f▷
```
  (* dots: '...' can to match any statememt *)
  | A.ExprStmt (A.Ellipsis _i, _), _b -> return ()
```

7.4.3 Optional code matching

⟨Generic_vs_generic.m_stmt() *ellipsis cases* 84f⟩+≡ (73a) ◁84e
```
  | A.Return (a0, a1, asc), B.Return (b0, b1, bsc) ->
      let* () = m_tok a0 b0 in
      let* () = m_option_ellipsis_ok m_expr a1 b1 in
      m_tok asc bsc
```

⟨*signature* Matching_generic.m_option_ellipsis_ok 84g⟩≡ (440)
```
  val m_option_ellipsis_ok :
    (AST_generic.expr -> 'a -> tin -> tout) ->
    AST_generic.expr option ->
    'a option ->
    tin ->
    tout
```

⟨*function* Matching_generic.m_option_ellipsis_ok 85a⟩≡ (442)
```
(* dots: *)
let m_option_ellipsis_ok f a b =
  match (a, b) with
  | None, None -> return ()
  (* dots: ... can match 0 or 1 expression *)
  | Some (A.Ellipsis _), None -> return ()
  | Some xa, Some xb -> f xa xb
  | None, _ | Some _, _ -> fail ()
```

7.4.4 Binary operations sequence matching

7.4.5 String matching

⟨Generic_vs_generic.m_literal() *ellipsis case* 85b⟩≡ (71) 85c ▷
```
(* dots: '...' on string or regexps *)
| A.String a, B.String b -> m_wrap m_string_ellipsis_or_regexp_or_default a b
```

⟨Generic_vs_generic.m_literal() *ellipsis case* 85c⟩+≡ (71) ◁85b
```
| A.Regexp ("/.../", a), B.Regexp (_s, b) -> m_info a b
```

⟨Generic_vs_generic.m_expr() *interpolated strings case* 85d⟩≡ (70b)
```
(* dots '...' for string literal:
 * Interpolated strings are transformed into Call(Special(Concat, ...),
 * hence we want Python patterns like f"...{$X}...", which are expanded to
 * Call(Special(Concat, [L"..."; Id "$X"; L"..."])) to
 * match concrete code like f"foo{a}" such that "..." is seemingly
 * matching 0 or more literal expressions.
 * bugfix: note that we want to do that only when inside
 * Call(Special(Concat(...))), hence the special m_arguments_concat below,
 * otherwise regular call patterns like foo("...") would match code like
 * foo().
 *)
| ( A.Call (A.IdSpecial (A.ConcatString akind, _a1), a2),
    B.Call (B.IdSpecial (B.ConcatString bkind, _b1), b2) ) ->
    m_concat_string_kind akind bkind >>= fun () ->
    m_bracket m_arguments_concat a2 b2
```

⟨*function* Generic_vs_generic.m_arguments_concat 85e⟩≡ (428b)
```
and m_arguments_concat a b =
  match (a, b) with
  | [], [] -> return ()
  ⟨Generic_vs_generic.m_arguments_concat() ellipsis cases 86a⟩
  (* the general case *)
  | xa :: aas, xb :: bbs -> (
      (* exception: for concat strings, don't have ellipsis match   *)
      (* string literals since string literals are implicitly not   *)
      (* interpolated, and ellipsis implicitly is                   *)
      match (xa, xb) with
      | A.Arg (A.Ellipsis _), A.Arg (A.L (A.String _)) -> fail ()
      | _ -> m_argument xa xb >>= fun () -> m_arguments_concat aas bbs )
  | [], _ | _ :: _, _ -> fail ()
```

⟨Generic_vs_generic.m_arguments_concat() *ellipsis cases* 86a⟩≡ (85e)
```
(* dots '...' for string literal, can match any number of arguments *)
| [ A.Arg (A.L (A.String ("...", _))) ], _xsb -> return ()
(* specific case: f"...{$X}..." will properly extract $X from f"foo {bar} baz" *)
| A.Arg (A.L (A.String ("...", a))) :: xsa, B.Arg bexpr :: xsb ->
    (* can match nothing *)
    m_arguments_concat xsa (B.Arg bexpr :: xsb)
    >|!> (* can match more *)
    m_arguments_concat (A.Arg (A.L (A.String ("...", a))) :: xsa) xsb
```

7.5 Out-of-order matching

7.5.1 Matching attributes

7.5.2 Matching keyword arguments

⟨Generic_vs_generic.m_list__m_argument() ArgKwd *pattern case* 86b⟩≡ (83a)
```
(* unordered kwd argument matching *)
| (A.ArgKwd (((s, _tok) as ida), ea) as a) :: xsa, xsb -> (
    if MV.is_metavar_name s then
      ⟨Generic_vs_generic.m_list__m_argument() if metavar keyword argument 86c⟩
    else
      try
        let before, there, after =
          xsb
          |> Common2.split_when (function
               | A.ArgKwd ((s2, _), _) when s =$= s2 -> true
               | _ -> false)
        in
        match there with
        | A.ArgKwd (idb, eb) ->
            m_ident ida idb >>= fun () ->
            m_expr ea eb >>= fun () -> m_list__m_argument xsa (before @ after)
        | _ -> raise Impossible
      with Not_found -> fail () )
```

7.5.3 Matching metavariable keyword arguments

⟨Generic_vs_generic.m_list__m_argument() *if metavar keyword argument* 86c⟩≡ (86b)
```
let candidates = all_elem_and_rest_of_list xsb in
(* less: could use a fold *)
let rec aux xs =
  match xs with
  | [] -> fail ()
  | (b, xsb) :: xs ->
      m_argument a b
      >>= (fun () -> m_list__m_argument xsa (lazy_rest_of_list xsb))
      >|!> aux xs
in
aux candidates
```

⟨*signature* Matching_generic.all_elem_and_rest_of_list 86d⟩≡ (440)
```
val all_elem_and_rest_of_list : 'a list -> ('a * 'a list Lazy.t) list
```

⟨*function* `Matching_generic.all_elem_and_rest_of_list` 87a⟩≡ (442)
```
(* todo? optimize, probably not the optimal version ... *)
let all_elem_and_rest_of_list xs =
  let xs = Common.index_list xs |> List.map (fun (x, i) -> (i, x)) in
  xs
  |> List.map (fun (i, x) -> (x, lazy (List.remove_assq i xs |> List.map snd)))
  [@@profiling]
```

⟨*toplevel* `Matching_generic._1` 87b⟩≡ (442)
```
(* let _ = Common2.example
    (all_elem_and_rest_of_list ['a';'b';'c'] =
     [('a', ['b';'c']); ('b', ['a';'c']); ('c', ['a';'b'])]) *)
```

7.5.4 Matching XML attributes

⟨*function* `Generic_vs_generic.m_attrs` 87c⟩≡ (428b)
```
and m_attrs a b =
  let _has_ellipsis, a = has_xml_ellipsis_and_filter_ellipsis a in
  (* always implicit ... *)
  m_list_in_any_order ~less_is_ok:true m_xml_attr a b
```

7.5.5 Matching metavariable XML attributes

7.5.6 Matching fields

⟨`Generic_vs_generic.m_list__m_field() DefStmt` *pattern case* 87d⟩≡ (84b)
```
| ( ( A.FieldStmt
        {
          s = A.DefStmt (({ A.name = A.EN (A.Id ((s1, _), _)); _ }, _) as adef);
          _;
        } as a )
    :: xsa,
    xsb ) -> (
    if MV.is_metavar_name s1 || Pattern.is_regexp_string s1 then
      ⟨Generic_vs_generic.m_list__m_field() in DefStmt case if metavar field 88a⟩
    else
      try
        let before, there, after =
          xsb
          |> Common2.split_when (function
               | A.FieldStmt
                   {
                     s =
                       A.DefStmt
                         ({ B.name = B.EN (B.Id ((s2, _tok), _)); _ }, _);
                     _;
                   }
                 when s2 = s1 ->
                   true
               | _ -> false)
        in
        match there with
        | A.FieldStmt { s = A.DefStmt bdef; _ } ->
            m_definition adef bdef >>= fun () ->
            m_list__m_field xsa (before @ after)
        | _ -> raise Impossible
      with Not_found -> fail () )
```

7.5.7 Matching metavariable fields

⟨Generic_vs_generic.m_list__m_field() *in* DefStmt *case if metavar field* 88a⟩≡ (87d)
```
  let candidates = all_elem_and_rest_of_list xsb in
  (* less: could use a fold *)
  let rec aux xs =
    match xs with
    | [] -> fail ()
    | (b, xsb) :: xs ->
        m_field a b
        >>= (fun () -> m_list__m_field xsa (lazy_rest_of_list xsb))
        >||> aux xs
  in
  aux candidates
```

7.6 Regular expressions matching

7.6.1 Matching strings

⟨Generic_vs_generic.m_literal() *regexp case* 88b⟩≡ (71)

⟨*constant* Matching_generic.regexp_regexp_string 88c⟩≡ (442)

⟨*signature* Matching_generic.is_regexp_string 88d⟩≡ (440)

⟨*function* Matching_generic.is_regexp_string 88e⟩≡ (442)

⟨*signature* Matching_generic.regexp_of_regexp_string 88f⟩≡ (440)
```
  val regexp_matcher_of_regexp_string : string -> string -> bool
```

⟨*function* Matching_generic.regexp_of_regexp_string 88g⟩≡ (442)
```
  let regexp_matcher_of_regexp_string s =
    if s =~ Pattern.regexp_regexp_string then (
      let x, flags = Common.matched2 s in
      let flags =
        match flags with
        | "" -> []
        | "i" -> [ `CASELESS ]
        | "m" -> [ `MULTILINE ]
        | _ -> failwith (spf "This is not a valid PCRE regexp flag: %s" flags)
      in
      (* old: let re = Str.regexp x in (fun s -> Str.string_match re s 0) *)
      (* TODO: add `ANCHORED to be consistent with Python re.match (!re.search)*)
      let re = Re.Pcre.regexp ~flags x in
      fun s2 ->
        Re.Pcre.pmatch ~rex:re s2 |> fun b ->
        logger#debug "regexp match: %s on %s, result = %b" s s2 b;
        b )
    else failwith (spf "This is not a PCRE-compatible regexp: " ^ s)
```

7.6.2 Matching fields

⟨Generic_vs_generic.m_ident() *regexp case* 89a⟩≡ (78a 70a)
```
  (* in some languages such as Javascript certain entities like
   * fields can use strings for identifiers (e.g., {"myfield": 1}),
   * which gives the opportunity to use regexp string for fields
   * (e.g., {"=~/.*field/": $X}).
   *)
  | (stra, _), (strb, _) when Pattern.is_regexp_string stra ->
      let re_match = Matching_generic.regexp_matcher_of_regexp_string stra in
      if re_match strb then return () else fail ()
```

7.7 Deep matching

7.7.1 Deep (implicit) expression matching

⟨Generic_vs_generic.m_stmt() *deep matching cases* 89b⟩≡ (73a) 90c ▷
```
  (* deeper: go deep by default implicitly (no need for explicit <... ...>) *)
  | A.ExprStmt (a1, a2), B.ExprStmt (b1, b2) ->
      m_expr_deep a1 b1 >>= fun () -> m_tok a2 b2
```

⟨*constant* Flag_semgrep.go_deeper_expr 89c⟩≡ (384)

⟨*function* Generic_vs_generic.m_expr_deep 89d⟩≡ (428b)
```
  and m_expr_deep a b =
    if_config
      (fun x -> not x.go_deeper_expr)
      ~then_:(m_expr a b)
      ~else_:
        ( m_expr a b >!> fun () ->
          let subs = SubAST_generic.subexprs_of_expr b in
          (* less: could use a fold *)
          let rec aux xs =
            match xs with [] -> fail () | x :: xs -> m_expr_deep a x >||> aux xs
          in
          aux subs )
```

⟨*function* SubAST_generic.subexprs_of_expr 89e⟩≡ (451c)
```
  (* used for deep expression matching *)
  let subexprs_of_expr e =
    match e with
    | L _ | N _ | IdSpecial _ | Ellipsis _ | TypedMetavar _ -> []
    | DotAccess (e, _, _)
    | Await (_, e)
    | Cast (_, e)
    | Ref (_, e)
    | DeRef (_, e)
    | DeepEllipsis (_, e, _)
    | DotAccessEllipsis (e, _) ->
        [ e ]
    | Assign (e1, _, e2)
    | AssignOp (e1, _, e2)
    | ArrayAccess (e1, (_, e2, _))
    (* not sure we always want to return 'e1' here *) ->
        [ e1; e2 ]
    | Conditional (e1, e2, e3) -> [ e1; e2; e3 ]
    | Tuple (_, xs, _) | Seq xs -> xs
```

```
  | Record (_, flds, _) ->
      flds
      |> Common2.map_flatten (function
           | FieldStmt st -> subexprs_of_stmt st
           | FieldSpread (_, e) -> [ e ])
  | Container (_, xs) -> unbracket xs
  | Call (e, args) ->
      (* not sure we want to return 'e' here *)
      e
      :: ( args |> unbracket
         |> Common.map_filter (function
              | Arg e | ArgKwd (_, e) -> Some e
              | ArgType _ | ArgOther _ -> None) )
  | SliceAccess (e1, e2) ->
      e1
      :: ( e2 |> unbracket
         |> (fun (a, b, c) -> [ a; b; c ])
         |> List.map Common.opt_to_list
         |> List.flatten )
  | Yield (_, eopt, _) -> Common.opt_to_list eopt
  | OtherExpr (_, anys) ->
      (* in theory we should go deeper in any *)
      subexprs_of_any_list anys
  | Lambda def -> subexprs_of_stmt def.fbody
  (* currently skipped over but could recurse *)
  | Constructor _ | AnonClass _ | Xml _ | LetPattern _ | MatchPattern _ -> []
  | DisjExpr _ -> raise Common.Impossible
  [@@profiling]
```

7.7.2 Deep (explicit) ellipsis operator <... ...>

⟨AST_generic.expr *semgrep extensions cases* 90a⟩+≡ (24d) ◁33b 102e▷

```
  | DeepEllipsis of expr bracket (* <... ...> *)
```

⟨Generic_vs_generic.m_expr() *ellipsis cases* 90b⟩+≡ (70b) ◁84d

```
  | A.DeepEllipsis (_, a1, _), a2 -> m_expr_deep a1 a2
```

7.7.3 Deep (implicit) statememt matching

⟨Generic_vs_generic.m_stmt() *deep matching cases* 90c⟩+≡ (73a) ◁89b

```
  (* opti: specialization to avoid going in the deep stmt matching!
   * TODO: we should not need this; '...' should not enumerate all
   * possible subset of stmt list and take forever.
   * Note that as a side effect it returns also less equivalent
   * matches (which again, should not happen), which used to introduce
   * some regressions (see tests/OTHER/rules/regression_uniq...) but this
   * has been fixed now.
   *)
  | A.Block (_, [ { s = A.ExprStmt (A.Ellipsis _i, _); _ } ], _), B.Block _b1 ->
      return ()
  (* opti: another specialization; again we should not need it *)
  | ( A.Block
        ( _,
          [
            { s = A.ExprStmt (A.Ellipsis _, _); _ };
            a;
            { s = A.ExprStmt (A.Ellipsis _, _); _ };
```

```
        ],
        _ ),
     B.Block (_, bs, _) ) ->
     let bs =
       match SubAST_generic.flatten_substmts_of_stmts bs with
       (* already flat  *)
       | None -> bs
       | Some (xs, _) -> xs
     in
     or_list m_stmt a bs
  (* the general case *)
  (* TODO: ... will allow a subset of stmts? good? *)
  | A.Block a1, B.Block b1 -> m_bracket (m_stmts_deep ~less_is_ok:false) a1 b1
```

⟨*function* Generic_vs_generic.m_stmts_deep 91⟩≡ (428b)

```
  and m_stmts_deep_uncached ~less_is_ok (xsa : A.stmt list) (xsb : A.stmt list) =
    (* opti: this was the old code:
     *   if !Flag.go_deeper_stmt && (has_ellipsis_stmts xsa)
     *   then
     *   m_list__m_stmt xsa xsb >!> (fun () ->
     *     let xsb' = SubAST_generic.flatten_substmts_of_stmts xsb in
     *     m_list__m_stmt xsa xsb'
     *   )
     *   else m_list__m_stmt xsa xsb
     *
     * but this was really slow on huge files because with a pattern like
     * 'foo(); ...; bar();' we would call flatten_substmts_of_stmts
     * on each sequences in the program, even though foo(); was not
     * matched first.
     * Better to first match the first element, and if it matches and
     * we have a '...' that was not matched on the current sequence,
     * then we try with flatten_substmts_of_stmts.
     *
     * The code below is mostly a copy paste of m_list__m_stmt. We could
     * factorize, but I prefer to control and limit the number of places
     * where we call m_stmts_deep. Once we call m_list__m_stmt, we
     * are in a simpler world where the list of stmts will not grow.
     *)
    match (xsa, xsb) with
    | [], [] -> return ()
    (* less-is-ok:
     * it's ok to have statements after in the concrete code as long as we
     * matched all the statements in the pattern (there is an implicit
     * '...' at the end, in addition to implicit '...' at the beginning
     * handled by kstmts calling the pattern for each subsequences).
     * TODO: sgrep_generic though then display the whole sequence as a match
     * instead of just the relevant part.
     *)
    | [], _ :: _ -> if less_is_ok then return () else fail ()
    (* dots: '...', can also match no statement *)
    | [ { s = A.ExprStmt (A.Ellipsis _i, _); _ } ], [] -> return ()
    | ({ s = A.ExprStmt (A.Ellipsis _i, _); _ } :: _ as xsa), (_ :: _ as xsb) ->
        (* let's first try without going deep *)
        m_list__m_stmt ~list_kind:CK.Original xsa xsb >!> fun () ->
        if_config
          (fun x -> x.go_deeper_stmt)
          ~then_:
            ( match SubAST_generic.flatten_substmts_of_stmts xsb with
            | None -> fail () (* was already flat *)
            | Some (xsb, last_stmt) ->
```

```
              m_list__m_stmt ~list_kind:(CK.Flattened_until last_stmt.s_id) xsa
                xsb )
          ~else_:(fail ())
    (* the general case *)
    | xa :: aas, xb :: bbs ->
        m_stmt xa xb >>= fun () ->
        env_add_matched_stmt xb >>= fun () -> m_stmts_deep ~less_is_ok aas bbs
    | _ :: _, _ -> fail ()
```

⟨*signature* `Matching_generic.has_ellipsis_stmts` 92a⟩≡ (440)

```
  val has_ellipsis_stmts : AST_generic.stmt list -> bool
```

⟨*function* `Matching_generic.has_ellipsis_stmts` 92b⟩≡ (442)

```
  (* guard for deep stmt matching *)
  let has_ellipsis_stmts xs =
    xs
    |> List.exists (fun st ->
           match st.A.s with A.ExprStmt (A.Ellipsis _, _) -> true | _ -> false)
```

⟨*signature* `Generic_vs_generic.m_stmts_deep` 92c⟩≡ (428a)

```
  val m_stmts_deep :
    less_is_ok:bool ->
    (AST_generic.stmt list, AST_generic.stmt list) Matching_generic.matcher
```

⟨*constant* `Flag_semgrep.go_deeper_stmt` 92d⟩≡ (384)

⟨*constant* `Flag_semgrep.go_really_deeper_stmt` 92e⟩≡ (384)

⟨*function* `SubAST_generic.subexprs_of_stmt` 92f⟩≡ (451c)

⟨*function* `SubAST_generic.substmts_of_stmt` 92g⟩≡ (451c)

```
  (* used for deep statement matching *)
  let substmts_of_stmt st =
    match st.s with
    (* 0 *)
    | DirectiveStmt _ | ExprStmt _ | Return _ | Continue _ | Break _ | Goto _
    | Throw _ | Assert _ | OtherStmt _ ->
        []
    (* 1 *)
    | While (_, _, st)
    | DoWhile (_, st, _)
    | For (_, _, st)
    | Label (_, st)
    | OtherStmtWithStmt (_, _, st) ->
        [ st ]
    (* 2 *)
    | If (_, _, st1, st2) -> st1 :: Common.opt_to_list st2
    | WithUsingResource (_, st1, st2) -> [ st1; st2 ]
    (* n *)
    | Block (_, xs, _) -> xs
    | Switch (_, _, xs) ->
        xs
        |> List.map (function
             | CasesAndBody (_, st) -> [ st ]
             | CaseEllipsis _ -> [])
        |> List.flatten
```

```
    | Try (_, st, xs, opt) -> (
        [ st ]
        @ (xs |> List.map Common2.thd3)
        @ match opt with None -> [] | Some (_, st) -> [ st ] )
    | DisjStmt _ -> raise Common.Impossible
    (* this may slow down things quite a bit *)
    | DefStmt (_ent, def) -> (
        if not !go_really_deeper_stmt then []
        else
          match def with
          | VarDef _ | FieldDefColon _ | TypeDef _ | MacroDef _ | Signature _
          | UseOuterDecl _
          (* recurse? *)
          | ModuleDef _ | OtherDef _ ->
              []
          (* this will add lots of substatements *)
          | FuncDef def -> [ def.fbody ]
          | ClassDef def ->
              def.cbody |> unbracket
              |> Common.map_filter (function
                   | FieldStmt st -> Some st
                   | FieldSpread _ -> None) )
```

⟨*function* SubAST_generic.flatten_substmts_of_stmts 93⟩≡ (451c)

```
  let flatten_substmts_of_stmts xs =
    (* opti: using a ref, List.iter, and Common.push instead of a mix of
     * List.map, List.flatten and @ below speed things up
     * (but it is still slow when called many many times)
     *)
    let res = ref [] in
    let changed = ref false in

    let rec aux x =
      (* return the current statement first, and add substmts *)
      Common.push x res;

      (* this can be really slow because lambdas_in_expr() below can be called
       * a zillion times on big files (see tests/PERF/) if we do the
       * matching naively in m_stmts_deep.
       *)
      ( if !go_really_deeper_stmt then
        let es = subexprs_of_stmt x in
        (* getting deeply nested lambdas stmts *)
        let lambdas = es |> List.map lambdas_in_expr_memo |> List.flatten in
        lambdas |> List.map (fun def -> def.fbody) |> List.iter aux );

      let xs = substmts_of_stmt x in
      match xs with
      | [] -> ()
      | xs ->
          changed := true;
          xs |> List.iter aux
    in
    xs |> List.iter aux;
    if !changed then
      match !res with
      | [] -> None
      | last :: _ ->
          (* Return the last element of the list as a pair.
```

```
          This is used as part of the caching optimization. *)
        Some (List.rev !res, last)
    else None
    [@@profiling]
```

7.8 Unspecified matches more

7.8.1 Incomplete lists

⟨Matching_generic.m_list_with_dots() *¿ empty list vs list case* 94a⟩≡ (82c)
```
  (* less-is-ok: empty list can sometimes match non-empty list *)
  | [], _ :: _ when less_is_ok -> return ()
```

⟨Generic_vs_generic.m_list__m_field() *empty list vs list case* 94b⟩≡ (84b)
```
  (* less-is-ok:
   * it's ok to have fields after in the concrete code as long as we
   * matched all the fields in the pattern.
   * TODO? should we impose to use '...' if you allow extra fields?
   *)
  | [], _ :: _ -> return ()
```

⟨Generic_vs_generic.m_list__m_stmt() *empty list vs list case* 94c⟩≡ (83c)
```
  (* less-is-ok:
   * it's ok to have statements after in the concrete code as long as we
   * matched all the statements in the pattern (there is an implicit
   * '...' at the end, in addition to implicit '...' at the beginning
   * handled by kstmts calling the pattern for each subsequences).
   * TODO: sgrep_generic though then display the whole sequence as a match
   * instead of just the relevant part.
   *)
  | [], _ :: _ -> return ()
```

7.8.2 Unspecified types

⟨*signature* Matching_generic.m_option_none_can_match_some 94d⟩≡ (440)
```
  val m_option_none_can_match_some :
    ('a -> 'b -> tin -> tout) -> 'a option -> 'b option -> tin -> tout
```

⟨*function* Matching_generic.m_option_none_can_match_some 94e⟩≡ (442)
```
  (* less-is-ok: *)
  let m_option_none_can_match_some f a b =
    match (a, b) with
    (* Nothing specified in the pattern can match Some stuff *)
    | None, _ -> return ()
    | Some xa, Some xb -> f xa xb
    | Some _, _ -> fail ()
```

7.8.3 Unspecified module aliases

7.8.4 Unspecified XML body

⟨*function* Generic_vs_generic.m_bodies 94f⟩≡ (428b)
```
  and m_bodies a b = m_list__m_body a b
```

⟨*function* Generic_vs_generic.m_list__m_body 95a⟩≡ (428b)
```
  and m_list__m_body a b =
    match a with
    (* less-is-ok: it's ok to have an empty body in the pattern *)
    | [] -> return ()
    | _ -> m_list m_body a b
```

7.8.5 Unspecified inheritance

7.9 Prefix matching

7.9.1 Module name prefixes

⟨Generic_vs_generic.m_directive_basic *import cases* 95b⟩≡ (76b)
```
  (* metavar: import $LIB should bind $LIB to the full qualified name *)
  | ( A.ImportFrom (a0, DottedName [], (str, tok), a3),
      B.ImportFrom (b0, DottedName xs, x, b3) )
    when MV.is_metavar_name str ->
      let name = H.name_of_ids (xs @ [ x ]) in
      let* () = m_tok a0 b0 in
      let* () = envf (str, tok) (MV.N name) in
      (m_option_none_can_match_some m_ident_and_id_info) a3 b3
  | A.ImportFrom (a0, a1, a2, a3), B.ImportFrom (b0, b1, b2, b3) ->
      m_tok a0 b0 >>= fun () ->
      m_module_name_prefix a1 b1 >>= fun () ->
      m_ident_and_empty_id_info a2 b2 >>= fun () ->
      (m_option_none_can_match_some m_ident_and_id_info) a3 b3
  | A.ImportAs (a0, a1, a2), B.ImportAs (b0, b1, b2) ->
      m_tok a0 b0 >>= fun () ->
      m_module_name_prefix a1 b1 >>= fun () ->
      (m_option_none_can_match_some m_ident_and_id_info) a2 b2
  | A.ImportAll (a0, a1, a2), B.ImportAll (b0, b1, b2) ->
      m_tok a0 b0 >>= fun () ->
      m_module_name_prefix a1 b1 >>= fun () -> m_tok a2 b2
```

⟨*function* Generic_vs_generic.m_module_name_prefix 95c⟩≡ (428b)
```
  (* less-is-ok: prefix matching is supported for imports, eg.:
   *  pattern: import foo.x should match: from foo.x.z.y
   *)
  let m_module_name_prefix a b =
    match (a, b) with
    (* metavariable case *)
    | A.FileName ((a_str, _) as a1), B.FileName b1 when MV.is_metavar_name a_str
      ->
        (* Bind as a literal string expression so that pretty-printing works.
         * This also means that this metavar can match both literal strings and
         * filenames with the same string content. *)
        envf a1 (MV.E (B.L (B.String b1)))
    (* dots: '...' on string or regexp *)
    | A.FileName a, B.FileName b ->
        m_wrap
          (m_string_ellipsis_or_regexp_or_default
           (* TODO figure out what prefix support means here *)
             ~m_string_for_default:m_string_prefix)
          a b
    | A.DottedName a1, B.DottedName b1 -> m_dotted_name_prefix_ok a1 b1
    | A.FileName _, _ | A.DottedName _, _ -> fail ()
```

7.9.2 List prefixes

⟨*signature* Matching_generic.m_list_prefix 96a⟩≡ (440)
```
val m_list_prefix : ('a, 'b) matcher -> ('a list, 'b list) matcher
```

⟨*function* Matching_generic.m_list_prefix 96b⟩≡ (442)
```
let rec m_list_prefix f a b =
  match (a, b) with
  | [], [] -> return ()
  | xa :: aas, xb :: bbs -> f xa xb >>= fun () -> m_list_prefix f aas bbs
  (* less-is-ok: prefix is ok *)
  | [], _ -> return ()
  | _ :: _, _ -> fail ()
```

7.9.3 String prefixes

⟨*signature* Matching_generic.string_is_prefix 96c⟩≡ (440)
```
val string_is_prefix : string -> string -> bool
```

⟨*function* Matching_generic.string_is_prefix 96d⟩≡ (442)
```
let string_is_prefix s1 s2 =
  let len1 = String.length s1 and len2 = String.length s2 in
  if len1 < len2 then false
  else
    let sub = Str.first_chars s1 len2 in
    sub = s2
```

⟨*signature* Matching_generic.m_string_prefix 96e⟩≡ (440)
```
val m_string_prefix : string -> string -> tin -> tout
```

⟨*function* Matching_generic.m_string_prefix 96f⟩≡ (442)
```
(* less-is-ok: *)
let m_string_prefix a b = if string_is_prefix b a then return () else fail ()
```

7.10 Builtin code equivalences

7.10.1 Variable definitions versus assignments

⟨Generic_vs_generic.m_stmt() *builtin equivalences cases* 96g⟩≡ (73a) 97c ▷
```
(* equivalence: vardef ==> assign, and go deep *)
| ( A.ExprStmt (a1, _),
    B.DefStmt (ent, B.VarDef ({ B.vinit = Some _; _ } as def)) ) ->
    let b1 = H.vardef_to_assign (ent, def) in
    m_expr_deep a1 b1
```

⟨*function* AST_generic.vardef_to_assign 97a⟩≡ (274)
```
(* used in controlflow_build and semgrep *)
let vardef_to_assign (ent, def) =
  let name = name_or_dynamic_to_expr ent.name None in
  let v =
    match def.vinit with
    | Some v -> v
    | None -> L (Null (Parse_info.fake_info "null"))
  in
  Assign (name, Parse_info.fake_info "=", v)
```

⟨*constant* AST_generic.special_multivardef_pattern 97b⟩≡ (263c)
```
(* In JS one can do 'var {x,y} = foo();'. We used to transpile that
 * in multiple vars, but in sgrep one may want to match over those patterns.
 * However those multivars do not fit well with the (entity * definition_kind)
 * model we currently use, so for now we need this ugly hack of converting
 * the statement above in
 * ({name = "!MultiVarDef"}, VarDef {vinit = Assign (Record {...}, foo())}).
                                    * This is bit ugly, but at some point we may want to remove completely
                                    * VarDef by transforming them in Assign (see vardef_to_assign() below)
                                    * so this temporary hack is not too bad.
 *)
let special_multivardef_pattern = AST_generic_.special_multivardef_pattern
```

7.10.2 Expression statements versus `return`

⟨Generic_vs_generic.m_stmt() *builtin equivalences cases* 97c⟩+≡ (73a) ◁96g
```
(* equivalence: *)
| A.ExprStmt (a1, _), B.Return (_, Some b1, _) -> m_expr_deep a1 b1
```

7.10.3 Import variations

⟨*function* Generic_vs_generic.m_directive 97d⟩≡ (428b)
```
and m_directive a b =
  m_directive_basic a b >!> fun () ->
  match a with
  (* normalize only if very simple import pattern (no alias) *)
  | A.ImportFrom (_, _, _, None) | A.ImportAs (_, _, None) -> (
      (* equivalence: *)
      let normal_a = Normalize_generic.normalize_import_opt true a in
      let normal_b = Normalize_generic.normalize_import_opt false b in
      match (normal_a, normal_b) with
      | Some (a0, a1), Some (b0, b1) ->
          m_tok a0 b0 >>= fun () -> m_module_name_prefix a1 b1
      | _ -> fail () )
  (* more complex pattern should not be normalized *)
  | A.ImportFrom _ | A.ImportAs _
  (* definitely do not normalize the pattern for ImportAll *)
  | A.ImportAll _ | A.Package _ | A.PackageEnd _ | A.Pragma _
  | A.OtherDirective _ ->
      fail ()
```

⟨*signature* Normalize_generic.normalize_import_opt 98a⟩≡ (453a)
```
val normalize_import_opt :
  bool ->
  AST_generic.directive ->
  (AST_generic.tok * AST_generic.module_name) option
```

⟨*function* Normalize_generic.full_module_name 98b⟩≡ (453b)
```
(* Normalize imports for matching purposes.
 * Examples (for Python):
 *   from foo import bar -> import foo.bar
 *   from foo.bar import baz -> import foo.bar.baz
 *)

let full_module_name is_pattern from_module_name import_opt =
  match (from_module_name, import_opt) with
  | DottedName idents, Some import_ident_name ->
      let new_module_name : dotted_ident = idents @ [ import_ident_name ] in
      Some (DottedName new_module_name)
  | DottedName idents, None -> Some (DottedName idents)
  | FileName s, None -> Some (FileName s)
  | FileName s, _ when not is_pattern ->
      (* bugfix: for languages such as JS, 'import x from "path"' should not
       * be converted in just "path". We should return None here as it
       * does not make sense to allow this pattern to match
       * import y from "path". Use just 'import "path"' if you just want
       * to check you vaguely imported a package.
       *)
      Some (FileName s)
  | FileName _, Some _ -> None
```

⟨*function* Normalize_generic.normalize_import_opt 98c⟩≡ (453b)
```
let normalize_import_opt is_pattern i =
  match i with
  | ImportFrom (t, module_name, m, _alias_opt) ->
      full_module_name is_pattern module_name (Some m) >>= fun x -> Some (t, x)
  | ImportAs (t, module_name, _alias_opt) ->
      full_module_name is_pattern module_name None >>= fun x -> Some (t, x)
  | ImportAll (t, module_name, _t2) ->
      full_module_name is_pattern module_name None >>= fun x -> Some (t, x)
  | Package _ | PackageEnd _ | Pragma _ | OtherDirective _ -> None
```

7.11 Language-specific equivalences

7.11.1 Javascript `var` versus `let/const`

⟨*function* Generic_vs_generic.m_keyword_attribute 98d⟩≡ (428b)
```
and m_keyword_attribute a b =
  match (a, b) with
  (* equivalent: quite JS-specific *)
  | A.Var, (A.Var | A.Let | A.Const) -> return ()
  | _ -> m_other_xxx a b
```

7.11.2 Python inheritance list

⟨*function* Generic_vs_generic.m_list__m_type_ 99a⟩≡ (428b)
```
and m_list__m_type_ (xsa : A.type_ list) (xsb : A.type_ list) =
  m_list_with_dots m_type_
    (* dots: '...', this is very Python Specific I think *)
      (function
      | A.OtherType (A.OT_Arg, [ A.Ar (A.Arg (A.Ellipsis _i)) ]) -> true
      | _ -> false)
    (* less-is-ok: it's ok to not specify all the parents I think *)
    true (* empty list can not match non-empty list *) xsa xsb
```

7.11.3 JSX/XHP/XML matching

⟨*function* Generic_vs_generic.m_body 99b⟩≡ (428b)
```
and m_body a b =
  match (a, b) with
  | A.XmlText a1, B.XmlText b1 ->
      m_wrap
        (m_string_ellipsis_or_regexp_or_default
           ~m_string_for_default:m_string_xhp_text)
        a1 b1
  (* boilerplate *)
  ⟨Generic_vs_generic.m_body boilerplate cases 230d⟩
```

⟨*function* Generic_vs_generic.m_string_xhp_text 99c⟩≡ (428b)
```
(* equivalence: on different indentation
 * todo? work? was copy-pasted from XHP sgrep matcher
 *)
let m_string_xhp_text sa sb =
  if sa =$= sb || (sa =~ "^[\n ]+$" && sb =~ "^[\n ]+$") then return ()
  else fail ()
```

⟨*function* Generic_vs_generic.m_xml_attr_value 99d⟩≡ (428b)
```
and m_xml_attr_value a b =
  (* less: deep? *)
  m_expr a b
```

7.12 Unit testing

⟨*signature* Generic_vs_generic.m_any 99e⟩≡ (428a)
```
(* used only for unit testing *)
val m_any : (AST_generic.any, AST_generic.any) Matching_generic.matcher
```

⟨*signature* Unit_matcher.unittest 99f⟩≡ (427c)
```
(* Returns the testsuite for this directory To be concatenated by
 * the caller (e.g. in pfff/main_test.ml ) with other testsuites and
 * run via OUnit.run_test_tt
 *)
val unittest : any_gen_of_string:(string -> AST_generic.any) -> OUnit.test
```

⟨*function* Unit_matcher.unittest 100⟩≡ (427d)

```
let unittest ~any_gen_of_string =
  "sgrep(generic) features" >:: fun () ->
  (* spec: pattern string, code string, should_match boolean *)
  let triples =
    [
      (* right now any_gen_of_string use the Python sgrep_spatch_pattern
       * parser so the syntax below must be valid Python code
       *)

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

      (* basic string-match of course *)
      ("foo(1,2)", "foo(1,2)", true);
      ("foo(1,3)", "foo(1,2)", false);
      (* matches even when space or newline differs *)
      ("foo(1,2)", "foo(1,     2)", true);
      ("foo(1,2)", "foo(1,\n                          2)", true);
      (* matches even when have comments in the middle *)
      ("foo(1,2)", "foo(1, #foo\n                        2)", true);
      (* ------------ *)
      (* metavariables *)
      (* ------------ *)

      (* for identifiers *)
      ("import $X", "import Foo", true);
      ("x.$X", "x.foo", true);
      (* for expressions *)
      ("foo($X)", "foo(1)", true);
      ("foo($X)", "foo(1+1)", true);
      (* for lvalues *)
      ("$X.method()", "foo.method()", true);
      ("$X.method()", "foo.bar.method()", true);
      (* "linear" patterns, a la Prolog *)
      ("$X & $X", "(a | b) & (a | b)", true);
      ("foo($X, $X)", "foo(a, a)", true);
      ("foo($X, $X)", "foo(a, b)", false);
      (* metavariable on function name *)
      ("$X(1,2)", "foo(1,2)", true);
      (* metavariable on method call *)
      ("$X.foo()", "Bar.foo()", true);
      (* should not match infix expressions though, even if those
       * are transformed internally in Calls *)
      ("$X(...)", "a+b", false);
      (* metavariable for statements *)
      ("if(True): $S\n", "if(True): return 1\n", true);
      (* metavariable for entity definitions *)
      ("def $X():  return 1\n", "def foo(): return 1\n", true);
      (* metavariable for parameter *)
      ("def foo($A, b):  return 1\n", "def foo(x, b): return 1\n", true);
      (* metavariable string for identifiers *)
      (*     "foo('X');", "foo('a_func');", true; *)
      (* many arguments metavariables *)
      (*      "foo($MANYARGS);", "foo(1,2,3);", true; *)

      (* ------------ *)
      (* '...' *)
      (* ------------ *)
```

```
    (* '...' in funcall *)
    ("foo(...)", "foo()", true);
    ("foo(...)", "foo(1)", true);
    ("foo(...)", "foo(1,2)", true);
    ("foo($X,...)", "foo(1,2)", true);
    (* ... also match when there is no additional arguments *)
    ("foo($X,...)", "foo(1)", true);
    ("foo(..., 3, ...)", "foo(1,2,3,4)", true);
    (* ... in more complex expressions *)
    ("strstr(...) == False", "strstr(x)==False", true);
    (* in strings *)
    ("foo(\"...\")", "foo(\"this is a long string\")", true);
    (* "foo(\"...\");", "foo(\"a string\" . \"another string\");", true;*)

    (* for stmts *)
    ("if True: foo(); ...; bar()\n", "if True: foo(); foobar(); bar()\n", true);
    (* for parameters *)
    ("def foo(...): ...\n", "def foo(a, b): return a+b\n", true);
    ( "def foo(..., foo=..., ...): ...\n",
      "def foo(a, b, foo = 1, bar = 2): return a+b\n",
      true );
    (*      "class Foo { ... }", "class Foo { int x; }", true; *)
    (* '...' in arrays *)
    (*      "foo($X, array(...));",  "foo(1, array(2, 3));", true; *)

    (* ------------ *)
    (* Misc isomorphisms *)
    (* ------------ *)
    (* flexible keyword argument matching, the order does not matter *)
    ("foo(kwd1=$X, kwd2=$Y)", "foo(kwd2=1, kwd1=3)", true);
    (* regexp matching in strings *)
    ("foo(\"=~/a+/\")", "foo(\"aaaa\")", true);
    ("foo(\"=~/a+/\")", "foo(\"bbbb\")", false);
    (*      "new Foo(...);","new Foo;", true; *)
  ]
in
triples
|> List.iter (fun (spattern, scode, should_match) ->
       try
         let pattern = any_gen_of_string spattern in
         let code = any_gen_of_string scode in
         let cache = None in
         let config = Config_semgrep.default_config in
         let env = Matching_generic.empty_environment cache config in
         let matches_with_env =
           Semgrep_generic.match_any_any pattern code env
         in
         if should_match then
           assert_bool
             (spf "pattern:|%s| should match |%s" spattern scode)
             (matches_with_env <> [])
         else
           assert_bool
             (spf "pattern:|%s| should not match |%s" spattern scode)
             (matches_with_env = [])
       with Parsing.Parse_error ->
         failwith (spf "problem parsing %s or %s" spattern scode))
```

Chapter 8

User-defined Code Equivalences

⟨*signature* `Apply_equivalences.apply` 102a⟩≡ (447a)
```
val apply : Equivalence.t list -> Pattern.t -> Pattern.t
```

8.1 `semgrep -equivalences <file> ...`

⟨*constant* `Main_semgrep_core.equivalences_file` 102b⟩≡ (369)
```
let equivalences_file = ref ""
```

⟨`Main_semgrep_core.options` *user-defined equivalences case* 102c⟩≡ (46d)
```
( "-equivalences",
  Arg.Set_string equivalences_file,
  " <file> obtain list of code equivalences from YAML file" );
```

⟨*function* `Main_semgrep_core.parse_equivalences` 102d⟩≡ (369)
```
let parse_equivalences () =
  match !equivalences_file with
  | "" -> []
  | file -> Parse_equivalences.parse file
  [@@profiling]
```

8.2 Core data structures

8.2.1 `Equivalence.t`

8.2.2 Extensions to the generic AST: disjunction patterns

⟨`AST_generic.expr` *semgrep extensions cases* 102e⟩+≡ (24d) ◁90a 107a▷
```
| DisjExpr of expr * expr
```

⟨`AST_generic.stmt` *semgrep extensions cases* 102f⟩≡ (26e)
```
(* sgrep: *)
| DisjStmt of stmt * stmt
```

⟨`AST_generic.pattern` *semgrep extensions cases* 102g⟩≡ (152a)
```
(* sgrep: *)
| PatEllipsis of tok
| DisjPat of pattern * pattern
```

8.3 Matching with disjunction patterns

⟨Generic_vs_generic.m_expr() *disjunction case* 103a⟩≡ (70b)
```
  (* equivalence: user-defined equivalence! *)
  | A.DisjExpr (a1, a2), b -> m_expr a1 b >||> m_expr a2 b
```

⟨Generic_vs_generic.m_stmt() *disjunction case* 103b⟩≡ (73a)
```
  (* equivalence: user-defined equivalence! *)
  | A.DisjStmt (a1, a2), _b -> m_stmt a1 b >||> m_stmt a2 b
```

⟨Generic_vs_generic.m_pattern() *disjunction case* 103c⟩≡ (76c)
```
  (* equivalence: user-defined equivalence! *)
  | A.DisjPat (a1, a2), b -> m_pattern a1 b >||> m_pattern a2 b
```

8.4 Applying equivalences on patterns

⟨Semgrep_generic.check2() *apply equivalences to rule pattern* any 103d⟩≡ (61b)
```
  let any = Apply_equivalences.apply equivs any in
```

⟨*function* Apply_equivalences.apply 103e⟩≡ (447b)
```
  let apply equivs any =
    let expr_rules = ref [] in
    let stmt_rules = ref [] in

    equivs
    |> List.iter (fun { Eq.left; op; right; _ } ->
           match (left, op, right) with
           | E l, Eq.Equiv, E r ->
               Common.push (l, r) expr_rules;
               Common.push (r, l) expr_rules
           | E l, Eq.Imply, E r -> Common.push (l, r) expr_rules
           | S l, Eq.Equiv, S r ->
               Common.push (l, r) stmt_rules;
               Common.push (r, l) stmt_rules
           | S l, Eq.Imply, S r -> Common.push (l, r) stmt_rules
           | _ -> failwith "only expr and stmt equivalence patterns are supported");
    (* the order matters, keep the original order reverting Common.push *)
    let expr_rules = List.rev !expr_rules in
    let _stmt_rulesTODO = List.rev !stmt_rules in

    let visitor =
      M.mk_visitor
        {
          M.default_visitor with
          M.kexpr =
            (fun (k, _) x ->
              (* transform the children *)
              let x' = k x in

              let rec aux xs =
                match xs with
                | [] -> x'
                | (l, r) :: xs -> (
                    (* look for a match on original x, not x' *)
                    let matches_with_env =
                      match_e_e_for_equivalences "<equivalence>" l x
                    in
                    match matches_with_env with
```

```
                (* todo: should generate a Disj for each possibilities? *)
                | env :: _xs ->
                    (* Found a match *)
                    let alt = subst_e env.mv r (* recurse on r? *) in
                    (* TODO: use AST_generic.equal_any*)
                    if
                      H.abstract_for_comparison_any (E x)
                      =*= H.abstract_for_comparison_any (E alt)
                    then x' (* disjunction (if different) *)
                    else DisjExpr (x', alt)
                (* no match yet, trying another equivalence *)
                | [] -> aux xs )
            in
            aux expr_rules);
        M.kstmt = (fun (_k, _) x -> x);
      }
  in
  visitor.M.vany any
  [@@profiling]
```

8.4.1 Finding patterns on patterns

⟨*constant* Flag_semgrep.equivalence_mode 104a⟩≡ (384)

```
(* special mode to set before using generic_vs_generic to match
 * code equivalences.
 *)
let equivalence_mode = ref false
```

⟨*function* Apply_equivalences.match_e_e_for_equivalences 104b⟩≡ (447b)

```
let match_e_e_for_equivalences _ruleid a b =
  Common.save_excursion Flag.equivalence_mode true (fun () ->
      let config =
        {
          Config_semgrep.default_config with
          go_deeper_expr = false;
          go_deeper_stmt = false;
        }
      in
      let cache = None in
      let env = Matching_generic.empty_environment cache config in
      Generic_vs_generic.m_expr a b env)
```

8.4.2 Pattern metavariables matching and substiution

⟨*function* Apply_equivalences.subst_e 104c⟩≡ (447b)

```
let subst_e (env : Env.t) e =
  let bindings = env.full_env in
  let visitor =
    M.mk_visitor
      {
        M.default_visitor with
        M.kexpr =
          (fun (k, _) x ->
            match x with
            | N (Id ((str, _tok), _id_info)) when MV.is_metavar_name str -> (
```

```
                match List.assoc_opt str bindings with
                | Some (MV.Id (id, Some idinfo)) ->
                    (* less: abstract-line? *)
                    N (Id (id, idinfo))
                | Some (MV.E e) ->
                    (* less: abstract-line? *)
                    e
                | Some _ ->
                    failwith
                      (spf "incompatible metavar: %s, was expecting an expr"
                         str)
                | None ->
                    failwith
                      (spf "could not find metavariable %s in environment" str)
                )
            | _ -> k x);
      }
  in
  visitor.M.vexpr e
```

8.5 semgrep -dump_equivalences

⟨Main_semgrep_core.all_actions *dumper cases* 105a⟩+≡ (48d) ◁50h 198b▷

```
("-dump_equivalences", " <file>", Common.mk_action_1_arg dump_equivalences);
```

⟨*function* Main_semgrep_core.dump_equivalences 105b⟩≡ (369)

```
let dump_equivalences file =
  let xs = Parse_equivalences.parse file in
  pr2_gen xs
```

Chapter 9

Semantic Matching

9.1 Scoped metavariables

⟨*function* Generic_vs_generic.m_id_info 106a⟩≡ (428b)
```
  (* Currently m_id_info is a Nop because the Semgrep pattern usually
   * does not have correct name resolution (see the comment below).
   * However, we do use id_info in equal_ast() to check
   * whether two $X refers to the same code. In that case we are using
   * the id_resolved tag and sid!
   *)
  and m_id_info a b =
    match (a, b) with
    | ( { A.id_resolved = _a1; id_type = _a2; id_constness = _a3 },
        { B.id_resolved = _b1; id_type = _b2; id_constness = _b3 } ) ->
        (* old: (m_ref m_resolved_name) a3 b3  >>= (fun () ->
         * but doing import flask in a source file means every reference
         * to flask.xxx will be tagged with a ImportedEntity, but
         * semgrep pattern will use flask.xxx directly, without the preceding
         * import, without this tag, which would prevent
         * matching. We need to correctly resolve names and always compare with
         * the resolved_name instead of the name used in the code
         * (which can be an alias)
         *
         * old: (m_ref (m_option m_type_)) a2 b2
         * the same is true for types! Now we sometimes propagate type annotations
         * in Naming_AST.ml, but we do that in the source file, not the pattern,
         * which would prevent a match.
         * More generally, id_info is something populated and used on the
         * generic AST of the source, not on the pattern, hence we should
         * not use it as a condition for matching here. Instead use
         * the information in the caller.
         *)
        return ()
```

⟨*function* Generic_vs_generic.m_sid 106b⟩≡ (428b)
```
  let m_sid a b = if a =|= b then return () else fail ()
```

⟨*function* Generic_vs_generic._m_resolved_name 106c⟩≡ (428b)
```
  let _m_resolved_name (a1, a2) (b1, b2) =
    let* () = m_resolved_name_kind a1 b1 in
    m_sid a2 b2
```

9.2 Typed metavariables

⟨AST_generic.expr *semgrep extensions cases* 107a⟩+≡ (24d) ◁102e
```
  | TypedMetavar of ident * tok (* : *) * type_
```

⟨Generic_vs_generic.m_expr() *typed metavariable case* 107b⟩≡ (70b)
```
  (* metavar: typed! *)
  | A.TypedMetavar ((str, tok), _, t), e2 when MV.is_metavar_name str ->
      m_compatible_type (str, tok) t e2
```

⟨*signature* Typechecking_generic.compatible_type 107c⟩≡ (459a)
```
  val compatible_type : AST_generic.type_ -> AST_generic.expr -> bool
```

⟨*function* Typechecking_generic.compatible_type 107d⟩≡ (459b)
```
  (* very Python specific for now where the type is currently an OT_Expr
   * TODO: fill AST_generic.expr_to_type at least.
   *)
  let compatible_type t e =
    match (t, e) with
    | OtherType (OT_Expr, [ E (N (Id (("int", _tok), _idinfo))) ]), L (Int _) ->
        true
    | OtherType (OT_Expr, [ E (N (Id (("float", _tok), _idinfo))) ]), L (Float _)
      ->
        true
    | OtherType (OT_Expr, [ E (N (Id (("str", _tok), _idinfo))) ]), L (String _)
      ->
        true
    | TyBuiltin (t1, _), N (Id (_, { id_type; _ })) -> (
        match !id_type with Some (TyBuiltin (t2, _)) -> t1 = t2 | _ -> false )
    | TyN (Id ((t1, _), _)), N (Id (_, { id_type; _ })) -> (
        match !id_type with Some (TyN (Id ((t2, _), _))) -> t1 = t2 | _ -> false )
    | TyArray (_, TyN (Id ((t1, _), _))), N (Id (_, { id_type; _ })) -> (
        match !id_type with
        | Some (TyArray (_, TyN (Id ((t2, _), _)))) -> t1 = t2
        | _ -> false )
    | _ -> false
```

9.3 Propagated constants

⟨Generic_vs_generic.m_expr() *propagated constant case* 107e⟩≡ (70b)
```
  (* equivalence: constant propagation and evaluation!
   * TODO: too late, must do that before 'metavar:' so that
   * const a = "foo"; ... a == "foo" would be catched by $X == $X.
   *)
  | A.L a1, b1 ->
      if_config
        (fun x -> x.Config.constant_propagation)
        ~then_:
          ( match
              Normalize_generic.constant_propagation_and_evaluate_literal b1
            with
          | Some b1 -> m_literal_constness a1 b1
          | None -> fail () )
        ~else_:(fail ())
```

⟨*signature* Normalize_generic.constant_propagation_and_evaluate_literal 108a⟩≡ (453a)
```
val constant_propagation_and_evaluate_literal :
  AST_generic.expr -> AST_generic.constness option
```

⟨*function* Normalize_generic.eval 108b⟩≡ (453b)
```
(* see also Constant_propagation.ml. At some point we should remove
 * the code below and rely only on Constant_propagation.ml
 *)
let rec eval x : constness option =
  match x with
  | L x -> Some (Lit x)
  | N (Id (_, { id_constness = { contents = Some x }; _ }))
  | DotAccess
      ( IdSpecial (This, _),
        _,
        EN (Id (_, { id_constness = { contents = Some x }; _ })) ) ->
      Some x
  | Call (IdSpecial ((Op (Plus | Concat) | ConcatString _), _), args) -> (
      let literals =
        args |> unbracket
        |> Common.map_filter (fun arg ->
               match arg with Arg e -> eval e | _ -> None)
      in
      let strs =
        literals
        |> Common.map_filter (fun lit ->
               match lit with Lit (String (s1, _)) -> Some s1 | _ -> None)
      in
      let concated = String.concat "" strs in
      let all_args_are_string =
        List.length strs = List.length (unbracket args)
      in
      match List.nth_opt literals 0 with
      | Some (Lit (String (_s1, t1))) when all_args_are_string ->
          Some (Lit (String (concated, t1)))
      | _ -> None )
  (* TODO: partial evaluation for ints/floats/... *)
  | _ -> None
```

⟨*constant* Normalize_generic.constant_propagation_and_evaluate_literal 108c⟩≡ (453b)
```
let constant_propagation_and_evaluate_literal = eval
```

9.4 Resolved alias imports

⟨Generic_vs_generic.m_expr() *resolving alias case* 108d⟩≡ (70b)
```
(* equivalence: name resolving! *)
| ( a,
    B.N
      (B.Id
        ( idb,
          {
            B.id_resolved =
              {
                contents =
                  Some
```

```
                    ( ( B.ImportedEntity dotted
                      | B.ImportedModule (B.DottedName dotted) ),
                      _sid );
                };
              _;
          } )) ) ->
      (* We used to force to fully qualify entities in the pattern
       * (e.g., with org.foo(...)) but this is confusing for users.
       * We now allow an unqualified pattern like 'foo' to match resolved
       * entities like import org.foo; foo(), just like for attributes.
       *)
      m_expr a (B.N (B.Id (idb, B.empty_id_info ())))
      >||> (* try this time a match with the resolved entity *)
      m_expr a (make_dotted dotted)
  (* Put this before the next case to prevent overly eager dealiasing *)
  | A.N (A.IdQualified (a1, a2)), B.N (B.IdQualified (b1, b2)) ->
      m_name_ a1 b1 >>= fun () -> m_id_info a2 b2
  (* Matches pattern
   *   a.b.C.x
   * to code
   *   import a.b.C
   *   C.x
   *)
  | ( A.N
        (A.IdQualified
          ((alabel, { A.name_qualifier = Some (A.QDots names); _ }), _id_info)),
      b ) ->
      let full = names @ [ alabel ] in
      m_expr (make_dotted full) b
```

⟨*function* Generic_vs_generic.make_dotted 109a⟩≡ (428b)

```
  (* This is for languages like Python where foo.arg.func is not parsed
   * as a qualified name but as a chain of DotAccess.
   *)
  let make_dotted xs =
    match xs with
    | [] -> raise Impossible
    | x :: xs ->
        let base = B.N (B.Id (x, B.empty_id_info ())) in
        List.fold_left
          (fun acc e ->
            let tok = Parse_info.fake_info "." in
            B.DotAccess (acc, tok, B.EN (B.Id (e, B.empty_id_info ()))))
          base xs
```

⟨Generic_vs_generic.m_attribute *resolving alias case* 109b⟩≡ (76a)

```
  (* equivalence: name resolving! *)
  (* TODO: factorize with m_name again *)
  | ( a,
      B.NamedAttr
        ( t1,
          B.Id
            ( b1,
              {
                B.id_resolved =
                  {
                    contents =
                      Some
                        ( ( B.ImportedEntity dotted
```

```
                    | B.ImportedModule (B.DottedName dotted) ),
                    _sid );
            };
          _;
        } ),
    b2 ) ) ->
(* We also allow an unqualified pattern like @Attr to match resolved
 * one like import org.foo.Attr; @Attr *)
m_attribute a (B.NamedAttr (t1, B.Id (b1, B.empty_id_info ()), b2))
>|> m_attribute a (B.NamedAttr (t1, H.name_of_ids dotted, b2))
```

Chapter 10

Reporting

10.1 The match result

10.2 Reporting gradually

⟨*constant* Main_semgrep_core.match_format 111a⟩≡ (369)
```
let match_format = ref Matching_report.Normal
```

⟨Main_semgrep_core.options *report match mode cases* 111b⟩≡ (46d) 112e▷
```
( "-emacs",
  Arg.Unit (fun () -> match_format := Matching_report.Emacs),
  " print matches on the same line than the match position" );
( "-oneline",
  Arg.Unit (fun () -> match_format := Matching_report.OneLine),
  " print matches on one line, in normalized form" );
```

⟨*type* Matching_report.match_format 111c⟩≡ (457a 456)
```
type match_format =
  (* ex: tests/misc/foo4.php:3
   *  foo(
   *   1,
   *   2);
   *)
  | Normal
  (* ex: tests/misc/foo4.php:3: foo( *)
  | Emacs
  (* ex: tests/misc/foo4.php:3: foo(1,2) *)
  | OneLine
```

⟨*function* Main_semgrep_core.print_match 111d⟩≡ (369)
```
let print_match ?str mvars mvar_binding ii_of_any tokens_matched_code =
  (* there are a few fake tokens in the generic ASTs now (e.g.,
   * for DotAccess generated outside the grammar) *)
  let toks = tokens_matched_code |> List.filter PI.is_origintok in
  ( if mvars = [] then
    Matching_report.print_match ?str ~format:!match_format toks
  else
    ⟨Main_semgrep_core.print_match() when non empty mvars 115c⟩
    () );
  ⟨Main_semgrep_core.print_match() hook 124d⟩
```

⟨*signature* Matching_report.print_match 112a⟩≡ (456)
```
val print_match :
  ?format:match_format -> ?str:string -> Parse_info.t list -> unit
```

⟨*function* Matching_report.print_match 112b⟩≡ (457a)
```
let print_match ?(format = Normal) ?(str = "") ii =
  try
    let mini, maxi = PI.min_max_ii_by_pos ii in
    let file, line = (PI.file_of_info mini, PI.line_of_info mini) in
    let prefix = spf "%s:%d" file line in
    let arr = Common2.cat_array file in
    let lines = Common2.enum (PI.line_of_info mini) (PI.line_of_info maxi) in

    match format with
    | Normal ->
        let prefix = if str = "" then prefix else prefix ^ " " ^ str in
        pr prefix;
        (* todo? some context too ? *)
        lines
        |> List.map (fun i -> arr.(i))
        |> List.iter (fun s -> pr (" " ^ s))
    | Emacs -> pr (prefix ^ ": " ^ arr.(List.hd lines))
    | OneLine ->
        pr
          ( prefix ^ ": "
          ^ (ii |> List.map PI.str_of_info |> join_with_space_if_needed) )
  with Failure "get_pos: Ab or FakeTok" ->
    pr "<could not locate match, FakeTok or AbstractTok>"
```

⟨*signature* Matching_report.join_with_space_if_needed 112c⟩≡ (456)
```
val join_with_space_if_needed : string list -> string
```

⟨*function* Matching_report.join_with_space_if_needed 112d⟩≡ (457a)
```
(* When we print in the OneLine format we want to normalize the matched
 * expression or code and so only print the tokens in the AST (and not
 * the extra whitespace, newlines or comments). It's not enough though
 * to just List.map str_of_info because some PHP expressions such as
 * '$x = print FOO' would then be transformed into $x=printFOO, hence
 * this function
 *)
let rec join_with_space_if_needed xs =
  match xs with
  | [] -> ""
  | [ x ] -> x
  | x :: y :: xs ->
      if x =~ ".*[a-zA-Z0-9_]$" && y =~ "^[a-zA-Z0-9_]" then
        x ^ " " ^ join_with_space_if_needed (y :: xs)
      else x ^ join_with_space_if_needed (y :: xs)
```

10.3 JSON output

⟨Main_semgrep_core.options *report match mode cases* 112e⟩+≡ (46d) ◁111b
```
("-json", Arg.Unit (fun () -> output_format := Json), " output JSON format");
( "-json_time",
```

```
    Arg.Unit
      (fun () ->
        output_format := Json;
        report_time := true),
    " report detailed matching times as part of the JSON response. Implies \
     '-json'." );
```

⟨*function* Main_semgrep_core.print_matches_and_errors 113a⟩≡ (369)

⟨*signature* JSON_report.match_to_json 113b⟩≡ (453d)

```
  (* Can return an Error because when have a NoTokenLocation exn when
   * trying to get the range of a match or metavar.
   *)
  val match_to_json : Pattern_match.t -> (JSON.t, Error_code.error) Common.either
```

⟨*function* JSON_report.match_to_json 113c⟩≡ (454d)

```
  (* similar to pfff/h_program-lang/R2c.ml *)
  let match_to_json x =
    try
      let min_loc, max_loc = x.range_loc in
      let startp, endp = json_range min_loc max_loc in
      Left
        (J.Object
           [
             ("check_id", J.String x.rule_id.id);
             ("path", J.String x.file);
             ("start", startp);
             ("end", endp);
             ( "extra",
               J.Object
                 [
                   ("message", J.String x.rule_id.message);
                   ("metavars", J.Object (x.env |> List.map (json_metavar startp)));
                 ] );
           ])
      (* raised by min_max_ii_by_pos in range_of_any when the AST of the
       * pattern in x.code or the metavar does not contain any token
       *)
    with Parse_info.NoTokenLocation s ->
      let loc = Parse_info.first_loc_of_file x.file in
      let s =
        spf "NoTokenLocation with pattern %s, %s" x.rule_id.pattern_string s
      in
      let err = E.mk_error_loc loc (E.MatchingError s) in
      Right err
    [@@profiling]
```

⟨*constant* Main_semgrep_core.output_format_json 113d⟩≡ (369)

```
  let output_format = ref Text
```

10.4 Match ranges and Semgrep set-based operators

⟨*function* JSON_report.range_of_any 113e⟩≡ (454d)

```
  let range_of_any any =
    let min_loc, max_loc = V.range_of_any any in
```

```
  let startp, endp = json_range min_loc max_loc in
  (startp, endp)
```

⟨*function* JSON_report.json_range 114a⟩≡ (454d)
```
let json_range min_loc max_loc =
  (* pfff (and Emacs) have the first column at index 0, but not r2c *)
  let adjust_column x = x + 1 in

  let len_max = String.length max_loc.PI.str in
  ( J.Object
      [
        ("line", J.Int min_loc.PI.line);
        ("col", J.Int (adjust_column min_loc.PI.column));
        ("offset", J.Int min_loc.PI.charpos);
      ],
    J.Object
      [
        ("line", J.Int max_loc.PI.line);
        ("col", J.Int (adjust_column (max_loc.PI.column + len_max)));
        ("offset", J.Int (max_loc.PI.charpos + len_max));
      ] )
```

10.5 Match metavariables and single unique identifier

⟨*function* JSON_report.json_metavar 114b⟩≡ (454d)
```
let json_metavar startp (s, mval) =
  let any = MV.mvalue_to_any mval in
  let startp, endp =
    try range_of_any any
    with Parse_info.NoTokenLocation _exn ->
      raise
        (Parse_info.NoTokenLocation
           (spf "NoTokenLocation with metavar %s, close location = %s" s
              (J.string_of_json startp)))
  in
  ( s,
    J.Object
      [
        ("start", startp);
        ("end", endp);
        ( "abstract_content",
          J.String
            ( any |> V.ii_of_any
            |> List.filter PI.is_origintok
            |> List.sort Parse_info.compare_pos
            |> List.map PI.str_of_info
            |> Matching_report.join_with_space_if_needed ) );
        ("unique_id", unique_id any);
      ] )
```

⟨*function* JSON_report.unique_id 114c⟩≡ (454d)
```
(* Returning scoping-aware information about a metavariable, so that
 * the callers of sgrep (sgrep-lint) can check if multiple metavariables
 * reference the same entity, or reference exactly the same code.
 * See pfff/.../Naming_AST.ml for more information.
```

```
 *
 * TODO: provide a typed interface for the json object because it's really
 *       hard to see how to produce correct output. 'Semgrep.atd' in
 *       spacegrep already has definitions for about every type except this
 *       one.
 *)
let unique_id any =
  match any with
  | E
      (N
        (Id
          ((str, _tok), { id_resolved = { contents = Some (resolved, sid) }; _ })))
    ->
      J.Object
        [
          ("type", J.String "id");
          ("value", J.String str);
          ("kind", J.String (string_of_resolved resolved));
          (* single unique id *)
          ("sid", J.Int sid);
        ]
  (* not an Id, return a md5sum of its AST as a "single unique id" *)
  | _ ->
      (* todo? note that if the any use a parameter, or a local,
       * as in foo(x): return complex(x), then they will have different
       * md5sum because the parameter will be different! We may
       * want to abstract also the resolved information in those cases.
       *)
      let any = AST_generic_helpers.abstract_for_comparison_any any in
      (* alt: Using the AST dumper should work also.
       * let v = Meta_AST.vof_any any in
       * let s = OCaml.string_of_v v in
       *)
      let s = Marshal.to_string any [] in
      let md5 = Digest.string s in
      J.Object
        [ ("type", J.String "AST"); ("md5sum", J.String (Digest.to_hex md5)) ]
```

10.6 Reporting metavariable bindings: `semgrep -mvar`

⟨*constant* `Main_semgrep_core.mvars` 115a⟩≡ (369)

```
let mvars = ref ([] : Metavariable.mvar list)
```

⟨`Main_semgrep_core.options` *other cases* 115b⟩≡ (46d) 123j ▷

```
( "-pvar",
  Arg.String (fun s -> mvars := Common.split "," s),
  " <metavars> print the metavariables, not the matched code" );
```

⟨`Main_semgrep_core.print_match()` *when non empty* `mvars` 115c⟩≡ (111d)

```
(* similar to the code of Lib_matcher.print_match, maybe could
 * factorize code a bit.
 *)
let mini, _maxi = PI.min_max_ii_by_pos toks in
let file, line = (PI.file_of_info mini, PI.line_of_info mini) in

let strings_metavars =
```

```
  mvars
  |> List.map (fun x ->
         match Common2.assoc_opt x mvar_binding with
         | Some any ->
             any |> ii_of_any
             |> List.filter PI.is_origintok
             |> List.map PI.str_of_info
             |> Matching_report.join_with_space_if_needed
         | None -> failwith (spf "the metavariable '%s' was not binded" x))
in
pr (spf "%s:%d: %s" file line (Common.join ":" strings_metavars));
```

10.7 Reporting internal errors

⟨*signature* R2c.error_to_json 116a⟩≡ (257a)

```
val error_to_json: Error_code.error -> JSON.t
```

⟨*signature* R2c.string_of_errors 116b⟩≡ (257a)

```
val string_of_errors: Error_code.error list -> string
```

Chapter 11

Optimizations

11.1 Running search in parallel: `semgrep -j <cpus>`

⟨*constant* `Main_semgrep_core.ncores` 117a⟩≡ (369)
```
(* -j *)
let ncores = ref 1
```

⟨`Main_semgrep_core.options -j` *case* 117b⟩≡ (46d)
```
("-j", Arg.Set_int ncores, " <int> number of cores to use (default = 1)");
```

⟨*function* `Main_semgrep_core.map` 117c⟩≡ (369)
```
let map f xs =
  if !ncores <= 1 then List.map f xs
  else
    let n = List.length xs in
    (* Heuristic. Note that if you don't set a chunksize, Parmap
     * will evenly split the list xs, which does not provide any load
     * balancing.
     *)
    let chunksize =
      match n with
      | _ when n > 1000 -> 10
      | _ when n > 100 -> 5
      | _ when n = 0 -> 1
      | _ when n <= !ncores -> 1
      | _ -> n / !ncores
    in
    assert (!ncores > 0 && chunksize > 0);
    Parmap.parmap ~ncores:!ncores ~chunksize f (Parmap.L xs)
```

11.2 Tuning the GC

⟨`Main.main()` *tune the GC* 117d⟩≡ (37b)
```
Gc.set {(Gc.get ()) with Gc.stack_limit = 1000 * 1024 * 1024};
```

⟨*function* `Main_semgrep_core.set_gc` 117e⟩≡ (369)
```
let set_gc () =
  (*
  if !Flag.debug_gc
  then Gc.set { (Gc.get()) with Gc.verbose = 0x01F };
*)
```

```
(* only relevant in bytecode, in native the stacklimit is the os stacklimit,
 * which usually requires a ulimit -s 40000
 *)
Gc.set { (Gc.get ()) with Gc.stack_limit = 1000 * 1024 * 1024 };
(* see www.elehack.net/michael/blog/2010/06/ocaml-memory-tuning *)
Gc.set { (Gc.get ()) with Gc.minor_heap_size = 4_000_000 };
Gc.set { (Gc.get ()) with Gc.major_heap_increment = 8_000_000 };
Gc.set { (Gc.get ()) with Gc.space_overhead = 300 };
()
```

Chapter 12

Other Features

12.1 Filtering files

⟨Main_semgrep_core.options *file filters cases* 119a⟩≡ (46d)

⟨Main_semgrep_core.semgrep_with_one_pattern() *filter* files 119b⟩≡ (42c)

⟨*constant* Main_semgrep_core.excludes 119c⟩≡ (369)

⟨*constant* Main_semgrep_core.includes 119d⟩≡ (369)

⟨*constant* Main_semgrep_core.exclude_dirs 119e⟩≡ (369)

⟨*constant* Main_semgrep_core.include_dirs 119f⟩≡ (369)

⟨*function* Main_semgrep_core.filter_files 119g⟩≡ (369)

⟨*function* Main_semgrep_core.get_final_files 119h⟩≡ (369)

```
(* Small wrapper around Lang.files_of_dirs_or_files to also accept
 * an explicit list of files that may not be recognized as part
 * of the language (e.g., files without an extension but that we still
 * want to process)
 *)
let get_final_files lang xs =
  let files = Lang.files_of_dirs_or_files lang xs in
  let explicit_files =
    xs
    |> List.filter (fun file ->
           Sys.file_exists file && not (Sys.is_directory file))
  in
  Common2.uniq_eff (files @ explicit_files)
  [@@profiling]
```

⟨*signature* Files_finder.files_of_dirs_or_files 119i⟩≡ (405a)

```
val files_of_dirs_or_files: string list -> Common.filename list
```

⟨*function* Files_finder.files_of_dirs_or_files 119j⟩≡ (405b)

```
let files_of_dirs_or_files xs =
  Common.files_of_dir_or_files_no_vcs_nofilter xs
```

⟨*type* Files_filter.filters 119k⟩≡ (406 405c)

```
type filters = {
  excludes: glob list;
  includes: glob list;
  exclude_dirs: glob list;
  include_dirs: glob list;
}
```

⟨*type* Files_filter.glob 120a⟩≡ (406)
```
(* see https://dune.readthedocs.io/en/stable/concepts.html#glob *)
type glob = Glob.t
```

⟨*exception* Files_filter.GlobSyntaxError 120b⟩≡ (406 405c)
```
exception GlobSyntaxError of string
```

⟨*signature* Files_filter.mk_filters 120c⟩≡ (405c)
```
(* may raise GlobSyntaxError *)
val mk_filters:
  excludes: string list -> includes: string list -> exclude_dirs: string list -> include_dirs: string list ->
  filters
```

⟨*signature* Files_filter.filter 120d⟩≡ (405c)
```
(* entry point *)
val filter: filters -> Common.filename list -> Common.filename list
```

⟨*function* Files_filter.mk_filters 120e⟩≡ (406)
```
let mk_filters ~excludes ~includes ~exclude_dirs ~include_dirs =
 try
  { excludes = excludes |> List.map Glob.of_string;
    includes =
      if includes = []
      then [Glob.universal]
      else includes |> List.map Glob.of_string;
    exclude_dirs = exclude_dirs |> List.map Glob.of_string;
    include_dirs =
      if include_dirs = []
      then [Glob.universal]
      else include_dirs |> List.map Glob.of_string;
  }
 with Invalid_argument s -> raise (GlobSyntaxError s)
```

⟨*function* Files_filter.filter 120f⟩≡ (406)
```
let filter filters xs =
  xs |> List.filter (fun file ->
    let base = Filename.basename file in
    let dir = Filename.dirname file in
    let dirs = Str.split (Str.regexp "/") dir in
    (* todo? includes have priority over excludes? *)
    (filters.excludes |> List.for_all (fun glob -> not (Glob.test glob base)))
    &&
    (filters.includes |> List.exists (fun glob -> Glob.test glob base))
    &&
    (filters.exclude_dirs |> List.for_all
       (fun glob -> not (dirs |> List.exists (fun dir -> Glob.test glob dir))))
    &&
    (filters.include_dirs |> List.exists
       (fun glob -> (dirs |> List.exists (fun dir -> Glob.test glob dir))))
 )
```

⟨*signature* Unit_files.unittest 120g⟩≡ (407a)
```
(* Returns the testsuite for this directory To be concatenated by
 * the caller (e.g. in pfff/main_test.ml ) with other testsuites and
 * run via OUnit.run_test_tt
 *)
val unittest:
  OUnit.test
```

⟨*constant* `Unit_files.unittest` 121a⟩≡ (407b)

```
let unittest =
  "file filtering" >::: [
    "basic exclude/include" >:: (fun () ->
        let files = [
            "a/b/foo.c";
            "a/b/foo.js";
            "a/b/bar.c";
            "a/b/bar.js";
            "a/b/foo.go";
            "a/c/foo.c";
            "a/c/foo.js";
        ] in
        let filters = Files_filter.mk_filters
          ~excludes:["*.{c,h}"; "*.go"]
          ~includes:["foo.*"]
          ~exclude_dirs:["c"]
          ~include_dirs:["a"; "b"] in
        assert_equal ~msg:"it should filter files"
          ["a/b/foo.js"]
          (Files_filter.filter filters files)
    )
  ]
```

12.2 Dumping the AST for Semgrep live

⟨*function* `Main_semgrep_core.json_of_v` 121b⟩≡ (369)

```
let json_of_v (v : OCaml.v) =
  let rec aux v =
    match v with
    | OCaml.VUnit -> J.String "()"
    | OCaml.VBool v1 -> if v1 then J.String "true" else J.String "false"
    | OCaml.VFloat v1 -> J.Float v1 (* ppf "%f" v1 *)
    | OCaml.VChar v1 -> J.String (spf "'%c'" v1)
    | OCaml.VString v1 -> J.String v1
    | OCaml.VInt i -> J.Int i
    | OCaml.VTuple xs -> J.Array (List.map aux xs)
    | OCaml.VDict xs -> J.Object (List.map (fun (k, v) -> (k, aux v)) xs)
    | OCaml.VSum (s, xs) -> (
        match xs with
        | [] -> J.String (spf "%s" s)
        | [ one_element ] -> J.Object [ (s, aux one_element) ]
        | _ -> J.Object [ (s, J.Array (List.map aux xs)) ] )
    | OCaml.VVar (s, i64) -> J.String (spf "%s_%d" s (Int64.to_int i64))
    | OCaml.VArrow _ -> failwith "Arrow TODO"
    | OCaml.VNone -> J.Null
    | OCaml.VSome v -> J.Object [ ("some", aux v) ]
    | OCaml.VRef v -> J.Object [ ("ref@", aux v) ]
    | OCaml.VList xs -> J.Array (List.map aux xs)
    | OCaml.VTODO _ -> J.String "VTODO"
  in
  aux v
```

12.3 Linting the linter

⟨*signature* `Check_semgrep.check_pattern` 121c⟩≡ (407c)

```
val check : Lang.t -> AST_generic.any -> unit
(** Check semgrep patterns for potential issues. Will raise exn if issue. *)
```

⟨*constant* Check_semgrep.lang_has_no_dollar_ids 122a⟩≡ (407d)
```
(* for these languages, we are sure that $x is an error *)
let lang_has_no_dollar_ids =
  Lang.(
    function
    | Python | Python2 | Python3 | Java | Go | C | Cplusplus | OCaml | JSON
    | Yaml | Csharp | Kotlin | Lua | R ->
        true
    | Javascript | Ruby | Typescript | PHP | Rust -> false)
```

⟨*function* Check_semgrep.check_pattern_metavars 122b⟩≡ (407d)
```
let check_pattern_metavars error lang ast =
  let kident_metavar (k, _out) ((str, _tok) as ident) =
    if
      str.[0] = '$'
      && (not (Metavariable.is_metavar_name str))
      && not (Metavariable.is_metavar_ellipsis str)
    then
      error
        (Common.spf
           "`%s' is neither a valid identifier in %s nor a valid meta-variable"
           str (Lang.string_of_lang lang));
    k ident
  in
  if lang_has_no_dollar_ids lang then
    Visitor_AST.(
      mk_visitor { default_visitor with kident = kident_metavar } ast)
```

⟨*function* Check_semgrep.check_pattern 122c⟩≡ (407d)
```
let check lang ast =
  let error s = failwith s in
  check_pattern_metavars error lang ast
```

12.4 Validating the pattern

⟨*function* Main_semgrep_core.read_all 122d⟩≡ (369)
```
(* We do not use the easier Stdlib.input_line here because this function
 * does remove newlines (and may do other clever things), but
 * newlines have a special meaning in some languages
 * (e.g., Python), so we use the lower-level Stdlib.input instead.
 *)
let rec read_all chan =
  let buf = Bytes.create 4096 in
  let len = input chan buf 0 4096 in
  if len = 0 then ""
  else
    let rest = read_all chan in
    Bytes.sub_string buf 0 len ^ rest
```

⟨Main_semgrep_core.all_actions *other cases* 123a⟩≡ (48d)
```
( "--validate-pattern-stdin",
  " check the syntax of a pattern ",
  Common.mk_action_0_arg validate_pattern );
```

⟨*function* Main_semgrep_core.validate_pattern 123b⟩≡ (369)
```
(* works with -lang *)
let validate_pattern () =
  let chan = stdin in
  let s = read_all chan in
  try
    let lang = lang_of_string !lang in
    let _ = parse_pattern lang s in
    exit 0
  with _exn -> exit 1
```

12.5 Fuzzy matching

⟨*type* Parse_info.token_kind 123c⟩≡ (252k)
```
type token_kind =
  | LPar | RPar
  | LBrace | RBrace
  | LBracket | RBracket
  | LAngle | RAngle
  | Esthet of esthet
  | Eof
  | Other
```

⟨*type* Parse_info.esthet 123d⟩≡ (252k)
```
and esthet =
  | Comment
  | Newline
  | Space
```

⟨Main_semgrep_core.ast *other cases* 123e⟩≡ (43a)

⟨Main_semgrep_core.pattern *other cases* 123f⟩≡ (43b)

⟨Main_semgrep_core.sgrep_ast() *match* pattern *and* any_ast *other cases* 123g⟩≡ (43c)

⟨Main_semgrep_core.create_ast() *when not a supported language* 123h⟩≡ (52a)

⟨Main_semgrep_core.parse_pattern() *when not a supported language* 123i⟩≡ (55a)

12.6 Codemap layer

⟨Main_semgrep_core.options *other cases* 123j⟩+≡ (46d) ◁115b 193b▷
```
( "-gen_layer",
  Arg.String (fun s -> Experiments.layer_file := Some s),
  " <file> save result in a codemap layer file" );
```

⟨Main_semgrep_core.semgrep_with_one_pattern() *optional layer generation* 123k⟩≡ (42c)
```
Experiments.gen_layer_maybe _matching_tokens pattern_string xs
```

⟨Main.pfff_extra_actions *other cases* 124a⟩+≡ (39b) ◁39d
```
"-layer_stat", " <file>",
Common.mk_action_1_arg Test_program_lang.layer_stat;
```

⟨*constant* Main_semgrep_core.layer_file 124b⟩≡ (369)

⟨*constant* Main_semgrep_core._matching_tokens 124c⟩≡ (369)
```
(* for -gen_layer, see Experiments.ml *)
let _matching_tokens = ref []
```

⟨Main_semgrep_core.print_match() *hook* 124d⟩≡ (111d)
```
toks |> List.iter (fun x -> Common.push x _matching_tokens)
```

⟨*function* Main_semgrep_core.gen_layer 124e⟩≡ (369)

Part II

Static Analysis Components

Chapter 13

The Concrete Syntax Tree (CST)

Chapter 14

The Abstract Syntax Tree (AST): Python example

14.1 Positions and tokens

⟨*type* AST_python.tok 127a⟩≡ (329)
```
(* Contains among other things the position of the token through
 * the Parse_info.token_location embedded inside it, as well as the
 * transformation field that makes possible spatch on the code.
*)
type tok = Parse_info.t
```

⟨*type* AST_python.wrap 127b⟩≡ (329)
```
(* a shortcut to annotate some information with token/position information *)
type 'a wrap = 'a * tok
```

⟨*type* AST_python.bracket 127c⟩≡ (329)
```
(* round(), square[], curly{}, angle<> brackets *)
type 'a bracket = tok * 'a * tok
```

14.2 Identifiers and names

⟨*type* AST_python.name 127d⟩≡ (329)
```
type name = string wrap
```

⟨*constant* AST_python.str_of_name 127e⟩≡ (329)
```
let str_of_name = fst
```

⟨*type* AST_python.dotted_name 127f⟩≡ (329)
```
(* note that name can be also the special "*" in an import context. *)
type dotted_name = name list
```

⟨*type* AST_python.module_name 127g⟩≡ (329)
```
type module_name =
  dotted_name *
  (* https://realpython.com/absolute-vs-relative-python-imports/ *)
  (tok (* . or ... toks *) list) option (* levels, for relative imports *)
```

⟨*type* AST_python.resolved_name 128a⟩≡ (329)
```
  (* TODO: reuse AST_generic one? *)
  type resolved_name =
    (* this can be computed by a visitor *)
    | LocalVar
    | Parameter
    | GlobalVar
    | ClassField
    (* both dotted_name should contain at least one element! *)
    | ImportedModule of dotted_name
    | ImportedEntity of dotted_name

    (* default case *)
    | NotResolved
```

14.3 Expressions

⟨*type* AST_python.expr 128b⟩≡ (329)
```
  type expr =
    | Num of number (* n *)
    | Str of string wrap (* s *)
    (* todo: we should split the token in r'foo' in two, one string wrap
     * for the prefix and a string wrap for the string itself. *)
    | EncodedStr of string wrap * string (* prefix *)
    (* python3: now officially reserved keywords *)
    | Bool of bool wrap
    | None_ of tok

    (* introduce new vars when expr_context = Store *)
    | Name of name (* id *) * expr_context (* ctx *) * resolved_name ref

    | Tuple of expr list_or_comprehension * expr_context
    | List  of expr list_or_comprehension * expr_context
    | DictOrSet of dictorset_elt list_or_comprehension

    (* python3: *)
    | ExprStar of expr (* less: expr_context? always Store anyway no? *)
    (* python3: f-strings
     * reference: https://www.python.org/dev/peps/pep-0498/ *)
    | InterpolatedString of interpolated list
    | ConcatenatedString of interpolated list (* always Str *)

    (* python3: *)
    (* inside an Assign (or ExprStmt) *)
    | TypedExpr of expr * type_
    (* sgrep-ext: *)
    | Ellipsis of tok (* should be only in .pyi, types Dict[str,...], or sgrep *)
    | DeepEllipsis of expr bracket
    | TypedMetavar of name * tok * type_

    | BoolOp of boolop wrap (* op *) * expr list (* values *)
    | BinOp of expr (* left *) * operator wrap (* op *) * expr (* right *)
    | UnaryOp of unaryop wrap (* op *) * expr (* operand *)
    | Compare of expr (* left *) * cmpop wrap list (* ops *) * expr list (* comparators *)

    (* note that Python does not have a 'new' keyword, a call with the name
     * of a class is a New *)
    | Call of expr (* func *) * argument list bracket (* args *)
```

```
    | Subscript of expr (* value *) * slice list bracket (* slice *) *
                 expr_context

    (* the parameters do not have types here *)
    | Lambda of tok (* lambda *) * parameters (* args *) * tok (* : *) *
                expr (* body *)

    | IfExp of expr (* test *) * expr (* body *) * expr (* orelse *)

    | Yield of tok * expr option (* value *) * bool (* is_yield_from *)
    (* python3: *)
    | Await of tok * expr

    (* python 3.8+; see https://www.python.org/dev/peps/pep-0572/ *)
    | NamedExpr of expr * tok * expr
    | Repr of expr bracket (* `` *)
    (* =~ ObjAccess *)
    | Attribute of expr (* value *) * tok (* . *) * name (* attr *) *
                 expr_context (* ctx *)
```

⟨*type* AST_python.number 129a⟩≡ (329)
```
  and number =
    | Int of int option wrap
    | LongInt of int option wrap
    | Float of float option wrap
    | Imag of string wrap
```

⟨*type* AST_python.boolop 129b⟩≡ (329)
```
  and boolop = And | Or
```

⟨*type* AST_python.operator 129c⟩≡ (329)
```
  and operator =
    | Add | Sub | Mult | Div
    | Mod | Pow | FloorDiv
    | LShift | RShift
    | BitOr | BitXor | BitAnd
    | MatMult (* Matrix Multiplication *)
```

⟨*type* AST_python.unaryop 129d⟩≡ (329)
```
  and unaryop = Invert | Not | UAdd | USub
```

⟨*type* AST_python.cmpop 129e⟩≡ (329)
```
  and cmpop =
    | Eq | NotEq
    | Lt | LtE | Gt | GtE
    | Is | IsNot
    | In | NotIn
```

⟨*type* AST_python.interpolated 129f⟩≡ (329)
```
  and interpolated = expr
```

⟨*type* AST_python.list_or_comprehension 129g⟩≡ (329)
```
  and 'a list_or_comprehension =
    | CompList of 'a list bracket
    | CompForIf of 'a comprehension
```

⟨*type* AST_python.comprehension 129h⟩≡ (329)
```
  and 'a comprehension = 'a * for_if list
```

⟨*type* AST_python.for_if 130a⟩≡ (329)
```
  and for_if =
    | CompFor of expr (* introduce new vars *) * (* in *) expr
    | CompIf of expr
```

⟨*type* AST_python.dictorset_elt 130b⟩≡ (329)
```
  and dictorset_elt =
    | KeyVal of expr * expr
    | Key of expr
    (* python3: *)
    | PowInline of expr
```

⟨*type* AST_python.expr_context 130c⟩≡ (329)
```
  and expr_context =
    | Load | Store
    | Del
    | AugLoad | AugStore
    | Param
```

⟨*type* AST_python.slice 130d⟩≡ (329)
```
  and slice =
    | Slice of expr option (* lower *) * expr option (* upper *) * expr option (* step *)
    | Index of expr (* value *)
```

⟨*type* AST_python.parameter 130e⟩≡ (329)
```
  and parameter =
    (* the first expr can only be a Name or a Tuple (pattern?),
     * and the Name can have a type associated with it
    *)
    | ParamDefault of (name * type_ option) * expr (* default value *)
    (* pattern can be either a name or a tuple pattern *)
    | ParamPattern of param_pattern * type_ option
    | ParamStar of tok (* '*' *)  * (name * type_ option)
    | ParamPow  of tok (* '**' *) * (name * type_ option)
    (* python3: single star delimiter to force keyword-only arguments after.
     * reference: https://www.python.org/dev/peps/pep-3102/ *)
    | ParamSingleStar of tok
    (* python3: single slash delimiter to force positional-only arg prior. *)
    | ParamSlash of tok
    (* sgrep-ext: *)
    | ParamEllipsis of tok
```

⟨*type* AST_python.parameters 130f⟩≡ (329)
```
  and parameters = parameter list
```

⟨*type* AST_python.argument 130g⟩≡ (329)
```
  and argument =
    | Arg of expr (* this can be Ellipsis for sgrep *)
    | ArgKwd of name (* arg *) * expr (* value *)
    | ArgStar of (* '*' *)  tok * expr
    | ArgPow  of (* '**' *) tok * expr
    | ArgComp of expr * for_if list
```

14.4 Statements

⟨*type* AST_python.stmt 131⟩≡ (329)

```
type stmt =
  | ExprStmt of expr (* value *)

  (* the left expr should be an lvalue: Name, List, Tuple, Subscript,
   * or Attribute, or ExprStar, which are anything with an expr_context
   * (see also Parser_python.set_expr_ctx).
   * This can introduce new vars.
   * TODO: why take an expr list? can reuse Tuple for tuple assignment
  *)
  | Assign of expr list (* targets *) * tok * expr (* value *)
  | AugAssign of expr (* target *) * operator wrap (* op *) * expr (* value *)

  | For of tok * pattern (* (pattern) introduce new vars *) *
           tok * expr (* 'in' iter *) *
           stmt list (* body *) * stmt list (* orelse *)
  | While of tok * expr (* test *) * stmt list (* body *) *
             stmt list (* orelse *)
  | If of tok * expr (* test *) * stmt list (* body *) *
          stmt list option (* orelse *)
  (* https://docs.python.org/2.5/whatsnew/pep-343.html *)
  | With of tok * expr (* context_expr *) * expr option (* optional_vars *) *
            stmt list (* body *)

  | Return of tok * expr option (* value *)
  | Break of tok | Continue of tok
  | Pass of tok

  | Raise of tok * (expr * expr option (* from *)) option
  | RaisePython2 of tok * expr * expr option (* arguments *) * expr option (* location *)
  | TryExcept of tok * stmt list (* body *) * excepthandler list (* handlers *)
                 * stmt list (* orelse *)
  | TryFinally of tok * stmt list (* body *) * tok * stmt list (* finalbody *)
  | Assert of tok * expr (* test *) * expr option (* msg *)

  | Global of tok * name list (* names *)
  | Delete of tok * expr list (* targets *)
  (* python3: *)
  | NonLocal of tok * name list (* names *)

  (* python2: *)
  | Print of tok * expr option (* dest *) * expr list (* values *) * bool (* nl *)
  | Exec of tok * expr (* body *) * expr option (* glob *) * expr option (* local *)

  (* python3: for With, For, and FunctionDef *)
  | Async of tok * stmt

  | ImportAs   of tok * module_name (* name *) * name option (* asname *)
  | ImportAll  of tok * module_name * tok (* * *)
  | ImportFrom of tok * module_name (* module *) * alias list (* names *)

  (* should be allowed just at the toplevel *)
  | FunctionDef of
    tok (* 'def' *) *
    name (* name *) *
    parameters (* args *) *
    type_ option * (* return type *)
    stmt list (* body *) *
```

```
    decorator list (* decorator_list *)

  | ClassDef of
    tok (* 'class' *) *
    name (* name *) *
    type_parent list (* bases *) *
    stmt list (* body *) *
    decorator list (* decorator_list *)
```

⟨*type* AST_python.excepthandler 132a⟩≡ (329)
```
and excepthandler =
    ExceptHandler of
      tok *
      expr option (* type, possibly a list of types as in (Error,Fatal) *) *
      name option (* name, introduce new var, todo: only if pattern is Some *) *
      stmt list (* body *)
```

14.5 Directives

⟨*type* AST_python.alias 132b⟩≡ (329)
```
and alias = name (* name *) * name option (* asname *)
```

14.6 Types, patterns, and attributes

⟨*type* AST_python.type_ 132c⟩≡ (329)
```
and type_ = expr
```

⟨*type* AST_python.type_parent 132d⟩≡ (329)
```
and type_parent = argument
```

⟨*type* AST_python.pattern 132e⟩≡ (329)
```
(* Name, or Tuple? or more? *)
and pattern = expr
```

⟨*type* AST_python.decorator 132f⟩≡ (329)
```
and decorator = tok (* @ *) * dotted_name * argument list bracket option
```

14.7 The program

⟨*type* AST_python.program 132g⟩≡ (329)
```
type program = stmt list
```

⟨*type* AST_python.any 132h⟩≡ (329)
```
type any =
  | Expr of expr
  | Stmt of stmt
  | Stmts of stmt list
  | Program of program

  | DictElem of dictorset_elt
```

14.8 Semgrep extensions to the Python AST

14.9 pfff -dump_python <file>

⟨*function* Test_parsing_python.test_dump_python 133⟩≡ (366b)

```
let test_dump_python file =
  Common.save_excursion Flag.error_recovery true (fun () ->
    Common.save_excursion Flag.exn_when_lexical_error false (fun () ->
      let ast = Parse_python.parse_program file in
      let s = AST_python.show_program ast in
      pr s
    ))
```

Chapter 15

Parsing code: Python example

15.1 The Python lexer

⟨*function* Parse_python.error_msg_tok 134a⟩≡ (360)
```
let error_msg_tok tok =
  Parse_info.error_message_info (TH.info_of_tok tok)
```

15.1.1 Lexing states

⟨*function* Parse_python.tokens2 134b⟩≡ (360)
```
let tokens parsing_mode file =
  let state = Lexer.create () in
  let python2 = parsing_mode = Python2 in
  let token lexbuf =
    match Lexer.top_mode state with
    | Lexer.STATE_TOKEN ->
        Lexer.token python2 state lexbuf
    | Lexer.STATE_OFFSET ->
        failwith "impossibe STATE_OFFSET in python lexer"
    | Lexer.STATE_UNDERSCORE_TOKEN ->
        let tok = Lexer._token python2 state lexbuf in
        (match tok, Lexer.top_mode state with
         | T.TCommentSpace _, _ -> ()
         | T.FSTRING_START _, _ -> ()
         | _, Lexer.STATE_UNDERSCORE_TOKEN ->
             (* Note that _token() may have changed the top state.
              * For example, after having lexed 'f"{foo '", which puts us in a state
              * ST_UNDERSCORE_TOKEN with a full stack of [ST_UNDERSCORE_TOKEN;
              * ST_IN_F_STRING_DOUBLE; ST_UNDERSCORE_TOKEN], encountering a '}' will
              * pop the stack and leave ST_IN_FSTRING_DOUBLE at the top, which we
              * don't want to replace with ST_TOKEN. This is why we should switch
              * back to ST_TOKEN only when the current state is
              * STATE_UNDERSCORE_TOKEN. *)
             Lexer.set_mode state Lexer.STATE_TOKEN
         | _ -> ()
        );
        tok
    | Lexer.STATE_IN_FSTRING_SINGLE pre ->
        Lexer.fstring_single state pre lexbuf
    | Lexer.STATE_IN_FSTRING_DOUBLE pre ->
        Lexer.fstring_double state pre lexbuf
    | Lexer.STATE_IN_FSTRING_TRIPLE_SINGLE pre ->
        Lexer.fstring_triple_single state pre lexbuf
    | Lexer.STATE_IN_FSTRING_TRIPLE_DOUBLE pre ->
```

```
      Lexer.fstring_triple_double state pre lexbuf
  in
  Parse_info.tokenize_all_and_adjust_pos ~unicode_hack:true
    file token TH.visitor_info_of_tok TH.is_eof
```

⟨*function* Parse_python.tokens 135a⟩≡ (360)

15.1.2 Post-lexing position setting

⟨*signature* Token_helpers_python.is_eof 135b⟩≡ (363a)
```
val is_eof          : Parser_python.token -> bool
```

⟨*signature* Token_helpers_python.is_comment 135c⟩≡ (363a)
```
val is_comment      : Parser_python.token -> bool
```

⟨*function* Token_helpers_python.is_eof 135d⟩≡ (365)
```
let is_eof = function
  | EOF _ -> true
  | _ -> false
```

⟨*function* Token_helpers_python.is_comment 135e⟩≡ (365)
```
let is_comment = function
  | TComment _ | TCommentSpace _ -> true
  | _ -> false
```

⟨*signature* Token_helpers_python.visitor_info_of_tok 135f⟩≡ (363a)
```
val visitor_info_of_tok :
  (Parse_info.t -> Parse_info.t) -> Parser_python.token -> Parser_python.token
```

⟨*signature* Token_helpers_python.info_of_tok 135g⟩≡ (363a)
```
val info_of_tok :
  Parser_python.token -> Parse_info.t
```

⟨*function* Token_helpers_python.info_of_tok 135h⟩≡ (365)
```
let info_of_tok tok =
  let res = ref None in
  visitor_info_of_tok (fun ii -> res := Some ii; ii) tok |> ignore;
  match !res with
  | Some x -> x
  | None -> Parse_info.fake_info "NOTOK"
```

15.1.3 pfff -tokens_python <file>

⟨*signature* Test_parsing_python.test_tokens_python 135i⟩≡ (366a)
```
(* Print the set of tokens in a .py file *)
val test_tokens_python : Common.filename -> unit
```

⟨*function* Test_parsing_python.test_tokens_python 135j⟩≡ (366b)
```
let test_tokens_python file =
  if not (file =~ ".*\\.py")
  then pr2 "warning: seems not a python file";

  Flag.verbose_lexing := true;
  Flag.verbose_parsing := true;
  Flag.exn_when_lexical_error := true;
  let parsing_mode = Parse_python.Python in

  let toks = Parse_python.tokens parsing_mode file
             |> Parsing_hacks_python.fix_tokens in
  toks |> List.iter (fun x -> pr2_gen x);
  ()
```

15.2 Semgrep extensions to the Python lexer

15.3 The Python grammar

15.4 Semgrep extensions to the Python grammar

15.5 The Python parser

⟨*type* Parse_python.program_and_tokens 136a⟩≡ (360 359c)

⟨*type* Parse_python.parsing_mode 136b⟩≡ (360 359c)
```
type parsing_mode =
  | Python2
  | Python3
  (* will start with Python3 and fallback to Python2 in case of an error *)
  | Python
```

⟨*signature* Parse_python.parse 136c⟩≡ (359c)
```
(* This is the main function.
 * can throw Parse_info.Lexical_error and Parse_info.Parsing_error.
 * The token list in parsing_result contains also the comment-tokens.
*)
val parse:
  ?parsing_mode:parsing_mode (* default mode is Python *) ->
  Common.filename ->
  (AST_python.program, Parser_python.token) Parse_info.parsing_result
```

⟨*signature* Parse_python.parse_program 136d⟩≡ (359c)
```
val parse_program:
  ?parsing_mode:parsing_mode ->
  Common.filename -> AST_python.program
```

⟨*function* Parse_python.parse_basic 136e⟩≡ (360)
```
let rec parse_basic ?(parsing_mode=Python) filename =
  let stat = Parse_info.default_stat filename in

  (* this can throw Parse_info.Lexical_error *)
  let toks = tokens parsing_mode filename in
  let toks = Parsing_hacks_python.fix_tokens toks in

  let tr, lexer, lexbuf_fake =
    Parse_info.mk_lexer_for_yacc toks TH.is_comment in

  try
    (* ---------------------------------------------------- *)
    (* Call parser *)
    (* ---------------------------------------------------- *)
    let xs =
      Common.profile_code "Parser_python.main" (fun () ->
        Parser_python.main lexer lexbuf_fake
      )
    in
    { Parse_info. ast = xs; tokens = toks; stat }

  with Parsing.Parse_error ->

    (* There are still lots of python2 code out there, it would be sad to
     * not parse them just because they use the print and exec special
```

```
       * statements, which are not compatible with python3, hence
       * the special error recovery trick below.
       * For the rest Python2 is mostly compatible with Python3.
       *
       * Note that we do the error recovery only when we think a print
       * or exec identifiers was involved. Otherwise every parse errors
       * would trigger a parsing with a python2 mode, which change the
       * significance of the print and exec identifiers, which may give
       * strange error messages for python3 code.
      *)
      if parsing_mode = Python &&
         (tr.PI.passed |> Common.take_safe 10 |> List.exists (function
            | T.NAME (("print" | "exec"), _)
            | T.ASYNC _ | T.AWAIT _ | T.NONLOCAL _ | T.TRUE _ | T.FALSE _
              -> true
            | _ -> false))
      then
        (* note that we cant use tokens as the tokens are actually different
         * in Python2 mode, but we could optimize things a bit and just
         * transform those tokens here *)
        parse_basic ~parsing_mode:Python2 filename
      else begin
        let cur = tr.PI.current in
        if not !Flag.error_recovery
        then raise (PI.Parsing_error (TH.info_of_tok cur));

        if !Flag.show_parsing_error
        then begin
          pr2 ("parse error \n = " ^ error_msg_tok cur);

          let filelines = Common2.cat_array filename in
          let checkpoint2 = Common.cat filename |> List.length in
          let line_error = PI.line_of_info (TH.info_of_tok cur) in
          Parse_info.print_bad line_error (0, checkpoint2) filelines;
        end;
        stat.PI.error_line_count <- stat.PI.total_line_count;
        { Parse_info. ast = []; tokens = toks; stat }
      end
```

⟨*function* Parse_python.parse 137a⟩≡ (360)

```
  let parse ?parsing_mode a =
    Common.profile_code "Parse_python.parse" (fun () ->
      parse_basic ?parsing_mode a)
```

⟨*function* Parse_python.parse_program 137b⟩≡ (360)

```
  let parse_program ?parsing_mode file =
    let res = parse ?parsing_mode file in
    res.PI.ast
```

15.5.1 Parsing hacks: whitespace layout

⟨*signature* Parsing_hacks_python.fix_tokens 137c⟩≡ (362a)

```
  val fix_tokens: Parser_python.token list -> Parser_python.token list
```

⟨*function* Parsing_hacks_python.add_dedent_aux 137d⟩≡ (362b)

```
  let rec add_dedent_aux num ii xs =
    if num <= 0
    then xs
    else T.DEDENT ii::add_dedent_aux (num - 1) ii xs
```

⟨*function* Parsing_hacks_python.add_dedent 138a⟩≡ (362b)
```
let add_dedent num ii xs =
  if num <= 0
  then xs
  (* this closes the small_stmt from the stmt_list in suite (see grammar)
   * which then can be reduced by the series of DEDENT created by
   * add_dedent_aux.
  *)
  else T.NEWLINE ii::add_dedent_aux num ii xs
```

⟨*function* Parsing_hacks_python.fix_tokens 138b⟩≡ (362b)
```
let fix_tokens toks =
  let rec aux indent xs =
    match xs with
    | [T.NEWLINE ii; T.EOF _] -> add_dedent indent ii xs
    | [T.EOF ii] -> add_dedent indent ii [T.NEWLINE ii; T.EOF ii]
    | [] -> raise Common.Impossible
    | x::xs ->
        let new_indent =
          match x with
          | T.INDENT _ -> indent + 1
          | T.DEDENT _ -> indent - 1
          | _ -> indent
        in
        x::aux new_indent xs
  in
  aux 0 toks
```

15.5.2 pfff -parse_python <file>

⟨*signature* Test_parsing_python.actions 138c⟩≡ (366a)
```
(* This makes accessible the different test_xxx functions above from
 * the command line, e.g. '$ pfff -parse_python foo.py will call the
 * test_parse_python function.
*)
val actions : unit -> Common.cmdline_actions
```

⟨*function* Test_parsing_python.test_parse_python_common 138d⟩≡ (366b)
```
let test_parse_python_common parsing_mode xs =
  let xs = List.map Common.fullpath xs in

  let fullxs =
    Lib_parsing_python.find_source_files_of_dir_or_files xs
    |> Skip_code.filter_files_if_skip_list ~root:xs
  in

  let stat_list = ref [] in
  let newscore  = Common2.empty_score () in
  let ext = "python" in

  fullxs |> Console.progress (fun k -> List.iter (fun file ->
    k();

    let { Parse_info. stat; _ } =
      Common.save_excursion Flag.error_recovery true (fun () ->
        Common.save_excursion Flag.exn_when_lexical_error false (fun () ->
          Parse_python.parse ~parsing_mode file
        )) in
    Common.push stat stat_list;
```

```
    let s = spf "bad = %d" stat.Parse_info.error_line_count in
    if stat.Parse_info.error_line_count = 0
    then Hashtbl.add newscore file (Common2.Ok)
    else Hashtbl.add newscore file (Common2.Pb s)
  ));
  Parse_info.print_parsing_stat_list !stat_list;
  Parse_info.print_regression_information ~ext xs newscore;
  ()
```

⟨*function* Test_parsing_python.actions 139a⟩≡ (366b)

```
let actions () = [
  "-tokens_python", "   <file>",
  Common.mk_action_1_arg test_tokens_python;
  "-parse_python", "   <files or dirs>",
  Common.mk_action_n_arg (test_parse_python_common Parse_python.Python);
  "-parse_python2", "   <files or dirs>",
  Common.mk_action_n_arg (test_parse_python_common Parse_python.Python2);
  "-parse_python3", "   <files or dirs>",
  Common.mk_action_n_arg (test_parse_python_common Parse_python.Python3);
  "-dump_python", "   <file>",
  Common.mk_action_1_arg test_dump_python;
]
```

15.6 Semgrep extensions to the Python parser

⟨*signature* Parse_python.any_of_string 139b⟩≡ (359c)

```
(* for semgrep *)
val any_of_string:
  ?parsing_mode:parsing_mode -> string -> AST_python.any
```

⟨*function* Parse_python.any_of_string 139c⟩≡ (360)

```
(* for sgrep/spatch *)
let any_of_string ?(parsing_mode=Python) s =
  Common.save_excursion Flag_parsing.sgrep_mode true (fun () ->
    Common2.with_tmp_file ~str:s ~ext:"py" (fun file ->
      let toks = tokens parsing_mode file in
      let toks = Parsing_hacks_python.fix_tokens toks in
      let _tr, lexer, lexbuf_fake = PI.mk_lexer_for_yacc toks TH.is_comment in
      (* ------------------------------------------------------ *)
      (* Call parser *)
      (* ------------------------------------------------------ *)
      Parser_python.sgrep_spatch_pattern lexer lexbuf_fake
    ))
```

15.7 Python AST utilities

15.7.1 Visitor

⟨*type* Visitor_python.visitor_in 139d⟩≡ (367a)

```
type visitor_in = {
  kexpr: (expr  -> unit) * visitor_out -> expr  -> unit;
  kstmt: (stmt  -> unit) * visitor_out -> stmt  -> unit;
  ktype_: (type_  -> unit) * visitor_out -> type_  -> unit;
  kdecorator: (decorator  -> unit) * visitor_out -> decorator  -> unit;
  kparameter: (parameter  -> unit) * visitor_out -> parameter  -> unit;
  kinfo: (tok -> unit)  * visitor_out -> tok  -> unit;
}
```

⟨*type* Visitor_python.visitor_out 140a⟩≡ (367a)
```
and visitor_out = any -> unit
```

⟨*signature* Visitor_python.default_visitor 140b⟩≡ (367a)
```
val default_visitor: visitor_in
```

⟨*signature* Visitor_python.mk_visitor 140c⟩≡ (367a)
```
val mk_visitor: visitor_in -> visitor_out
```

⟨*signature* Lib_parsing_python.find_source_files_of_dir_or_files 140d⟩≡ (361a)
```
val find_source_files_of_dir_or_files:
  Common.path list -> Common.filename list
```

⟨*signature* Lib_parsing_python.ii_of_any 140e⟩≡ (361a)
```
val ii_of_any: AST_python.any -> Parse_info.t list
```

⟨*function* Lib_parsing_python.find_source_files_of_dir_or_files 140f⟩≡ (361b)
```
let find_source_files_of_dir_or_files xs =
  Common.files_of_dir_or_files_no_vcs_nofilter xs
  |> List.filter (fun filename ->
    let ftype = File_type.file_type_of_file filename in
    match ftype with
    | File_type.PL (File_type.Python) -> true
    | _ -> false
  ) |> Common.sort
```

⟨*function* Lib_parsing_python.extract_info_visitor 140g⟩≡ (361b)
```
let extract_info_visitor recursor =
  let globals = ref [] in
  let hooks = { V.default_visitor with
                V.kinfo = (fun (_k, _) i -> Common.push i globals)
              } in
  begin
    let vout = V.mk_visitor hooks in
    recursor vout;
    List.rev !globals
  end
```

⟨*function* Lib_parsing_python.ii_of_any 140h⟩≡ (361b)
```
let ii_of_any any =
  extract_info_visitor (fun visitor -> visitor any)
```

15.7.2 Dumper

⟨*signature* Meta_AST_python.vof_program 140i⟩≡ (367b)
```
val vof_program:
  AST_python.program -> OCaml.v
```

⟨*signature* Meta_AST_python.vof_any 140j⟩≡ (367b)
```
val vof_any:
  AST_python.any -> OCaml.v
```

15.8 Advanced features

15.8.1 Python2 parsing mode

⟨*function* `Parse_generic.lang_to_python_parsing_mode` 141a⟩≡ (279e)
```
let lang_to_python_parsing_mode = function
  | Lang.Python -> Parse_python.Python
  | Lang.Python2 -> Parse_python.Python2
  | Lang.Python3 -> Parse_python.Python3
  | s -> failwith (spf "not a python language:%s" (Lang.string_of_lang s))
```

⟨`Lang.t` *extra Python cases* 141b⟩≡ (23b)
```
(* Python will start with Python3 mode and fall back to Python2 in case
 * of error. Python2 and Python3 are for specific version of Python
 * (no fallback) *)
| Python2
| Python3
```

Chapter 16

The generic AST

16.1 Positions and tokens

16.2 Identifiers and names

⟨*type* AST_generic.name 142a⟩≡ (263c)
```
  (* old: Id below used to be called Name and was generalizing also IdQualified
   * but some analysis are easier when they just need to
   * handle a simple Id, hence the split. For example, there was some bugs
   * in sgrep because sometimes an identifier was an ident (in function header)
   * and sometimes a name (when called). For naming, we also need to do
   * things differently for Id vs IdQualified and would need many times to
   * inspect the name.name_qualifier to know if we have an Id or IdQualified.
   * We do the same split for Fid vs FName for fields.
   *
   * newvar: Id is sometimes abused to also introduce a newvar (as in Python)
   * but ultimately those cases should be rewritten to first introduce
   * a VarDef.
   *
   * todo: Sometimes some DotAccess should really be transformed in IdQualified
   * with a better qualifier because the obj is actually the name of a package
   * or module, but you may need advanced semantic information and global
   * analysis to disambiguate.
   *
   * less: factorize the id_info in both and inline maybe name_info
   *)
  type name =
    | Id of ident * id_info
    | IdQualified of (ident * name_info) * id_info
```

⟨*type* AST_generic.name_info 142b⟩≡ (263c)
```
  and name_info = {
    name_qualifier : qualifier option;
    name_typeargs : type_arguments option; (* Java/Rust *)
  }

  (* todo: not enough in OCaml with functor and type args or C++ templates*)
```

⟨*type* AST_generic.qualifier 142c⟩≡ (263c)
```
  and qualifier =
    | QTop of tok (* ::, Ruby, C++, also ''' abuse for PolyVariant in OCaml *)
    | QDots of dotted_ident (* Java, OCaml *)
    | QExpr of expr * tok
```

```
  (* Ruby *)
```

⟨*constant* AST_generic.empty_name_info 143a⟩≡ (263c)
```
  let empty_name_info = { name_qualifier = None; name_typeargs = None }
```

16.3 Expressions

⟨AST_generic.expr *other composite cases* 143b⟩+≡ (24d) ◁25c 143c▷
```
  (* And-type (field.vinit should be a Some) *)
  | Record of field list bracket
```

⟨AST_generic.expr *other composite cases* 143c⟩+≡ (24d) ◁143b
```
  (* Or-type (could be used instead of Container, Cons, Nil, etc.).
   * (ab)used also for polymorphic variants where qualifier is QTop with
   * the ''`'' token.
   *)
  | Constructor of dotted_ident * expr list
  (* see also Call(IdSpecial (New,_), [ArgType _;...] for other values *)
```

⟨AST_generic.expr *anonymous entity cases* 143d⟩≡ (24d)
```
  (* very special value *)
  | Lambda of function_definition
  (* usually an argument of a New (used in Java, Javascript) *)
  | AnonClass of class_definition
```

⟨AST_generic.expr *other identifier cases* 143e⟩≡ (24d) 143f▷
```
```

⟨AST_generic.expr *other identifier cases* 143f⟩+≡ (24d) ◁143e
```
  | IdSpecial of special wrap
```

⟨AST_generic.expr *other assign cases* 143g⟩≡ (24d)
```
  (* less: could desugar in Assign, should be only binary_operator *)
  | AssignOp of expr * operator wrap * expr
  (* newvar:! newscope:? in OCaml yes but we miss the 'in' part here  *)
  | LetPattern of pattern * expr
```

⟨AST_generic.expr *other call cases* 143h⟩≡ (24d)
```
  (* (XHP, JSX, TSX), could be transpiled also (done in IL.ml?) *)
  | Xml of xml
  (* IntepolatedString of expr list is simulated with a
   * Call(IdSpecial (Concat ...)) *)
```

⟨AST_generic.expr *array access cases* 143i⟩+≡ (24d) ◁26d
```
  (* could also use ArrayAccess with a Tuple rhs, or use a special *)
  | SliceAccess of
      expr
      * (expr option (* lower *) * expr option (* upper *) * expr option)
        (* step *)
        bracket
```

⟨AST_generic.expr *other cases* 144a⟩≡ (24d)
```
  (* a.k.a ternary expression, or regular if in OCaml *)
  | Conditional of expr * expr * expr
  | MatchPattern of expr * action list
  (* less: TryFunctional *)
  | Yield of tok * expr option * bool (* 'from' for Python *)
  | Await of tok * expr
  (* Send/Recv of Go are currently in OtherExpr *)
  | Cast of type_ (* TODO: bracket or colon *) * expr
  (* less: should be in statement *)
  | Seq of expr list (* at least 2 elements *)
  (* less: could be in Special, but pretty important so I've lifted them here*)
  | Ref of tok (* &, address of *) * expr
  | DeRef of tok (* '*' in C, '!' or '<-' in OCaml, ^ in Reason *) * expr
```

⟨*type* AST_generic.special 144b⟩≡ (263c)
```
  and special =
    (* special vars *)
    | This
    | Super (* called 'base' in C# *)
    | Self
    | Parent (* different from This/Super? *)
    | NextArrayIndex (* Lua, todo: just remove it, create Dict without key *)
    (* special calls *)
    | Eval
    | Typeof (* for C? and Go in switch x.(type) *)
    | Instanceof
    | Sizeof (* takes a ArgType *)
    | Defined (* defined? in Ruby, other? *)
    (* note that certain languages do not have a 'new' keyword
     * (e.g., Python, Scala 3), instead certain 'Call' are really 'New' *)
    | New (* usually associated with Call(New, [ArgType _;...]) *)
    (* new by itself is not a valid expression*)

    (* used for interpolated strings constructs *)
    | ConcatString of concat_string_kind
    | EncodedString of string (* only for Python for now (e.g., b"foo") *)
    (* TaggedString? for Javascript, for styled.div`bla{xx}`?
     * We could have this TaggedString where the first arg of Call
     * will be the tagging function, and the rest will be a Call ConcatString.
     * However, it is simpler to just transform those special calls as
     * regular calls even though they do not have parenthesis
     * (not all calls have parenthesis anyway, as in OCaml or Ruby).
     *)
    (* Use this to separate interpolated elements in interpolated strings
     * but this is a bit of a hack. We should probably add InterpolatedConcat
     * as an expression
     *)
    | InterpolatedElement
    (* "Inline" the content of a var containing a list (a.k.a a Splat in Ruby).
     * Used in a Container or Call argument context.
     * The corresponding constructor in a parameter context is ParamRest.
     *)
    | Spread (* ...x in JS, *x in Python/Ruby *)
    (* Similar to Spread, but for a var containing a hashtbl.
     * The corresponding constructor in a parameter context is ParamHashSplat.
     *)
    | HashSplat (* **x in Python/Ruby
                 * (not to confused with Pow below which is a Binary op *)
    | ForOf (* Javascript, for generators, used in ForEach *)
```

```
    (* used for unary and binary operations *)
    | Op of operator
    (* less: should be lift up and transformed in Assign at stmt level *)
    | IncrDecr of (incr_decr * prefix_postfix)
```

⟨*type* AST_generic.arithmetic_operator 145⟩≡ (263c)

```
  and operator =
    | Plus
    (* unary too *)
    | Minus (* unary too *)
    | Mult
    | Div
    | Mod
    | Pow (* ** binary op; for unary see HashSplat above *)
    | FloorDiv
    | MatMult (* Python *)
    | LSL
    | LSR
    | ASR (* L = logic, A = Arithmetic, SL = shift left *)
    | BitOr
    | BitXor
    | BitAnd
    | BitNot
    (* unary *)
    | BitClear (* Go *)
    (* todo? rewrite in CondExpr? have special behavior *)
    | And
    | Or
    (* also shortcut operator *)
    | Xor
    (* PHP*)
    | Not (* unary *)
    | Eq (* '=' in OCaml, '==' in Go/... *)
    | NotEq (* less: could be desugared to Not Eq *)
    | PhysEq (* '==' in OCaml, '===' in JS/... *)
    | NotPhysEq (* less: could be desugared to Not PhysEq *)
    | Lt
    | LtE
    | Gt
    | GtE (* less: could be desugared to Or (Eq Lt) *)
    | Cmp (* <=>, PHP *)
    | Concat (* '.' PHP, '..' Lua *)
    | Append (* x[] = ... in PHP, just in AssignOp *)
    | RegexpMatch (* =~, Ruby (and Perl) *)
    | NotMatch (* !~ Ruby less: could be desugared to Not RegexpMatch *)
    | Range (* .. or ..., Ruby, one arg can be nil for endless range *)
    | RangeInclusive (* '..=' in Rust *)
    | NotNullPostfix (* ! in Typescript, postfix operator *)
    | Length (* '#' in Lua *)
    (* See https://en.wikipedia.org/wiki/Elvis_operator.
     * In PHP we currently generate a Conditional instead of a Binary Elvis.
     * It looks like the Nullish operator is quite similar to the Elvis
     * operator, so we may want to merge those operators at some point.
     *)
    | Elvis (* ?: in Kotlin, can compare possible null value *)
    | Nullish (* ?? in Javascript *)
    | In
    (* in: checks that value belongs to a collection *)
    | NotIn (* !in *)
```

```
    | Is
    (* is: checks value has type *)
    | NotIs

  (* !is: *)
```

⟨*type* AST_generic.incr_decr 146a⟩≡ (263c)
```
  and incr_decr = Incr | Decr

  (* '++', '--' *)
```

⟨*type* AST_generic.prefix_postfix 146b⟩≡ (263c)
```
  and prefix_postfix = Prefix | Postfix
```

⟨AST_generic.field_ident *other cases* 146c⟩≡ (26c)

⟨*type* AST_generic.action 146d⟩≡ (263c)
```
  and action = pattern * expr
```

⟨*type* AST_generic.xml 146e⟩≡ (263c)
```
  (* this is for JSX/TSX in javascript land (old: and XHP in PHP land) *)
  and xml = {
    xml_kind : xml_kind;
    xml_attrs : xml_attribute list;
    xml_body : xml_body list;
  }
```

⟨*type* AST_generic.xml_attribute 146f⟩≡ (263c)
```
  and xml_attribute =
    | XmlAttr of ident * tok (* = *) * xml_attr_value
    (* less: XmlAttrNoValue of ident. <foo a /> <=> <foo a=true /> *)
    (* jsx: usually a Spread operation, e.g., <foo {...bar} /> *)
    | XmlAttrExpr of expr bracket
    (* sgrep: *)
    | XmlEllipsis of tok
```

⟨*type* AST_generic.xml_attr_value 146g⟩≡
```
         and xml_attr_value = expr
```

⟨*type* AST_generic.xml_body 146h⟩≡ (263c)
```
  and xml_body =
    (* sgrep-ext: can contain "..." *)
    | XmlText of string wrap
    (* this can be None when people abuse {} to put comments in it *)
    | XmlExpr of expr option bracket
    | XmlXml of xml
```

⟨AST_generic.argument *other cases* 146i⟩+≡ (26a) ◁26b
```
  (* type argument for New, instanceof/sizeof/typeof, C macros *)
  | ArgType of type_
```

⟨AST_generic.argument *OtherXxx case* 146j⟩≡ (26a)
```
  | ArgOther of other_argument_operator * any list
```

⟨*type* AST_generic.other_argument_operator 147a⟩≡ (263c)
```
and other_argument_operator =
  (* Python *)
  | OA_ArgComp (* comprehension *)
  (* OCaml *)
  | OA_ArgQuestion
```

⟨AST_generic.expr *OtherXxx case* 147b⟩≡ (24d)
```
(* TODO: other_expr_operator wrap, so enforce at least one token instead
 * of relying that the any list contains at least one token *)
| OtherExpr of other_expr_operator * any list
```

⟨*type* AST_generic.other_expr_operator 147c⟩≡ (263c)
```
and other_expr_operator =
  (* Javascript *)
  | OE_Exports
  | OE_Module
  | OE_Define
  | OE_Arguments
  | OE_NewTarget
  | OE_Delete
  | OE_YieldStar
  (* note: some of them are transformed in ImportFrom in js_to_generic.ml *)
  | OE_Require
  | OE_UseStrict (* todo: lift up to program attribute/directive? *)
  (* Python *)
  | OE_Invert
  | OE_Slices (* see also SliceAccess *)
  (* todo: newvar: *)
  | OE_CompForIf
  | OE_CompFor
  | OE_CompIf
  | OE_CmpOps
  | OE_Repr (* todo: move to special, special Dump *)
  (* Java *)
  | OE_NameOrClassType
  | OE_ClassLiteral
  | OE_NewQualifiedClass
  | OE_Annot
  (* C *)
  | OE_GetRefLabel
  | OE_ArrayInitDesignator (* [x] = ... todo? use ArrayAccess in container?*)
  (* PHP *)
  | OE_Unpack
  | OE_ArrayAppend (* $x[]. The AST for $x[] = 1 used to be
                    * handled as an AssignOp with special Append, but we now
                    * use OE_ArrayAppend for everything to simplify.
                    *)
  (* OCaml *)
  | OE_RecordWith
  | OE_RecordFieldName
  (* Go *)
  | OE_Send
  | OE_Recv
  (* Ruby *)
  (* Rust *)
  | OE_MacroInvocation
  (* C# *)
```

```
    | OE_Checked
    | OE_Unchecked
    (* Other *)
    | OE_StmtExpr (* OCaml/Ruby have just expressions, no statements *)
    | OE_Todo
```

16.4 Statements

⟨AST_generic.stmt *other cases* 148a⟩≡ (26e)
```
  | DoWhile of tok * stmt * expr
  (* newscope: *)
  | For of tok (* 'for', 'foreach'*) * for_header * stmt
  (* The expr can be None for Go and Ruby.
   * less: could be merged with ExprStmt (MatchPattern ...) *)
  | Switch of
      tok (* 'switch' or also 'select' in Go *)
      * expr option
      * case_and_body list
  | Continue of tok * label_ident * sc
  | Break of tok * label_ident * sc
  (* todo? remove stmt argument? more symetric to Goto *)
  | Label of label * stmt
  | Goto of tok * label
  | Throw of tok (* 'raise' in OCaml, 'throw' in Java/PHP *) * expr * sc
  | Try of tok * stmt * catch list * finally option
  | WithUsingResource of
      tok (* 'with' in Python, 'using' in C# *)
      * stmt (* resource acquisition *)
      * stmt (* newscope: block *)
  | Assert of tok * expr * expr option (* message *) * sc
```

⟨*type* AST_generic.case_and_body 148b⟩≡ (263c)
```
  and case_and_body =
    | CasesAndBody of (case list * stmt)
    (* sgrep: *)
    | CaseEllipsis of tok

  (* ... *)
```

⟨*type* AST_generic.case 148c⟩≡ (263c)
```
  and case =
    | Case of tok * pattern
    | Default of tok
    (* For Go, expr can contain some Assign bindings.
     * todo? could merge with regular Case? can 'case x := <-chan' be
     * transformed in a pattern?
     *)
    | CaseEqualExpr of tok * expr
```

⟨*type* AST_generic.catch 148d⟩≡ (263c)
```
  and catch = tok (* 'catch', 'except' in Python *) * pattern * stmt

  (* newscope: *)
```

⟨*type* AST_generic.finally 148e⟩≡ (263c)
```
  and finally = tok (* 'finally' *) * stmt
```

⟨*type* AST_generic.label 149a⟩≡ (263c)
```
  and label = ident
```

⟨*type* AST_generic.label_ident 149b⟩≡ (263c)
```
  and label_ident =
    | LNone (* C/Python *)
    | LId of label (* Java/Go *)
    | LInt of int wrap (* PHP *)
    | LDynamic of expr

  (* PHP, woohoo, dynamic break! bailout for CFG *)
```

⟨*type* AST_generic.for_header 149c⟩≡ (263c)
```
  and for_header =
    (* todo? copy Go and have instead
     * ForClassic of simple option * expr * simple option?
     *)
    | ForClassic of
        for_var_or_expr list (* init *) * expr option (* cond *) * expr option (* next *)
    (* newvar: *)
    | ForEach of
        pattern * tok (* 'in' Python, 'range' Go, 'as' PHP, '' Java *) * expr (* pattern 'in' expr *)
    (* sgrep: *)
    | ForEllipsis of tok (* ... *)
    (* Lua. todo: merge with ForEach? *)
    | ForIn of for_var_or_expr list (* init *) * expr list

  (* pattern 'in' expr *)
```

⟨*type* AST_generic.for_var_or_expr 149d⟩≡ (263c)
```
  and for_var_or_expr =
    (* newvar: *)
    | ForInitVar of entity * variable_definition
    | ForInitExpr of expr
```

⟨*type* AST_generic.other_stmt_with_stmt_operator 149e⟩≡ (263c)
```
  and other_stmt_with_stmt_operator =
    (* Python/Javascript *)
    (* TODO: used in C# with 'Using', make new stmt TryWithResource? do Java?*)
    | OSWS_With (* newscope: newvar: in OtherStmtWithStmt with LetPattern *)
    (* Ruby *)
    | OSWS_BEGIN
    | OSWS_END (* also in Awk, Perl? *)
    | OSWS_Else_in_try
    (* Rust *)
    | OSWS_UnsafeBlock
    | OSWS_AsyncBlock
    | OSWS_ConstBlock
    | OSWS_ForeignBlock
    | OSWS_ImplBlock
    (* C# *)
    | OSWS_CheckedBlock
    | OSWS_UncheckedBlock
```

⟨*type* AST_generic.other_stmt_operator 150a⟩≡ (263c)
```
and other_stmt_operator =
  (* Python *)
  | OS_Delete
  (* todo: reduce? transpile? *)
  | OS_ForOrElse
  | OS_WhileOrElse
  | OS_TryOrElse
  | OS_ThrowFrom
  | OS_ThrowNothing
  | OS_ThrowArgsLocation (* Python2: `raise expr, expr` and `raise expr, expr, exr` *)
  | OS_Pass
  | OS_Async
  (* Java *)
  | OS_Sync
  (* C *)
  | OS_Asm
  (* Go *)
  | OS_Go
  | OS_Defer
  | OS_Fallthrough (* only in Switch *)
  (* PHP *)
  | OS_GlobalComplex (* e.g., global $$x, argh *)
  (* Ruby *)
  | OS_Redo
  | OS_Retry
  (* OCaml *)
  | OS_ExprStmt2
  (* Other *)
  | OS_Todo
```

⟨AST_generic.stmt *OtherXxx case* 150b⟩≡ (26e)
```
  (* this is important to correctly compute a CFG *)
  | OtherStmtWithStmt of other_stmt_with_stmt_operator * expr option * stmt
  (* any here should not contain any statement! otherwise the CFG will be
   * incorrect and some analysis (e.g., liveness) will be incorrect.
   * TODO: other_stmt_operator wrap, so enforce at least one token instead
   * of relying that the any list contains at least one token
   *)
  | OtherStmt of other_stmt_operator * any list
```

16.5 Types

⟨AST_generic.type_ *other cases* 150c⟩≡ (30c)
```
  (* old: was originally TyApply (name, []), but better to differentiate.
   * todo? may need also TySpecial because the name can actually be
   *  self/parent/static (e.g., in PHP)
   *)
  | TyN of name
  (* covers tuples, list, etc.
   * TODO: merge with TyN IdQualified? name_info has name_typeargs
   *)
  | TyNameApply of dotted_ident * type_arguments
  | TyVar of ident (* type variable in polymorphic types (not a typedef) *)
  | TyAny of tok (* anonymous type, '_' in OCaml *)
  | TyPointer of tok * type_
```

```
  | TyRef of tok * type_ (* C++/Rust *)
  | TyQuestion of type_ * tok (* a.k.a option type *)
  | TyRest of tok * type_ (* '...foo' e.g. in a typescript tuple type *)
  (* intersection types, used for Java Cast, and in Typescript *)
  | TyAnd of type_ * tok (* & *) * type_
  (* union types in Typescript *)
  | TyOr of type_ * tok (* | *) * type_
  (* Anonymous record type, a.k.a shape in PHP/Hack. See also AndType.
   * Most record types are defined via a TypeDef and are then referenced
   * via a TyName. Here we have flexible record types (a.k.a. rows in OCaml).
   *)
  | TyRecordAnon of tok (* 'struct/shape', fake in other *) * field list bracket
  (* for Go *)
  | TyInterfaceAnon of tok (* 'interface' *) * field list bracket
  (* sgrep-ext: *)
  | TyEllipsis of tok
```

⟨AST_generic.type_ *OtherXxx case* 151a⟩≡ (30c)
```
  | OtherType of other_type_operator * any list
```

⟨*type* AST_generic.type_arguments 151b⟩≡ (263c)
```
  and type_arguments = type_argument list
```

⟨*type* AST_generic.type_argument 151c⟩≡ (263c)
```
  and type_argument =
    | TypeArg of type_
    (* Java only *)
    | TypeWildcard of
        tok (* '?' *) * (bool wrap (* extends|super, true=super *) * type_) option
    (* Rust *)
    | TypeLifetime of ident
    | OtherTypeArg of other_type_argument_operator * any list
```

⟨*type* AST_generic.other_type_argument_operator 151d⟩≡ (263c)

⟨*type* AST_generic.other_type_operator 151e⟩≡ (263c)
```
  and other_type_operator =
    (* C *)
    (* todo? convert in unique names with TyName? *)
    | OT_StructName
    | OT_UnionName
    | OT_EnumName
    (* PHP *)
    | OT_Variadic (* ???? *)
    (* Rust *)
    | OT_Lifetime
    (* Other *)
    | OT_Expr
    | OT_Arg (* Python: todo: should use expr_to_type() when can *)
    | OT_Todo
```

16.6 Patterns

⟨AST_generic.parameter *other cases* 151f⟩≡ (28c)
```
  | ParamPattern of pattern (* in OCaml, but also now JS, and Python2 *)
```

⟨*type* AST_generic.pattern 152a⟩≡ (263c)
```
  and pattern =
    | PatLiteral of literal
    (* Or-Type, used also to match OCaml exceptions *)
    (* Used with Rust path expressions, with an empty pattern list *)
    | PatConstructor of dotted_ident * pattern list
    (* And-Type*)
    | PatRecord of (dotted_ident * pattern) list bracket
    (* newvar:! *)
    | PatId of ident * id_info (* Usually Local/Param, Global in toplevel let *)
    (* special cases of PatConstructor *)
    | PatTuple of pattern list bracket (* at least 2 elements *)
    (* less: generalize to other container_operator? *)
    | PatList of pattern list bracket
    | PatKeyVal of pattern * pattern (* a kind of PatTuple *)
    (* special case of PatId *)
    | PatUnderscore of tok
    (* OCaml *)
    | PatDisj of pattern * pattern (* also abused for catch in Java *)
    | PatTyped of pattern * type_
    | PatWhen of pattern * expr
    | PatAs of pattern * (ident * id_info)
    (* For Go also in switch x.(type) { case int: ... } *)
    | PatType of type_
    (* In catch for Java/PHP, and foreach in Java.
     * less: do instead PatAs (PatType(TyApply, var))?
     *       or even    PatAs (PatConstructor(id, []), var)?
     *)
    | PatVar of type_ * (ident * id_info) option
    ⟨AST_generic.pattern semgrep extensions cases 102g⟩
    | OtherPat of other_pattern_operator * any list
```

⟨*type* AST_generic.other_pattern_operator 152b⟩≡ (263c)
```
  and other_pattern_operator =
    (* Other *)
    | OP_Expr (* todo: Python should transform via expr_to_pattern() below *)
    | OP_Todo
```

16.7 Definitions

⟨AST_generic.entity *other fields* 152c⟩≡ (27b)

⟨AST_generic.definition_kind *other cases* 152d⟩≡ (27e) 152e▷
```
  | TypeDef of type_definition
  | ModuleDef of module_definition
  | MacroDef of macro_definition
```

⟨AST_generic.definition_kind *other cases* 152e⟩+≡ (27e) ◁152d 152f▷
```
  (* in a header file (e.g., .mli in OCaml or 'module sig') *)
  | Signature of type_
```

⟨AST_generic.definition_kind *other cases* 152f⟩+≡ (27e) ◁152e
```
  (* Only used inside a function.
   * Needed for languages without local VarDef (e.g., Python/PHP)
   * where the first use is also its declaration. In that case when we
   * want to access a global we need to disambiguate with creating a new
```

```
   * local.
   *)
  | UseOuterDecl of tok (* 'global' or 'nonlocal' in Python, 'use' in PHP *)
  | OtherDef of other_def_operator * any list
```

⟨*type* AST_generic.type_parameter 153a⟩≡ (263c)
```
  and type_parameter = ident * type_parameter_constraint list
```

⟨*type* AST_generic.type_parameter_constraints 153b⟩≡ (263c)

⟨*type* AST_generic.type_parameter_constraint 153c⟩≡ (263c)
```
  and type_parameter_constraint =
    | Extends of type_
    | HasConstructor of tok
    | OtherTypeParam of other_type_parameter_operator * any list
```

⟨AST_generic.parameter *OtherXxx case* 153d⟩≡ (28c)
```
  | OtherParam of other_parameter_operator * any list
```

⟨*type* AST_generic.other_parameter_operator 153e⟩≡ (263c)
```
  and other_parameter_operator =
    (* Python *)
    (* single '*' or '/' to delimit regular parameters from special one *)
    | OPO_SingleStarParam
    | OPO_SlashParam
    (* Go *)
    | OPO_Receiver (* of parameter_classic, used to tag the "self" parameter*)
    (* PHP/Ruby *)
    | OPO_Ref (* of parameter_classic *)
    (* Other *)
    | OPO_Todo
```

⟨*type* AST_generic.type_definition 153f⟩≡ (263c)
```
  and type_definition = { tbody : type_definition_kind }
```

⟨*type* AST_generic.type_definition_kind 153g⟩≡ (263c)
```
  and type_definition_kind =
    | OrType of or_type_element list (* enum/ADTs *)
    (* field.vtype should be defined here
     * record/struct (for class see class_definition)
     *)
    | AndType of field list bracket
    (* a.k.a typedef in C (and alias type in Go) *)
    | AliasType of type_
    (* Haskell/Hack/Go ('type x foo' vs 'type x = foo' in Go) *)
    | NewType of type_
    | Exception of ident (* same name than entity *) * type_ list
    | OtherTypeKind of other_type_kind_operator * any list
```

⟨*type* AST_generic.or_type_element 154a⟩≡ (263c)
```
and or_type_element =
  (* OCaml *)
  | OrConstructor of ident * type_ list
  (* C *)
  | OrEnum of ident * expr option
  (* Java? *)
  | OrUnion of ident * type_
  | OtherOr of other_or_type_element_operator * any list
```

⟨*type* AST_generic.other_or_type_element_operator 154b⟩≡ (263c)
```
and other_or_type_element_operator =
  (* Java, Kotlin *)
  | OOTEO_EnumWithMethods
  | OOTEO_EnumWithArguments
```

⟨AST_generic.field *other cases* 154c⟩≡ (30b)
```
(* less: could abuse FieldStmt(ExprStmt(IdSpecial(Spread))) for that *)
| FieldSpread of tok (* ... *) * expr
```

⟨*type* AST_generic.other_type_kind_operator 154d⟩≡ (263c)
```
and other_type_kind_operator = (* OCaml *)
  | OTKO_AbstractType
```

⟨*type* AST_generic.module_definition 154e⟩≡ (263c)
```
and module_definition = { mbody : module_definition_kind }
```

⟨*type* AST_generic.module_definition_kind 154f⟩≡ (263c)
```
and module_definition_kind =
  | ModuleAlias of dotted_ident
  (* newscope: *)
  | ModuleStruct of dotted_ident option * item list
  | OtherModule of other_module_operator * any list
```

⟨*type* AST_generic.other_module_operator 154g⟩≡ (263c)
```
and other_module_operator =
  (* OCaml (functors and their applications) *)
  | OMO_Todo
```

⟨*type* AST_generic.macro_definition 154h⟩≡ (263c)
```
and macro_definition = { macroparams : ident list; macrobody : any list }
```

16.8 Attributes

⟨AST_generic.attribute *OtherXxx case* 154i⟩≡ (30f)
```
| OtherAttribute of other_attribute_operator * any list
```

⟨*type* AST_generic.other_attribute_operator 155a⟩≡ (263c)
```
  and other_attribute_operator =
    (* Java *)
    | OA_StrictFP
    | OA_Transient
    | OA_Synchronized
    | OA_Native
    | OA_Default
    | OA_AnnotThrow
    (* Other *)
    | OA_Expr

  (* todo: Python, should transform in NamedAttr when can *)
```

16.9 Directives

⟨AST_generic.directive *package cases* 155b⟩≡ (31b)
```
  (* packages are different from modules in that multiple files can reuse
   * the same package name; they are agglomarated in the same package
   *)
  | Package of tok * dotted_ident (* a.k.a namespace *)
  (* for languages such as C++/PHP with scoped namespaces
   * alt: Package of tok * dotted_ident * item list bracket, but less
   * consistent with other directives, so better to use PackageEnd.
   *)
  | PackageEnd of tok
```

⟨AST_generic.directive *OtherXxx cases* 155c⟩≡ (31b)
```
  | OtherDirective of other_directive_operator * any list
```

⟨*type* AST_generic.other_directive_operator 155d⟩≡ (263c)
```
  and other_directive_operator =
    (* Javascript *)
    | OI_Export
    | OI_ReExportNamespace
```

16.10 The final program

16.11 AST_generic.any

16.12 pfff -dump_ast <file>

Chapter 17

Generic AST Utilities

17.1 The dumper

⟨*signature* `Meta_AST.vof_any` 156a⟩≡ (279c)
```
val vof_any : AST_generic.any -> OCaml.v
```

17.2 The visitors

⟨*type* `Visitor_AST.visitor_in` 156b⟩≡ (277)
```
type visitor_in = {
  (* those are the one used by semgrep *)
  kexpr : (expr -> unit) * visitor_out -> expr -> unit;
  kstmt : (stmt -> unit) * visitor_out -> stmt -> unit;
  kstmts : (stmt list -> unit) * visitor_out -> stmt list -> unit;
  ktype_ : (type_ -> unit) * visitor_out -> type_ -> unit;
  kpattern : (pattern -> unit) * visitor_out -> pattern -> unit;
  kfield : (field -> unit) * visitor_out -> field -> unit;
  kattr : (attribute -> unit) * visitor_out -> attribute -> unit;
  kpartial : (partial -> unit) * visitor_out -> partial -> unit;
  kdef : (definition -> unit) * visitor_out -> definition -> unit;
  kdir : (directive -> unit) * visitor_out -> directive -> unit;
  kparam : (parameter -> unit) * visitor_out -> parameter -> unit;
  kident : (ident -> unit) * visitor_out -> ident -> unit;
  kname : (name -> unit) * visitor_out -> name -> unit;
  kentity : (entity -> unit) * visitor_out -> entity -> unit;
  kfunction_definition :
    (function_definition -> unit) * visitor_out -> function_definition -> unit;
  kclass_definition :
    (class_definition -> unit) * visitor_out -> class_definition -> unit;
  kinfo : (tok -> unit) * visitor_out -> tok -> unit;
  kid_info : (id_info -> unit) * visitor_out -> id_info -> unit;
  kconstness : (constness -> unit) * visitor_out -> constness -> unit;
}
```

⟨*type* `Visitor_AST.visitor_out` 156c⟩≡ (277)
```
and visitor_out = any -> unit
```

⟨*signature* `Visitor_AST.default_visitor` 156d⟩≡ (277)
```
val default_visitor : visitor_in
```

⟨*signature* `Visitor_AST.mk_visitor` 157a⟩≡ (277)
```
val mk_visitor : visitor_in -> visitor_out
```

⟨*function* `Visitor_AST.mk_visitor` 157b⟩≡
```
TODO
```

17.2.1 Example: getting all the tokens

⟨*signature* `Lib_AST.ii_of_any` 157c⟩≡ (280c)
```
val ii_of_any: AST.any -> Parse_info.t list
```

⟨*function* `Lib_AST.extract_info_visitor` 157d⟩≡ (281a)
```
let extract_info_visitor recursor =
  let globals = ref [] in
  let hooks = { V.default_visitor with
    V.kinfo = (fun (_k, _) i -> Common.push i globals)
  } in
  begin
    let vout = V.mk_visitor hooks in
    recursor vout;
    List.rev !globals
  end
```

⟨*function* `Lib_AST.ii_of_any` 157e⟩≡ (281a)
```
let ii_of_any any =
  extract_info_visitor (fun visitor -> visitor any)
```

17.3 The mappers

⟨*type* `Map_AST.visitor_in` 157f⟩≡ (278a)
```
type visitor_in = {
  kexpr : (expr -> expr) * visitor_out -> expr -> expr;
  kstmt : (stmt -> stmt) * visitor_out -> stmt -> stmt;
  kinfo : (tok -> tok) * visitor_out -> tok -> tok;
  kidinfo : (id_info -> id_info) * visitor_out -> id_info -> id_info;
}
```

⟨*type* `Map_AST.visitor_out` 157g⟩≡ (278a)
```
and visitor_out = {
  vitem : item -> item;
  vprogram : program -> program;
  vexpr : expr -> expr;
  vany : any -> any;
}
```

⟨*signature* `Map_AST.default_visitor` 157h⟩≡ (278a)
```
val default_visitor : visitor_in
```

⟨*signature* `Map_AST.mk_visitor` 157i⟩≡ (278a)
```
val mk_visitor : visitor_in -> visitor_out
```

17.3.1 Example: abstracting token positions for comparing ASTs

⟨*signature* `Lib_AST.abstract_position_info_any` 158a⟩≡ (280c)
```
val abstract_position_info_any: AST.any -> AST.any
```

⟨*function* `Lib_AST.abstract_position_info_any` 158b⟩≡ (281a)
```
let abstract_position_info_any x =
  abstract_position_visitor (fun visitor -> visitor.M.vany x)
```

⟨*function* `Lib_AST.abstract_position_visitor` 158c⟩≡ (281a)
```
let abstract_position_visitor recursor =
  let hooks = { M.default_visitor with
    M.kinfo = (fun (_k, _) i ->
      { i with Parse_info.token = Parse_info.Ab }
    )
  } in
  begin
    let vout = M.mk_visitor hooks in
    recursor vout;
  end
```

Chapter 18

Converting an AST to the Generic AST: Python example

⟨*signature* Python_to_generic.program 159a⟩≡ (367c)
```
val program : AST_python.program -> AST_generic.program
```

⟨*signature* Python_to_generic.any 159b⟩≡ (367c)
```
val any : AST_python.any -> AST_generic.any
```

⟨*constant* Python_to_generic.id 159c⟩≡ (367d)
```
let id x = x
```

⟨*constant* Python_to_generic.option 159d⟩≡ (367d)
```
let option = Common.map_opt
```

⟨*constant* Python_to_generic.list 159e⟩≡ (367d)
```
let list = List.map
```

⟨*function* Python_to_generic.vref 159f⟩≡ (367d)
```
let vref f x = ref (f !x)
```

⟨*constant* Python_to_generic.string 159g⟩≡ (367d)
```
let string = id
```

⟨*constant* Python_to_generic.bool 159h⟩≡ (367d)
```
let bool = id
```

18.1 Converting tokens

⟨*function* Python_to_generic.fake 159i⟩≡ (367d)
```
let fake s = Parse_info.fake_info s
```

⟨*function* Python_to_generic.fake_bracket 159j⟩≡ (367d)
```
let fb = AST_generic.fake_bracket
```

⟨*function* Python_to_generic.info 160a⟩≡ (367d)
```
let info x = x
```

⟨*constant* Python_to_generic.wrap 160b⟩≡ (367d)
```
let wrap _of_a (v1, v2) =
  let v1 = _of_a v1 and v2 = info v2 in
  (v1, v2)
```

⟨*function* Python_to_generic.bracket 160c⟩≡ (367d)
```
let bracket of_a (t1, x, t2) = (info t1, of_a x, info t2)
```

18.2 Converting identifiers

⟨*function* Python_to_generic.name 160d⟩≡ (367d)
```
let name v = wrap string v
```

⟨*function* Python_to_generic.dotted_name 160e⟩≡ (367d)
```
let dotted_name v = list name v
```

⟨*function* Python_to_generic.module_name 160f⟩≡ (367d)
```
let module_name (v1, dots) =
  let v1 = dotted_name v1 in
  match dots with
  | None -> G.DottedName v1
  (* transforming '. foo.bar' in G.Filename "./foo/bar" *)
  | Some toks ->
      let count =
        toks
        |> List.map Parse_info.str_of_info
        |> String.concat "" |> String.length
      in
      let tok = List.hd toks in
      let elems = v1 |> List.map fst in
      let prefixes =
        match count with
        | 1 -> [ "." ]
        | 2 -> [ ".." ]
        | n -> Common2.repeat ".." (n - 1)
      in
      let s = String.concat "/" (prefixes @ elems) in
      G.FileName (s, tok)
```

⟨*function* Python_to_generic.resolved_name 160g⟩≡ (367d)
```
let resolved_name = function
  | LocalVar -> Some (G.Local, G.sid_TODO)
  | Parameter -> Some (G.Param, G.sid_TODO)
  | GlobalVar -> Some (G.Global, G.sid_TODO)
  | ClassField -> None
  | ImportedModule xs -> Some (G.ImportedModule (G.DottedName xs), G.sid_TODO)
  | ImportedEntity xs -> Some (G.ImportedEntity xs, G.sid_TODO)
  | NotResolved -> None
```

⟨*function* Python_to_generic.expr_context 161a⟩≡ (367d)
```
let expr_context = function
  | Load -> ()
  | Store -> ()
  | Del -> ()
  | AugLoad -> ()
  | AugStore -> ()
  | Param -> ()
```

⟨*function* Python_to_generic.program 161b⟩≡ (367d)
```
let program v =
  let v = list stmt v in
  v
```

⟨*function* Python_to_generic.any 161c⟩≡ (367d)
```
let any = function
  | Expr v1 ->
      let v1 = expr v1 in
      G.E v1
  | Stmt v1 -> (
      let v1 = stmt v1 in
      (* in Python Assign is a stmt but in the generic AST it's an expression*)
      match v1.G.s with G.ExprStmt (x, _t) -> G.E x | _ -> G.S v1 )
  (* TODO? should use list stmt_aux here? Some intermediate Block
   * could be inserted preventing some sgrep matching?
   *)
  | Stmts v1 ->
      let v1 = list stmt v1 in
      G.Ss v1
  | Program v1 ->
      let v1 = program v1 in
      G.Pr v1
  | DictElem v1 ->
      let v1 = dictorset_elt v1 in
      G.E v1
```

18.3 Converting expressions

⟨*function* Python_to_generic.expr 161d⟩≡ (367d)
```
let rec expr (x : expr) =
  match x with
  | Bool v1 ->
      let v1 = wrap bool v1 in
      G.L (G.Bool v1)
  | None_ x ->
      let x = info x in
      G.L (G.Null x)
  | Ellipsis x ->
      let x = info x in
      G.Ellipsis x
  | DeepEllipsis x ->
      let x = bracket expr x in
      G.DeepEllipsis x
  | Num v1 ->
      let v1 = number v1 in
```

```
    G.L v1
| Str v1 ->
    let v1 = wrap string v1 in
    G.L (G.String v1)
| EncodedStr (v1, pre) ->
    let v1 = wrap string v1 in
    (* bugfix: do not reuse the same tok! otherwise in semgrep
     * if a metavar is bound to an encoded string (e.g., r'foo'), and
     * the metavar is used in the message, r'foo' will be displayed
     * three times.
     * todo: the right fix is to have EncodedStr of string wrap * string wrap
     *)
    G.Call
      ( G.IdSpecial (G.EncodedString pre, fake ""),
        fb [ G.Arg (G.L (G.String v1)) ] )
| InterpolatedString xs ->
    G.Call
      ( G.IdSpecial (G.ConcatString G.FString, fake "concat"),
        fb
          ( xs
          |> List.map (fun x ->
                 let x = expr x in
                 G.Arg x) ) )
| ConcatenatedString xs ->
    G.Call
      ( G.IdSpecial (G.ConcatString G.SequenceConcat, fake "concat"),
        fb
          ( xs
          |> List.map (fun x ->
                 let x = expr x in
                 G.Arg x) ) )
| TypedExpr (v1, v2) ->
    let v1 = expr v1 in
    let v2 = type_ v2 in
    G.Cast (v2, v1)
| TypedMetavar (v1, v2, v3) ->
    let v1 = name v1 in
    let v3 = type_ v3 in
    G.TypedMetavar (v1, v2, v3)
| ExprStar v1 ->
    let v1 = expr v1 in
    G.Call (G.IdSpecial (G.Spread, fake "spread"), fb [ G.arg v1 ])
| Name (v1, v2, v3) ->
    let v1 = name v1
    and _v2TODO = expr_context v2
    and v3 = vref resolved_name v3 in
    G.N (G.Id (v1, { (G.empty_id_info ()) with G.id_resolved = v3 }))
| Tuple (CompList v1, v2) ->
    let v1 = bracket (list expr) v1 and _v2TODO = expr_context v2 in
    G.Tuple v1
| Tuple (CompForIf (v1, v2), v3) ->
    let e1 = comprehension expr v1 v2 in
    let _v4TODO = expr_context v3 in
    G.Tuple (G.fake_bracket e1)
| List (CompList v1, v2) ->
    let v1 = bracket (list expr) v1 and _v2TODO = expr_context v2 in
    G.Container (G.List, v1)
| List (CompForIf (v1, v2), v3) ->
    let e1 = comprehension expr v1 v2 in
    let _v3TODO = expr_context v3 in
```

```
    G.Container (G.List, fb e1)
| Subscript (v1, v2, v3) -> (
    let e = expr v1 and _v3TODO = expr_context v3 in
    match v2 with
    | l1, [ x ], l2 -> slice1 e (l1, x, l2)
    | _, xs, _ ->
        let xs = list (slice e) xs in
        G.OtherExpr (G.OE_Slices, xs |> List.map (fun x -> G.E x)) )
| Attribute (v1, t, v2, v3) ->
    let v1 = expr v1
    and t = info t
    and v2 = name v2
    and _v3TODO = expr_context v3 in
    G.DotAccess (v1, t, G.EN (G.Id (v2, G.empty_id_info ())))
| DictOrSet (CompList (t1, v, t2)) ->
    let v' = list dictorset_elt v in
    let kind =
      if
        v
        |> List.for_all (function
             | KeyVal _
             (* semgrep-ext: ... should not count *)
             | Key (Ellipsis _) ->
                 true
             | _ -> false)
        || v = []
      then G.Dict
      else G.Set
    in
    G.Container (kind, (t1, v', t2))
| DictOrSet (CompForIf (v1, v2)) ->
    let e1 = comprehension2 dictorset_elt v1 v2 in
    G.Container (G.Dict, fb e1)
| BoolOp ((v1, tok), v2) ->
    let v1 = boolop v1 and v2 = list expr v2 in
    G.Call (G.IdSpecial (G.Op v1, tok), fb (v2 |> List.map G.arg))
| BinOp (v1, (v2, tok), v3) ->
    let v1 = expr v1 and v2 = operator v2 and v3 = expr v3 in
    G.Call (G.IdSpecial (G.Op v2, tok), fb ([ v1; v3 ] |> List.map G.arg))
| UnaryOp ((v1, tok), v2) -> (
    let v1 = unaryop v1 and v2 = expr v2 in
    match v1 with
    | Left op ->
        G.Call (G.IdSpecial (G.Op op, tok), fb ([ v2 ] |> List.map G.arg))
    | Right oe -> G.OtherExpr (oe, [ G.E v2 ]) )
| Compare (v1, v2, v3) -> (
    let v1 = expr v1 and v2 = list cmpop v2 and v3 = list expr v3 in
    match (v2, v3) with
    | [ (op, tok) ], [ e ] ->
        G.Call (G.IdSpecial (G.Op op, tok), fb ([ v1; e ] |> List.map G.arg))
    | _ ->
        let anyops =
          v2
          |> List.map (function arith, tok ->
                 G.E (G.IdSpecial (G.Op arith, tok)))
        in
        let any = anyops @ (v3 |> List.map (fun e -> G.E e)) in
        G.OtherExpr (G.OE_CmpOps, any) )
| Call (v1, v2) ->
    let v1 = expr v1 in
```

```
      let v2 = bracket (list argument) v2 in
      G.Call (v1, v2)
  | Lambda (t0, v1, _t2, v2) ->
      let v1 = parameters v1 and v2 = expr v2 in
      G.Lambda
        {
          G.fparams = v1;
          fbody = G.exprstmt v2;
          frettype = None;
          fkind = (G.LambdaKind, t0);
        }
  | IfExp (v1, v2, v3) ->
      let v1 = expr v1 and v2 = expr v2 and v3 = expr v3 in
      G.Conditional (v1, v2, v3)
  | Yield (t, v1, v2) ->
      let v1 = option expr v1 and v2 = v2 in
      G.Yield (t, v1, v2)
  | Await (t, v1) ->
      let v1 = expr v1 in
      G.Await (t, v1)
  | Repr v1 ->
      let _, v1, _ = bracket expr v1 in
      G.OtherExpr (G.OE_Repr, [ G.E v1 ])
  | NamedExpr (v, t, e) -> G.Assign (expr v, t, expr e)
```

⟨*function* Python_to_generic.argument 164a⟩≡ (367d)

```
and argument = function
  | Arg e ->
      let e = expr e in
      G.Arg e
  | ArgStar (t, e) ->
      let e = expr e in
      G.Arg (G.Call (G.IdSpecial (G.Spread, t), fb [ G.arg e ]))
  | ArgPow (t, e) ->
      let e = expr e in
      G.Arg (G.Call (G.IdSpecial (G.HashSplat, t), fb [ G.arg e ]))
  | ArgKwd (n, e) ->
      let n = name n in
      let e = expr e in
      G.ArgKwd (n, e)
  | ArgComp (e, xs) ->
      let e = expr e in
      G.ArgOther (G.OA_ArgComp, G.E e :: list for_if xs)
```

⟨*function* Python_to_generic.for_if 164b⟩≡ (367d)

```
and for_if = function
  | CompFor (e1, e2) ->
      let e1 = expr e1 in
      let e2 = expr e2 in
      G.E (G.OtherExpr (G.OE_CompFor, [ G.E e1; G.E e2 ]))
  | CompIf e1 ->
      let e1 = expr e1 in
      G.E (G.OtherExpr (G.OE_CompIf, [ G.E e1 ]))
```

⟨*function* Python_to_generic.dictorset_elt 164c⟩≡ (367d)

```
and dictorset_elt = function
  | KeyVal (v1, v2) ->
```

```
      let v1 = expr v1 in
      let v2 = expr v2 in
      G.Tuple (G.fake_bracket [ v1; v2 ])
  | Key v1 ->
      let v1 = expr v1 in
      v1
  | PowInline v1 ->
      let v1 = expr v1 in
      G.Call (G.IdSpecial (G.Spread, fake "spread"), fb [ G.arg v1 ])
```

⟨*function* Python_to_generic.number 165a⟩≡ (367d)

```
and number = function
  | Int v1 ->
      let v1 = wrap id v1 in
      G.Int v1
  | LongInt v1 ->
      let v1 = wrap id v1 in
      G.Int v1
  | Float v1 ->
      let v1 = wrap id v1 in
      G.Float v1
  | Imag v1 ->
      let v1 = wrap string v1 in
      G.Imag v1
```

⟨*function* Python_to_generic.boolop 165b⟩≡ (367d)

```
and boolop = function And -> G.And | Or -> G.Or
```

⟨*function* Python_to_generic.operator 165c⟩≡ (367d)

```
and operator = function
  | Add -> G.Plus
  | Sub -> G.Minus
  | Mult -> G.Mult
  | Div -> G.Div
  | Mod -> G.Mod
  | Pow -> G.Pow
  | FloorDiv -> G.FloorDiv
  | LShift -> G.LSL
  | RShift -> G.LSR
  | BitOr -> G.BitOr
  | BitXor -> G.BitXor
  | BitAnd -> G.BitAnd
  | MatMult -> G.MatMult
```

⟨*function* Python_to_generic.unarop 165d⟩≡ (367d)

```
and unaryop = function
  | Invert -> Right G.OE_Invert
  | Not -> Left G.Not
  | UAdd -> Left G.Plus
  | USub -> Left G.Minus
```

⟨*function* Python_to_generic.cmpop 165e⟩≡ (367d)

```
and cmpop (a, b) =
  match a with
  | Eq -> (G.Eq, b)
```

```
    | NotEq -> (G.NotEq, b)
    | Lt -> (G.Lt, b)
    | LtE -> (G.LtE, b)
    | Gt -> (G.Gt, b)
    | GtE -> (G.GtE, b)
    | Is -> (G.PhysEq, b)
    | IsNot -> (G.NotPhysEq, b)
    | In -> (G.In, b)
    | NotIn -> (G.NotIn, b)
```

⟨*function* Python_to_generic.comprehension 166a⟩≡ (367d)

```
  and comprehension f v1 v2 =
    let v1 = f v1 in
    let v2 = list for_if v2 in
    [ G.OtherExpr (G.OE_CompForIf, G.E v1 :: v2) ]
```

⟨*function* Python_to_generic.comprehension2 166b⟩≡ (367d)

```
  and comprehension2 f v1 v2 =
    let v1 = f v1 in
    let v2 = list for_if v2 in
    [ G.OtherExpr (G.OE_CompForIf, G.E v1 :: v2) ]
```

⟨*function* Python_to_generic.slice 166c⟩≡ (367d)

```
  and slice1 e1 (t1, e2, t2) =
    match e2 with
    | Index v1 ->
        let v1 = expr v1 in
        G.ArrayAccess (e1, (t1, v1, t2))
    | Slice (v1, v2, v3) ->
        let v1 = option expr v1 and v2 = option expr v2 and v3 = option expr v3 in
        G.SliceAccess (e1, (t1, (v1, v2, v3), t2))

  and slice e = function
    | Index v1 ->
        let v1 = expr v1 in
        G.ArrayAccess (e, fb v1)
    | Slice (v1, v2, v3) ->
        let v1 = option expr v1 and v2 = option expr v2 and v3 = option expr v3 in
        G.SliceAccess (e, fb (v1, v2, v3))
```

⟨*function* Python_to_generic.type_ 166d⟩≡ (367d)

```
  and type_ v =
    let v = expr v in
    H.expr_to_type v
```

⟨*function* Python_to_generic.parameters 166e⟩≡ (367d)

```
  and parameters xs =
    xs
    |> List.map (function
         | ParamDefault ((n, topt), e) ->
             let n = name n in
             let topt = option type_ topt in
             let e = expr e in
             G.ParamClassic
               { (G.param_of_id n) with G.ptype = topt; pdefault = Some e }
```

```
    | ParamPattern (PatternName n, topt) ->
        let n = name n and topt = option type_ topt in
        G.ParamClassic { (G.param_of_id n) with G.ptype = topt }
    | ParamPattern (PatternTuple pat, _) ->
        let pat = list param_pattern pat in
        G.ParamPattern (G.PatTuple (G.fake_bracket pat))
    | ParamStar (t, (n, topt)) ->
        let n = name n in
        let topt = option type_ topt in
        G.ParamRest (t, { (G.param_of_id n) with G.ptype = topt })
    | ParamPow (t, (n, topt)) ->
        let n = name n in
        let topt = option type_ topt in
        G.ParamHashSplat (t, { (G.param_of_id n) with G.ptype = topt })
    | ParamEllipsis tok -> G.ParamEllipsis tok
    | ParamSingleStar tok ->
        G.OtherParam (G.OPO_SingleStarParam, [ G.Tk tok ])
    | ParamSlash tok -> G.OtherParam (G.OPO_SlashParam, [ G.Tk tok ]))
```

⟨*function* Python_to_generic.type_parent 167a⟩≡ (367d)

```
and type_parent v =
  let v = argument v in
  G.OtherType (G.OT_Arg, [ G.Ar v ])
```

18.4 Converting statements

⟨*function* Python_to_generic.list_stmt1 167b⟩≡ (367d)

```
and list_stmt1 xs =
  match list stmt xs with
  (* bugfix: We do not want actually to optimize and remove the
   * intermediate Block because otherwise sgrep will not work
   * correctly with a list of stmt.
   *
   * old: | [e] -> e
   *
   * For example
   * if $E:
   *   ...
   *   foo()
   *
   * will not match code like
   *
   * if True:
   *   foo()
   *
   * because above we have a Block ([Ellipsis; foo()] and down we would
   * have just (foo()). We do want Block ([foo()]].
   *
   * Unless the body is actually just a metavar, in which case we probably
   * want to match a list of stmts, as in
   *
   *  if $E:
   *    $S
   *
   * in which case we remove the G.Block around it.
   * hacky ...
```

```
     *)
    | [ ({ G.s = G.ExprStmt (G.N (G.Id ((s, _), _)), _); _ } as x) ]
      when AST_generic_.is_metavar_name s ->
        x
    | xs -> G.Block (fb xs) |> G.s
```

⟨*function* Python_to_generic.stmt_aux 168⟩≡ (367d)

```
  and stmt_aux x =
    match x with
    | FunctionDef (t, v1, v2, v3, v4, v5) ->
        let v1 = name v1
        and v2 = parameters v2
        and v3 = option type_ v3
        and v4 = list_stmt1 v4
        and v5 = list decorator v5 in
        let ent = G.basic_entity v1 v5 in
        let def =
          { G.fparams = v2; frettype = v3; fbody = v4; fkind = (G.Function, t) }
        in
        [ G.DefStmt (ent, G.FuncDef def) |> G.s ]
    | ClassDef (v0, v1, v2, v3, v4) ->
        let v1 = name v1
        and v2 = list type_parent v2
        and v3 = list stmt v3
        and v4 = list decorator v4 in
        let ent = G.basic_entity v1 v4 in
        let def =
          {
            G.ckind = (G.Class, v0);
            cextends = v2;
            cimplements = [];
            cmixins = [];
            cparams = [];
            cbody = fb (v3 |> List.map (fun x -> G.FieldStmt x));
          }
        in
        [ G.DefStmt (ent, G.ClassDef def) |> G.s ]
    | Assign (v1, v2, v3) -> (
        let v1 = list expr v1 and v2 = info v2 and v3 = expr v3 in
        match v1 with
        | [] -> raise Impossible
        | [ a ] -> (
            match a with
            (* x: t = ... is definitely a VarDef *)
            | G.Cast (t, G.N (G.Id (id, idinfo))) ->
                let ent =
                  { G.name = G.EN (G.Id (id, idinfo)); attrs = []; tparams = [] }
                in
                let var = G.VarDef { G.vinit = Some v3; vtype = Some t } in
                [ G.DefStmt (ent, var) |> G.s ]
            (* TODO: We should turn more Assign in G.VarDef!
             * Is it bad for semgrep to turn only the typed assign in VarDef?
             * No because we have some magic equivalences to convert some
             * Vardef back in Assign in Generic_vs_generic.
             *)
            | _ -> [ G.exprstmt (G.Assign (a, v2, v3)) ] )
        | xs -> [ G.exprstmt (G.Assign (G.Tuple (G.fake_bracket xs), v2, v3)) ] )
    | AugAssign (v1, (v2, tok), v3) ->
        let v1 = expr v1 and v2 = operator v2 and v3 = expr v3 in
```

```
    [ G.exprstmt (G.AssignOp (v1, (v2, tok), v3)) ]
| Return (t, v1) ->
    let v1 = option expr v1 in
    [ G.Return (t, v1, G.sc) |> G.s ]
| Delete (_t, v1) ->
    let v1 = list expr v1 in
    [ G.OtherStmt (G.OS_Delete, v1 |> List.map (fun x -> G.E x)) |> G.s ]
| If (t, v1, v2, v3) ->
    let v1 = expr v1 and v2 = list_stmt1 v2 and v3 = option list_stmt1 v3 in
    [ G.If (t, v1, v2, v3) |> G.s ]
| While (t, v1, v2, v3) -> (
    let v1 = expr v1 and v2 = list_stmt1 v2 and v3 = list stmt v3 in
    match v3 with
    | [] -> [ G.While (t, v1, v2) |> G.s ]
    | _ ->
        [
          G.Block
            (fb
               [
                 G.While (t, v1, v2) |> G.s;
                 G.OtherStmt
                   (G.OS_WhileOrElse, v3 |> List.map (fun x -> G.S x))
                 |> G.s;
               ])
          |> G.s;
        ] )
| For (t, v1, t2, v2, v3, v4) -> (
    let foreach = pattern v1
    and ins = expr v2
    and body = list_stmt1 v3
    and orelse = list stmt v4 in
    let header = G.ForEach (foreach, t2, ins) in
    match orelse with
    | [] -> [ G.For (t, header, body) |> G.s ]
    | _ ->
        [
          G.Block
            (fb
               [
                 G.For (t, header, body) |> G.s;
                 G.OtherStmt
                   (G.OS_ForOrElse, orelse |> List.map (fun x -> G.S x))
                 |> G.s;
               ])
          |> G.s;
        ] )
(* TODO: unsugar in sequence? *)
| With (_t, v1, v2, v3) ->
    let v1 = expr v1 and v2 = option expr v2 and v3 = list_stmt1 v3 in
    let e =
      match v2 with
      | None -> v1
      | Some e2 -> G.LetPattern (H.expr_to_pattern e2, v1)
    in
    [ G.OtherStmtWithStmt (G.OSWS_With, Some e, v3) |> G.s ]
| Raise (t, v1) -> (
    match v1 with
    | Some (e, None) ->
        let e = expr e in
        [ G.Throw (t, e, G.sc) |> G.s ]
```

```
    | Some (e, Some from) ->
        let e = expr e in
        let from = expr from in
        let st = G.Throw (t, e, G.sc) |> G.s in
        [ G.OtherStmt (G.OS_ThrowFrom, [ G.E from; G.S st ]) |> G.s ]
    | None -> [ G.OtherStmt (G.OS_ThrowNothing, [ G.Tk t ]) |> G.s ] )
| RaisePython2 (t, e, v2, v3) -> (
    let e = expr e in
    let st = G.Throw (t, e, G.sc) |> G.s in
    match (v2, v3) with
    | Some args, Some loc ->
        let args = expr args and loc = expr loc in
        [
          G.OtherStmt (G.OS_ThrowArgsLocation, [ G.E loc; G.E args; G.S st ])
          |> G.s;
        ]
    | Some args, None ->
        let args = expr args in
        [ G.OtherStmt (G.OS_ThrowArgsLocation, [ G.E args; G.S st ]) |> G.s ]
    | None, _ -> [ st ] )
| TryExcept (t, v1, v2, v3) -> (
    let v1 = list_stmt1 v1
    and v2 = list excepthandler v2
    and orelse = list stmt v3 in
    match orelse with
    | [] -> [ G.Try (t, v1, v2, None) |> G.s ]
    | _ ->
        [
          G.Block
            (fb
               [
                 G.Try (t, v1, v2, None) |> G.s;
                 G.OtherStmt
                   (G.OS_TryOrElse, orelse |> List.map (fun x -> G.S x))
                 |> G.s;
               ])
          |> G.s;
        ] )
| TryFinally (t, v1, t2, v2) ->
    let v1 = list_stmt1 v1 and v2 = list_stmt1 v2 in
    (* could lift down the Try in v1 *)
    [ G.Try (t, v1, [], Some (t2, v2)) |> G.s ]
| Assert (t, v1, v2) ->
    let v1 = expr v1 and v2 = option expr v2 in
    [ G.Assert (t, v1, v2, G.sc) |> G.s ]
| ImportAs (t, v1, v2) ->
    let mname = module_name v1 and nopt = option ident_and_id_info v2 in
    [ G.DirectiveStmt (G.ImportAs (t, mname, nopt)) |> G.s ]
| ImportAll (t, v1, v2) ->
    let mname = module_name v1 and v2 = info v2 in
    [ G.DirectiveStmt (G.ImportAll (t, mname, v2)) |> G.s ]
| ImportFrom (t, v1, v2) ->
    let v1 = module_name v1 and v2 = list alias v2 in
    List.map
      (fun (a, b) -> G.DirectiveStmt (G.ImportFrom (t, v1, a, b)) |> G.s)
      v2
| Global (t, v1) | NonLocal (t, v1) ->
    let v1 = list name v1 in
    v1
    |> List.map (fun x ->
```

```
            let ent = G.basic_entity x [] in
            G.DefStmt (ent, G.UseOuterDecl t) |> G.s)
  | ExprStmt v1 ->
      let v1 = expr v1 in
      [ G.exprstmt v1 ]
  | Async (t, x) -> (
      let x = stmt x in
      match x.G.s with
      | G.DefStmt (ent, func) ->
          [
            G.DefStmt
              ({ ent with G.attrs = G.attr G.Async t :: ent.G.attrs }, func)
            |> G.s;
          ]
      | _ -> [ G.OtherStmt (G.OS_Async, [ G.S x ]) |> G.s ] )
  | Pass t -> [ G.OtherStmt (G.OS_Pass, [ G.Tk t ]) |> G.s ]
  | Break t -> [ G.Break (t, G.LNone, G.sc) |> G.s ]
  | Continue t -> [ G.Continue (t, G.LNone, G.sc) |> G.s ]
  (* python2: *)
  | Print (tok, _dest, vals, _nl) ->
      let id = Name (("print", tok), Load, ref NotResolved) in
      stmt_aux (ExprStmt (Call (id, fb (vals |> List.map (fun e -> Arg e)))))
  | Exec (tok, e, _eopt, _eopt2) ->
      let id = Name (("exec", tok), Load, ref NotResolved) in
      stmt_aux (ExprStmt (Call (id, fb [ Arg e ])))
```

⟨*function* Python_to_generic.stmt 171a⟩≡ (367d)

```
  and stmt x = G.stmt1 (stmt_aux x)
```

⟨*function* Python_to_generic.pattern 171b⟩≡ (367d)

```
  and pattern e =
    let e = expr e in
    H.expr_to_pattern e
```

⟨*function* Python_to_generic.excepthandler 171c⟩≡ (367d)

```
  and excepthandler = function
    | ExceptHandler (t, v1, v2, v3) ->
        let v1 = option expr v1 (* a type actually, even tuple of types *)
        and v2 = option name v2
        and v3 = list_stmt1 v3 in
        ( t,
          ( match (v1, v2) with
          | Some e, None -> G.PatVar (H.expr_to_type e, None)
          | None, None -> G.PatUnderscore (fake "_")
          | None, Some _ -> raise Impossible (* see the grammar *)
          | Some e, Some n ->
              G.PatVar (H.expr_to_type e, Some (n, G.empty_id_info ())) ),
          v3 )
```

⟨*function* Python_to_generic.expr_to_attribute 171d⟩≡ (367d)

⟨*function* Python_to_generic.decorator 171e⟩≡ (367d)

```
  and decorator (t, v1, v2) =
    let v1 = dotted_name v1 in
    let v2 = option (bracket (list argument)) v2 in
    let args =
```

```
    match v2 with Some (t1, x, t2) -> (t1, x, t2) | None -> G.fake_bracket []
  in
  let name = H.name_of_ids v1 in
  G.NamedAttr (t, name, args)
```

⟨*function* `Python_to_generic.alias` 172⟩≡ (367d)

```
and alias (v1, v2) =
  let v1 = name v1 and v2 = option ident_and_id_info v2 in
  (v1, v2)
```

Chapter 19

Naming (a.k.a Scope Resolution)

⟨Main_semgrep_core.parse_generic() *resolve names in the AST* 173a⟩≡ (52b)

19.1 The single unique ID (sid)

⟨*type* AST_generic.id_info 173b⟩≡ (263c)
```
and id_info = {
  id_resolved : resolved_name option ref;
  (* variable tagger (naming) *)
  (* sgrep: in OCaml we also use that to store the type of
   * a typed entity, which can be interpreted as a TypedMetavar in semgrep.
   * alt: have an explicity type_ field in entity.
   *)
  id_type : type_ option ref;
  (* type checker (typing) *)
  (* sgrep: this is for sgrep constant propagation hack.
   * todo? associate only with Id?
   * note that we do not use the constness for equality (hence the adhoc
   * @equal below) because the constness analysis is now controlflow-sensitive
   * meaning the same variable might have different id_constness value
   * depending where it is used.
   *)
  id_constness : constness option ref; [@equal fun _a _b -> true]
      (* THINK: Drop option? *)
}
```

⟨*function* AST_generic.empty_id_info 173c⟩≡ (263c)
```
let empty_id_info () =
  { id_resolved = ref None; id_type = ref None; id_constness = ref None }
```

⟨*function* AST_generic.basic_id_info 173d⟩≡ (263c)
```
let basic_id_info resolved =
  {
    id_resolved = ref (Some resolved);
    id_type = ref None;
    id_constness = ref None;
  }
```

⟨*type* AST_generic.resolved_name 173e⟩≡ (263c)
```
and resolved_name = resolved_name_kind * sid
```

⟨*type* AST_generic.resolved_name_kind 174a⟩≡ (263c)
```
and resolved_name_kind =
  (* Global is useful in codemap/efuns to highlight differently and warn
   * about the use of globals inside functions.
   * old: Global was merged with ImportedEntity before but simpler to split, as
   * anyway I was putting often an empty list for dotted_ident with a
   * todo note in the code.
   *)
  | Global
  (* Those could be merged, but again this is useful in codemap/efuns *)
  | Local
  | Param
  (* For closures; can refer to a Local or Param.
   * With sid this is potentially less useful for scoping-related issues,
   * but this can be useful in codemap to again highlight specially
   * enclosed vars.
   * todo: this is currently used also for fields, but we should use another
   * constructor.
   * Note that it's tempting to add a depth parameter to EnclosedVar, but
   * that would prevent semgrep to work because whatever the depth you are,
   * if you reference the same entity, this entity must have the same
   * resolved_name (sid and resolved_name_kind).
   *)
  | EnclosedVar (* less: add depth? *)
  (* sgrep: those cases allow to match entities/modules even if they were
   * aliased when imported.
   * both dotted_ident must at least contain one element *)
  | ImportedEntity of dotted_ident (* can also use 0 for gensym *)
  | ImportedModule of module_name
  (* used in Go, where you can pass types as arguments and where we
   * need to resolve those cases
   *)
  | TypeName
  (* used for C *)
  | Macro
  | EnumConstant
```

⟨*type* AST_generic.sid 174b⟩≡ (263c)
```
type sid = int

(* a single unique gensym'ed number. See gensym() below *)
```

⟨*constant* AST_generic.gensym_counter 174c⟩≡ (274)
```
(* You can use 0 for globals, even though this will work only on a single
 * file. Any global analysis will need to set a unique ID for globals too. *)
let gensym_counter = ref 0

(* see sid type in resolved_name *)
```

⟨*function* AST_generic.gensym 174d⟩≡ (274)
```
(* see sid type in resolved_name *)
let gensym () =
  incr gensym_counter;
  !gensym_counter

(* before Naming_AST.resolve can do its job *)
```

⟨*constant* AST_generic.sid_TODO 174e⟩≡ (263c)
```
(* before Naming_AST.resolve can do its job *)
let sid_TODO = -1
```

19.1.1 Naming_AST.resolve()

⟨*signature* Naming_AST.resolve 175a⟩≡ (281b)
```
(* Works by side effect on the generic AST by modifying its refs.
 * We pass the lang because some name resolution algorithm may be
 * specific to a language.
 *)
val resolve : Lang.t -> AST_generic.program -> unit
```

19.2 Managing scopes

19.3 Managing import aliases

19.4 pfff -naming_generic <file>

⟨*function* Test_analyze_generic.test_naming_generic 175b⟩≡ (327)
```
let test_naming_generic file =
  let ast = Parse_target.parse_program file in
  let lang = List.hd (Lang.langs_of_filename file) in
  Naming_AST.resolve lang ast;
  let s = AST_generic.show_any (AST_generic.Pr ast) in
  pr2 s
```

Chapter 20

Typing

Chapter 21

The Intermediate Language (IL)

21.1 Positions and tokens

⟨*type* IL.tok 177a⟩≡ (295)
```
(* the classic *)
type tok = G.tok [@@deriving show]
```

⟨*type* IL.wrap 177b⟩≡ (295)
```
type 'a wrap = 'a G.wrap [@@deriving show]
```

⟨*type* IL.bracket 177c⟩≡ (295)
```
(* useful mainly for empty containers *)
type 'a bracket = tok * 'a * tok [@@deriving show]
```

21.2 Identifier and resolved names

⟨*type* IL.ident 177d⟩≡ (295)
```
type ident = G.ident [@@deriving show]
```

⟨*type* IL.name 177e⟩≡ (295)
```
(* 'sid' below is the result of name resolution and variable disambiguation
 * using a gensym (see Naming_AST.ml). The pair is guaranteed to be
 * global and unique (no need to handle variable shadowing, block scoping,
 * etc; this has been done already).
 * TODO: use it to also do SSA! so some control-flow insensitive analysis
 * can become control-flow sensitive? (e.g., DOOP)
 *
 *)
type name = ident * G.sid [@@deriving show]
```

⟨*function* IL.str_of_name 177f⟩≡ (295)
```
let str_of_name ((s, _tok), _sid) = s
```

21.3 Lvalues

⟨*type* IL.lval 178a⟩≡ (295)
```
type lval = {
  base : base;
  offset : offset;
  constness : G.constness option ref;
      (* THINK: Drop option? *)
      (* todo: ltype: typ; *)
}
```

⟨*type* IL.base 178b⟩≡ (295)
```
and base =
  | Var of name
  | VarSpecial of var_special wrap
  (* aka DeRef, e.g. *E in C *)
  (* THINK: Mem of exp -> Deref of name *)
  | Mem of exp
```

⟨*type* IL.offset 178c⟩≡ (295)
```
and offset =
  | NoOffset
  (* What about nested field access? foo.x.y?
   * - use intermediate variable for that. TODO? same semantic?
   * - do as in CIL and have recursive offset and stop with NoOffset.
   * What about computed field names?
   * - handle them in Special?
   * - convert in Index with a string exp?
   * Note that Dot is used to access many different kinds of entities:
   *  objects/records (fields), classes (static fields), but also
   *  packages, modules, namespaces depending on the type of 'var' above.
   *
   * old: Previously the field was an `ident` but in some cases we want a
   *      proper (resolved) name here. For example, we want to resolve this.foo()
   *      to the class method "foo"; this is useful for our poor's man
   *      interprocedural analysis.
   *)
  | Dot of name
  | Index of exp
```

⟨*type* IL.var_special 178d⟩≡ (295)
```
and var_special = This | Super | Self | Parent
```

21.4 Side-effect-free expressions

⟨*type* IL.exp 178e⟩≡ (295)
```
and exp = { e : exp_kind; (* todo: etype: typ; *)
                          eorig : G.expr }
```

⟨*type* IL.exp_kind 179a⟩≡ (295)
```
and exp_kind =
  | Lvalue of lval (* lvalue used in a rvalue context *)
  | Literal of G.literal
  | Composite of composite_kind * exp list bracket
  (* Record could be a Composite where the arguments are CTuple with
   * the Literal (String) as a key, but they are pretty important I think
   * for some analysis so better to support them more directly.
   * TODO should we transform that in a New followed by a series of Assign
   * with Dot? simpler?
   * This could also be used for Dict.
   *)
  | Record of (ident * exp) list
  | Cast of G.type_ * exp
  (* This could be put in call_special, but dumped IL are then less readable
   * (they are too many intermediate _tmp variables then) *)
  | Operator of G.operator wrap * exp list
  | FixmeExp of fixme_kind * G.any
```

⟨*type* IL.composite_kind 179b⟩≡ (295)
```
and composite_kind =
  | CTuple
  | CArray
  | CList
  | CSet
  | CDict (* could be merged with Record *)
  | Constructor of name (* OCaml *)
```

⟨*type* IL.argument 179c⟩≡ (295)
```
type argument = exp [@@deriving show]
```

21.5 Instructions

⟨*type* IL.instr 179d⟩≡ (295)
```
type instr = { i : instr_kind; iorig : G.expr }
```

⟨*type* IL.instr_kind 179e⟩≡ (295)
```
and instr_kind =
  (* was called Set in CIL, but a bit ambiguous with Set module *)
  | Assign of lval * exp
  | AssignAnon of lval * anonymous_entity
  | Call of lval option * exp (* less: enforce lval? *) * argument list
  | CallSpecial of lval option * call_special wrap * argument list
  (* todo: PhiSSA! *)
  | FixmeInstr of fixme_kind * G.any
```

⟨*type* IL.call_special 179f⟩≡ (295)
```
and call_special =
  | Eval
  (* Note that in some languages (e.g., Python) some regular calls are
   * actually New under the hood.
   * The type_ argument is usually a name, but it can also be an name[] in
   * Java/C++.
   *)
```

```
    | New (* TODO: lift up and add 'of type_ * argument list'? *)
    | Typeof
    | Instanceof
    | Sizeof
    (* old: better in exp: | Operator of G.arithmetic_operator *)
    | Concat
    | Spread
    | Yield
    | Await
    (* was in stmt before, but with a new clean 'instr' type, better here *)
    | Assert
    (* was in expr before (only in C/PHP) *)
    | Ref (* TODO: lift up, have AssignRef? *)
    (* when transpiling certain features (e.g., patterns, foreach) *)
    | ForeachNext
    | ForeachHasNext

  (* primitives called under the hood *)
```

⟨*type* IL.anonymous_entity 180a⟩≡ (295)
```
  and anonymous_entity =
    | Lambda of G.function_definition
    | AnonClass of G.class_definition
```

21.6 Statements

⟨*type* IL.stmt 180b⟩≡ (295)
```
  type stmt = { s : stmt_kind (* sorig: G.stmt; ?*) }
```

⟨*type* IL.stmt_kind 180c⟩≡ (295)
```
  and stmt_kind =
    | Instr of instr
    (* Switch are converted to a series of If *)
    | If of tok * exp * stmt list * stmt list
    (* While/DoWhile/For are converted in a unified Loop construct.
     * Break/Continue are handled via Label.
     * alt: we could go further and transform in If+Goto, but nice to
     * not be too far from the original code.
     *)
    | Loop of tok * exp * stmt list
    | Return of tok * exp (* use Unit instead of 'exp option' *)
    (* alt: do as in CIL and resolve that directly in 'Goto of stmt' *)
    | Goto of tok * label
    | Label of label
    | Try of stmt list * (name * stmt list) list * stmt list
    | Throw of tok * exp (* less: enforce lval here? *)
    | MiscStmt of other_stmt
    | FixmeStmt of fixme_kind * G.any
```

⟨*type* IL.other_stmt 180d⟩≡ (295)
```
  and other_stmt =
    (* everything except VarDef (which is transformed in a Set instr) *)
    | DefStmt of G.definition
    | DirectiveStmt of G.directive
```

⟨*type* IL.label 181a⟩≡ (295)
```
and label = ident * G.sid
```

21.7 IL.any

⟨*type* IL.any 181b⟩≡ (295)
```
type any = L of lval | E of exp | I of instr | S of stmt | Ss of stmt list
(*  | N of node *)
```

⟨*signature* Meta_IL.vof_any 181c⟩≡ (299b)
```
val vof_any: IL.any -> OCaml.v
```

21.8 pfff -dump_il <file>

⟨*signature* Test_analyze_generic.actions 181d⟩≡ (326a)
```
val actions : unit -> Common.cmdline_actions
```

⟨*function* Test_analyze_generic.actions 181e⟩≡ (327)
```
let actions () =
  [
    ("-typing_generic", " <file>", Common.mk_action_1_arg test_typing_generic);
    ("-cfg_generic", " <file>", Common.mk_action_1_arg test_cfg_generic);
    ("-dfg_generic", " <file>", Common.mk_action_1_arg test_dfg_generic);
    ("-naming_generic", " <file>", Common.mk_action_1_arg test_naming_generic);
    ( "-constant_propagation",
      " <file>",
      Common.mk_action_1_arg test_constant_propagation );
    ("-il_generic", " <file>", Common.mk_action_1_arg test_il_generic);
    ("-cfg_il", " <file>", Common.mk_action_1_arg test_cfg_il);
    ("-dfg_tainting", " <file>", Common.mk_action_1_arg test_dfg_tainting);
    ("-dfg_constness", " <file>", Common.mk_action_1_arg test_dfg_constness);
  ]
```

⟨*function* Test_analyze_generic.test_il_generic 181f⟩≡ (327)
```
let test_il_generic file =
  let ast = Parse_target.parse_program file in
  let lang = List.hd (Lang.langs_of_filename file) in
  Naming_AST.resolve lang ast;

  let v =
    V.mk_visitor
      {
        V.default_visitor with
        V.kfunction_definition =
          (fun (_k, _) def ->
            let s = AST_generic.show_any (S def.fbody) in
            pr2 s;
            pr2 "==>";

            let xs = AST_to_IL.stmt def.fbody in
            let s = IL.show_any (IL.Ss xs) in
            pr2 s);
      }
  in
  v (Pr ast)
```

Chapter 22

Converting the Generic AST to IL

⟨*signature* `AST_to_IL.stmt` 182⟩≡ (300a)

```
val stmt : AST_generic.stmt -> IL.stmt list
```

22.1 Converting expressions

22.2 Converting statements

Chapter 23

The Control Flow Graph (CFG)

23.1 Control-flow node

⟨*type* IL.node 183a⟩≡ (295)
```
type node = {
  n : node_kind;
      (* old: there are tok in the nodes anyway
       * t: Parse_info.t option;
       *)
}
```

⟨*type* IL.node_kind 183b⟩≡ (295)
```
and node_kind =
  | Enter
  | Exit
  | TrueNode
  | FalseNode (* for Cond *)
  | Join (* after Cond *)
  | NInstr of instr
  | NCond of tok * exp
  | NGoto of tok * label
  | NReturn of tok * exp
  | NThrow of tok * exp
  | NOther of other_stmt
  | NTodo of stmt
```

23.2 The graph

⟨*type* IL.edge 183c⟩≡ (295)
```
(* For now there is just one kind of edge. Later we may have more,
 * see the ShadowNode idea of Julia Lawall.
 *)
type edge = Direct
```

⟨*type* IL.cfg 183d⟩≡ (295)
```
type cfg = (node, edge) Ograph_extended.ograph_mutable
```

⟨*function* IL.find_node 183e⟩≡ (295)
```
let find_node f cfg =
  cfg#nodes#tolist
  |> Common.find_some (fun (nodei, node) -> if f node then Some nodei else None)
```

⟨*function* IL.find_exit 184a⟩≡ (295)
```
let find_exit cfg = find_node (fun node -> node.n = Exit) cfg
```

⟨*function* IL.find_enter 184b⟩≡ (295)
```
let find_enter cfg = find_node (fun node -> node.n = Enter) cfg
```

⟨*type* IL.nodei 184c⟩≡ (295)
```
(* an int representing the index of a node in the graph *)
type nodei = Ograph_extended.nodei
```

⟨*signature* Meta_IL.display_cfg 184d⟩≡ (299b)
```
val display_cfg: IL.cfg -> unit
```

23.3 pfff -cfg_il <file>

⟨*function* Test_analyze_generic.test_cfg_il 184e⟩≡ (327)
```
let test_cfg_il file =
  let ast = Parse_target.parse_program file in
  let lang = List.hd (Lang.langs_of_filename file) in
  Naming_AST.resolve lang ast;

  ast
  |> List.iter (fun item ->
         match item.s with
         | DefStmt (_ent, FuncDef def) ->
             let xs = AST_to_IL.stmt def.fbody in
             let cfg = CFG_build.cfg_of_stmts xs in
             Display_IL.display_cfg cfg
         | _ -> ())
```

Chapter 24

Converting the IL to CFG

⟨*signature* `CFG_build.cfg_of_stmts` 185⟩≡ (314a)

```
val cfg_of_stmts : IL.stmt list -> IL.cfg
```

Chapter 25

Dataflow analysis

25.1 Data structures

⟨*type* Dataflow.nodei 186a⟩≡ (320c 318)
```
type nodei = int
```

⟨*type* Dataflow.var 186b⟩≡ (320c 318)
```
(* The comparison function uses only the name of a variable (a string), so
 * two variables at different positions in the code will be agglomerated
 * correctly in the Set or Map.
 *)
type var = string
```

⟨*module* Dataflow.VarMap 186c⟩≡ (318)
```
module VarMap : Map.S with type key = String.t
```

⟨*module* Dataflow.VarSet 186d⟩≡ (318)
```
module VarSet : Set.S with type elt = String.t
```

⟨*type* Dataflow.mapping 186e⟩≡ (320c 318)
```
type 'a mapping = 'a inout array
```

⟨*type* Dataflow.inout 186f⟩≡ (320c 318)
```
and 'a inout = { in_env : 'a env; out_env : 'a env }
```

⟨*type* Dataflow.env 186g⟩≡ (320c 318)
```
and 'a env = 'a VarMap.t
```

⟨*signature* Dataflow.empty_env 186h⟩≡ (318)
```
val empty_env : unit -> 'a VarMap.t
```

⟨*signature* Dataflow.empty_inout 186i⟩≡ (318)
```
val empty_inout : unit -> 'a inout
```

25.2 The transfer function

⟨*type* Dataflow.transfn 187a⟩≡ (320c 318)
```
(* The transition/transfer function. It is usually made from the
 * gens and kills.
 *
 * todo? having only a transfer function is enough ? do we need to pass
 * extra information to it ? maybe only the mapping is not enough. For
 * instance if in the code there is $x = &$g, a reference, then
 * we may want later to have access to this information. Maybe we
 * should pass an extra env argument ? Or maybe can encode this
 * sharing of reference in the 'a, so that when one update the
 * value associated to a var, its reference variable get also
 * the update.
 *)
type 'a transfn = 'a mapping -> nodei -> 'a inout
```

25.3 The fixpoint algorithm

⟨*module type* Dataflow.Flow 187b⟩≡ (318)
```
module type Flow = sig
  type node

  type edge

  type flow = (node, edge) Ograph_extended.ograph_mutable

  val short_string_of_node : node -> string
end
```

⟨*functor signature* Dataflow.Make 187c⟩≡ (318)
```
module Make (F : Flow) : sig
  (* main entry point *)
  val fixpoint :
    eq:('a -> 'a -> bool) ->
    init:'a mapping ->
    trans:'a transfn ->
    flow:F.flow ->
    forward:bool ->
    'a mapping

  val new_node_array : F.flow -> 'a -> 'a array

  (* debugging output *)
  val display_mapping : F.flow -> 'a mapping -> ('a -> string) -> unit
end
```

25.4 Helpers

⟨*module* Dataflow.NodeiSet 187d⟩≡ (318)
```
module NodeiSet : Set.S with type elt = Int.t
```

⟨*signature* Dataflow.varmap_union 188a⟩≡ (318)
```
val varmap_union : ('a -> 'a -> 'a) -> 'a env -> 'a env -> 'a env
```

⟨*signature* Dataflow.varmap_diff 188b⟩≡ (318)
```
val varmap_diff : ('a -> 'a -> 'a) -> ('a -> bool) -> 'a env -> 'a env -> 'a env
```

⟨*signature* Dataflow.union_env 188c⟩≡ (318)
```
(* helpers *)
val union_env : NodeiSet.t env -> NodeiSet.t env -> NodeiSet.t env
```

⟨*signature* Dataflow.diff_env 188d⟩≡ (318)
```
val diff_env : NodeiSet.t env -> NodeiSet.t env -> NodeiSet.t env
```

⟨*signature* Dataflow.add_var_and_nodei_to_env 188e⟩≡ (318)
```
val add_var_and_nodei_to_env : var -> nodei -> NodeiSet.t env -> NodeiSet.t env
```

⟨*signature* Dataflow.add_vars_and_nodei_to_env 188f⟩≡ (318)
```
val add_vars_and_nodei_to_env :
  VarSet.t -> nodei -> NodeiSet.t env -> NodeiSet.t env
```

⟨*signature* Dataflow.ns_to_str 188g⟩≡ (318)
```
val ns_to_str : NodeiSet.t -> string
```

25.5 pfff -dfg_generic <file>

⟨*function* Test_analyze_generic.test_dfg_generic 188h⟩≡ (327)
```
let test_dfg_generic file =
  let ast = Parse_target.parse_program file in
  ast
  |> List.iter (fun item ->
         match item.s with
         | DefStmt (_ent, FuncDef def) ->
             let flow = Controlflow_build.cfg_of_func def in
             pr2 "Reaching definitions";
             let mapping = Dataflow_reaching.fixpoint flow in
             DataflowX.display_mapping flow mapping Dataflow.ns_to_str;
             pr2 "Liveness";
             let mapping = Dataflow_liveness.fixpoint flow in
             DataflowX.display_mapping flow mapping (fun () -> "()")
         | _ -> ())
```

Chapter 26

Tainting analysis

26.1 Dataflow instanciation

⟨*type* Dataflow_tainting.mapping 189a⟩≡ (325b 324a)
```
type mapping = unit Dataflow.mapping
(** Map for each node/var whether a variable is "tainted" *)
```

⟨*signature* Dataflow_tainting.fixpoint 189b⟩≡ (324a)
```
val fixpoint : config -> fun_env -> IL.name option -> IL.cfg -> mapping
(** Main entry point, [fixpoint config fun_env opt_name cfg] returns a mapping
  * (effectively a set) containing all the tainted variables in [cfg]. Besides,
  * if it finds an instruction that consumes tainted data, then it will invoke
  * [config.found_tainted_sink] which can perform any side-effectful action.
  *
  * Parameter [fun_env] is a set of tainted functions in the overall program;
  * it provides basic interprocedural capabilities.
  *
  * Parameter [opt_name] is the name of the function being analyzed, if it has
  * a name. When [Some name] is given, and there is a tainted return statement in
  * [cfg], the function [name] itself will be added to [fun_env] by side-effect.
*)
```

⟨*constant* Dataflow_tainting.union 189c⟩≡ (325b)
```
let union = Dataflow.varmap_union (fun () () -> ())
```

⟨*constant* Dataflow_tainting.diff 189d⟩≡ (325b)
```
let diff = Dataflow.varmap_diff (fun () () -> ()) (fun () -> true)
```

⟨*module* Dataflow.Make(Il) 189e⟩≡ (325b)
```
module DataflowX = Dataflow.Make (struct
  type node = F.node

  type edge = F.edge

  type flow = (node, edge) Ograph_extended.ograph_mutable

  let short_string_of_node n = Display_IL.short_string_of_node_kind n.F.n
end)
```

⟨*function* Dataflow_tainting.transfer 190⟩≡ (325b)

```
 let (transfer :
       config -> fun_env -> IL.name option -> flow:F.cfg -> unit Dataflow.transfn)
     =
  fun config fun_env opt_name ~flow
      (* the transfer function to update the mapping at node index ni *)
       mapping ni ->
   let in' =
     (flow#predecessors ni)#fold
       (fun acc (ni_pred, _) -> union acc mapping.(ni_pred).D.out_env)
       VarMap.empty
   in
   let node = flow#nodes#assoc ni in

   (* TODO: do that later? once everything if finished? *)
   ( match node.F.n with
   | NInstr x ->
       (* TODO: use metavar in sink to know which argument we should check
        * for taint? *)
       if config.is_sink x && tainted_instr config fun_env in' x then
         config.found_tainted_sink x in'
   (* if just a single return is tainted then the function is tainted *)
   | NReturn (_, e) when tainted config fun_env in' e -> (
       match opt_name with
       | Some var -> Hashtbl.add fun_env (str_of_name var) ()
       | None -> () )
   | Enter | Exit | TrueNode | FalseNode | Join | NCond _ | NGoto _ | NReturn _
   | NThrow _ | NOther _ | NTodo _ ->
       () );

   let gen_ni_opt =
     match node.F.n with
     | NInstr x ->
         if tainted_instr config fun_env in' x then IL.lvar_of_instr_opt x
         else None
     | Enter | Exit | TrueNode | FalseNode | Join | NCond _ | NGoto _ | NReturn _
     | NThrow _ | NOther _ | NTodo _ ->
         None
   in
   let kill_ni_opt =
     (* old:
      *  if gen_ni_opt <> None
      *  then None
      * but now gen_ni <> None does not necessarily mean we had a source().
      * It can also be one tainted rvars which propagate to the lvar
      *)
     match node.F.n with
     | NInstr x ->
         if tainted_instr config fun_env in' x then None
         else
           (* all clean arguments should reset the taint *)
           IL.lvar_of_instr_opt x
     | Enter | Exit | TrueNode | FalseNode | Join | NCond _ | NGoto _ | NReturn _
     | NThrow _ | NOther _ | NTodo _ ->
         None
   in
   let gen_ni = option_to_varmap gen_ni_opt in
   let kill_ni = option_to_varmap kill_ni_opt in

   let out' = diff (union in' gen_ni) kill_ni in
```

```
  { D.in_env = in'; out_env = out' }
```

⟨*function* Dataflow_tainting.fixpoint 191a⟩≡ (325b)
```
let (fixpoint : config -> fun_env -> IL.name option -> F.cfg -> mapping) =
 fun config fun_env opt_name flow ->
  DataflowX.fixpoint
    ~eq:(fun () () -> true)
    ~init:(DataflowX.new_node_array flow (Dataflow.empty_inout ()))
    ~trans:
      (transfer config fun_env opt_name ~flow)
      (* tainting is a forward analysis! *)
    ~forward:true ~flow
```

26.2 Lvalues and rvalues helpers

⟨*function* IL.lvar_of_instr_opt 191b⟩≡ (295)
```
let lvar_of_instr_opt x =
  let open Common in
  lval_of_instr_opt x >>= fun lval ->
  match lval.base with Var n -> Some n | VarSpecial _ | Mem _ -> None
```

⟨*function* IL.exps_of_instr 191c⟩≡ (295)
```
let exps_of_instr x =
  match x.i with
  | Assign (_, exp) -> [ exp ]
  | AssignAnon _ -> []
  | Call (_, e1, args) -> e1 :: args
  | CallSpecial (_, _, args) -> args
  | FixmeInstr _ -> []
```

⟨*function* IL.rvars_of_exp 191d⟩≡
```
(* opti: could use a set *)
let rec rvars_of_exp e =
  match e.e with
  | Lvalue ({base=Var var;_}) -> [var]
  | Lvalue _ -> []
  | Literal _ -> []
  | Cast (_, e) -> rvars_of_exp e
  | Composite (_, (_, xs, _)) | Operator (_, xs) -> rvars_of_exps xs
  | Record ys -> rvars_of_exps (ys |> List.map snd)
  | TodoExp _ -> []

and rvars_of_exps xs =
  xs |> List.map (rvars_of_exp) |> List.flatten
```

⟨*function* IL.rvars_of_instr 191e⟩≡
```
let rvars_of_instr x =
  let exps = exps_of_instr x in
  rvars_of_exps exps
```

26.3 pfff -dfg_tainting <file>

⟨*function* Test_analyze_generic.test_dfg_tainting 192⟩≡ (327)

```
let test_dfg_tainting file =
  let ast = Parse_target.parse_program file in
  let lang = List.hd (Lang.langs_of_filename file) in
  Naming_AST.resolve lang ast;
  let fun_env = Hashtbl.create 8 in
  ast
  |> List.iter (fun item ->
         match item.s with
         | DefStmt (ent, FuncDef def) ->
             let xs = AST_to_IL.stmt def.fbody in
             let flow = CFG_build.cfg_of_stmts xs in
             pr2 "Tainting";
             let config =
               {
                 Dataflow_tainting.is_source = (fun _ -> false);
                 is_source_exp = (fun _ -> false);
                 is_sink = (fun _ -> false);
                 is_sanitizer = (fun _ -> false);
                 found_tainted_sink = (fun _ _ -> ());
               }
             in
             let opt_name = AST_to_IL.name_of_entity ent in
             let mapping =
               Dataflow_tainting.fixpoint config fun_env opt_name flow
             in
             DataflowY.display_mapping flow mapping (fun () -> "()")
         | _ -> ())
```

Chapter 27

Tainting analysis and Semgrep integration

⟨*constant* Main_semgrep_core.tainting_rules_file 193a⟩≡ (369)
```
(* -tainting_rules_file *)
let tainting_rules_file = ref ""
```

⟨Main_semgrep_core.options *other cases* 193b⟩+≡ (46d) ◁123j 224e▷
```
( "-tainting_rules_file",
  Arg.Set_string tainting_rules_file,
  " <file> obtain source/sink/sanitizer patterns from YAML file" );
```

⟨Main_semgrep_core.main() *main entry match cases* 193c⟩+≡ (40c) ◁43l
```
| _ when !tainting_rules_file <> "" ->
    let lang = lang_of_string !lang in
    tainting_with_rules lang !tainting_rules_file (x :: xs)
```

⟨*signature* Tainting_generic.check 193d⟩≡ (457b)
```
val check :
  Tainting_rule.rules -> Common.filename -> Target.t -> Pattern_match.t list
```

⟨*type* Dataflow_tainting.config 193e⟩≡ (325b 324a)
```
type config = {
  is_source : IL.instr -> bool;
  is_source_exp : IL.exp -> bool;
  is_sink : IL.instr -> bool;
  is_sanitizer : IL.instr -> bool;
  found_tainted_sink : IL.instr -> unit Dataflow.env -> unit;
}
(** This can use semgrep patterns under the hood. Note that a source can be an
  * instruction but also an expression. *)
```

⟨*function* Main_semgrep_core.tainting_with_rules 193f⟩≡ (369)
```
let tainting_with_rules lang rules_file files_or_dirs =
  try
    logger#info "Parsing %s" rules_file;
    let rules = Parse_tainting_rules.parse rules_file in

    let files = get_final_files lang files_or_dirs in
    let file_results =
      files
      |> iter_files_and_get_matches_and_exn_to_errors (fun file ->
             let ast, errors = parse_generic lang file in
             let rules =
               rules |> List.filter (fun r -> List.mem lang r.TR.languages)
```

```
            in
            {
              matches = Tainting_generic.check rules file ast;
              errors;
              profiling = RP.empty_partial_profiling file (* TODO? *);
            })
      in
      let res =
        RP.make_rule_result file_results ~report_time:false ~rule_parse_time:0.0
      in
      let flds = JSON_report.json_fields_of_matches_and_errors files res in
      let s = J.string_of_json (J.Object flds) in
      pr s
    with exn ->
      let json = JSON_report.json_of_exn exn in
      let s = J.string_of_json json in
      pr s;
      exit 2
```

⟨*signature* Parse_tainting_rules.parse 194a⟩≡ (426a)

```
val parse : Common.filename -> Tainting_rule.rules
```

⟨*function* Parse_tainting_rules.parse_patterns 194b⟩≡ (426b)

```
let parse_patterns ~id ~lang xs =
  xs
  |> List.map (function
       | `String s -> Parse_mini_rule.parse_pattern ~id ~lang s
       | x ->
           pr2_gen x;
           raise (InvalidYamlException "wrong pattern field"))
```

⟨*function* Parse_tainting_rules.parse 194c⟩≡ (426b)

```
let parse file =
  let str = Common.read_file file in
  let yaml_res = Yaml.of_string str in
  match yaml_res with
  | Result.Ok v -> (
      match v with
      | `O [ ("rules", `A xs) ] ->
          xs
          |> List.map (fun v ->
                 match v with
                 | `O xs ->
                     let id = ref None in
                     let languages = ref None in
                     let lang = ref None in
                     let severity = ref None in
                     let message = ref "" in
                     let sources = ref [] in
                     let sanitizers = ref [] in
                     let sinks = ref [] in

                     let current_id () =
                       match !id with
                       | Some s -> s
                       | None -> raise (InvalidYamlException "no id field")
                     in
```

```
let current_lang () =
  match !lang with
  | Some s -> s
  | None ->
      raise (InvalidYamlException "no languages field")
in

(* ugly: use sort so id/languages are before the source/sink/...
 * which need an id and lang set
 *)
xs |> Common.sort_by_key_lowfirst
|> List.iter (fun x ->
       match x with
       | "id", `String s -> id := Some s
       | "languages", `A langs ->
           let a, b =
             Parse_mini_rule.parse_languages
               ~id:(current_id ()) langs
           in
           languages := Some a;
           lang := Some b
       | "message", `String s -> message := s
       | "severity", `String s ->
           severity :=
             Some
               (Parse_mini_rule.parse_severity
                  ~id:(current_id ()) s)
       | "pattern-sources", `A xs ->
           sources :=
             parse_patterns ~id:(current_id ())
               ~lang:(current_lang ()) xs
       | "pattern-sinks", `A xs ->
           sinks :=
             parse_patterns ~id:(current_id ())
               ~lang:(current_lang ()) xs
       | "pattern-sanitizers", `A xs ->
           sanitizers :=
             parse_patterns ~id:(current_id ())
               ~lang:(current_lang ()) xs
       | x ->
           pr2_gen x;
           raise (InvalidYamlException "wrong rule field"));
let id =
  match !id with Some s -> s | None -> raise Todo
in
let message = !message in
let languages =
  match !languages with
  | Some xs -> xs
  | None -> raise Todo
in
let severity =
  match !severity with Some x -> x | None -> raise Todo
in
let source =
  match !sources with
  | _ :: _ -> !sources
  | [] -> raise Todo
in
let sink =
```

```
                  match !sinks with _ :: _ -> !sinks | [] -> raise Todo
                in
                let sanitizer = !sanitizers in
                {
                  R.id;
                  message;
                  languages;
                  severity;
                  source;
                  sink;
                  sanitizer;
                }
            | x ->
                pr2_gen x;
                raise (InvalidYamlException "wrong rule fields"))
      | _ -> raise (InvalidYamlException "missing rules entry as top-level key")
      )
  | Result.Error (`Msg s) -> raise (UnparsableYamlException s)
```

⟨*function* Tainting_generic.match_pat_instr 196a⟩≡ (458)
```
let match_pat_instr pat instr =
  let eorig = instr.IL.iorig in
  match_pat_eorig pat eorig
```

⟨*function* Tainting_generic.config_of_rule 196b⟩≡ (458)
```
let config_of_rule found_tainted_sink rule =
  {
    Dataflow_tainting.is_source = match_pat_instr rule.R.source;
    is_source_exp = match_pat_exp rule.R.source;
    is_sanitizer = match_pat_instr rule.R.sanitizer;
    is_sink = match_pat_instr rule.R.sink;
    found_tainted_sink;
  }
```

27.1 The Semgrep tainting rule

⟨*type* Tainting_rule.pattern 196c⟩≡ (389a)
```
(* right now only Expr is supported *)
type pattern = AST_generic.any
```

⟨*type* Tainting_rule.rule 196d⟩≡ (389a)
```
type rule = {
  id : string;
  (* the list below are used to express disjunction *)
  source : pattern list;
  sanitizer : pattern list;
  sink : pattern list;
  message : string;
  severity : Mini_rule.severity;
  languages : Lang.t list; (* at least one element *)
}
```

⟨*type* Tainting_rule.rules 196e⟩≡ (389a)
```
and rules = rule list
```

⟨*type* Tainting_rule.t 197a⟩≡ (389a)
```
(* alias *)
type t = rule
```

⟨*function* Tainting_rule.rule_of_tainting_rule 197b⟩≡ (389a)
```
(* for Pattern_match.t.rule compatibility *)

let rule_id_of_tainting_rule tr =
  {
    Pattern_match.id = tr.id;
    message = tr.message;
    pattern_string = "TODO: no pattern_string";
  }
```

27.2 Tainting_generic.check()

⟨*function* Tainting_generic.check2 197c⟩≡ (458)
```
let check rules file ast =
  let matches = ref [] in

  let fun_env = Hashtbl.create 8 in

  let check_fdef opt_name def =
    let xs = AST_to_IL.stmt def.AST.fbody in
    let flow = CFG_build.cfg_of_stmts xs in

    rules
    |> List.iter (fun rule ->
           let found_tainted_sink instr _env =
             let code = AST.E instr.IL.iorig in
             let range_loc = V.range_of_any code in
             let tokens = lazy (V.ii_of_any code) in
             let rule_id = Tainting_rule.rule_id_of_tainting_rule rule in
             (* todo: use env from sink matching func?  *)
             Common.push
               { PM.rule_id; file; range_loc; tokens; env = [] }
               matches
           in
           let config = config_of_rule found_tainted_sink rule in
           let mapping =
             Dataflow_tainting.fixpoint config fun_env opt_name flow
           in
           ignore mapping
           (* TODO
              logger#sdebug (DataflowY.mapping_to_str flow
               (fun () -> "()") mapping);
           *))
  in

  let v =
    V.mk_visitor
      {
        V.default_visitor with
        V.kdef =
          (fun (k, _) ((ent, def_kind) as def) ->
            match def_kind with
```

```
            | AST.FuncDef fdef ->
                let opt_name = AST_to_IL.name_of_entity ent in
                check_fdef opt_name fdef
            | __else__ -> k def);
        V.kfunction_definition = (fun (_k, _) def -> check_fdef None def);
      }
  in
  v (AST.Pr ast);

  !matches
  [@@profiling]
```

⟨*function* `Tainting_generic.check` 198a⟩≡
```
let check a b c =
  Common.profile_code "Tainting_generic.check" (fun () -> check2 a b c)
```

27.3 semgrep -dump_tainting_rules

⟨`Main_semgrep_core.all_actions` *dumper cases* 198b⟩+≡ (48d) ◁105a
```
( "-dump_tainting_rules",
  " <file>",
  Common.mk_action_1_arg dump_tainting_rules );
```

⟨*function* `Main_semgrep_core.dump_tainting_rules` 198c⟩≡ (369)
```
let dump_tainting_rules file =
  let xs = Parse_tainting_rules.parse file in
  pr2_gen xs
```

Chapter 28

Other Topics

28.1 Language-specific adjustments

28.1.1 C

⟨Main_semgrep_core.parse_generic() *use standard macros if parsing C* 199a⟩≡ (52b)
```
if lang = Lang.C && Sys.file_exists !Flag_parsing_cpp.macros_h then
  Parse_cpp.init_defs !Flag_parsing_cpp.macros_h;
```

⟨Main_semgrep_core.options *concatenated flags* 199b⟩≡ (46d) 210a ▷
```
@ Flag_parsing_cpp.cmdline_flags_macrofile ()
```

28.2 Preprocessing

⟨*type* Parse_info.token_origin 199c⟩≡ (252k)
```
(* to deal with expanded tokens, e.g. preprocessor like cpp for C *)
type token_origin =
  | OriginTok  of token_location
  | FakeTokStr of string  * (token_location * int) option (* next to *)
  | ExpandedTok of token_location * token_location * int
  | Ab (* abstract token, see Parse_info.ml comment *)
```

28.3 Program transformations

⟨*type* Parse_info.token_mutable 199d⟩≡ (252k)
```
type token_mutable = {
  token: token_origin;
  (* for spatch *)
  mutable transfo: transformation;
}
```

⟨*type* Parse_info.transformation 199e⟩≡ (252k)
```
and transformation =
  | NoTransfo
  | Remove
  | AddBefore of add
  | AddAfter of add
  | Replace of add
  | AddArgsBefore of string list
```

⟨*type* `Parse_info.add` 200a⟩≡ (252k)
```
and add =
  | AddStr of string
  | AddNewlineAndIdent
```

28.4 Fake tokens and `NoTokenLocation` errors

⟨*exception* `Parse_info.NoTokenLocation` 200b⟩≡ (252k)
```
exception NoTokenLocation of string
```

⟨*signature* `Parse_info.fake_token_location` 200c⟩≡ (252k)
```
val fake_token_location : token_location
```

⟨*signature* `Parse_info.fake_info` 200d⟩≡ (252k)
```
val fake_info : string -> t
```

⟨*signature* `Parse_info.min_max_ii_by_pos` 200e⟩≡ (252k)
```
val min_max_ii_by_pos: t list -> t * t
```

⟨*function* `AST_generic.fake` 200f⟩≡ (263c)
```
(* Try avoid using them! if you build new constructs, you should try
 * to derive the tokens in those new constructs from existing constructs.
 *)
let fake s = Parse_info.fake_info s
```

⟨*function* `AST_generic.fake_bracket` 200g⟩≡ (263c)
```
let fake_bracket x = (fake "(", x, fake ")")
```

⟨*signature* `Parse_info.is_origintok` 200h⟩≡ (252k)
```
val is_origintok: t -> bool
```

Chapter 29

Conclusion

Appendix A

Utilities

A.1 Extended standard library: `Common.mli`

⟨*signature* `Common.pr` 202a⟩≡ (241b)
```
val pr : string -> unit
```

A.1.1 Strings

⟨*signature* `Common.spf` 202b⟩≡ (241b)
```
val spf : ('a, unit, string) format -> 'a
```

⟨*signature* `Common.join` 202c⟩≡ (241b)
```
val join : string (* sep *) -> string list -> string
```

⟨*signature* `Common.split` 202d⟩≡ (241b)
```
val split : string (* sep regexp *) -> string -> string list
```

⟨*signature* `Common.null_string` 202e⟩≡ (241b)
```
val null_string : string -> bool
```

⟨*signature* `Common.i_to_s` 202f⟩≡ (241b)
```
val i_to_s : int -> string
```

⟨*signature* `Common.s_to_i` 202g⟩≡ (241b)
```
val s_to_i : string -> int
```

A.1.2 Regexps

⟨*signature match operator* 202h⟩≡ (241b)
```
val (=~) : string -> string -> bool
```

⟨*signature matched functions* 202i⟩≡ (241b)
```
val matched1 : string -> string
val matched2 : string -> string * string
val matched3 : string -> string * string * string
val matched4 : string -> string * string * string * string
val matched5 : string -> string * string * string * string * string
val matched6 : string -> string * string * string * string * string * string
val matched7 : string -> string * string * string * string * string * string * string
```

A.1.3 Filenames and paths

⟨*type* `Common.filename` 202j⟩≡ (241b)
```
type filename = string
```

⟨*type* Common.dirname 203a⟩≡ (241b)
```
type dirname = string
```

⟨*type* Common.path 203b⟩≡ (241b)
```
type path = string
```

⟨*signature* Common.fullpath 203c⟩≡ (241b)
```
(* for realpath, see efuns_c library  *)
val fullpath: filename -> filename
```

⟨*signature* Common.filename_without_leading_path 203d⟩≡ (241b)
```
val filename_without_leading_path : string -> filename -> filename
```

⟨*signature* Common.readable 203e⟩≡ (241b)
```
val readable: root:string -> filename -> filename
```

⟨*signature* Common.follow_symlinks 203f⟩≡ (241b)
```
val follow_symlinks: bool ref
```

⟨*signature* Common.files_of_dir_or_files_no_vcs_nofilter 203g⟩≡ (241b)
```
val files_of_dir_or_files_no_vcs_nofilter:
  string list -> filename list
```

A.1.4 File content

⟨*signature* Common.cat 203h⟩≡ (241b)
```
val cat :      filename -> string list
```

⟨*signature* Common.write_file 203i⟩≡ (241b)
```
val write_file : file:filename -> string -> unit
```

⟨*signature* Common.read_file 203j⟩≡ (241b)
```
val read_file : filename -> string
```

⟨*signature* Common.with_open_outfile 203k⟩≡ (241b)
```
val with_open_outfile :
  filename -> ((string -> unit) * out_channel -> 'a) -> 'a
```

⟨*signature* Common.with_open_infile 203l⟩≡ (241b)
```
val with_open_infile :
  filename -> (in_channel -> 'a) -> 'a
```

A.1.5 Running commands

⟨*exception* Common.CmdError 203m⟩≡ (241b)
```
exception CmdError of Unix.process_status * string
```

⟨*signature* Common.command2 203n⟩≡ (241b)
```
val command2 : string -> unit
```

⟨*signature* Common.cmd_to_list 203o⟩≡ (241b)
```
val cmd_to_list :  ?verbose:bool -> string -> string list (* alias *)
```

⟨*signature* Common.cmd_to_list_and_status 203p⟩≡ (241b)
```
val cmd_to_list_and_status:
  ?verbose:bool -> string -> string list * Unix.process_status
```

A.1.6 Lists

⟨*signature* Common.null 204a⟩≡ (241b)
```
val null : 'a list -> bool
```

⟨*signature* Common.exclude 204b⟩≡ (241b)
```
val exclude : ('a -> bool) -> 'a list -> 'a list
```

⟨*signature* Common.sort 204c⟩≡ (241b)
```
val sort : 'a list -> 'a list
```

⟨*signature* Common.take 204d⟩≡ (241b)
```
val take : int -> 'a list -> 'a list
```

⟨*signature* Common.take_safe 204e⟩≡ (241b)
```
val take_safe : int -> 'a list -> 'a list
```

⟨*signature* Common.drop 204f⟩≡ (241b)
```
val drop : int -> 'a list -> 'a list
```

⟨*signature* Common.span 204g⟩≡ (241b)
```
val span : ('a -> bool) -> 'a list -> 'a list * 'a list
```

⟨*signature* Common.index_list 204h⟩≡ (241b)
```
val index_list   : 'a list -> ('a * int) list
```

⟨*signature* Common.index_list_0 204i⟩≡ (241b)
```
val index_list_0 : 'a list -> ('a * int) list
```

⟨*signature* Common.index_list_1 204j⟩≡ (241b)
```
val index_list_1 : 'a list -> ('a * int) list
```

A.1.7 Optional values

⟨*signature* Common.map_filter 204k⟩≡ (241b)
```
val map_filter : ('a -> 'b option) -> 'a list -> 'b list
```

⟨*signature* Common.find_opt 204l⟩≡ (241b)
```
val find_opt: ('a -> bool) -> 'a list -> 'a option
```

⟨*signature* Common.find_some 204m⟩≡ (241b)
```
val find_some : ('a -> 'b option) -> 'a list -> 'b
```

⟨*signature* Common.find_some_opt 204n⟩≡ (241b)
```
val find_some_opt : ('a -> 'b option) -> 'a list -> 'b option
```

⟨*signature* Common.filter_some 204o⟩≡ (241b)
```
val filter_some: 'a option list -> 'a list
```

⟨*signature* Common.map_opt 204p⟩≡ (241b)
```
val map_opt: ('a -> 'b) -> 'a option -> 'b option
```

⟨*signature* Common.opt 204q⟩≡ (241b)
```
val opt: ('a -> unit) -> 'a option -> unit
```

⟨*signature* Common.do_option 204r⟩≡ (241b)
```
val do_option : ('a -> unit) -> 'a option -> unit
```

⟨*signature* Common.opt_to_list 204s⟩≡ (241b)
```
val opt_to_list: 'a option -> 'a list
```

⟨*signature* Common.TODOOPERATOR (pfff/commons/Common.mli)6 205a⟩≡ (241b)
```
val (>>=): 'a option -> ('a -> 'b option) -> 'b option
```

⟨*signature* Common.TODOOPERATOR (pfff/commons/Common.mli)7 205b⟩≡ (241b)
```
val (|||): 'a option -> 'a -> 'a
```

A.1.8 Alternative values

⟨*type* Common.either 205c⟩≡ (241b)
```
type ('a, 'b) either = Left of 'a | Right of 'b
```

⟨*type* Common.either3 205d⟩≡ (241b)
```
type ('a, 'b, 'c) either3 = Left3 of 'a | Middle3 of 'b | Right3 of 'c
```

⟨*signature* Common.partition_either 205e⟩≡ (241b)
```
val partition_either :
  ('a -> ('b, 'c) either) -> 'a list -> 'b list * 'c list
```

⟨*signature* Common.partition_either3 205f⟩≡ (241b)
```
val partition_either3 :
  ('a -> ('b, 'c, 'd) either3) -> 'a list -> 'b list * 'c list * 'd list
```

A.1.9 Association lists, sorting, and grouping

⟨*type* Common.assoc 205g⟩≡ (241b)
```
type ('a, 'b) assoc = ('a * 'b) list
```

⟨*signature* Common.sort_by_val_lowfirst 205h⟩≡ (241b)
```
val sort_by_val_lowfirst: ('a,'b) assoc -> ('a * 'b) list
```

⟨*signature* Common.sort_by_val_highfirst 205i⟩≡ (241b)
```
val sort_by_val_highfirst: ('a,'b) assoc -> ('a * 'b) list
```

⟨*signature* Common.sort_by_key_lowfirst 205j⟩≡ (241b)
```
val sort_by_key_lowfirst: ('a,'b) assoc -> ('a * 'b) list
```

⟨*signature* Common.sort_by_key_highfirst 205k⟩≡ (241b)
```
val sort_by_key_highfirst: ('a,'b) assoc -> ('a * 'b) list
```

⟨*signature* Common.group_by 205l⟩≡ (241b)
```
val group_by: ('a -> 'b) -> 'a list -> ('b * 'a list) list
```

⟨*signature* Common.group_assoc_bykey_eff 205m⟩≡ (241b)
```
val group_assoc_bykey_eff : ('a * 'b) list -> ('a * 'b list) list
```

⟨*signature* Common.group_by_mapped_key 205n⟩≡ (241b)
```
val group_by_mapped_key: ('a -> 'b) -> 'a list -> ('b * 'a list) list
```

⟨*signature* Common.group_by_multi 205o⟩≡ (241b)
```
val group_by_multi: ('a -> 'b list) -> 'a list -> ('b * 'a list) list
```

A.1.10 Stacks

⟨*type* Common.stack 205p⟩≡ (241b)
```
type 'a stack = 'a list
```

⟨*signature* Common.push 205q⟩≡ (241b)
```
val push : 'a -> 'a stack ref -> unit
```

A.1.11 Hash tables and sets

⟨*signature* Common.hash_of_list 206a⟩≡ (241b)
```
val hash_of_list : ('a * 'b) list -> ('a, 'b) Hashtbl.t
```

⟨*signature* Common.hash_to_list 206b⟩≡ (241b)
```
val hash_to_list : ('a, 'b) Hashtbl.t -> ('a * 'b) list
```

⟨*type* Common.hashset 206c⟩≡ (241b)
```
type 'a hashset = ('a, bool) Hashtbl.t
```

⟨*signature* Common.hashset_of_list 206d⟩≡ (241b)
```
val hashset_of_list : 'a list -> 'a hashset
```

⟨*signature* Common.hashset_to_list 206e⟩≡ (241b)
```
val hashset_to_list : 'a hashset -> 'a list
```

A.1.12 Exceptions

⟨*signature* Common.unwind_protect 206f⟩≡ (241b)
```
(* emacs spirit *)
val unwind_protect : (unit -> 'a) -> (exn -> 'b) -> 'a
(* java spirit *)
```

⟨*signature* Common.finalize 206g⟩≡ (241b)
```
(* java spirit *)
val finalize :       (unit -> 'a) -> (unit -> 'b) -> 'a
```

A.1.13 Command-line arguments

⟨*type* Common.arg_spec_full 206h⟩≡ (241b)
```
type arg_spec_full = Arg.key * Arg.spec * Arg.doc
```

⟨*type* Common.cmdline_options 206i⟩≡ (241b)
```
type cmdline_options = arg_spec_full list
```

⟨*type* Common.options_with_title 206j⟩≡ (241b)
```
type options_with_title = string * string * arg_spec_full list
```

⟨*type* Common.cmdline_sections 206k⟩≡ (241b)
```
type cmdline_sections = options_with_title list
```

⟨*signature* Common.parse_options 206l⟩≡ (241b)
```
(* A wrapper around Arg modules that have more logical argument order,
 * and returns the remaining args.
*)
val parse_options :
  cmdline_options -> Arg.usage_msg -> string array -> string list
(* Another wrapper that does Arg.align automatically *)
```

⟨*signature* Common.usage 206m⟩≡ (241b)
```
(* Another wrapper that does Arg.align automatically *)
val usage : Arg.usage_msg -> cmdline_options -> unit
```

⟨*signature* Common.short_usage 206n⟩≡ (241b)
```
(* Work with the options_with_title type way to organize a long
 * list of command line switches.
*)
val short_usage :
  Arg.usage_msg -> short_opt:cmdline_options -> unit
```

⟨*signature* Common.long_usage 207a⟩≡ (241b)
```
val long_usage :
  Arg.usage_msg -> short_opt:cmdline_options -> long_opt:cmdline_sections ->
  unit
```

⟨*signature* Common.arg_align2 207b⟩≡ (241b)
```
(* With the options_with_title way, we don't want the default -help and --help
 * so need adapter of Arg module, not just wrapper.
*)
val arg_align2 : cmdline_options -> cmdline_options
```

⟨*signature* Common.arg_parse2 207c⟩≡ (241b)
```
val arg_parse2 :
  cmdline_options -> Arg.usage_msg -> (unit -> unit) (* short_usage func *) ->
  string list
```

⟨*type* Common.flag_spec 207d⟩≡ (241b)
```
(* The action lib. Useful to debug supart of your system. cf some of
 * my Main.ml for example of use. *)
type flag_spec   = Arg.key * Arg.spec * Arg.doc
```

A.1.14 Command-line actions

⟨*type* Common.action_spec 207e⟩≡ (241b)
```
type action_spec = Arg.key * Arg.doc * action_func
```

⟨*type* Common.action_func 207f⟩≡ (241b)
```
and action_func = (string list -> unit)
```

⟨*type* Common.cmdline_actions 207g⟩≡ (241b)
```
type cmdline_actions = action_spec list
```

⟨*exception* Common.WrongNumberOfArguments 207h⟩≡ (241b)
```
exception WrongNumberOfArguments
```

⟨*signature* Common.mk_action_0_arg 207i⟩≡ (241b)
```
val mk_action_0_arg : (unit -> unit)                        -> action_func
```

⟨*signature* Common.mk_action_1_arg 207j⟩≡ (241b)
```
val mk_action_1_arg : (string -> unit)                      -> action_func
```

⟨*signature* Common.mk_action_2_arg 207k⟩≡ (241b)
```
val mk_action_2_arg : (string -> string -> unit)            -> action_func
```

⟨*signature* Common.mk_action_3_arg 207l⟩≡ (241b)
```
val mk_action_3_arg : (string -> string -> string -> unit) -> action_func
```

⟨*signature* Common.mk_action_4_arg 207m⟩≡ (241b)
```
val mk_action_4_arg : (string -> string -> string -> string -> unit) ->
  action_func
```

⟨*signature* Common.mk_action_n_arg 207n⟩≡ (241b)
```
val mk_action_n_arg : (string list -> unit) -> action_func
```

⟨*signature* Common.options_of_actions 207o⟩≡ (241b)
```
val options_of_actions:
  string ref (* the action ref *) -> cmdline_actions -> cmdline_options
```

⟨*signature* Common.do_action 208a⟩≡ (241b)
```
val do_action:
  Arg.key -> string list (* args *) -> cmdline_actions -> unit
```

⟨*signature* Common.action_list 208b⟩≡ (241b)
```
val action_list:
  cmdline_actions -> Arg.key list
```

A.1.15 Debugging

A.1.16 Profiling

A.1.17 Temporary files

⟨*signature* Common.new_temp_file 208c⟩≡ (241b)
```
val new_temp_file : string (* prefix *) -> string (* suffix *) -> filename
```

⟨*signature* Common._temp_files_created 208d⟩≡ (241b)
```
(* creation of /tmp files, a la gcc
 * ex: new_temp_file "cocci" ".c" will give "/tmp/cocci-3252-434465.c"
*)
val _temp_files_created : (string, unit) Hashtbl.t
```

⟨*signature* Common.save_tmp_files 208e⟩≡ (241b)
```
val save_tmp_files : bool ref
```

⟨*signature* Common.erase_temp_files 208f⟩≡ (241b)
```
val erase_temp_files : unit -> unit
```

⟨*signature* Common.erase_this_temp_file 208g⟩≡ (241b)
```
val erase_this_temp_file : filename -> unit
```

A.1.18 Advanced utilities

⟨*signature* Common.save_excursion 208h⟩≡ (241b)
```
val save_excursion : 'a ref -> 'a -> (unit -> 'b) -> 'b
```

⟨*exception* Common.UnixExit 208i⟩≡ (241b)
```
exception UnixExit of int
```

⟨*exception* Common.Timeout 208j⟩≡ (241b)
```
exception Timeout
```

⟨*signature* Common.timeout_function 208k⟩≡ (241b)
```
val timeout_function: ?verbose:bool -> int -> (unit -> 'a) -> 'a
```

⟨*signature equality operators* 208l⟩≡ (241b)
```
val (=|=) : int    -> int    -> bool
val (=<=) : char   -> char   -> bool
val (=$=) : string -> string -> bool
val (=:=) : bool   -> bool   -> bool
```

⟨*signature* Common.TODOOPERATOR (pfff/commons/Common.mli)4 208m⟩≡ (241b)
```
val (=*=): 'a -> 'a -> bool
```

⟨*exception* Common.Multi_found 208n⟩≡ (241b)
```
exception Multi_found
```

⟨*signature* `Common.memoized` 209a⟩≡ (241b)

```
val memoized :
  ?use_cache:bool -> ('a, 'b) Hashtbl.t -> 'a -> (unit -> 'b) -> 'b
```

⟨*signature* `Common.cache_computation` 209b⟩≡ (241b)

```
val cache_computation :
  ?verbose:bool -> ?use_cache:bool -> filename  -> string (* extension *) ->
  (unit -> 'a) -> 'a
```

⟨*signature* `Common.main_boilerplate` 209c⟩≡ (241b)

```
(* do some finalize, signal handling, unix exit conversion, etc *)
val main_boilerplate : (unit -> unit) -> unit
```

Appendix B

Debugging

⟨Main_semgrep_core.options *concatenated flags* 210a⟩+≡ (46d) ◁199b 212c▷
```
  (* inlining of: Common2.cmdline_flags_devel () @ *)
  @ [
      ( "-debugger",
        Arg.Set Common.debugger,
        " option to set if launched inside ocamldebug" );
      ( "-profile",
        Arg.Unit
          (fun () ->
            Common.profile := Common.ProfAll;
            profile := true),
        " output profiling information" );
    ]
```

⟨*signature* Common.debugger 210b⟩≡ (241b)
```
  (* if set then certain functions like unwind_protect will not
   * do a try and finalize and instead just call the function, which
   * helps in ocamldebug and also in getting better backtraces.
   * This is also useful to set in a js_of_ocaml (jsoo) context to
   * again get better backtraces.
  *)
  val debugger : bool ref
```

⟨*signature* Common.pr2 210c⟩≡ (241b)
```
  val pr2 : string -> unit
```

⟨*signature* Common._already_printed 210d⟩≡ (241b)
```
  (* forbid pr2_once to do the once "optimisation" *)
  val _already_printed : (string, bool) Hashtbl.t
```

⟨*signature* Common.disable_pr2_once 210e⟩≡ (241b)
```
  val disable_pr2_once : bool ref
```

⟨*signature* Common.pr2_once 210f⟩≡ (241b)
```
  val pr2_once : string -> unit
```

B.1 Dumping raw values: `Dumper.mli`

⟨*signature* Dumper.dump 210g⟩≡ (247a)
```
  (* Dump an OCaml value into a printable string.
   * By Richard W.M. Jones (rich@annexia.org).
   * Dumper.mli 1.1 2005/02/03 23:07:47 rich Exp
  *)

  val dump : 'a -> string
```

⟨*signature* `Common.pr2_gen` 211a⟩≡ (241b)
```
val pr2_gen: 'a -> unit
```

⟨*signature* `Common.dump` 211b⟩≡ (241b)
```
val dump: 'a -> string
```

B.2 Dumping structured values: `OCaml.mli`

⟨*type* `OCaml.v` 211c⟩≡ (246b)
```
(* OCaml values (a restricted form of expressions) *)
type v =
  | VUnit
  | VBool of bool | VFloat of float | VInt of int
  | VChar of char | VString of string

  | VTuple of v list
  | VDict of (string * v) list
  | VSum of string * v list

  | VVar of (string * int64)
  | VArrow of string

  (* special cases *)
  | VNone | VSome of v
  | VList of v list
  | VRef of v

  | VTODO of string
```

⟨*signature* `OCaml.string_of_v` 211d⟩≡ (246b)
```
(* regular pretty printer (not via sexp, but using Format) *)
val string_of_v: v -> string
```

⟨*signature* `OCaml.vof_xxx` *functions* 211e⟩≡ (246b)
```
(* building blocks, used by code generated using ocamltarzan *)
val vof_unit   : unit -> v
val vof_bool   : bool -> v
val vof_int    : int -> v
val vof_float   : float -> v
val vof_string : string -> v
val vof_list   : ('a -> v) -> 'a list -> v
val vof_option : ('a -> v) -> 'a option -> v
val vof_ref    : ('a -> v) -> 'a ref -> v
val vof_either    : ('a -> v) -> ('b -> v) -> ('a, 'b) Common.either -> v
val vof_either3    : ('a -> v) -> ('b -> v) -> ('c -> v) ->
  ('a, 'b, 'c) Common.either3 -> v
val vof_all3   : ('a -> v) -> ('b -> v) -> ('c -> v) -> 'a * 'b * 'c -> v
```

B.3 The OCaml debugger and `run-ocamldebug.sh`

B.4 `pfff -debug`

⟨*constant* `Main_semgrep_core.debug` 211f⟩≡ (369)
```
let debug = ref false
```

⟨*constant* `Flag_semgrep.debug` 212a⟩≡ (384)
```
(* note that this will stop at the first fail(), but if you restrict
 * enough your pattern, this can help you debug your problem.*)
let debug_matching = ref false
```

⟨*constant* `Flag_semgrep.debug_with_full_position` 212b⟩≡ (384)
```
let debug_with_full_position = ref false
```

⟨`Main_semgrep_core.options` *concatenated flags* 212c⟩+≡ (46d) ◁210a 224f▷
```
@ Meta_parse_info.cmdline_flags_precision ()
```

B.5 semgrep -debug

⟨*function* `Matching_generic.str_of_any` 212d⟩≡ (442)

Appendix C

Testing

C.1 Unit testing library: `OUnit.mli`

⟨*signature* `OUnit.assert_equal` 213a⟩≡ (248l)
```
(** Compares two values, when they are not equal a failure is signaled.
    The cmp parameter can be used to pass a different compare function.
    This parameter defaults to ( = ). The optional printer can be used
    to convert the value to string, so a nice error message can be
    formatted. When msg is also set it can be used to identify the failure.

    @raise Failure description *)
val assert_equal : ?cmp:('a -> 'a -> bool) ->  ?printer:('a -> string) ->
                   ?msg:string -> 'a -> 'a -> unit
```

⟨*signature* `OUnit.assert_failure` 213b⟩≡ (248l)
```
(** Signals a failure. This will raise an exception with the specified
    string.

    @raise Failure to signal a failure *)
val assert_failure : string -> 'a
```

⟨*type* `OUnit.test` 213c⟩≡ (248l)
```
(** The type of tests *)
type test =
    TestCase of test_fun
  | TestList of test list
  | TestLabel of string * test
```

⟨*type* `OUnit.test_fun` 213d⟩≡ (248l)
```
(** The type of test function *)
type test_fun = unit -> unit
```

⟨*signature* `OUnit.TODOOPERATOR (pfff/commons/OUnit.mli)2` 213e⟩≡ (248l)
```
(** Create a TestLabel for a TestCase *)
val (>::) : string -> test_fun -> test
```

⟨*signature* `OUnit.TODOOPERATOR (pfff/commons/OUnit.mli)3` 213f⟩≡ (248l)
```
(** Create a TestLabel for a TestList *)
val (>:::) : string -> test list -> test
```

⟨*signature* `OUnit.run_test_tt` 213g⟩≡ (248l)
```
(** A simple text based test runner. It prints out information
    during the Test. *)
val run_test_tt : ?verbose:bool -> test -> test_result list
```

⟨*signature* OUnit.run_test_tt_main 214a⟩≡ (248l)
```
(** Main version of the text based test runner. It reads the supplied command
    line arguments to set the verbose level and limit the number of test to run
  *)
val run_test_tt_main : test -> test_result list
```

⟨*type* OUnit.test_result 214b⟩≡ (248l)
```
(** The possible results of a test *)
type test_result =
    RSuccess of path
  | RFailure of path * string
  | RError of path * string
  | RSkip of path * string
  | RTodo of path * string
```

C.2 Example: Python parsing regression

⟨*signature* Unit_parsing_python.unittest 214c⟩≡ (366c)
```
(* Returns the testsuite for parsing/. To be concatenated by
 * the caller (e.g. in pfff/main_test.ml ) with other testsuites and
 * run via OUnit.run_test_tt
*)
val unittest: OUnit.test
```

⟨*constant* Unit_parsing_python.unittest 214d⟩≡ (366d)
```
let unittest =
  "parsing_python" >::: [

    "regression files" >:: (fun () ->
      let dir = Config_pfff.tests_path "python/parsing" in
      let files = Common2.glob (spf "%s/*.py" dir)in
      files |> List.iter (fun file ->
        try
          let _ = Parse_python.parse_program file in
          ()
        with Parse_info.Parsing_error _ ->
          assert_failure (spf "it should correctly parse %s" file)
      )
    );
  ]
```

C.3 Regression testing library

⟨*signature* Error_code.expected_error_lines_of_files 214e⟩≡ (256b)
```
(* extract all the lines with ERROR: comment in test files *)
val expected_error_lines_of_files:
  ?regexp:string -> Common.filename list ->
  (Common.filename * int (* line with ERROR *)) list
```

⟨*signature* Error_code.compare_actual_to_expected 214f⟩≡ (256b)
```
(* use Ounit *)
val compare_actual_to_expected:
  error list -> (Common.filename * int) list -> unit
```

C.4 Example: Semgrep regression testing

⟨*constant* Test.tests_path 215a⟩≡ (460h)
```
(* ran from _build/default/tests/ hence the '..'s below *)
let tests_path = "../../../tests"
```

⟨*constant* Test.data_path 215b⟩≡ (460h)
```
let data_path = "../../../data"
```

⟨*function* Test.ast_fuzzy_of_string 215c⟩≡ (460h)

⟨*function* Test.regression_tests_for_lang 215d⟩≡ (460h)
```
let regression_tests_for_lang ~with_caching files lang =
  files |> List.map (fun file ->
   (Filename.basename file) >:: (fun () ->
    let sgrep_file =
      let (d,b,_e) = Common2.dbe_of_filename file in
      let candidate1 = Common2.filename_of_dbe (d,b,"sgrep") in
      if Sys.file_exists candidate1
      then candidate1
      else
        let d = Filename.concat tests_path "POLYGLOT" in
        let candidate2 = Common2.filename_of_dbe (d,b,"sgrep") in
        if Sys.file_exists candidate2
        then candidate2
        else failwith (spf "could not find sgrep file for %s" file)
    in
    let ast =
        try
          let { Parse_target. ast; errors; _ } =
            Parse_target.parse_and_resolve_name_use_pfff_or_treesitter lang file
          in
          if errors <> []
          then pr2 (spf "WARNING: fail to fully parse %s" file);
          ast
        with exn ->
          failwith (spf "fail to parse %s (exn = %s)" file
                    (Common.exn_to_s exn))
    in
    let pattern =
      Common.save_excursion Flag_parsing.sgrep_mode true (fun () ->
        try
          Parse_pattern.parse_pattern lang ~print_errors:true (Common.read_file sgrep_file)
        with exn ->
          failwith (spf "fail to parse pattern %s with lang = %s (exn = %s)"
                        sgrep_file
                      (Lang.string_of_lang lang)
                    (Common.exn_to_s exn))
      )
    in
    Error_code.g_errors := [];

    let rule = { R.
      id = "unit testing"; pattern; message = "";
      severity = R.Error; languages = [lang];
      pattern_string = "test: no need for pattern string";
    } in
    let equiv =
     (* Python == is not the same than !(==) *)
     if lang <> Lang.Python
```

```
      then Parse_equivalences.parse
        (Filename.concat data_path "basic_equivalences.yml")
      else []
    in
    Common.save_excursion Flag_semgrep.with_opt_cache with_caching (fun() ->
     Semgrep_generic.check
       ~hook:(fun _env matched_tokens ->
       (* there are a few fake tokens in the generic ASTs now (e.g.,
        * for DotAccess generated outside the grammar) *)
         let xs = Lazy.force matched_tokens in
         let toks = xs |> List.filter Parse_info.is_origintok in
         let (minii, _maxii) = Parse_info.min_max_ii_by_pos toks in
         Error_code.error minii (Error_code.SemgrepMatchFound ("",""))
       )
       Config_semgrep.default_config
       [rule] equiv (file, lang, ast)
     |> ignore;

     let actual = !Error_code.g_errors in
     let expected = Error_code.expected_error_lines_of_files [file] in
       Error_code.compare_actual_to_expected actual expected;
    )
   )
 )
```

⟨*constant* `Test.lang_regression_tests 216`⟩≡ (460h)

```
 let lang_regression_tests ~with_caching =
   let regression_tests_for_lang files lang =
     regression_tests_for_lang ~with_caching files lang
   in
   let name_suffix =
     if with_caching then " with caching"
     else " without caching"
   in
   "lang regression testing" ^ name_suffix >::: [
   "semgrep Python" >::: (
     let dir = Filename.concat tests_path "python" in
     let files = Common2.glob (spf "%s/*.py" dir) in
     let lang = Lang.Python in
     regression_tests_for_lang files lang
   );
   "semgrep Javascript" >::: (
     let dir = Filename.concat tests_path "js" in
     let files = Common2.glob (spf "%s/*.js" dir) in
     let lang = Lang.Javascript in
     regression_tests_for_lang files lang
   );
   "semgrep Typescript" >::: (
     let dir = Filename.concat tests_path "ts" in
     let files = Common2.glob (spf "%s/*.ts" dir) in
     let lang = Lang.Typescript in
     regression_tests_for_lang files lang
   );
   "semgrep Typescript on Javascript (no JSX)" >::: (
     let dir = Filename.concat tests_path "js" in
     let files = Common2.glob (spf "%s/*.js" dir) in
     let files = Common.exclude (fun s -> s =~ ".*xml" || s =~ ".*jsx") files in
     let lang = Lang.Typescript in
     regression_tests_for_lang files lang
   );
```

```
"semgrep JSON" >::: (
  let dir = Filename.concat tests_path "json" in
  let files = Common2.glob (spf "%s/*.json" dir) in
  let lang = Lang.JSON in
  regression_tests_for_lang files lang
);
"semgrep Java" >::: (
  let dir = Filename.concat tests_path "java" in
  let files = Common2.glob (spf "%s/*.java" dir) in
  let lang = Lang.Java in
  regression_tests_for_lang files lang
);
"semgrep C" >::: (
  let dir = Filename.concat tests_path "c" in
  let files = Common2.glob (spf "%s/*.c" dir) in
  let lang = Lang.C in
  regression_tests_for_lang files lang
);
"semgrep Go" >::: (
  let dir = Filename.concat tests_path "go" in
  let files = Common2.glob (spf "%s/*.go" dir) in
  let lang = Lang.Go in
  regression_tests_for_lang files lang
);
"semgrep OCaml" >::: (
  let dir = Filename.concat tests_path "ocaml" in
  let files = Common2.glob (spf "%s/*.ml" dir) in
  let lang = Lang.OCaml in
  regression_tests_for_lang files lang
);
"semgrep Ruby" >::: (
  let dir = Filename.concat tests_path "ruby" in
  let files = Common2.glob (spf "%s/*.rb" dir) in
  let lang = Lang.Ruby in
  regression_tests_for_lang files lang
);
"semgrep PHP" >::: (
  let dir = Filename.concat tests_path "php" in
  let files = Common2.glob (spf "%s/*.php" dir) in
  let lang = Lang.PHP in
  regression_tests_for_lang files lang
);
"semgrep C#" >::: (
  let dir = Filename.concat tests_path "csharp" in
  let files = Common2.glob (spf "%s/*.cs" dir) in
  let lang = Lang.Csharp in
  regression_tests_for_lang files lang
);
"semgrep Lua" >::: (
  let dir = Filename.concat tests_path "lua" in
  let files = Common2.glob (spf "%s/*.lua" dir) in
  let lang = Lang.Lua in
  regression_tests_for_lang files lang
);
"semgrep Rust" >::: (
  let dir = Filename.concat tests_path "rust" in
  let files = Common2.glob (spf "%s/*.rs" dir) in
  let lang = Lang.Rust in
  regression_tests_for_lang files lang
);
```

```
    "semgrep Yaml" >::: (
      let dir = Filename.concat tests_path "yaml" in
      let files = Common2.glob (spf "%s/*.yaml" dir) in
      let lang = Lang.Yaml in
      regression_tests_for_lang files lang
    );
   ]
```

⟨*constant* Test.lint_regression_tests 218a⟩≡ (460h)

```
  (* mostly a copy paste of pfff/linter/unit_linter.ml *)
  let lint_regression_tests ~with_caching =
    let name =
      if with_caching then
        "lint regression testing with caching"
      else
        "lint regression testing without caching"
    in
    name >:: (fun () ->
    let p path = Filename.concat tests_path path in
    let rule_file = Filename.concat data_path "basic.yml" in
    let lang = Lang.Python in

    let test_files = [
     p "OTHER/mini_rules/stupid.py";
    ] in

    (* expected *)
    let expected_error_lines = E.expected_error_lines_of_files test_files in

    (* actual *)
    E.g_errors := [];
    let rules = Parse_mini_rule.parse rule_file in
    let equivs = [] in

    test_files |> List.iter (fun file ->
      E.try_with_exn_to_error file (fun () ->
      let { Parse_target. ast; _} =
          Parse_target.just_parse_with_lang lang file in
      Common.save_excursion Flag_semgrep.with_opt_cache with_caching (fun() ->
        Semgrep_generic.check ~hook:(fun _ _ -> ())
           Config_semgrep.default_config
           rules equivs
           (file, lang, ast)
        |> List.iter JSON_report.match_to_error;
      )
    ));

    (* compare *)
    let actual_errors = !E.g_errors in
    if !verbose
    then actual_errors |> List.iter (fun e -> pr (E.string_of_error e));
    E.compare_actual_to_expected actual_errors expected_error_lines
    )
```

C.5 Semgrep test launcher: `semgrep/tests/Test.ml`

⟨*constant* Test.usage 218b⟩≡ (460h)

```
  let usage =
```

```
    Common.spf "Usage: %s [options] [regexp]> \nrun the unit tests matching the regexp\nOptions:"
       (Filename.basename Sys.argv.(0))
```

⟨*function* `Test.main` 219a⟩≡ (460h)
```
  let main () =
    let args = ref [] in
    Arg.parse options (fun arg -> args := arg::!args) usage;

    (match List.rev !args with
    | [] -> test "all"
    | [x] -> test x
    | _::_::_ ->
      print_string "too many arguments\n";
      Arg.usage options usage;
    )
```

⟨*toplevel* `Test._1` 219b⟩≡ (460h)
```
  let _ = main ()
```

⟨*function* `Test.test` 219c⟩≡ (460h)
```
  let test regexp =
    (* There is no reflection in OCaml so the unit test framework OUnit requires
     * us to explicitely build the test suites (which is not too bad).
     *)
    let tests =
      "all" >::: [

        (* just expression vs expression testing for one language (Python) *)
        Unit_matcher.unittest ~any_gen_of_string;
        Unit_synthesizer.unittest;
        Unit_dataflow.unittest Parse_target.parse_program;
        Unit_typing_generic.unittest
          Parse_target.parse_program
          (fun lang file -> Parse_pattern.parse_pattern lang file);
        Unit_naming_generic.unittest Parse_target.parse_program;

        lang_parsing_tests;
        (* full testing for many languages *)
        lang_regression_tests ~with_caching:false;
        lang_regression_tests ~with_caching:true;
        (* TODO Unit_matcher.spatch_unittest ~xxx *)
        (* TODO Unit_matcher_php.unittest; (* sgrep, spatch, refactoring, unparsing *) *)
        lint_regression_tests ~with_caching:false;
        lint_regression_tests ~with_caching:true;
        eval_regression_tests;
        full_rule_regression_tests;
        lang_tainting_tests;
      ]
    in
    let suite =
      if regexp = "all"
      then tests
      else
        let paths =
          OUnit.test_case_paths tests |> List.map OUnit.string_of_path in
        let keep = paths |> List.filter (fun path ->
          path =~ (".*" ^ regexp))
        in
        Common2.some (OUnit.test_filter keep tests)
    in
```

```
let results = OUnit.run_test_tt ~verbose:!verbose suite in
let has_an_error =
  results |> List.exists (function
  | OUnit.RSuccess _ | OUnit.RSkip _ | OUnit.RTodo _ -> false
  | OUnit.RFailure _ | OUnit.RError _ -> true
  )
in
if has_an_error
then exit 1
else exit 0
```

Appendix D

Profiling

D.1 The OCaml profilers

D.2 Common.profile_code()

⟨*type* Common.prof 221a⟩≡ (241b)
```
type prof = ProfAll | ProfNone | ProfSome of string list
```

⟨*signature* Common.profile 221b⟩≡ (241b)
```
val profile : prof ref
```

⟨*signature* Common.show_trace_profile 221c⟩≡ (241b)
```
val show_trace_profile : bool ref
```

⟨*signature* Common._profile_table 221d⟩≡ (241b)
```
val _profile_table : (string, (float ref * int ref)) Hashtbl.t ref
```

⟨*signature* Common.profile_code 221e⟩≡ (241b)
```
val profile_code : string -> (unit -> 'a) -> 'a
```

⟨*signature* Common.profile_diagnostic 221f⟩≡ (241b)
```
val profile_diagnostic : unit -> string
```

⟨*signature* Common.profile_code_exclusif 221g⟩≡ (241b)
```
val profile_code_exclusif : string -> (unit -> 'a) -> 'a
```

⟨*signature* Common.profile_code_inside_exclusif_ok 221h⟩≡ (241b)
```
val profile_code_inside_exclusif_ok : string -> (unit -> 'a) -> 'a
```

⟨*signature* Common.report_if_take_time 221i⟩≡ (241b)
```
val report_if_take_time : int -> string -> (unit -> 'a) -> 'a
(* similar to profile_code but print some information during execution too *)
```

⟨*signature* Common.profile_code2 221j⟩≡ (241b)
```
(* similar to profile_code but print some information during execution too *)
val profile_code2 : string -> (unit -> 'a) -> 'a
```

Appendix E

Error Management

⟨*exception* `Common.Todo` 222a⟩≡ (241b)
```
exception Todo
```

⟨*exception* `Common.Impossible` 222b⟩≡ (241b)
```
exception Impossible
```

⟨*signature* `Common.exn_to_s` 222c⟩≡ (241b)
```
val exn_to_s : exn -> string
```

E.1 Semgrep parsing errors

E.2 pfff parsing errors

⟨*signature* `Parse_info.lexical_error` 222d⟩≡ (252k)
```
val lexical_error: string -> Lexing.lexbuf -> unit
```

⟨*signature* `Parse_info.string_of_info` 222e⟩≡ (252k)
```
(* small error reporting, for longer reports use error_message above *)
val string_of_info: t -> string
```

E.3 pfff/h_program_lang/Error_code.mli

⟨*type* `Error_code.error` 222f⟩≡ (256b)
```
type error = {
  typ: error_kind;
  loc: Parse_info.token_location;
  sev: severity;
}
```

⟨*type* `Error_code.severity` 222g⟩≡ (256b)
```
and severity = Error | Warning | Info
```

⟨*type* `Error_code.error_kind` 222h⟩≡ (256b)
```
and error_kind =
  (* parsing related *)
  | LexicalError of string
  | ParseError (* a.k.a SyntaxError *)
  | AstBuilderError of string
  | AstGenericError of string
  | OtherParsingError of string
```

```
    (* matching (semgrep) related *)
    | MatchingError of string (* internal error, e.g., NoTokenLocation *)
    | SemgrepMatchFound of (string (* check_id *) * string (* msg *))
    | TooManyMatches of string (* can contain offending pattern *)

    (* global analysis *)
    | Deadcode of entity

    (* use/def entities *)
    | UndefinedDefOfDecl of entity
    | UnusedExport of entity * Common.filename
    | UnusedVariable of string * Scope_code.t

    (* CFG/DFG *)
    | UnusedStatement
    | UnusedAssign of string (* var name *)
    | UseOfUninitialized of string
    | CFGError of string

    (* other *)
    | FatalError of string
    | Timeout of string option
    | OutOfMemory of string option
```

⟨*type* Error_code.entity 223a⟩≡ (256b)
```
  and entity = (string * Entity_code.entity_kind)
```

⟨*signature* Error_code.g_errors 223b⟩≡ (256b)
```
  val g_errors: error list ref
```

⟨*signature* Error_code.error 223c⟩≡ (256b)
```
  (* !modify g_errors! *)
  val error  : Parse_info.t -> error_kind -> unit
```

⟨*signature* Error_code.warning 223d⟩≡ (256b)
```
  val warning: Parse_info.t -> error_kind -> unit
```

⟨*signature* Error_code.info 223e⟩≡ (256b)
```
  val info: Parse_info.t -> error_kind -> unit
```

⟨*signature* Error_code.string_of_error 223f⟩≡ (256b)
```
  val string_of_error: error -> string
```

⟨*signature* Error_code.report_parse_errors 223g⟩≡ (256b)
```
  val report_parse_errors: bool ref
```

⟨*signature* Error_code.report_fatal_errors 223h⟩≡ (256b)
```
  val report_fatal_errors: bool ref
```

⟨*signature* Error_code.options 223i⟩≡ (256b)
```
  val options: unit -> Common.cmdline_options
```

⟨*signature* Error_code.filter_maybe_parse_and_fatal_errors 223j⟩≡ (256b)
```
  (* use the flags above to filter certain errors *)
  val filter_maybe_parse_and_fatal_errors: error list -> error list
  (* convert parsing and other fatal exceptions in regular 'error'
   * added to g_errors
  *)
```

⟨*signature* `Error_code.exn_to_error` 224a⟩≡ (256b)
```
val exn_to_error: Common.filename -> exn -> error
```

⟨*signature* `Error_code.try_with_exn_to_error` 224b⟩≡ (256b)
```
val try_with_exn_to_error:
  Common.filename -> (unit -> unit) -> unit
```

⟨*signature* `Error_code.try_with_print_exn_and_reraise` 224c⟩≡ (256b)
```
val try_with_print_exn_and_reraise:
  Common.filename -> (unit -> unit) -> unit
```

E.4 Error recovery

⟨*constant* `Main_semgrep_core.error_recovery` 224d⟩≡ (369)
```
(* try to continue processing files, even if one has a parse error with -e/f.
 * note that -rules_file does its own error recovery.
 *)
let error_recovery = ref false
```

⟨`Main_semgrep_core.options` *other cases* 224e⟩+≡ (46d) ◁193b
```
( "-error_recovery",
  Arg.Unit
    (fun () ->
      error_recovery := true;
      Flag_parsing.error_recovery := true),
  " do not stop at first parsing error with -e/-f" );
( "-fail_fast",
  Arg.Set fail_fast,
  " stop at first exception (and get a backtrace)" );
```

⟨`Main_semgrep_core.options` *concatenated flags* 224f⟩+≡ (46d) ◁212c
```
@ Error_code.options ()
```

Appendix F

Extra Code

F.1 Misc flags and actions

⟨Main.options *concatenated flags* 225a⟩≡ (38e)
```
  Flag_parsing.cmdline_flags_verbose () @
  Flag_parsing.cmdline_flags_debugging () @
  Meta_parse_info.cmdline_flags_precision () @

  Flag_parsing_cpp.cmdline_flags_debugging () @

  Flag_parsing_php.cmdline_flags_pp () @
  Flag_parsing_cpp.cmdline_flags_macrofile () @

  Common2.cmdline_flags_devel () @
  Common2.cmdline_flags_other () @
```

⟨Main.all_actions *concatenated actions* 225b⟩+≡ (39a) ◁39e
```
  Test_parsing_ml.actions()@
  Test_analyze_ml.actions()@
  Test_parsing_skip.actions()@

  Test_parsing_php.actions()@
  Test_parsing_js.actions()@
  Test_parsing_json.actions()@
  Test_analyze_js.actions()@
  Test_parsing_python.actions()@
  Test_parsing_ruby.actions()@
  Test_analyze_ruby.actions()@

  Test_parsing_c.actions()@
  Test_parsing_cpp.actions()@
  Test_parsing_java.actions()@
  Test_parsing_go.actions()@

  Test_parsing_nw.actions()@
  Test_parsing_hs.actions()@
  Test_parsing_csharp.actions()@
  Test_parsing_rust.actions()@
  Test_parsing_erlang.actions()@

  Test_parsing_text.actions()@
  Test_parsing_html.actions()@
  (* TODO need dune file
      Test_parsing_css.actions()@
      Test_parsing_web.actions()@
```

```
    Test_parsing_sql.actions()@
*)

Test_parsing_lisp.actions()@

(* beta *)


(*  Test_analyze_generic.actions() @ *)
(*
  Test_analyze_cpp.actions () ++
  Test_analyze_php.actions () ++
  Test_analyze_ml.actions () ++
  Test_analyze_c.actions() ++
*)
```

F.2 Misc any

⟨AST_generic.any *other cases* 226a⟩≡ (32i)

```
  (* todo: get rid of some? *)
  | Modn of module_name
  | ModDk of module_definition_kind
  | En of entity
  | Pa of parameter
  | Ar of argument
  | Dk of definition_kind
  | Di of dotted_ident
  | Lbli of label_ident
  | NoD of name_or_dynamic
  | Tk of tok
  | TodoK of todo_kind
```

F.3 Boilerplate generic vs generic

⟨*function* Generic_vs_generic.m_dotted_name 226b⟩≡ (428b)

```
  let m_dotted_name a b =
    match (a, b) with
    (* TODO: [$X] should match any list *)
    | a, b -> (m_list m_ident) a b
```

⟨*function* Generic_vs_generic.m_name 226c⟩≡ (428b)

```
  (* TODO: factorize metavariable and aliasing logic in m_expr, m_type, m_attr
   * here, and use a new MV.N instead of MV.Id
   *)
  let rec m_name a b =
    match (a, b) with
    | A.Id (a1, a2), B.Id (b1, b2) -> m_ident a1 b1 >>= fun () -> m_id_info a2 b2
    | A.IdQualified (a1, a2), B.IdQualified (b1, b2) ->
        m_name_ a1 b1 >>= fun () -> m_id_info a2 b2
    | A.Id _, _ | A.IdQualified _, _ -> fail ()

  and m_name_ a b =
    match (a, b) with
    | (a1, a2), (b1, b2) -> m_ident a1 b1 >>= fun () -> m_name_info a2 b2
```

⟨*signature* `Matching_generic.m_other_xxx` 227a⟩≡ (440)
```
val m_other_xxx : 'a -> 'a -> tin -> tout
```

⟨*function* `Matching_generic.m_other_xxx` 227b⟩≡ (442)
```
let m_other_xxx a b =
  match (a, b) with a, b when a =*= b -> return () | _ -> fail ()
```

⟨*function* `Generic_vs_generic.m_module_name` 227c⟩≡ (428b)
```
let m_module_name a b =
  match (a, b) with
  | A.FileName a1, B.FileName b1 -> (m_wrap m_string) a1 b1
  | A.DottedName a1, B.DottedName b1 -> m_dotted_name a1 b1
  | A.FileName _, _ | A.DottedName _, _ -> fail ()
```

⟨*function* `Generic_vs_generic.m_resolved_name_kind` 227d⟩≡ (428b)
```
let m_resolved_name_kind a b =
  match (a, b) with
  | A.Local, B.Local -> return ()
  | A.EnclosedVar, B.EnclosedVar -> return ()
  | A.Param, B.Param -> return ()
  | A.Global, B.Global -> return ()
  | A.ImportedEntity a1, B.ImportedEntity b1 -> m_dotted_name a1 b1
  | A.ImportedModule a1, B.ImportedModule b1 -> m_module_name a1 b1
  | A.Macro, B.Macro -> return ()
  | A.EnumConstant, B.EnumConstant -> return ()
  | A.TypeName, B.TypeName -> return ()
  | A.Local, _
  | A.Param, _
  | A.Global, _
  | A.EnclosedVar, _
  | A.Macro, _
  | A.EnumConstant, _
  | A.TypeName, _
  | A.ImportedEntity _, _
  | A.ImportedModule _, _ ->
      fail ()
```

⟨`Generic_vs_generic.m_expr()` *boilerplate cases* 227e⟩≡ (70b)
```
| A.Record a1, B.Record b1 -> (m_bracket m_fields) a1 b1
| A.Constructor (a1, a2), B.Constructor (b1, b2) ->
    m_dotted_name a1 b1 >>= fun () -> (m_list m_expr) a2 b2
| A.Lambda a1, B.Lambda b1 ->
    m_function_definition a1 b1 >>= fun () -> return ()
| A.AnonClass a1, B.AnonClass b1 -> m_class_definition a1 b1
| A.IdSpecial a1, B.IdSpecial b1 -> m_wrap m_special a1 b1
(* This is mainly for Go which generates an AssignOp (Eq)
 * for the x := a short variable declaration.
 * TODO: this should be a configurable equivalence: $X = $Y ==> $X := $Y.
 * Some people want it, some people may not want it.
 * At least we dont do the opposite (AssignOp matching Assign) so
 * using := in a pattern will not match code using just =
 * (but pattern using = will match both code using = or :=).
 *)
| A.Assign (a1, a2, a3), B.AssignOp (b1, (B.Eq, b2), b3) ->
    m_expr (A.Assign (a1, a2, a3)) (B.Assign (b1, b2, b3))
| A.AssignOp (a1, a2, a3), B.AssignOp (b1, b2, b3) ->
```

```
    m_expr a1 b1 >>= fun () ->
    m_wrap m_arithmetic_operator a2 b2 >>= fun () -> m_expr a3 b3
| A.Xml a1, B.Xml b1 -> m_xml a1 b1
| A.LetPattern (a1, a2), B.LetPattern (b1, b2) ->
    m_pattern a1 b1 >>= fun () -> m_expr a2 b2
| A.SliceAccess (a1, a2), B.SliceAccess (b1, b2) ->
    let f = m_option m_expr in
    m_expr a1 b1 >>= fun () -> m_bracket (m_tuple3 f f f) a2 b2
| A.Conditional (a1, a2, a3), B.Conditional (b1, b2, b3) ->
    m_expr a1 b1 >>= fun () ->
    m_expr a2 b2 >>= fun () -> m_expr a3 b3
| A.MatchPattern (a1, a2), B.MatchPattern (b1, b2) ->
    m_expr a1 b1 >>= fun () -> (m_list m_action) a2 b2
| A.Yield (a0, a1, a2), B.Yield (b0, b1, b2) ->
    m_tok a0 b0 >>= fun () ->
    m_option m_expr a1 b1 >>= fun () -> m_bool a2 b2
| A.Await (a0, a1), B.Await (b0, b1) -> m_tok a0 b0 >>= fun () -> m_expr a1 b1
| A.Cast (a1, a2), B.Cast (b1, b2) -> m_type_ a1 b1 >>= fun () -> m_expr a2 b2
| A.Seq a1, B.Seq b1 -> (m_list m_expr) a1 b1
| A.Ref (a0, a1), B.Ref (b0, b1) -> m_tok a0 b0 >>= fun () -> m_expr a1 b1
| A.DeRef (a0, a1), B.DeRef (b0, b1) -> m_tok a0 b0 >>= fun () -> m_expr a1 b1
| A.OtherExpr (a1, a2), B.OtherExpr (b1, b2) ->
    m_other_expr_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
| A.Container _, _
| A.Tuple _, _
| A.Record _, _
| A.Constructor _, _
| A.Lambda _, _
| A.AnonClass _, _
| A.N _, _
| A.IdSpecial _, _
| A.Call _, _
| A.Xml _, _
| A.Assign _, _
| A.AssignOp _, _
| A.LetPattern _, _
| A.DotAccess _, _
| A.ArrayAccess _, _
| A.SliceAccess _, _
| A.Conditional _, _
| A.MatchPattern _, _
| A.Yield _, _
| A.Await _, _
| A.Cast _, _
| A.Seq _, _
| A.Ref _, _
| A.DeRef _, _
| A.OtherExpr _, _
| A.TypedMetavar _, _
| A.DotAccessEllipsis _, _ ->
    fail ()
```

⟨Generic_vs_generic.m_argument() *boilerplate cases* 228⟩≡ (72b)

```
| A.ArgType a1, B.ArgType b1 -> m_type_ a1 b1
| A.ArgKwd (a1, a2), B.ArgKwd (b1, b2) ->
    m_ident a1 b1 >>= fun () -> m_expr a2 b2
| A.ArgOther (a1, a2), B.ArgOther (b1, b2) ->
    m_other_argument_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
| A.Arg _, _ | A.ArgKwd _, _ | A.ArgType _, _ | A.ArgOther _, _ -> fail ()
```

⟨*function* `Generic_vs_generic.m_other_expr_operator` 229a⟩≡ (428b)
```
and m_other_expr_operator = m_other_xxx
```

⟨*function* `Generic_vs_generic.m_other_argument_operator` 229b⟩≡ (428b)
```
and m_other_argument_operator = m_other_xxx
```

⟨`Generic_vs_generic.m_field_ident()` *boilerplate cases* 229c⟩≡ (72c)
```
| A.EN a, B.EN b -> m_name a b
| A.EDynamic a, B.EDynamic b -> m_expr a b
| A.EN _, _ | A.EDynamic _, _ -> fail ()
```

⟨*function* `Generic_vs_generic.m_label_ident` 229d⟩≡ (428b)
```
and m_label_ident a b =
  match (a, b) with
  | A.LNone, B.LNone -> return ()
  | A.LId a, B.LId b -> m_label a b
  | A.LInt a, B.LInt b -> m_wrap m_int a b
  | A.LDynamic a, B.LDynamic b -> m_expr a b
  | A.LNone, _ | A.LId _, _ | A.LInt _, _ | A.LDynamic _, _ -> fail ()
```

⟨*function* `Generic_vs_generic.m_action` 229e⟩≡ (428b)
```
and m_action a b =
  match (a, b) with
  | (a1, a2), (b1, b2) -> m_pattern a1 b1 >>= fun () -> m_expr a2 b2
```

⟨*function* `Generic_vs_generic.m_arithmetic_operator` 229f⟩≡ (428b)
```
and m_arithmetic_operator a b =
  match (a, b) with _ when a =*= b -> return () | _ -> fail ()
```

⟨*function* `Generic_vs_generic.m_special` 229g⟩≡ (428b)
```
and m_special a b =
  match (a, b) with
  | A.This, B.This -> return ()
  | A.Super, B.Super -> return ()
  | A.Self, B.Self -> return ()
  | A.Parent, B.Parent -> return ()
  | A.Eval, B.Eval -> return ()
  | A.Typeof, B.Typeof -> return ()
  | A.Instanceof, B.Instanceof -> return ()
  | A.Sizeof, B.Sizeof -> return ()
  | A.New, B.New -> return ()
  | A.Defined, B.Defined -> return ()
  | A.ConcatString a, B.ConcatString b -> m_concat_string_kind a b
  | A.InterpolatedElement, B.InterpolatedElement -> return ()
  | A.Spread, B.Spread -> return ()
  | A.HashSplat, B.HashSplat -> return ()
  | A.ForOf, B.ForOf -> return ()
  | A.Op a1, B.Op b1 -> m_arithmetic_operator a1 b1
  | A.EncodedString a1, B.EncodedString b1 -> m_string a1 b1
  | A.IncrDecr (a1, a2), B.IncrDecr (b1, b2) ->
      m_eq a1 b1 >>= fun () -> m_eq a2 b2
  | A.NextArrayIndex, B.NextArrayIndex -> return ()
```

```
    | A.This, _
    | A.Super, _
    | A.Self, _
    | A.Parent, _
    | A.Eval, _
    | A.Typeof, _
    | A.Instanceof, _
    | A.Sizeof, _
    | A.New, _
    | A.ConcatString _, _
    | A.Spread, _
    | A.Op _, _
    | A.IncrDecr _, _
    | A.EncodedString _, _
    | A.HashSplat, _
    | A.Defined, _
    | A.ForOf, _
    | A.NextArrayIndex, _
    | InterpolatedElement, _ ->
        fail ()
```

⟨*function* Generic_vs_generic.m_name_info 230a⟩≡ (428b)

```
  and m_name_info a b =
    match (a, b) with
    | ( { A.name_qualifier = a1; name_typeargs = a2 },
        { B.name_qualifier = b1; name_typeargs = b2 } ) ->
        (m_option m_qualifier) a1 b1 >>= fun () ->
        (m_option m_type_arguments) a2 b2
```

⟨*function* Generic_vs_generic.m_xml 230b⟩≡ (428b)

```
  and m_xml a b =
    match (a, b) with
    | ( { A.xml_kind = a1; xml_attrs = a2; xml_body = a3 },
        { B.xml_kind = b1; xml_attrs = b2; xml_body = b3 } ) ->
        m_xml_kind a1 b1 >>= fun () ->
        m_attrs a2 b2 >>= fun () -> m_bodies a3 b3
```

⟨*function* Generic_vs_generic.m_xml_attr 230c⟩≡ (428b)

```
  and m_xml_attr a b =
    match (a, b) with
    | A.XmlAttr (a1, at, a2), B.XmlAttr (b1, bt, b2) ->
        let* () = m_ident a1 b1 in
        let* () = m_tok at bt in
        m_xml_attr_value a2 b2
    | A.XmlAttrExpr a1, B.XmlAttrExpr b1 -> m_bracket m_expr a1 b1
    | A.XmlEllipsis a1, B.XmlEllipsis b1 -> m_tok a1 b1
    | A.XmlAttr _, _ | A.XmlAttrExpr _, _ | A.XmlEllipsis _, _ -> fail ()
```

⟨Generic_vs_generic.m_body *boilerplate cases* 230d⟩≡ (99b)

```
  | A.XmlExpr a1, B.XmlExpr b1 ->
      (* less: could allow ... to match also a None *)
      m_bracket (m_option m_expr) a1 b1
  | A.XmlXml a1, B.XmlXml b1 -> m_xml a1 b1
  | A.XmlText _, _ | A.XmlExpr _, _ | A.XmlXml _, _ -> fail ()
```

⟨Generic_vs_generic.m_type_ *boilerplate cases* 231a⟩≡ (75c)

```
  (* TODO: do via m_name *)
  | A.TyN (A.Id (a1, a2)), B.TyN (B.Id (b1, b2)) ->
      m_ident_and_id_info (a1, a2) (b1, b2)
  | A.TyN (A.IdQualified (a1, a2)), B.TyN (B.IdQualified (b1, b2)) ->
      let* () = m_name_ a1 b1 in
      m_id_info a2 b2
  | A.TyAny a1, B.TyAny b1 -> m_tok a1 b1
  | A.TyNameApply (a1, a2), B.TyNameApply (b1, b2) ->
      m_dotted_name a1 b1 >>= fun () -> m_type_arguments a2 b2
  | A.TyVar a1, B.TyVar b1 -> m_ident a1 b1
  | A.TyPointer (a0, a1), B.TyPointer (b0, b1) ->
      m_tok a0 b0 >>= fun () -> m_type_ a1 b1
  | A.TyRef (a0, a1), B.TyRef (b0, b1) ->
      m_tok a0 b0 >>= fun () -> m_type_ a1 b1
  | A.TyQuestion (a1, a2), B.TyQuestion (b1, b2) ->
      m_type_ a1 b1 >>= fun () -> m_tok a2 b2
  | A.TyRest (a1, a2), B.TyRest (b1, b2) ->
      m_tok a1 b1 >>= fun () -> m_type_ a2 b2
  | A.TyRecordAnon (a0, a1), B.TyRecordAnon (b0, b1) ->
      let* () = m_tok a0 b0 in
      m_bracket m_fields a1 b1
  | A.TyInterfaceAnon (a0, a1), B.TyInterfaceAnon (b0, b1) ->
      let* () = m_tok a0 b0 in
      m_bracket m_fields a1 b1
  | A.TyOr (a1, a2, a3), B.TyOr (b1, b2, b3) ->
      m_type_ a1 b1 >>= fun () ->
      m_tok a2 b2 >>= fun () -> m_type_ a3 b3
  | A.TyAnd (a1, a2, a3), B.TyAnd (b1, b2, b3) ->
      m_type_ a1 b1 >>= fun () ->
      m_tok a2 b2 >>= fun () -> m_type_ a3 b3
  | A.OtherType (a1, a2), B.OtherType (b1, b2) ->
      m_other_type_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
  | A.TyBuiltin _, _
  | A.TyFun _, _
  | A.TyNameApply _, _
  | A.TyVar _, _
  | A.TyArray _, _
  | A.TyPointer _, _
  | A.TyTuple _, _
  | A.TyQuestion _, _
  | A.TyRest _, _
  | A.TyN _, _
  | A.TyAny _, _
  | A.TyOr _, _
  | A.TyAnd _, _
  | A.TyRecordAnon _, _
  | A.TyInterfaceAnon _, _
  | A.TyRef _, _
  | A.OtherType _, _ ->
      fail ()
```

⟨*function* Generic_vs_generic.m_ident_and_type_ 231b⟩≡ (428b)

⟨*function* Generic_vs_generic.m_type_arguments 231c⟩≡ (428b)

```
  and m_type_arguments a b =
    match (a, b) with a, b -> (m_list m_type_argument) a b
```

⟨*function* Generic_vs_generic.m_type_argument 232a⟩≡ (428b)

```
and m_type_argument a b =
  match (a, b) with
  | A.TypeArg a1, B.TypeArg b1 -> m_type_ a1 b1
  | A.TypeWildcard (a1, a2), B.TypeWildcard (b1, b2) ->
      let* () = m_tok a1 b1 in
      m_option m_wildcard a2 b2
  | A.TypeLifetime a1, B.TypeLifetime b1 -> m_ident a1 b1
  | A.OtherTypeArg (a1, a2), B.OtherTypeArg (b1, b2) ->
      m_other_type_argument_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
  | A.TypeArg _, _
  | A.TypeWildcard _, _
  | A.TypeLifetime _, _
  | A.OtherTypeArg _, _ ->
      fail ()
```

⟨*function* Generic_vs_generic.m_other_type_operator 232b⟩≡ (428b)

```
and m_other_type_operator = m_other_xxx
```

⟨*function* Generic_vs_generic.m_other_type_argument_operator 232c⟩≡ (428b)

```
and m_other_type_argument_operator = m_other_xxx
```

⟨*function* Generic_vs_generic.m_other_attribute_operator 232d⟩≡ (428b)

```
and m_other_attribute_operator = m_other_xxx
```

⟨Generic_vs_generic.m_stmt *boilerplate cases* 232e⟩+≡ (73a) ◁73b

```
  | A.DoWhile (a0, a1, a2), B.DoWhile (b0, b1, b2) ->
      m_tok a0 b0 >>= fun () ->
      m_stmt a1 b1 >>= fun () -> m_expr a2 b2
  | A.For (a0, a1, a2), B.For (b0, b1, b2) ->
      m_tok a0 b0 >>= fun () ->
      m_for_header a1 b1 >>= fun () -> m_stmt a2 b2
  | A.Switch (at, a1, a2), B.Switch (bt, b1, b2) ->
      m_tok at bt >>= fun () ->
      m_option m_expr a1 b1 >>= fun () -> m_case_clauses a2 b2
  | A.Continue (a0, a1, asc), B.Continue (b0, b1, bsc) ->
      let* () = m_tok a0 b0 in
      let* () = m_label_ident a1 b1 in
      m_tok asc bsc
  | A.Break (a0, a1, asc), B.Break (b0, b1, bsc) ->
      let* () = m_tok a0 b0 in
      let* () = m_label_ident a1 b1 in
      m_tok asc bsc
  | A.Label (a1, a2), B.Label (b1, b2) ->
      m_label a1 b1 >>= fun () -> m_stmt a2 b2
  | A.Goto (a0, a1), B.Goto (b0, b1) -> m_tok a0 b0 >>= fun () -> m_label a1 b1
  | A.Throw (a0, a1, asc), B.Throw (b0, b1, bsc) ->
      let* () = m_tok a0 b0 in
      let* () = m_expr a1 b1 in
      m_tok asc bsc
  | A.Try (a0, a1, a2, a3), B.Try (b0, b1, b2, b3) ->
      let* () = m_tok a0 b0 in
      let* () = m_stmt a1 b1 in
      let* () = (m_list m_catch) a2 b2 in
      (m_option m_finally) a3 b3
  | A.Assert (a0, a1, a2, asc), B.Assert (b0, b1, b2, bsc) ->
```

```
      let* () = m_tok a0 b0 in
      let* () = m_expr a1 b1 in
      let* () = (m_option m_expr) a2 b2 in
      m_tok asc bsc
  | A.OtherStmt (a1, a2), B.OtherStmt (b1, b2) ->
      m_other_stmt_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
  | A.OtherStmtWithStmt (a1, a2, a3), B.OtherStmtWithStmt (b1, b2, b3) ->
      m_other_stmt_with_stmt_operator a1 b1 >>= fun () ->
      m_option m_expr a2 b2 >>= fun () -> m_stmt a3 b3
  | A.WithUsingResource (a1, a2, a3), B.WithUsingResource (b1, b2, b3) ->
      m_tok a1 b1 >>= fun () ->
      m_stmt a2 b2 >>= fun () -> m_stmt a3 b3
  | A.ExprStmt _, _
  | A.DefStmt _, _
  | A.DirectiveStmt _, _
  | A.Block _, _
  | A.If _, _
  | A.While _, _
  | A.DoWhile _, _
  | A.For _, _
  | A.Switch _, _
  | A.Return _, _
  | A.Continue _, _
  | A.Break _, _
  | A.Label _, _
  | A.Goto _, _
  | A.Throw _, _
  | A.Try _, _
  | A.Assert _, _
  | A.OtherStmt _, _
  | A.OtherStmtWithStmt _, _
  | A.WithUsingResource _, _ ->
      fail ()
```

⟨*function* `Generic_vs_generic.m_for_header` 233a⟩≡ (428b)

```
  and m_for_header a b =
    match (a, b) with
    (* dots: *)
    | A.ForEllipsis _, _ -> return ()
    | A.ForClassic (a1, a2, a3), B.ForClassic (b1, b2, b3) ->
        (m_list m_for_var_or_expr) a1 b1 >>= fun () ->
        m_option m_expr a2 b2 >>= fun () -> m_option m_expr a3 b3
    | A.ForEach (a1, at, a2), B.ForEach (b1, bt, b2) ->
        m_pattern a1 b1 >>= fun () ->
        m_tok at bt >>= fun () -> m_expr a2 b2
    | A.ForIn (a1, a2), B.ForIn (b1, b2) ->
        (m_list m_for_var_or_expr) a1 b1 >>= fun () -> m_list m_expr a2 b2
    | A.ForClassic _, _ | A.ForEach _, _ | A.ForIn _, _ -> fail ()
```

⟨*function* `Generic_vs_generic.m_for_var_or_expr` 233b⟩≡ (428b)

```
  and m_for_var_or_expr a b =
    match (a, b) with
    | A.ForInitVar (a1, a2), B.ForInitVar (b1, b2) ->
        m_entity a1 b1 >>= fun () -> m_variable_definition a2 b2
    | A.ForInitExpr a1, B.ForInitExpr b1 -> m_expr a1 b1
    | A.ForInitVar _, _ | A.ForInitExpr _, _ -> fail ()
```

⟨*function* Generic_vs_generic.m_label 234a⟩≡ (428b)
```
and m_label a b = match (a, b) with a, b -> m_ident a b
```

⟨*function* Generic_vs_generic.m_catch 234b⟩≡ (428b)
```
and m_catch a b =
  match (a, b) with
  | (at, a1, a2), (bt, b1, b2) ->
      m_tok at bt >>= fun () ->
      m_pattern a1 b1 >>= fun () -> m_stmt a2 b2
```

⟨*function* Generic_vs_generic.m_finally 234c⟩≡ (428b)
```
and m_finally a b =
  match (a, b) with (at, a), (bt, b) -> m_tok at bt >>= fun () -> m_stmt a b
```

⟨*function* Generic_vs_generic.m_case_and_body 234d⟩≡ (428b)
```
and m_case_and_body a b =
  match (a, b) with
  | CasesAndBody (a1, a2), CasesAndBody (b1, b2) ->
      (m_list m_case) a1 b1 >>= fun () -> m_stmt a2 b2
  | CaseEllipsis _, CasesAndBody _ -> return ()
  | CasesAndBody _, _ | CaseEllipsis _, _ -> fail ()
```

⟨*function* Generic_vs_generic.m_case 234e⟩≡ (428b)
```
and m_case a b =
  match (a, b) with
  | A.Case (a0, a1), B.Case (b0, b1) ->
      m_tok a0 b0 >>= fun () -> m_pattern a1 b1
  | A.CaseEqualExpr (a0, a1), B.CaseEqualExpr (b0, b1) ->
      m_tok a0 b0 >>= fun () -> m_expr a1 b1
  | A.Default a0, B.Default b0 -> m_tok a0 b0
  | A.Case _, _ | A.Default _, _ | A.CaseEqualExpr _, _ -> fail ()
```

⟨*function* Generic_vs_generic.m_other_stmt_operator 234f⟩≡ (428b)
```
and m_other_stmt_operator = m_other_xxx
```

⟨*function* Generic_vs_generic.m_other_stmt_with_stmt_operator 234g⟩≡ (428b)
```
and m_other_stmt_with_stmt_operator = m_other_xxx
```

⟨Generic_vs_generic.m_pattern *boilerplate cases* 234h⟩≡ (76c)
```
  | A.PatId (a1, a2), B.PatId (b1, b2) ->
      m_ident a1 b1 >>= fun () -> m_id_info a2 b2
  | A.PatLiteral a1, B.PatLiteral b1 -> m_literal a1 b1
  | A.PatType a1, B.PatType b1 -> m_type_ a1 b1
  | A.PatConstructor (a1, a2), B.PatConstructor (b1, b2) ->
      m_dotted_name a1 b1 >>= fun () -> (m_list m_pattern) a2 b2
  | A.PatTuple a1, B.PatTuple b1 -> m_bracket (m_list m_pattern) a1 b1
  | A.PatList a1, B.PatList b1 -> m_bracket (m_list m_pattern) a1 b1
  | A.PatRecord a1, B.PatRecord b1 -> m_bracket (m_list m_field_pattern) a1 b1
  | A.PatKeyVal (a1, a2), B.PatKeyVal (b1, b2) ->
      m_pattern a1 b1 >>= fun () -> m_pattern a2 b2
  | A.PatUnderscore a1, B.PatUnderscore b1 -> m_tok a1 b1
  | A.PatDisj (a1, a2), B.PatDisj (b1, b2) ->
      m_pattern a1 b1 >>= fun () -> m_pattern a2 b2
```

```
  | A.PatAs (a1, (a2, a3)), B.PatAs (b1, (b2, b3)) ->
      m_pattern a1 b1 >>= fun () -> m_ident_and_id_info (a2, a3) (b2, b3)
  | A.PatTyped (a1, a2), B.PatTyped (b1, b2) ->
      m_pattern a1 b1 >>= fun () -> m_type_ a2 b2
  | A.PatVar (a1, a2), B.PatVar (b1, b2) ->
      m_type_ a1 b1 >>= fun () -> m_option m_ident_and_id_info a2 b2
  | A.PatWhen (a1, a2), B.PatWhen (b1, b2) ->
      m_pattern a1 b1 >>= fun () -> m_expr a2 b2
  | A.OtherPat (a1, a2), B.OtherPat (b1, b2) ->
      m_other_pattern_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
  | A.PatId _, _
  | A.PatLiteral _, _
  | A.PatConstructor _, _
  | A.PatTuple _, _
  | A.PatList _, _
  | A.PatRecord _, _
  | A.PatKeyVal _, _
  | A.PatUnderscore _, _
  | A.PatDisj _, _
  | A.PatWhen _, _
  | A.PatAs _, _
  | A.PatTyped _, _
  | A.OtherPat _, _
  | A.PatType _, _
  | A.PatVar _, _ ->
      fail ()
```

⟨*function* Generic_vs_generic.m_field_pattern 235a⟩≡ (428b)

```
  and m_field_pattern a b =
    match (a, b) with
    | (a1, a2), (b1, b2) -> m_dotted_name a1 b1 >>= fun () -> m_pattern a2 b2
```

⟨*function* Generic_vs_generic.m_other_pattern_operator 235b⟩≡ (428b)

```
  and m_other_pattern_operator = m_other_xxx
```

⟨Generic_vs_generic.m_definition_kind *boilerplate cases* 235c⟩≡ (73d)

```
  | A.TypeDef a1, B.TypeDef b1 -> m_type_definition a1 b1
  | A.ModuleDef a1, B.ModuleDef b1 -> m_module_definition a1 b1
  | A.MacroDef a1, B.MacroDef b1 -> m_macro_definition a1 b1
  | A.Signature a1, B.Signature b1 -> m_type_ a1 b1
  | A.UseOuterDecl a1, B.UseOuterDecl b1 -> m_tok a1 b1
  | A.FuncDef _, _
  | A.VarDef _, _
  | A.ClassDef _, _
  | A.TypeDef _, _
  | A.ModuleDef _, _
  | A.MacroDef _, _
  | A.Signature _, _
  | A.UseOuterDecl _, _
  | A.FieldDefColon _, _
  | A.OtherDef _, _ ->
      fail ()
```

⟨*function* Generic_vs_generic.m_type_parameter_constraint 235d⟩≡ (428b)

```
  and m_type_parameter_constraint a b =
    match (a, b) with
```

```
    | A.Extends a1, B.Extends b1 -> m_type_ a1 b1
    | A.HasConstructor a1, B.HasConstructor b1 -> m_tok a1 b1
    | A.OtherTypeParam (a1, a2), B.OtherTypeParam (b1, b2) ->
        m_other_type_parameter_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
    | A.Extends _, _ | A.HasConstructor _, _ | A.OtherTypeParam _, _ -> fail ()
```

⟨*function* Generic_vs_generic.m_type_parameter_constraints 236a⟩≡ (428b)

```
  and m_type_parameter_constraints a b =
    match (a, b) with a, b -> (m_list m_type_parameter_constraint) a b
```

⟨*function* Generic_vs_generic.m_type_parameter 236b⟩≡ (428b)

```
  and m_type_parameter a b =
    match (a, b) with
    | (a1, a2), (b1, b2) ->
        m_ident a1 b1 >>= fun () -> m_type_parameter_constraints a2 b2
```

⟨Generic_vs_generic.m_parameter *boilerplate cases* 236c⟩≡ (74b)

```
  | A.ParamPattern a1, B.ParamPattern b1 -> m_pattern a1 b1
  | A.OtherParam (a1, a2), B.OtherParam (b1, b2) ->
      m_other_parameter_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
  | A.ParamEllipsis a1, B.ParamEllipsis b1 -> m_tok a1 b1
  | A.ParamClassic _, _
  | A.ParamPattern _, _
  | A.ParamRest _, _
  | A.ParamHashSplat _, _
  | A.ParamEllipsis _, _
  | A.OtherParam _, _ ->
      fail ()
```

⟨*function* Generic_vs_generic.m_other_parameter_operator 236d⟩≡ (428b)

```
  and m_other_parameter_operator = m_other_xxx
```

⟨Generic_vs_generic.m_field *boilerplate cases* 236e⟩≡ (75b)

```
  | A.FieldSpread (a0, a1), B.FieldSpread (b0, b1) ->
      m_tok a0 b0 >>= fun () -> m_expr a1 b1
  | A.FieldSpread _, _ | A.FieldStmt _, _ -> fail ()
```

⟨*function* Generic_vs_generic.m_type_definition 236f⟩≡ (428b)

```
  and m_type_definition a b =
    match (a, b) with
    | { A.tbody = a1 }, { B.tbody = b1 } -> m_type_definition_kind a1 b1
```

⟨*function* Generic_vs_generic.m_type_definition_kind 236g⟩≡ (428b)

```
  and m_type_definition_kind a b =
    match (a, b) with
    | A.OrType a1, B.OrType b1 -> (m_list m_or_type) a1 b1
    | A.AndType a1, B.AndType b1 -> m_bracket m_fields a1 b1
    | A.AliasType a1, B.AliasType b1 -> m_type_ a1 b1
    | A.NewType a1, B.NewType b1 -> m_type_ a1 b1
    | A.Exception (a1, a2), B.Exception (b1, b2) ->
        m_ident a1 b1 >>= fun () ->
        (* TODO: m_list__m_type_ ? *)
        (m_list m_type_) a2 b2
```

```
    | A.OtherTypeKind (a1, a2), B.OtherTypeKind (b1, b2) ->
        m_other_type_kind_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
    | A.OrType _, _
    | A.AndType _, _
    | A.AliasType _, _
    | A.Exception _, _
    | A.NewType _, _
    | A.OtherTypeKind _, _ ->
        fail ()
```

⟨*function* Generic_vs_generic.m_or_type 237a⟩≡ (428b)

```
  and m_or_type a b =
    match (a, b) with
    | A.OrConstructor (a1, a2), B.OrConstructor (b1, b2) ->
        m_ident a1 b1 >>= fun () ->
        (* TODO: m_list__m_type_ ? *)
        (m_list m_type_) a2 b2
    | A.OrEnum (a1, a2), B.OrEnum (b1, b2) ->
        m_ident a1 b1 >>= fun () -> m_option m_expr a2 b2
    | A.OrUnion (a1, a2), B.OrUnion (b1, b2) ->
        m_ident a1 b1 >>= fun () -> m_type_ a2 b2
    | A.OtherOr (a1, a2), B.OtherOr (b1, b2) ->
        m_other_or_type_element_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
    | A.OrConstructor _, _ | A.OrEnum _, _ | A.OrUnion _, _ | A.OtherOr _, _ ->
        fail ()
```

⟨*function* Generic_vs_generic.m_other_type_kind_operator 237b⟩≡ (428b)

```
  and m_other_type_kind_operator = m_other_xxx
```

⟨*function* Generic_vs_generic.m_other_or_type_element_operator 237c⟩≡ (428b)

```
  and m_other_or_type_element_operator = m_other_xxx
```

⟨*function* Generic_vs_generic.m_class_kind 237d⟩≡ (428b)

```
  and m_class_kind a b = m_wrap m_class_kind_bis a b

  and m_class_kind_bis a b =
    match (a, b) with
    | A.Class, B.Class
    | A.Interface, B.Interface
    | A.Trait, B.Trait
    | A.AtInterface, B.AtInterface
    | A.Object, B.Object
    | A.RecordClass, B.RecordClass ->
        return ()
    | A.Class, _
    | A.Interface, _
    | A.Trait, _
    | A.AtInterface, _
    | A.Object, _
    | A.RecordClass, _ ->
        fail ()
```

⟨*function* Generic_vs_generic.m_module_definition 237e⟩≡ (428b)

```
  and m_module_definition a b =
    match (a, b) with
    | { A.mbody = a1 }, { B.mbody = b1 } -> m_module_definition_kind a1 b1
```

⟨*function* Generic_vs_generic.m_module_definition_kind 238a⟩≡ (428b)
```
  and m_module_definition_kind a b =
    match (a, b) with
    | A.ModuleAlias a1, B.ModuleAlias b1 -> m_dotted_name a1 b1
    | A.ModuleStruct (a1, a2), B.ModuleStruct (b1, b2) ->
        (m_option m_dotted_name) a1 b1 >>= fun () -> (m_list m_item) a2 b2
    | A.OtherModule (a1, a2), B.OtherModule (b1, b2) ->
        m_other_module_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
    | A.ModuleAlias _, _ | A.ModuleStruct _, _ | A.OtherModule _, _ -> fail ()
```

⟨*function* Generic_vs_generic.m_other_module_operator 238b⟩≡ (428b)
```
  and m_other_module_operator = m_other_xxx
```

⟨*function* Generic_vs_generic.m_macro_definition 238c⟩≡ (428b)
```
  and m_macro_definition a b =
    match (a, b) with
    | ( { A.macroparams = a1; macrobody = a2 },
        { B.macroparams = b1; macrobody = b2 } ) ->
        (m_list m_ident) a1 b1 >>= fun () -> (m_list m_any) a2 b2
```

⟨Generic_vs_generic.m_directive_basic *boilerplate cases* 238d⟩≡ (76b)
```
  | A.Package (a0, a1), B.Package (b0, b1) ->
      m_tok a0 b0 >>= fun () -> m_dotted_name a1 b1
  | A.PackageEnd a1, B.PackageEnd b1 -> m_tok a1 b1
  | A.Pragma (a1, a2), B.Pragma (b1, b2) ->
      m_ident a1 b1 >>= fun () -> (m_list m_any) a2 b2
  | A.OtherDirective (a1, a2), B.OtherDirective (b1, b2) ->
      m_other_directive_operator a1 b1 >>= fun () -> (m_list m_any) a2 b2
  | A.ImportFrom _, _
  | A.ImportAs _, _
  | A.OtherDirective _, _
  | A.Pragma _, _
  | A.ImportAll _, _
  | A.Package _, _
  | A.PackageEnd _, _ ->
      fail ()
```

⟨*function* Generic_vs_generic.m_other_directive_operator 238e⟩≡ (428b)
```
  and m_other_directive_operator = m_other_xxx
```

⟨*function* Generic_vs_generic.m_program 238f⟩≡ (428b)
```
  and m_program a b = match (a, b) with a, b -> (m_list m_item) a b
```

⟨*function* Generic_vs_generic.m_item 238g⟩≡ (428b)
```
  and m_item a b = m_stmt a b
```

⟨*function* Generic_vs_generic.m_any 238h⟩≡ (428b)
```
  and m_any a b =
    match (a, b) with
    | A.Str a1, B.Str b1 -> m_wrap m_string_ellipsis_or_regexp_or_default a1 b1
    | A.Ss a1, B.Ss b1 -> m_stmts_deep ~less_is_ok:true a1 b1
    | A.E a1, B.E b1 -> m_expr a1 b1
    | A.S a1, B.S b1 -> m_stmt a1 b1
    ⟨Generic_vs_generic.m_any boilerplate cases 239a⟩
```

⟨Generic_vs_generic.m_any *boilerplate cases* 239a⟩≡ (238h)

```
  | A.Partial a1, B.Partial b1 -> m_partial a1 b1
  | A.Args a1, B.Args b1 -> m_list m_argument a1 b1
  (* boilerplate *)
  | A.Modn a1, B.Modn b1 -> m_module_name a1 b1
  | A.ModDk a1, B.ModDk b1 -> m_module_definition_kind a1 b1
  | A.Tk a1, B.Tk b1 -> m_tok a1 b1
  | A.TodoK a1, B.TodoK b1 -> m_ident a1 b1
  | A.Di a1, B.Di b1 -> m_dotted_name a1 b1
  | A.En a1, B.En b1 -> m_entity a1 b1
  | A.T a1, B.T b1 -> m_type_ a1 b1
  | A.P a1, B.P b1 -> m_pattern a1 b1
  | A.Def a1, B.Def b1 -> m_definition a1 b1
  | A.Dir a1, B.Dir b1 -> m_directive a1 b1
  | A.Fld a1, B.Fld b1 -> m_field a1 b1
  | A.Pa a1, B.Pa b1 -> m_parameter a1 b1
  | A.Ar a1, B.Ar b1 -> m_argument a1 b1
  | A.At a1, B.At b1 -> m_attribute a1 b1
  | A.Dk a1, B.Dk b1 -> m_definition_kind a1 b1
  | A.Pr a1, B.Pr b1 -> m_program a1 b1
  | A.I a1, B.I b1 -> m_ident a1 b1
  | A.Lbli a1, B.Lbli b1 -> m_label_ident a1 b1
  | A.NoD a1, B.NoD b1 -> m_name_or_dynamic a1 b1
  | A.I _, _
  | A.Modn _, _
  | A.Di _, _
  | A.En _, _
  | A.E _, _
  | A.S _, _
  | A.T _, _
  | A.P _, _
  | A.Def _, _
  | A.Dir _, _
  | A.Pa _, _
  | A.Ar _, _
  | A.At _, _
  | A.Dk _, _
  | A.Pr _, _
  | A.Fld _, _
  | A.Ss _, _
  | A.Tk _, _
  | A.Lbli _, _
  | A.NoD _, _
  | A.ModDk _, _
  | A.TodoK _, _
  | A.Partial _, _
  | A.Args _, _
  | A.Str _, _ ->
      fail ()
```

F.4 pfff/

⟨*function* Visitor_AST.visitor_in.kexpr 239b⟩≡

pfff/cli/Main.ml

⟨pfff/cli/Main_pfff.ml 240⟩≡

```
(*
 * Please imagine a long and boring GNU-style copyright notice
 * appearing just here.
 *)
open Common
module J = JSON

(*****************************************************************************)
(* Purpose *)
(*****************************************************************************)
(* A "driver" for the different parsers in pfff.
 *
 * Also useful to dump the CST or AST of a language (-dump_xxx).
 *
 * related:
 *  - https://astexplorer.net/, supports many languages, many parsers
*)

(*****************************************************************************)
(* Flags *)
(*****************************************************************************)

(* In addition to flags that can be tweaked via -xxx options (cf the
 * full list of options in the "the options" section below), this
 * program also depends on external files?
*)

⟨constant Main.verbose 38d⟩

⟨constant Main.lang 38b⟩

⟨constant Main.action 38g⟩

(*****************************************************************************)
(* Main action *)
(*****************************************************************************)

⟨function Main.main_action 38c⟩

(*****************************************************************************)
(* Extra Actions *)
(*****************************************************************************)

⟨function Main.test_json_pretty_printer 39c⟩

(* ---------------------------------------------------------------------- *)
⟨function Main.pfff_extra_actions 39b⟩

(*****************************************************************************)
(* The options *)
(*****************************************************************************)

⟨function Main.all_actions 39a⟩

⟨function Main.options 38e⟩

(*****************************************************************************)
```

```
(* Main entry point *)
(*****************************************************************************)
```

⟨*function* `Main.main` 37b⟩

```
(*****************************************************************************)
```
⟨*toplevel* `Main._1` 38a⟩

F.5 pfff/commons/

pfff/commons/Common.mli

⟨*type* `Common.smap` 241a⟩≡ (241b)
```
type 'a smap = 'a SMap.t
```

⟨`pfff/commons/Common.mli` 241b⟩≡
```
(* you should set this flag when you run code compiled by js_of_ocaml *)
val jsoo: bool ref
```

⟨*signature equality operators* 208l⟩

⟨*signature* `Common.TODOOPERATOR (pfff/commons/Common.mli)4` 208m⟩

⟨*signature* `Common.pr` 202a⟩
⟨*signature* `Common.pr2` 210c⟩

⟨*signature* `Common._already_printed` 210d⟩
⟨*signature* `Common.disable_pr2_once` 210e⟩
⟨*signature* `Common.pr2_once` 210f⟩

⟨*signature* `Common.pr2_gen` 211a⟩
⟨*signature* `Common.dump` 211b⟩

```
(* to be used in pipes as in foo() |> before_return (fun v -> pr2_gen v)*)
val before_return: ('a -> unit) -> 'a -> 'a
```

⟨*exception* `Common.Todo` 222a⟩
⟨*exception* `Common.Impossible` 222b⟩

⟨*exception* `Common.Multi_found` 208n⟩

⟨*signature* `Common.exn_to_s` 222c⟩

⟨*signature* `Common.i_to_s` 202f⟩
⟨*signature* `Common.s_to_i` 202g⟩

⟨*signature* `Common.null_string` 202e⟩

⟨*signature match operator* 202h⟩
⟨*signature matched functions* 202i⟩

⟨*signature* `Common.spf` 202b⟩

⟨*signature* `Common.join` 202c⟩
⟨*signature* `Common.split` 202d⟩

⟨*type* `Common.filename` 202j⟩
```
val pp_filename: Format.formatter -> filename -> unit
```

```
val equal_filename: filename -> filename -> bool
```
⟨*type* Common.dirname 203a⟩
⟨*type* Common.path 203b⟩

⟨*signature* Common.cat 203h⟩

⟨*signature* Common.write_file 203i⟩
⟨*signature* Common.read_file 203j⟩

⟨*signature* Common.with_open_outfile 203k⟩
⟨*signature* Common.with_open_infile 203l⟩

⟨*exception* Common.CmdError 203m⟩
⟨*signature* Common.command2 203n⟩
⟨*signature* Common.cmd_to_list 203o⟩
⟨*signature* Common.cmd_to_list_and_status 203p⟩

⟨*signature* Common.null 204a⟩
⟨*signature* Common.exclude 204b⟩
⟨*signature* Common.sort 204c⟩

```
val uniq_by: ('a -> 'a -> bool) -> 'a list -> 'a list
```

⟨*signature* Common.map_filter 204k⟩
⟨*signature* Common.find_opt 204l⟩
⟨*signature* Common.find_some 204m⟩
⟨*signature* Common.find_some_opt 204n⟩
⟨*signature* Common.filter_some 204o⟩

⟨*signature* Common.take 204d⟩
⟨*signature* Common.take_safe 204e⟩
⟨*signature* Common.drop 204f⟩
⟨*signature* Common.span 204g⟩

⟨*signature* Common.index_list 204h⟩
⟨*signature* Common.index_list_0 204i⟩
⟨*signature* Common.index_list_1 204j⟩

⟨*type* Common.assoc 205g⟩

⟨*signature* Common.sort_by_val_lowfirst 205h⟩
⟨*signature* Common.sort_by_val_highfirst 205i⟩

⟨*signature* Common.sort_by_key_lowfirst 205j⟩
⟨*signature* Common.sort_by_key_highfirst 205k⟩

⟨*signature* Common.group_by 205l⟩
⟨*signature* Common.group_assoc_bykey_eff 205m⟩
⟨*signature* Common.group_by_mapped_key 205n⟩
⟨*signature* Common.group_by_multi 205o⟩

⟨*type* Common.stack 205p⟩
⟨*signature* Common.push 205q⟩

⟨*signature* Common.hash_of_list 206a⟩
⟨*signature* Common.hash_to_list 206b⟩

⟨*type* Common.hashset 206c⟩
⟨*signature* Common.hashset_of_list 206d⟩
⟨*signature* Common.hashset_to_list 206e⟩

⟨*signature* Common.map_opt 204p⟩
⟨*signature* Common.opt 204q⟩
⟨*signature* Common.do_option 204r⟩
⟨*signature* Common.opt_to_list 204s⟩
⟨*signature* Common.TODOOPERATOR (pfff/commons/Common.mli)6 205a⟩
⟨*signature* Common.TODOOPERATOR (pfff/commons/Common.mli)7 205b⟩

⟨*type* Common.either 205c⟩
```
val pp_either: (Format.formatter -> 'a -> 'b) ->
  (Format.formatter -> 'c -> 'd) ->
  Format.formatter -> ('a, 'c) either -> unit
val equal_either:
  ('a -> 'a -> bool) ->
  ('b -> 'b -> bool) ->
  ('a, 'b) either -> ('a, 'b) either -> bool
```
⟨*type* Common.either3 205d⟩
```
val pp_either3: (Format.formatter -> 'a -> 'b) ->
  (Format.formatter -> 'c -> 'd) ->
  (Format.formatter -> 'e -> 'f) ->
  Format.formatter -> ('a, 'c, 'e) either3 -> unit
```
⟨*signature* Common.partition_either 205e⟩
⟨*signature* Common.partition_either3 205f⟩

⟨*type* Common.arg_spec_full 206h⟩
⟨*type* Common.cmdline_options 206i⟩

⟨*type* Common.options_with_title 206j⟩
⟨*type* Common.cmdline_sections 206k⟩

⟨*signature* Common.parse_options 206l⟩
⟨*signature* Common.usage 206m⟩

⟨*signature* Common.short_usage 206n⟩
⟨*signature* Common.long_usage 207a⟩

⟨*signature* Common.arg_align2 207b⟩
⟨*signature* Common.arg_parse2 207c⟩

⟨*type* Common.flag_spec 207d⟩
⟨*type* Common.action_spec 207e⟩
⟨*type* Common.action_func 207f⟩

⟨*type* Common.cmdline_actions 207g⟩
⟨*exception* Common.WrongNumberOfArguments 207h⟩

⟨*signature* Common.mk_action_0_arg 207i⟩
⟨*signature* Common.mk_action_1_arg 207j⟩
⟨*signature* Common.mk_action_2_arg 207k⟩
⟨*signature* Common.mk_action_3_arg 207l⟩
⟨*signature* Common.mk_action_4_arg 207m⟩

⟨*signature* Common.mk_action_n_arg 207n⟩

⟨*signature* Common.options_of_actions 207o⟩
⟨*signature* Common.do_action 208a⟩
⟨*signature* Common.action_list 208b⟩

⟨*signature* Common.debugger 210b⟩

⟨*signature* Common.unwind_protect 206f⟩
⟨*signature* Common.finalize 206g⟩

⟨*signature* Common.save_excursion 208h⟩

⟨*signature* Common.memoized 209a⟩

⟨*exception* Common.UnixExit 208i⟩

⟨*exception* Common.Timeout 208j⟩
⟨*signature* Common.timeout_function 208k⟩
```
val timeout_function_float :?verbose:bool -> float -> (unit -> 'a) -> 'a

val with_time: (unit -> 'a) -> 'a * float
```

⟨*type* Common.prof 221a⟩
⟨*signature* Common.profile 221b⟩
⟨*signature* Common.show_trace_profile 221c⟩

⟨*signature* Common._profile_table 221d⟩
⟨*signature* Common.profile_code 221e⟩
⟨*signature* Common.profile_diagnostic 221f⟩
⟨*signature* Common.profile_code_exclusif 221g⟩
⟨*signature* Common.profile_code_inside_exclusif_ok 221h⟩
⟨*signature* Common.report_if_take_time 221i⟩
⟨*signature* Common.profile_code2 221j⟩

⟨*signature* Common._temp_files_created 208d⟩
⟨*signature* Common.save_tmp_files 208e⟩
⟨*signature* Common.new_temp_file 208c⟩
⟨*signature* Common.erase_temp_files 208f⟩
⟨*signature* Common.erase_this_temp_file 208g⟩

⟨*signature* Common.fullpath 203c⟩

⟨*signature* Common.cache_computation 209b⟩

⟨*signature* Common.filename_without_leading_path 203d⟩
⟨*signature* Common.readable 203e⟩

⟨*signature* Common.follow_symlinks 203f⟩
⟨*signature* Common.files_of_dir_or_files_no_vcs_nofilter 203g⟩

⟨*signature* Common.main_boilerplate 209c⟩

```
(* type of maps from string to 'a *)
module SMap : Map.S with type key = String.t
```
⟨*type* Common.smap 241a⟩

pfff/commons/OCaml.mli

⟨*type* OCaml.t 244⟩≡ (246b)
```
  (* OCaml core type definitions (no objects, no modules) *)
  type t =
```

```
    | Unit
    | Bool | Float | Char | String | Int

    | Tuple of t list
    | Dict of (string * [`RW|`RO] * t) list (* aka record *)
    | Sum of (string * t list) list         (* aka variants *)

    | Var of string
    | Poly of string
    | Arrow of t * t

    | Apply of string * t

    (* special cases of Apply *)
    | Option of t
    | List of t

    | TTODO of string
```

⟨*signature* OCaml.add_new_type 245a⟩≡ (246b)
```
  val add_new_type: string -> t -> unit
```

⟨*signature* OCaml.get_type 245b⟩≡ (246b)
```
  val get_type: string -> t
```

⟨*signature* OCaml.xxx_ofv *functions* 245c⟩≡ (246b)
```
  val int_ofv:    v -> int
  val float_ofv:  v -> float
  val unit_ofv: v -> unit
  val string_ofv: v -> string
  val list_ofv: (v -> 'a) -> v -> 'a list
  val option_ofv: (v -> 'a) -> v -> 'a option
```

⟨*signature* OCaml.map_v 245d⟩≡ (246b)
```
  (* mapper/visitor *)
  val map_v:
    f:( k:(v -> v) -> v -> v) ->
    v ->
    v
```

⟨*signature* OCaml.map_of_xxx *functions* 245e⟩≡ (246b)
```
  (* other building blocks, used by code generated using ocamltarzan *)
  val map_of_unit: unit -> unit
  val map_of_bool: bool -> bool
  val map_of_int: int -> int
  val map_of_float: float -> float
  val map_of_char: char -> char
  val map_of_string: string -> string

  val map_of_ref: ('a -> 'b) -> 'a ref -> 'b ref
  val map_of_ref_do_nothing_share_ref: ('a -> 'a) -> 'a ref -> 'a ref
  val map_of_option: ('a -> 'b) -> 'a option -> 'b option
  val map_of_list: ('a -> 'b) -> 'a list -> 'b list
  val map_of_either:
    ('a -> 'b) -> ('c -> 'd) -> ('a, 'c) Common.either -> ('b, 'd) Common.either
  val map_of_either3:
    ('a -> 'b) -> ('c -> 'd) -> ('e -> 'f) ->
    ('a, 'c, 'e) Common.either3 -> ('b, 'd, 'f) Common.either3
  val map_of_all3: ('a -> 'b) -> ('c -> 'd) -> ('e -> 'f) -> 'a * 'c * 'e -> 'b * 'd * 'f
```

⟨*signature* OCaml.v_xxx *functions* 246a⟩≡ (246b)

```
(* pure visitor building blocks, used by code generated using ocamltarzan *)
val v_unit: unit -> unit
val v_bool: bool -> unit
val v_int: int -> unit
val v_float: float -> unit
val v_string: string -> unit
val v_option: ('a -> unit) -> 'a option -> unit
val v_list: ('a -> unit) -> 'a list -> unit
val v_ref_do_visit: ('a -> unit) -> 'a ref -> unit
val v_ref_do_not_visit: ('a -> unit) -> 'a ref -> unit
val v_either:
  ('a -> unit) -> ('b -> unit) ->
  ('a, 'b) Common.either -> unit
val v_either3:
  ('a -> unit) -> ('b -> unit) -> ('c -> unit) ->
  ('a, 'b, 'c) Common.either3 -> unit
```

⟨pfff/commons/OCaml.mli 246b⟩≡

```
(*
 * OCaml hacks to support reflection (works with ocamltarzan).
 *
 * See also sexp.ml, json.ml, and xml.ml for other "reflective" techniques.
 *)
```

⟨*type* OCaml.t 244⟩

⟨*signature* OCaml.add_new_type 245a⟩
⟨*signature* OCaml.get_type 245b⟩

⟨*type* OCaml.v 211c⟩

⟨*signature* OCaml.vof_xxx *functions* 211e⟩

⟨*signature* OCaml.xxx_ofv *functions* 245c⟩

⟨*signature* OCaml.string_of_v 211d⟩

```
(* sexp converters *)
(*
val sexp_of_t: t -> Sexp.t
val t_of_sexp: Sexp.t -> t
val sexp_of_v: v -> Sexp.t
val v_of_sexp: Sexp.t -> v
val string_sexp_of_t: t -> string
val t_of_string_sexp: string -> t
val string_sexp_of_v: v -> string
val v_of_string_sexp: string -> v
*)

(* json converters *)
(*
val v_of_json: Json_type.json_type -> v
val json_of_v: v -> Json_type.json_type
val save_json: Common.filename -> Json_type.json_type -> unit
val load_json: Common.filename -> Json_type.json_type
*)
```

⟨*signature* OCaml.map_v 245d⟩

⟨*signature* OCaml.map_of_xxx *functions* 245e⟩

⟨*signature* OCaml.v_xxx *functions* 246a⟩

pfff/commons/Dumper.mli

⟨pfff/commons/Dumper.mli 247a⟩≡
⟨*signature* Dumper.dump 210g⟩

pfff/commons/OUnit.mli

⟨*signature* OUnit.assert_bool 247b⟩≡ (248l)
```
(** Signals a failure when bool is false. The string identifies the
    failure.

    @raise Failure to signal a failure *)
val assert_bool : string -> bool -> unit
```

⟨*signature* OUnit.TODOOPERATOR 247c⟩≡ (248l)
```
(** Shorthand for assert_bool

    @raise Failure to signal a failure *)
val ( @? ) : string -> bool -> unit
```

⟨*signature* OUnit.assert_string 247d⟩≡ (248l)
```
(** Signals a failure when the string is non-empty. The string identifies the
    failure.

    @raise Failure to signal a failure *)
val assert_string : string -> unit
```

⟨*signature* OUnit.assert_raises 247e⟩≡ (248l)
```
(** Asserts if the expected exception was raised. When msg is set it can
    be used to identify the failure

    @raise Failure description *)
val assert_raises : ?msg:string -> exn -> (unit -> 'a) -> unit
```

⟨*signature* OUnit.skip_if 247f⟩≡ (248l)
```
(** [skip cond msg] If [cond] is true, skip the test for the reason explain in [msg].
  * For example [skip_if (Sys.os_type = "Win32") "Test a doesn't run on windows"].
  *)
val skip_if : bool -> string -> unit
```

⟨*signature* OUnit.todo 247g⟩≡ (248l)
```
(** The associated test is still to be done, for the reason given.
  *)
val todo : string -> unit
```

⟨*signature* OUnit.cmp_float 247h⟩≡ (248l)
```
(** Compare floats up to a given relative error. *)
val cmp_float : ?epsilon: float -> float -> float -> bool
```

⟨*signature* OUnit.bracket 247i⟩≡ (248l)
```
(** *)
val bracket : (unit -> 'a) -> ('a -> 'b) -> ('a -> 'c) -> unit -> 'c
```

⟨*signature* OUnit.TODOOPERATOR (pfff/commons/OUnit.mli) 248a⟩≡ (248l)
```
(** Create a TestLabel for a test *)
val (>:) : string -> test -> test
```

⟨*signature* OUnit.test_decorate 248b⟩≡ (248l)
```
(** [test_decorate g tst] Apply [g] to test function contains in [tst] tree. *)
val test_decorate : (test_fun -> test_fun) -> test -> test
```

⟨*signature* OUnit.test_filter 248c⟩≡ (248l)
```
(** [test_filter paths tst] Filter test based on their path string representation. *)
val test_filter : string list -> test -> test option
```

⟨*signature* OUnit.test_case_count 248d⟩≡ (248l)
```
(** Returns the number of available test cases *)
val test_case_count : test -> int
```

⟨*type* OUnit.node 248e⟩≡ (248l)
```
(** Types which represent the path of a test *)
type node = ListItem of int | Label of string
```

⟨*type* OUnit.path 248f⟩≡ (248l)
```
type path = node list (** The path to the test (in reverse order). *)
```

⟨*signature* OUnit.string_of_node 248g⟩≡ (248l)
```
(** Make a string from a node *)
val string_of_node : node -> string
```

⟨*signature* OUnit.string_of_path 248h⟩≡ (248l)
```
(** Make a string from a path. The path will be reversed before it is
    tranlated into a string *)
val string_of_path : path -> string
```

⟨*signature* OUnit.test_case_paths 248i⟩≡ (248l)
```
(** Returns a list with paths of the test *)
val test_case_paths : test -> path list
```

⟨*type* OUnit.test_event 248j⟩≡ (248l)
```
(** Events which occur during a test run *)
type test_event =
    EStart of path
  | EEnd of path
  | EResult of test_result
```

⟨*signature* OUnit.perform_test 248k⟩≡ (248l)
```
(** Perform the test, allows you to build your own test runner *)
val perform_test : (test_event -> 'a) -> test -> test_result list
```

⟨pfff/commons/OUnit.mli 248l⟩≡
```
(*****************************************************************************)
(* The OUnit library                                                         *)
(*                                                                           *)
(* Copyright (C) 2002, 2003, 2004, 2005, 2006, 2007, 2008                    *)
(* Maas-Maarten Zeeman.                                                      *)

(*
The package OUnit is copyright by Maas-Maarten Zeeman.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this document and the OUnit software ("the Software"), to
deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute,
```

```
sublicense, and/or sell copies of the Software, and to permit persons
to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

The Software is provided ''as is'', without warranty of any kind,
express or implied, including but not limited to the warranties of
merchantability, fitness for a particular purpose and noninfringement.
In no event shall Maas-Maarten Zeeman be liable for any claim, damages
or other liability, whether in an action of contract, tort or
otherwise, arising from, out of or in connection with the Software or
the use or other dealings in the software.
*)
(**************************************************************************)

(** The OUnit library can be used to implement unittests

    To uses this library link with
      [ocamlc OUnit.cmo]
    or
      [ocamlopt OUnit.cmx]

    @author Maas-Maarten Zeeman
*)

(** {5 Assertions}

    Assertions are the basic building blocks of unittests. *)
```

⟨*signature* OUnit.assert_failure 213b⟩

⟨*signature* OUnit.assert_bool 247b⟩

⟨*signature* OUnit.TODOOPERATOR 247c⟩

⟨*signature* OUnit.assert_string 247d⟩

⟨*signature* OUnit.assert_equal 213a⟩

⟨*signature* OUnit.assert_raises 247e⟩

```
(** {5 Skipping tests }

   In certain condition test can be written but there is no point running it, because they
   are not significant (missing OS features for example). In this case this is not a failure
   nor a success. Following function allow you to escape test, just as assertion but without
   the same error status.

   A test skipped is counted as success. A test todo is counted as failure.  *)
```

⟨*signature* OUnit.skip_if 247f⟩

⟨*signature* OUnit.todo 247g⟩

```
(** {5 Compare Functions} *)
```

⟨*signature* OUnit.cmp_float 247h⟩

```
(** {5 Bracket}

    A bracket is a functional implementation of the commonly used
    setUp and tearDown feature in unittests. It can be used like this:

    "MyTestCase" >:: (bracket test_set_up test_fun test_tear_down) *)
```

⟨*signature* OUnit.bracket 247i⟩

```
(** {5 Constructing Tests} *)
```

⟨*type* OUnit.test_fun 213d⟩

⟨*type* OUnit.test 213c⟩

⟨*signature* OUnit.TODOOPERATOR (pfff/commons/OUnit.mli) 248a⟩

⟨*signature* OUnit.TODOOPERATOR (pfff/commons/OUnit.mli)2 213e⟩

⟨*signature* OUnit.TODOOPERATOR (pfff/commons/OUnit.mli)3 213f⟩

```
(** Some shorthands which allows easy test construction.

   Examples:

   - ["test1" >: TestCase((fun _ -> ()))] =>
   [TestLabel("test2", TestCase((fun _ -> ())))]
   - ["test2" >:: (fun _ -> ())] =>
   [TestLabel("test2", TestCase((fun _ -> ())))]

   - ["test-suite" >::: ["test2" >:: (fun _ -> ());]] =>
   [TestLabel("test-suite", TestSuite([TestLabel("test2", TestCase((fun _ -> ()))))]))]
*)
```

⟨*signature* OUnit.test_decorate 248b⟩

⟨*signature* OUnit.test_filter 248c⟩

```
(** {5 Retrieve Information from Tests} *)
```

⟨*signature* OUnit.test_case_count 248d⟩

⟨*type* OUnit.node 248e⟩
⟨*type* OUnit.path 248f⟩

⟨*signature* OUnit.string_of_node 248g⟩

⟨*signature* OUnit.string_of_path 248h⟩

⟨*signature* OUnit.test_case_paths 248i⟩

```
(** {5 Performing Tests} *)
```

⟨*type* OUnit.test_result 214b⟩

⟨*type* OUnit.test_event 248j⟩

⟨*signature* OUnit.perform_test 248k⟩

⟨*signature* OUnit.run_test_tt 213g⟩

⟨*signature* OUnit.run_test_tt_main 214a⟩

F.6 pfff/h_program-lang/

pfff/h_program-lang/Parse_info.mli

⟨*signature* Parse_info.first_loc_of_file 251a⟩≡ (252k)
```
val first_loc_of_file: Common.filename -> token_location
```

⟨*signature* Parse_info.get_original_token_location 251b⟩≡ (252k)
```
val get_original_token_location: token_origin -> token_location
```

⟨*signature* Parse_info.compare_pos 251c⟩≡ (252k)
```
val compare_pos: t -> t -> int
```

⟨*type* Parse_info.parsing_stat 251d⟩≡ (252k)
```
type parsing_stat = {
  filename: Common.filename;
  total_line_count: int;
  mutable error_line_count: int;
  mutable have_timeout: bool;
  (* used only for cpp for now, to help diagnose problematic macros,
   * see print_recurring_problematic_tokens below.
  *)
  mutable commentized: int;
  mutable problematic_lines: (string list * int ) list;
}
```

⟨*signature* Parse_info.default_stat 251e⟩≡ (252k)
```
val default_stat: Common.filename -> parsing_stat
```

⟨*signature* Parse_info.print_parsing_stat_list 251f⟩≡ (252k)
```
val print_parsing_stat_list: ?verbose:bool -> parsing_stat list -> unit
```

⟨*signature* Parse_info.print_recurring_problematic_tokens 251g⟩≡ (252k)
```
val print_recurring_problematic_tokens: parsing_stat list -> unit
```

⟨*type* Parse_info.tokens_state 251h⟩≡ (252k)
```
(* lexer helpers *)
type 'tok tokens_state = {
  mutable rest:         'tok list;
  mutable current:      'tok;
  (* it's passed since last "checkpoint", not passed from the beginning *)
  mutable passed:       'tok list;
}
```

⟨*signature* Parse_info.mk_tokens_state 251i⟩≡ (252k)
```
val mk_tokens_state: 'tok list -> 'tok tokens_state
```

⟨*signature* Parse_info.tokinfo 251j⟩≡ (252k)
```
val tokinfo:
  Lexing.lexbuf -> t
```

⟨*signature* Parse_info.yyback 251k⟩≡ (252k)
```
val yyback: int -> Lexing.lexbuf -> unit
```

⟨*signature* Parse_info.tokinfo_str_pos 252a⟩≡ (252k)
```
(* can deprecate? *)
val tokinfo_str_pos:  string -> int -> t
```

⟨*signature* Parse_info.rewrap_str 252b⟩≡ (252k)
```
val rewrap_str: string -> t -> t
```

⟨*signature* Parse_info.tok_add_s 252c⟩≡ (252k)
```
val tok_add_s: string -> t -> t
```

⟨*signature* Parse_info.tokenize_all_and_adjust_pos 252d⟩≡ (252k)
```
(* to be used by the lexer *)
val tokenize_all_and_adjust_pos:
  ?unicode_hack:bool ->
  Common.filename ->
  (Lexing.lexbuf -> 'tok) (* tokenizer *) ->
  (('t -> 't) -> 'tok -> 'tok) (* token visitor *) ->
  ('tok -> bool) (* is_eof *) ->
  'tok list
```

⟨*signature* Parse_info.mk_lexer_for_yacc 252e⟩≡ (252k)
```
val mk_lexer_for_yacc: 'tok list -> ('tok -> bool) (* is_comment *) ->
  'tok tokens_state * (* token stream for error recovery *)
  (Lexing.lexbuf -> 'tok) * (* the lexer to pass to the ocamlyacc parser *)
  Lexing.lexbuf (* fake lexbuf needed by ocamlyacc API *)
```

⟨*signature* Parse_info.full_charpos_to_pos_large 252f⟩≡ (252k)
```
(* f(i) will contain the (line x col) of the i char position *)
val full_charpos_to_pos_large:
  Common.filename -> (int -> (int * int))
(* fill in the line and column field of token_location that were not set
 * during lexing because of limitations of ocamllex. *)
```

⟨*signature* Parse_info.complete_token_location_large 252g⟩≡ (252k)
```
(* fill in the line and column field of token_location that were not set
 * during lexing because of limitations of ocamllex. *)
val complete_token_location_large :
  Common.filename -> (int -> (int * int))  -> token_location -> token_location
```

⟨*signature* Parse_info.error_message 252h⟩≡ (252k)
```
val error_message : Common.filename -> (string * int) -> string
```

⟨*signature* Parse_info.error_message_info 252i⟩≡ (252k)
```
val error_message_info :  t -> string
```

⟨*signature* Parse_info.print_bad 252j⟩≡ (252k)
```
val print_bad: int -> int * int -> string array -> unit
```

⟨pfff/h_program-lang/Parse_info.mli 252k⟩≡

```
(*****************************************************************************)
(* Tokens *)
(*****************************************************************************)

(* ('token_location' < 'token_origin' < 'token_mutable') * token_kind *)

⟨type Parse_info.token_location 22a⟩
(* see also type filepos = { l: int; c: int; } in Common.mli *)

⟨type Parse_info.token_origin 199c⟩
```

```
(* to allow source to source transformation via token "annotations",
 * see the documentation for spatch.
*)
⟨type Parse_info.token_mutable 199d⟩

⟨type Parse_info.transformation 199e⟩

⟨type Parse_info.add 200a⟩

⟨type Parse_info.t 22b⟩
⟨type Parse_info.info_ 23a⟩

(* for ppx_deriving *)
val pp_full_token_info: bool ref
val pp : Format.formatter -> t -> unit
val pp_token_location: Format.formatter -> token_location -> unit
val equal_token_location: token_location -> token_location -> bool

(* mostly for the fuzzy AST builder *)
⟨type Parse_info.token_kind 123c⟩
⟨type Parse_info.esthet 123d⟩

(*****************************************************************************)
(* Errors during parsing *)
(*****************************************************************************)

(* note that those exceptions can be converted in Error_code.error with
 * Error_code.try_with_exn_to_error()
*)
⟨exception Parse_info.Lexical_error 54a⟩
⟨exception Parse_info.Parsing_error 54b⟩
⟨exception Parse_info.Ast_builder_error 54c⟩
⟨exception Parse_info.Other_error 54d⟩

⟨exception Parse_info.NoTokenLocation 200b⟩

⟨signature Parse_info.lexical_error 222d⟩

(*****************************************************************************)
(* Info accessors *)
(*****************************************************************************)

⟨signature Parse_info.fake_token_location 200c⟩
⟨signature Parse_info.fake_info 200d⟩
val abstract_info: t

val fake_bracket: 'a -> t * 'a * t
val unbracket: t * 'a * t -> 'a
val sc: t


val is_fake: t -> bool
⟨signature Parse_info.first_loc_of_file 251a⟩

⟨signature Parse_info.str_of_info 22d⟩
⟨signature Parse_info.line_of_info 22e⟩
⟨signature Parse_info.col_of_info 22f⟩
⟨signature Parse_info.pos_of_info 22g⟩
⟨signature Parse_info.file_of_info 22h⟩
```

⟨*signature* Parse_info.string_of_info 222e⟩

⟨*signature* Parse_info.is_origintok 200h⟩

⟨*signature* Parse_info.token_location_of_info 22c⟩
⟨*signature* Parse_info.get_original_token_location 251b⟩

⟨*signature* Parse_info.compare_pos 251c⟩
⟨*signature* Parse_info.min_max_ii_by_pos 200e⟩

```
(*****************************************************************************)
(* Parsing results *)
(*****************************************************************************)
```

⟨*type* Parse_info.parsing_stat 251d⟩
⟨*signature* Parse_info.default_stat 251e⟩
```
val bad_stat: Common.filename -> parsing_stat
val correct_stat: Common.filename -> parsing_stat
```
⟨*signature* Parse_info.print_parsing_stat_list 251f⟩
⟨*signature* Parse_info.print_recurring_problematic_tokens 251g⟩

```
val aggregate_stats: parsing_stat list -> int * int (* total * bad *)

val print_regression_information:
  ext:string -> Common2.path list -> Common2.score -> unit

(* a parser can also "return" an exception like Lexical_error,
 * or Parsing_error (unless Flag_parsing.error_recovery is true).
*)
type ('ast, 'toks) parsing_result = {
  ast: 'ast;
  (* Note that the token list contains usually also the comment-tokens *)
  tokens: 'toks list;
  stat: parsing_stat
}

(*****************************************************************************)
(* Lexer helpers *)
(*****************************************************************************)
```

⟨*type* Parse_info.tokens_state 251h⟩
⟨*signature* Parse_info.mk_tokens_state 251i⟩

⟨*signature* Parse_info.tokinfo 251j⟩
⟨*signature* Parse_info.yyback 251k⟩

⟨*signature* Parse_info.tokinfo_str_pos 252a⟩

⟨*signature* Parse_info.rewrap_str 252b⟩
⟨*signature* Parse_info.tok_add_s 252c⟩
```
(* used mainly by tree-sitter based parsers in semgrep *)
val combine_infos: t -> t list -> t
```

⟨*signature* Parse_info.tokenize_all_and_adjust_pos 252d⟩
⟨*signature* Parse_info.mk_lexer_for_yacc 252e⟩

```
(* can deprecate? just use tokenize_all_and_adjust_pos *)
```
⟨*signature* Parse_info.full_charpos_to_pos_large 252f⟩
⟨*signature* Parse_info.complete_token_location_large 252g⟩

⟨*signature* Parse_info.error_message 252h⟩
⟨*signature* Parse_info.error_message_info 252i⟩
⟨*signature* Parse_info.print_bad 252j⟩

pfff/h_program-lang/Error_code.mli

⟨*type* Error_code.annotation 255a⟩≡ (256b)
```
(* @xxx to acknowledge or explain false positives *)
type annotation =
  | AtScheck of string
```

⟨*signature* Error_code.error_loc 255b⟩≡ (256b)
```
val error_loc  : Parse_info.token_location -> error_kind -> unit
```

⟨*signature* Error_code.warning_loc 255c⟩≡ (256b)
```
val warning_loc: Parse_info.token_location -> error_kind -> unit
```

⟨*signature* Error_code.info_loc 255d⟩≡ (256b)
```
val info_loc: Parse_info.token_location -> error_kind -> unit
```

⟨*signature* Error_code.string_of_error_kind 255e⟩≡ (256b)
```
val string_of_error_kind: error_kind -> string
```

⟨*signature* Error_code.check_id_of_error_kind 255f⟩≡ (256b)
```
val check_id_of_error_kind: error_kind -> string
```

⟨*type* Error_code.rank 255g⟩≡ (256b)
```
type rank =
  | Never
  | OnlyStrict
  | Less
  | Ok
  | Important
  | ReallyImportant
```

⟨*signature* Error_code.score_of_rank 255h⟩≡ (256b)
```
val score_of_rank:  rank -> int
```

⟨*signature* Error_code.rank_of_error 255i⟩≡ (256b)
```
val rank_of_error:  error -> rank
```

⟨*signature* Error_code.score_of_error 255j⟩≡ (256b)
```
val score_of_error: error -> int
```

⟨*signature* Error_code.annotation_at 255k⟩≡ (256b)
```
val annotation_at:
  Parse_info.token_location -> annotation option
```

⟨*signature* Error_code.adjust_paths_relative_to_root 255l⟩≡ (256b)
```
(* convert parsing and other fatal exceptions in regular 'error'
 * added to g_errors
*)

val adjust_paths_relative_to_root:
  Common.path -> error list -> error list
```

⟨*type* Error_code.identifier_index 255m⟩≡ (256b)
```
(* to detect false positives (we use the Hashtbl.find_all property) *)
type identifier_index = (string, Parse_info.token_location) Hashtbl.t
```

⟨*signature* Error_code.adjust_errors 256a⟩≡ (256b)

```
(* have some approximations and Fps in graph_code_checker so filter them *)
val adjust_errors:
  error list -> error list
```

⟨pfff/h_program-lang/Error_code.mli 256b⟩≡

```
(* types *)
```

⟨*type* Error_code.error 222f⟩
⟨*type* Error_code.severity 222g⟩

⟨*type* Error_code.error_kind 222h⟩

⟨*type* Error_code.entity 223a⟩

```
(* internal: you should prefer to use the error function below *)
val mk_error_loc: Parse_info.token_location -> error_kind -> error
```

⟨*type* Error_code.annotation 255a⟩

```
(* main API *)
```

⟨*signature* Error_code.g_errors 223b⟩

⟨*signature* Error_code.error 223c⟩
⟨*signature* Error_code.warning 223d⟩
⟨*signature* Error_code.info 223e⟩

⟨*signature* Error_code.error_loc 255b⟩
⟨*signature* Error_code.warning_loc 255c⟩
⟨*signature* Error_code.info_loc 255d⟩

```
(* string-of *)
```

⟨*signature* Error_code.string_of_error 223f⟩
⟨*signature* Error_code.string_of_error_kind 255e⟩

⟨*signature* Error_code.check_id_of_error_kind 255f⟩

```
(* ranking *)
```

⟨*type* Error_code.rank 255g⟩

⟨*signature* Error_code.score_of_rank 255h⟩
⟨*signature* Error_code.rank_of_error 255i⟩
⟨*signature* Error_code.score_of_error 255j⟩

⟨*signature* Error_code.annotation_at 255k⟩

```
(* error adjustments *)
```

⟨*signature* Error_code.options 223i⟩

⟨*signature* Error_code.report_parse_errors 223g⟩
⟨*signature* Error_code.report_fatal_errors 223h⟩

⟨*signature* Error_code.filter_maybe_parse_and_fatal_errors 223j⟩
⟨*signature* Error_code.adjust_paths_relative_to_root 255l⟩

⟨*signature* Error_code.exn_to_error 224a⟩

⟨*signature* Error_code.try_with_exn_to_error 224b⟩

⟨*signature* Error_code.try_with_print_exn_and_reraise 224c⟩

⟨*type* Error_code.identifier_index 255m⟩

⟨*signature* Error_code.adjust_errors 256a⟩

```
(* helpers for unit testing code *)
```

⟨*signature* Error_code.expected_error_lines_of_files 214e⟩

⟨*signature* Error_code.compare_actual_to_expected 214f⟩

pfff/h_program-lang/R2c.mli

⟨pfff/h_program-lang/R2c.mli 257a⟩≡

⟨*signature* R2c.error_to_json 116a⟩

⟨*signature* R2c.string_of_errors 116b⟩

F.7 pfff/lang_GENERIC_base/

pfff/lang_GENERIC_base/Lang.mli

⟨*signature* Lang.lang_of_string_map 257b⟩≡ (257e)
```
val lang_of_string_map : (string, t) Hashtbl.t
```

⟨*signature* Lang.string_of_lang 257c⟩≡ (257e)
```
val string_of_lang : t -> string
```

⟨*signature* Lang.ext_of_lang 257d⟩≡ (257e)
```
val ext_of_lang : t -> string list
```

⟨pfff/lang_GENERIC/parsing/Lang.mli 257e⟩≡
⟨*type* Lang.t 23b⟩

```
(* from deriving eq *)
val pp : Format.formatter -> t -> unit

val show : t -> string

val equal : t -> t -> bool

(* accept any variants *)
val is_js : t -> bool

val is_python : t -> bool
```

⟨*signature* Lang.lang_of_string_map 257b⟩
⟨*signature* Lang.lang_of_string_opt 23c⟩

⟨*signature* `Lang.langs_of_filename` 23e⟩

⟨*signature* `Lang.files_of_dirs_or_files` 23d⟩

⟨*signature* `Lang.string_of_lang` 257c⟩
⟨*signature* `Lang.ext_of_lang` 257d⟩

pfff/lang_GENERIC_base/Lang.ml

⟨*constant* `Lang.list_of_lang` 258a⟩≡ (260c)
```
let list_of_lang =
  [
    ("py", Python);
    ("python", Python);
    ("python2", Python2);
    ("python3", Python3);
    ("js", Javascript);
    ("javascript", Javascript);
    ("json", JSON);
    ("ts", Typescript);
    ("typescript", Typescript);
    ("go", Go);
    ("golang", Go);
    ("c", C);
    ("cpp", Cplusplus);
    ("c++", Cplusplus);
    ("ml", OCaml);
    ("ocaml", OCaml);
    ("java", Java);
    ("ruby", Ruby);
    ("rb", Ruby);
    ("cs", Csharp);
    ("csharp", Csharp);
    ("c#", Csharp);
    ("php", PHP);
    ("kt", Kotlin);
    ("lua", Lua);
    ("rs", Rust);
    ("rust", Rust);
    ("r", R);
    ("yaml", Yaml);
  ]
```

⟨*constant* `Lang.lang_of_string_map` 258b⟩≡ (260c)
```
let lang_of_string_map = Common.hash_of_list list_of_lang
```

⟨*function* `Lang.lang_of_string_opt` 258c⟩≡ (260c)
```
let lang_of_string_opt x =
  Hashtbl.find_opt lang_of_string_map (String.lowercase_ascii x)
```

⟨*function* `Lang.langs_of_filename` 258d⟩≡ (260c)
```
let langs_of_filename filename =
  let typ = File_type.file_type_of_file filename in
  match typ with
  | FT.PL (FT.Web FT.Js) -> [ Javascript ] (* Add TypeScript too? *)
```

```
  | FT.PL (FT.Web FT.TypeScript) -> [ Typescript ]
  | FT.PL FT.Python -> [ Python; Python2; Python2 ]
  (* .h could also be Cpp at some point *)
  | FT.PL (FT.C "c") -> [ C ]
  | FT.PL (FT.C "h") -> [ C; Cplusplus ]
  | FT.PL (FT.Cplusplus _) -> [ Cplusplus ]
  | FT.PL (FT.ML _) -> [ OCaml ]
  | FT.PL FT.Java -> [ Java ]
  | FT.PL FT.Go -> [ Go ]
  | FT.Config FT.Json -> [ JSON ]
  | FT.Config FT.Yaml -> [ Yaml ]
  | FT.PL FT.Ruby -> [ Ruby ]
  | FT.PL FT.Csharp -> [ Csharp ]
  | FT.PL (FT.Web (FT.Php _)) -> [ PHP ]
  | FT.PL FT.Kotlin -> [ Kotlin ]
  | FT.PL FT.Lua -> [ Lua ]
  | FT.PL FT.Rust -> [ Rust ]
  | FT.PL FT.R -> [ R ]
  | _ -> []
```

⟨*function* Lang.string_of_lang 259a⟩≡ (260c)

```
let string_of_lang = function
  | Python -> "Python"
  | Python2 -> "Python2"
  | Python3 -> "Python3"
  | Javascript -> "Javascript"
  | Typescript -> "Typescript"
  | JSON -> "JSON"
  | Java -> "Java"
  | C -> "C"
  | Cplusplus -> "C++"
  | OCaml -> "OCaml"
  | Go -> "Golang"
  | Ruby -> "Ruby"
  | Csharp -> "C#"
  | PHP -> "PHP"
  | Kotlin -> "Kotlin"
  | Lua -> "Lua"
  | Rust -> "Rust"
  | R -> "R"
  | Yaml -> "Yaml"
```

⟨*function* Lang.ext_of_lang 259b⟩≡ (260c)

```
(* Manually pulled from file_type_of_file2 in file_type.ml *)
let ext_of_lang = function
  | Python | Python2 | Python3 -> [ "py"; "pyi" ]
  | Javascript -> [ "js"; "jsx" ]
  | Typescript -> [ "ts"; "tsx" ]
  | JSON -> [ "json" ]
  | Java -> [ "java" ]
  | C -> [ "c" ]
  | Cplusplus -> [ "cc"; "cpp" ]
  | OCaml -> [ "ml"; "mli" ] (* this is not parsed yet: "mly"; "mll" *)
  | Go -> [ "go" ]
  | Ruby -> [ "rb" ]
  | Csharp -> [ "cs" ]
  | PHP -> [ "php" ]
  | Kotlin -> [ "kt" ]
```

```
    | Lua -> [ "lua" ]
    | Rust -> [ "rs" ]
    | R -> [ "r"; "R" ]
    | Yaml -> [ "yaml"; "yml" ]
```

⟨*function* `Lang.find_source` 260a⟩≡ (260c)
```
  let find_source lang xs =
    Common.files_of_dir_or_files_no_vcs_nofilter xs
    |> List.filter (fun filename -> List.mem lang (langs_of_filename filename))
    |> Common.sort
```

⟨*function* `Lang.files_of_dirs_or_files` 260b⟩≡ (260c)
```
  (* this is used by sgrep, so it is probably better to keep the logic
   * simple and not perform any Skip_code filtering (bento already does that)
   *)
  let files_of_dirs_or_files lang xs =
    (* old: let xs = List.map Common.fullpath xs in
     * better to not transform in fullpath; does not interact
     * well with -exclude-dir and anyway this should be done in the caller
     * or not at all. Better just do one thing here.
     *)
    find_source lang xs
```

⟨`pfff/lang_GENERIC/parsing/Lang.ml` 260c⟩≡
```
  ⟨pad/r2c copyright 11⟩
  module FT = File_type

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

  (*****************************************************************************)
  (* Types *)
  (*****************************************************************************)

  (* coupling: if you add a language here, after fixing the compilation errors,
   * you probably still need to add also special code in list_of_lang and
   * langs_of_filename below.
   *)
  ⟨type Lang.t 23b⟩
  [@@ocamlformat "disable"]
  [@@deriving show, eq]

  let is_js = function Javascript | Typescript -> true | _ -> false

  let is_python = function Python | Python2 | Python3 -> true | _ -> false

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)

  ⟨constant Lang.list_of_lang 258a⟩

  ⟨constant Lang.lang_of_string_map 258b⟩

  ⟨function Lang.lang_of_string_opt 258c⟩
```

⟨*function* Lang.langs_of_filename 258d⟩

⟨*function* Lang.string_of_lang 259a⟩

⟨*function* Lang.ext_of_lang 259b⟩

⟨*function* Lang.find_source 260a⟩

⟨*function* Lang.files_of_dirs_or_files 260b⟩

pfff/lang_GENERIC_base/AST_generic.ml

⟨*constant* AST_generic.empty_var 261a⟩≡ (263c)
```
let empty_var = { vinit = None; vtype = None }
```

⟨*function* AST_generic.param_of_id 261b⟩≡ (263c)
```
let param_of_id id =
  {
    pname = Some id;
    pdefault = None;
    ptype = None;
    pattrs = [];
    pinfo = basic_id_info (Param, sid_TODO);
  }
```

⟨*function* AST_generic.param_of_type 261c⟩≡ (263c)
```
let param_of_type typ =
  {
    ptype = Some typ;
    pname = None;
    pdefault = None;
    pattrs = [];
    pinfo = empty_id_info ();
  }
```

⟨*function* AST_generic.basic_field 261d⟩≡ (263c)
```
let basic_field id vopt typeopt =
  let entity = basic_entity id [] in
  FieldStmt (s (DefStmt (entity, VarDef { vinit = vopt; vtype = typeopt })))
```

⟨*function* AST_generic.attr 261e⟩≡ (263c)
```
let attr kwd tok = KeywordAttr (kwd, tok)
```

⟨*function* AST_generic.arg 261f⟩≡ (263c)
```
let arg e = Arg e
```

⟨*function* AST_generic.expr_to_pattern 261g⟩≡ (274)
```
(* In Go a pattern can be a complex expressions. It is just
 * matched for equality with the thing it's matched against, so in that
 * case it should be a pattern like | _ when expr = x.
 * For Python you can actually have a PatDisj of exception classes.
 * coupling: see pattern_to_expr below
 *)
```

```
  let rec expr_to_pattern e =
    (* TODO: diconstruct e and generate the right pattern (PatLiteral, ...) *)
    match e with
    | N (Id (id, info)) -> PatId (id, info)
    | Tuple (t1, xs, t2) -> PatTuple (t1, xs |> List.map expr_to_pattern, t2)
    | L l -> PatLiteral l
    | Container (List, (t1, xs, t2)) ->
        PatList (t1, xs |> List.map expr_to_pattern, t2)
    (* Todo:  PatKeyVal *)
    | _ -> OtherPat (OP_Expr, [ E e ])
```

⟨*exception* AST_generic.NotAnExpr 262a⟩≡ (274)

```
  exception NotAnExpr
```

⟨*function* AST_generic.pattern_to_expr 262b⟩≡ (274)

```
  (* sgrep: this is to treat pattern metavars as expr metavars *)
  let rec pattern_to_expr p =
    match p with
    | PatId (id, info) -> N (Id (id, info))
    | PatTuple (t1, xs, t2) -> Tuple (t1, xs |> List.map pattern_to_expr, t2)
    | PatLiteral l -> L l
    | PatList (t1, xs, t2) ->
        Container (List, (t1, xs |> List.map pattern_to_expr, t2))
    | OtherPat (OP_Expr, [ E e ]) -> e
    | PatAs _ | PatVar _ -> raise NotAnExpr
    | _ -> raise NotAnExpr
```

⟨*function* AST_generic.expr_to_type 262c⟩≡ (274)

```
  let expr_to_type e =
    (* TODO: diconstruct e and generate the right type (TyBuiltin, ...) *)
    OtherType (OT_Expr, [ E e ])
```

⟨*function* AST_generic.opt_to_empty 262d⟩≡ (274)

⟨*function* AST_generic.opt_to_label_ident 262e⟩≡ (274)

```
  let opt_to_label_ident = function None -> LNone | Some id -> LId id
```

⟨*function* AST_generic.stmt1 262f⟩≡ (263c)

```
  let stmt1 xs =
    match xs with
    | [] -> s (Block (fake_bracket []))
    | [ st ] -> st
    | xs -> s (Block (fake_bracket xs))
```

⟨*function* AST_generic.is_boolean_operator 262g⟩≡ (274)

```
  (* used in abstract interpreter and type for PHP where we now reuse
   * 'AST_generic.arithmetic_operator' above *)
  (*
  let is_boolean_operator = function
    | Plus (* unary too *) | Minus (* unary too *)
    | Mult | Div | Mod
    | Pow | FloorDiv | MatMult (* Python *)
    | LSL | LSR | ASR (* L = logic, A = Arithmetic, SL = shift left *)
    | BitOr | BitXor | BitAnd | BitNot | BitClear (* unary *)
```

```
    | Range | Nullish | NotNullPostfix | Elvis | Length
    | RangeInclusive
      -> false
    | And | Or | Xor | Not
    | Eq     | NotEq
    | PhysEq | NotPhysEq
    | Lt | LtE | Gt | GtE
    | Cmp | Concat | Append
    | RegexpMatch | NotMatch
    | In | NotIn | Is | NotIs
      -> true
  *)
```

⟨*function* AST_generic.funcdef_to_lambda 263a⟩≡ (274)

```
  (* used in controlflow_build *)
  let funcdef_to_lambda (ent, def) resolved =
    let idinfo = { (empty_id_info ()) with id_resolved = ref resolved } in
    let name = name_or_dynamic_to_expr ent.name (Some idinfo) in
    let v = Lambda def in
    Assign (name, Parse_info.fake_info "=", v)
```

⟨*function* AST_generic.has_keyword_attr 263b⟩≡ (274)

```
  let has_keyword_attr kwd attrs =
    attrs
    |> List.exists (function KeywordAttr (kwd2, _) -> kwd =*= kwd2 | _ -> false)
```

⟨pfff/h_program-lang/AST_generic.ml 263c⟩≡

⟨*pad/r2c copyright* 11⟩

```
  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* A generic AST, to factorize similar analysis in different programming
   * languages (e.g., semgrep).
   *
   * Right now this generic AST is mostly the factorized union of:
   *  - Python, Ruby, Lua
   *  - Javascript, JSON, and Typescript
   *  - PHP
   *  - Java, CSharp
   *  - C (and some C++)
   *  - Go
   *  - OCaml
   *  - TODO next: Kotlin, Scala, Rust
   *
   * rational: In the end, programming languages have a lot in Common.
   * Even though most interesting analysis are probably better done on a
   * per-language basis, many useful analysis are trivial and require just an
   * AST and a visitor. One could duplicate those analysis for each language
   * or design an AST (this file) generic enough to factorize all those
   * analysis (e.g., unused entity). We want to remain
   * as precise as possible and not lose too much information while going
   * from the specific language AST to the generic AST. We also do not want
   * to be too generic as in ast_fuzzy.ml, where we have a very general
   * tree of nodes, but all the structure of the original AST is lost.
   *
   * The generic AST tries to be as close as possible to the original code but
   * not too close. When a programming language feature is really sugar or
```

```
* an alternative way to do a thing, we usually unsugar. Here are the
* simplifications done:
*  - we do not keep the comma tokens in arguments. More generally we
*    just keep the tokens to get the range right (see the discussions on
*    invariants below) and get rid of the other (e.g., we remove
*    parens around conditions in if). We keep the parens for 'Call'
*    because we want to get the right range for those so we need the
*    rightmost tokens in the AST.
*  - multiple var declarations in one declaration (e.g., int a,b; in C)
*    are expanded in multiple 'variable_definition'. Note that
*    tuple assignments (e.g., a,b=1,2) are not expanded in multiple assigns
*    because this is not always possible (e.g., a,b=foo()) and people may
*    want to explicitely match tuples assignments (we do some magic in
*    Generic_vs_generic though to let 'a=1' matches also 'a,b=1,2').
*  - multiple ways to define a function are converted all to a
*    'function_definition' (e.g., Javascript arrows are converted in that)
*  - we are more general and impose less restrictions on where certain
*    constructs can appear to simplify things.
*     * there is no special lhs/lvalue type (see IL.ml for that) and so
*       'Assign' takes a general 'expr' on its lhs.
*     * there is no special toplevel/item vs stmt. Certain programming
*       languages impose restrictions on where a function or directive can
*       appear (e.g., just at the toplevel), but we allow those constructs
*       at the stmt level.
*     * the Splat and HashSplat operator can usually appear just in arguments
*       or inside arrays or in struct definitions (a field) but we are more
*       general and put it at the 'expr' level.
*     * certain attributes are valid only for certain constructs but instead
*       we use one attribute type (no class_attribute vs func_attribute etc.)
*
* todo:
*  - add C++ (argh)
*  - add Scala (difficult)
*  - see ast_fuzzy.ml todos for ideas to use AST_generic for sgrep.
*
* related work:
*  - ast_fuzzy.ml (in this directory)
*  - github semantic
*    https://github.com/github/semantic
*  - UAST of babelfish
*    https://doc.bblf.sh/uast/uast-specification-v2.html
*  - Coverity common program representation?
*  - Semmle internal common representation?
*  - Infer SIL (for C++, Java, Objective-C)
*  - Dawson Engler and Fraser Brown micro-checkers for multiple languages
*  - Lightweight Multi-language syntax transformation paper, but does not
*    really operate on an AST
*  - https://tabnine.com/ which supports multiple languages, but probably
*    again does not operate on an AST
*  - srcML https://www.srcml.org/doc/srcMLGrammar.html
*    but just for C/C++/C#/Java and seems pretty heavy
*
* design choices to have a generic data structure:
*  - add some 'a, 'b, 'c around expr/stmt/...
*  - data-type a la carte like in github-semantic but IMHO too high-level
*    with astronaut-style architecture (too abstract, too advanced features).
*  - CURRENT SOLUTION: the OtherXxx strategy used in this file (simple)
*  - functorize and add some type hole (type tstmt; type texpr; ...),
*    todo? not a bad idea if later we want to add type information on each
*    expression nodes
```

```
 *
 * history:
 *  - started with crossproduct of Javascript, Python, PHP, Java, and C
 *    (and a bit of OCaml) after wanting to port checked_return from Js to
 *    Python and got the idea to factorize things
 *
 * INVARIANTS:
 *  - all the other_xxx types should contain only simple constructors (enums)
 *    without any parameter. I rely on that to simplify the code
 *    of the generic mapper and matcher.
 *    Same for keyword_attributes.
 *  - each expression or statement must have at least one token in it
 *    so that semgrep can track a location (e.g., 'Return of expr option'
 *    is not enough because with no expr, there is no location information
 *    for this return, so it must be 'Return of tok * expr option' instead)
 *  - each expression or statement should ideally have enough tokens in it
 *    to get its range, so at least the leftmost and rightmost token in
 *    all constructs, so the Return above should even be
 *    'Return of tok * expr option * tok' for the ending semicolon
 *    (alt: have the range info in each expr/stmt/pattern but big refactoring)
 *  - to correctly compute a CFG (Control Flow Graph), the stmt type
 *    should list all constructs that contains other statements and
 *    try to avoid to use the very generic 'OtherXxx of any'
 *  - to correctly compute a DFG (Data Flow Graph), and to correctly resolve
 *    names (see Naming_AST.ml), each constructs that introduce a new
 *    variable should have a relevant comment 'newvar:'
 *  - to correctly resolve names, each construct that introduces a new scope
 *    should have a relevant comment 'newscope:'
 *  - todo? each language should add the VarDefs that defines the locals
 *    used in a function (instead of having the first Assign play the role
 *    of a VarDef, as done in Python for example).
 *
 * See also pfff/lang_GENERIC/
 *)

(* Provide hash_* and hash_fold_* for the core ocaml types *)
open Ppx_hash_lib.Std.Hash.Builtin

(* ppx_hash refuses to hash mutable fields but we do it anyway. *)
let hash_fold_ref hash_fold_x acc x = hash_fold_x acc !x

(*****************************************************************************)
(* Token (leaf) *)
(*****************************************************************************)
⟨type AST_generic.tok 23f⟩

(* with tarzan *)

(* sgrep: we do not care about position when comparing for equality 2 ASTs.
 * related: Lib_AST.abstract_position_info_any and then use OCaml generic '='.
 *)
let equal_tok _t1 _t2 = true

let hash_tok _t = 0

let hash_fold_tok acc _t = acc

⟨type AST_generic.wrap 24a⟩
[@@deriving show, eq, hash]
```

```
(* with tarzan *)

⟨type AST_generic.bracket 25d⟩
[@@deriving show, eq, hash]

(* with tarzan *)

(* semicolon, a FakeTok in languages that do not require them (e.g., Python).
 * alt: tok option.
 * See the sc value also at the end of this file to build an sc.
 *)
type sc = tok [@@deriving show, eq, hash]

(* with tarzan *)

(* an AST element not yet handled; works with the Xx_Todo and Todo in any *)
type todo_kind = string wrap [@@deriving show, eq, hash]

(* with tarzan *)

(*****************************************************************************)
(* Names *)
(*****************************************************************************)

⟨type AST_generic.ident 24b⟩
[@@deriving show, eq, hash]

⟨type AST_generic.dotted_ident 32c⟩
[@@deriving show, eq, hash]

(* with tarzan *)

⟨type AST_generic.module_name 32b⟩
[@@deriving show { with_path = false }, eq, hash]

(* with tarzan *)

(* A single unique id: sid (uid would be a better name, but it usually
 * means "user id" for people).
 *
 * This single id simplifies further analysis which need less to care about
 * maintaining scoping information, for example to deal with variable
 * shadowing, or functions using the same parameter names
 * (even though you still need to handle specially recursive functions), etc.
 *
 * See Naming_AST.ml for more information.
 *
 * Most generic ASTs have a fake value (sid_TODO = -1) at first.
 * You need to call Naming_AST.resolve (or one of the lang-specific
 * Resolve_xxx.resolve) on the generic AST to set it correctly.
 *)
⟨type AST_generic.sid 174b⟩

⟨type AST_generic.resolved_name 173e⟩
⟨type AST_generic.resolved_name_kind 174a⟩
[@@deriving show { with_path = false }, eq, hash]

(* with tarzan *)

(* Start of big mutually recursive types because of the use of 'any'
```

```
 * in OtherXxx *)

⟨type AST_generic.name 142a⟩
⟨type AST_generic.name_info 142b⟩
⟨type AST_generic.qualifier 142c⟩

(* This is used to represent field names, where sometimes the name
 * can be a dynamic expression, or more recently also to
 * represent entities like in Ruby where a class name can be dynamic.
 *)
and name_or_dynamic =
  (* In the case of a field, it may be hard to resolve the id_info inside name.
   * For example, a method id can refer to many method definitions.
   * But for certain things, like a private field, we can resolve it
   * (right now we use an EnclosedVar for those fields).
   *
   * The IdQualified inside name is
   * Useful for OCaml field access, but also for Ruby class entity name.
   *)
  | EN of name
  (* for PHP/JS fields (even though JS use ArrayAccess for that), or Ruby *)
  | EDynamic of expr

(*****************************************************************************)
(* Naming/typing *)
(*****************************************************************************)

⟨type AST_generic.id_info 173b⟩

(*****************************************************************************)
(* Expression *)
(*****************************************************************************)

⟨type AST_generic.expr 24d⟩

⟨type AST_generic.literal 25a⟩

(* The type of an unknown constant. *)
and const_type = Cbool | Cint | Cstr | Cany

(* set by the constant propagation algorithm and used in semgrep *)
and constness = Lit of literal | Cst of const_type | NotCst

⟨type AST_generic.container_operator 25b⟩

(* It's useful to keep track in the AST of all those special identifiers.
 * They need to be handled in a special way by certain analysis and just
 * using Name for them would be error-prone.
 * Note though that by putting all of them together in a type, we lose
 * typing information, for example Eval takes only one argument and
 * InstanceOf takes a type and an expr. This is a tradeoff to also not
 * polluate too much expr with too many constructs.
 *)
⟨type AST_generic.special 144b⟩

(* mostly binary operators.
 * less: could be divided in really Arith vs Logical (bool) operators,
 * but see is_boolean_operator() helper below.
 * Note that Mod can be used for %style string formatting in Python.
 * Note that Plus can also be used for string concatenations in Go/??.
```

```
 * todo? use a Special operator intead for that? but need type info?
 *)
⟨type AST_generic.arithmetic_operator 145⟩
⟨type AST_generic.incr_decr 146a⟩
⟨type AST_generic.prefix_postfix 146b⟩
and concat_string_kind =
  (* many languages do not require a special syntax to use interpolated
   * strings e.g. simply "this is {a}". Javascript uses backquotes.
   *)
  | InterpolatedConcat (* Javascript/PHP/Ruby/Perl *)
  (* many languages have a binary Concat operator to concatenate strings,
   * but some languages also allow the simple juxtaposition of multiple
   * strings to be concatenated, e.g. "hello" "world" in Python.
   *)
  | SequenceConcat (* Python/C *)
  (* Python requires the special f"" syntax to use interpolated strings,
   * and some semgrep users may want to explicitely match only f-strings,
   * which is why we record this information here.
   *)
  | FString

(* Python *)

⟨type AST_generic.field_ident 26c⟩

(* newscope: newvar: *)
⟨type AST_generic.action 146d⟩

(* less: could make it more generic by adding a 'expr so it could be
 * reused in ast_js.ml, ast_php.ml
 *)
⟨type AST_generic.xml 146e⟩
and xml_kind =
  | XmlClassic of tok (*'<'*) * ident * tok (*'>'*) * tok (*'</foo>'*)
  | XmlSingleton of tok (*'<'*) * ident * tok (* '/>', with xml_body = [] *)
  (* React/JS specific *)
  | XmlFragment of tok (* '<>' *) * tok

(* '</>', with xml_attrs = [] *)

⟨type AST_generic.xml_attribute 146f⟩
(* either a String or a bracketed expr, but right now we just use expr *)
and xml_attr_value = expr

⟨type AST_generic.xml_body 146h⟩

⟨type AST_generic.arguments 25f⟩
⟨type AST_generic.argument 26a⟩

⟨type AST_generic.other_argument_operator 147a⟩

(* todo: reduce, or move in other_special? *)
⟨type AST_generic.other_expr_operator 147c⟩

(*****************************************************************************)
(* Statement *)
(*****************************************************************************)
⟨type AST_generic.stmt 26e⟩

(* newscope: *)
```

```
(* less: could merge even more with pattern
 * list = PatDisj and Default = PatUnderscore,
 * so case_and_body of Switch <=> action of MatchPattern
 *)
⟨type AST_generic.case_and_body 148b⟩
⟨type AST_generic.case 148c⟩

(* newvar: newscope: usually a PatVar *)
⟨type AST_generic.catch 148d⟩
⟨type AST_generic.finally 148e⟩

⟨type AST_generic.label 149a⟩
⟨type AST_generic.label_ident 149b⟩

⟨type AST_generic.for_header 149c⟩

⟨type AST_generic.for_var_or_expr 149d⟩

⟨type AST_generic.other_stmt_with_stmt_operator 149e⟩

⟨type AST_generic.other_stmt_operator 150a⟩

(*****************************************************************************)
(* Pattern *)
(*****************************************************************************)
(* This is quite similar to expr. A few constructs in expr have
 * equivalent here prefixed with Pat (e.g., PaLiteral, PatId). We could
 * maybe factorize with expr, and this may help sgrep, but I think it's
 * cleaner to have a separate type because the scoping rules for a pattern and
 * an expr are quite different and not any expr is allowed here.
 *)
⟨type AST_generic.pattern 152a⟩

⟨type AST_generic.other_pattern_operator 152b⟩

(*****************************************************************************)
(* Type *)
(*****************************************************************************)

⟨type AST_generic.type_ 30c⟩

⟨type AST_generic.type_arguments 151b⟩

⟨type AST_generic.type_argument 151c⟩
⟨type AST_generic.other_type_argument_operator 151d⟩

⟨type AST_generic.other_type_operator 151e⟩
and other_type_argument_operator =
  (* Rust *)
  | OTA_Literal
  | OTA_ConstBlock
  (* Other *)
  | OTA_Todo

(*****************************************************************************)
(* Attribute *)
(*****************************************************************************)
(* a.k.a decorators, annotations *)
⟨type AST_generic.attribute 30f⟩
```

⟨*type* AST_generic.keyword_attribute 31a⟩

⟨*type* AST_generic.other_attribute_operator 155a⟩

```
(*****************************************************************************)
(* Definitions *)
(*****************************************************************************)
(* definition (or just declaration sometimes) *)
```
⟨*type* AST_generic.definition 27a⟩

```
(* old: type_: type_ option; but redundant with the type information in
 * the different definition_kind, as well as in id_info, and does not
 * have any meanings for certain defs (e.g., ClassDef) so not worth
 * factoring.
 * update: this could be useful in OCaml when you explicitely type an
 * entity (as in 'let (f: int -> int) = fun i -> i + 1), but we
 * currently abuse id_info.id_type for that.
 *
 * see special_multivardef_pattern below for many vardefs in one entity in
 * ident.
 * less: could be renamed entity_def, and name is a kind of entity_use.
 *)
```
⟨*type* AST_generic.entity 27b⟩

⟨*type* AST_generic.definition_kind 27e⟩

```
(* template/generics/polymorphic-type *)
```
⟨*type* AST_generic.type_parameter 153a⟩
⟨*type* AST_generic.type_parameter_constraints 153b⟩
⟨*type* AST_generic.type_parameter_constraint 153c⟩
```
and other_type_parameter_operator =
  (* Rust *)
  | OTP_Lifetime
  | OTP_Ident
  | OTP_Constrained
  | OTP_Const
  (* Other *)
  | OTP_Todo

(* ------------------------------------------------------------------------ *)
(* Function (or method) definition *)
(* ------------------------------------------------------------------------ *)
(* less: could be merged with variable_definition *)
```
⟨*type* AST_generic.function_definition 28a⟩
```
(* We don't really care about the function_kind in semgrep, but who
 * knows maybe one day we will. We care about the token in the
 * function_kind wrap in fkind though for semgrep for accurate range.
 *)
and function_kind =
  | Function
  (* This is a bit redundant with having the func in a field *)
  | Method
  (* Also redundant; can just check if the fdef is in a Lambda *)
  | LambdaKind
  | Arrow

(* a.k.a short lambdas *)
```
⟨*type* AST_generic.parameters 28b⟩
⟨*type* AST_generic.parameter 28c⟩

```
(* less: could be merged with variable_definition, or pattern
 * less: could factorize pname/pattrs/pinfo with entity
 *)
⟨type AST_generic.parameter_classic 29a⟩
⟨type AST_generic.other_parameter_operator 153e⟩

(* ------------------------------------------------------------------------ *)
(* Variable definition *)
(* ------------------------------------------------------------------------ *)
(* Also used for constant_definition with attrs = [Const].
 * Also used for field definition in a class (and record).
 * less: could use for function_definition with vinit = Some (Lambda (...))
 *  but maybe useful to explicitely makes the difference for now?
 *)
⟨type AST_generic.variable_definition 29c⟩

(* ------------------------------------------------------------------------ *)
(* Type definition *)
(* ------------------------------------------------------------------------ *)
⟨type AST_generic.type_definition 153f⟩

⟨type AST_generic.type_definition_kind 153g⟩

⟨type AST_generic.or_type_element 154a⟩

⟨type AST_generic.other_or_type_element_operator 154b⟩

(* ------------------------------------------------------------------------ *)
(* Object/struct/record/class field definition *)
(* ------------------------------------------------------------------------ *)

(* Field definition and use, for classes, objects, and records.
 * note: I don't call it field_definition because it's used both to
 * define the shape of a field (a definition), and when creating
 * an actual field (a value).
 *
 * old: there used to be a FieldVar and FieldMethod similar to
 * VarDef and FuncDef but they are now converted into a FieldStmt(DefStmt).
 * This simplifies semgrep so that a function pattern can match
 * toplevel functions, nested functions, and methods.
 * Note that for FieldVar we sometimes converts it to a FieldDefColon
 * (which is very similar to a VarDef) because some people don't want a VarDef
 * to match a field definition in certain languages (e.g., Javascript) where
 * the variable declaration and field definition have a different syntax.
 * Note: the FieldStmt(DefStmt(FuncDef(...))) can have empty body
 * for interface methods.
 *
 * Note that not all stmt in FieldStmt are definitions. You can have also
 * a Block like in Kotlin for 'init' stmts.
 *)
⟨type AST_generic.field 30b⟩

⟨type AST_generic.other_type_kind_operator 154d⟩

(* ------------------------------------------------------------------------ *)
(* Class definition *)
(* ------------------------------------------------------------------------ *)
(* less: could be a special kind of type_definition *)
⟨type AST_generic.class_definition 29d⟩
```

⟨*type* AST_generic.class_kind 30a⟩

```
(* ------------------------------------------------------------------------ *)
(* Module definition  *)
(* ------------------------------------------------------------------------ *)
```
⟨*type* AST_generic.module_definition 154e⟩

⟨*type* AST_generic.module_definition_kind 154f⟩

⟨*type* AST_generic.other_module_operator 154g⟩

```
(* ------------------------------------------------------------------------ *)
(* Macro definition *)
(* ------------------------------------------------------------------------ *)
(* Used by cpp in C/C++ *)
```
⟨*type* AST_generic.macro_definition 154h⟩

```
(*****************************************************************************)
(* Directives (Module import/export, package) *)
(*****************************************************************************)
```
⟨*type* AST_generic.directive 31b⟩

⟨*type* AST_generic.alias 32d⟩

⟨*type* AST_generic.other_directive_operator 155d⟩
```
  (* PHP *)
  (* TODO: Declare, move OE_UseStrict here for JS? *)
  (* Ruby *)
  | OI_Alias
  | OI_Undef
  (* Rust *)
  | OI_Extern

(*****************************************************************************)
(* Toplevel *)
(*****************************************************************************)
(* item (a.k.a toplevel element, toplevel decl) is now equal to stmt.
 * Indeed, many languages allow nested functions, nested class definitions,
 * and even nested imports, so it is just simpler to merge item with stmt.
 * This simplifies sgrep too.
 * less: merge with field?
 *)
```
⟨*type* AST_generic.item 32f⟩

⟨*type* AST_generic.program 32e⟩

```
(*****************************************************************************)
(* Partial *)
(*****************************************************************************)
(* sgrep: this is only used by semgrep *)
and partial =
  (* partial defs. The fbody or cbody in definition will be empty. *)
  | PartialDef of definition
  (* partial stmts *)
  | PartialIf of tok * expr (* todo? bracket *)
  | PartialTry of tok * stmt
  | PartialCatch of catch
  | PartialFinally of tok * stmt
  (* partial objects (just used in JSON and YAML patterns for now)
   * alt: todo? could be considered a full thing and use Fld?
```

```
   *)
  | PartialSingleField of string wrap (* id or str *) * tok (*:*) * expr
  (* not really a partial, but the partial machinery can help with that *)
  | PartialLambdaOrFuncDef of function_definition

(*****************************************************************************)
(* Any *)
(*****************************************************************************)

(* mentioned in many OtherXxx so must be part of the mutually recursive type *)
⟨type AST_generic.any 32i⟩
[@@deriving show { with_path = false }, eq, hash]

(* with tarzan *)

⟨constant AST_generic.special_multivardef_pattern 97b⟩

(*****************************************************************************)
(* Error *)
(*****************************************************************************)

⟨exception AST_generic.Error 54e⟩

⟨function AST_generic.error 54f⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
(* see also AST_generic_helpers.ml *)

⟨constant AST_generic.sid_TODO 174e⟩

⟨constant AST_generic.empty_name_info 143a⟩

⟨constant AST_generic.empty_var 261a⟩

⟨function AST_generic.empty_id_info 173c⟩

⟨function AST_generic.basic_id_info 173d⟩

⟨function AST_generic.param_of_id 261b⟩
⟨function AST_generic.param_of_type 261c⟩

⟨function AST_generic.basic_entity 27d⟩

let s skind =
  {
    s = skind;
    s_id = AST_utils.Node_ID.create ();
    s_use_cache = false;
    s_backrefs = None;
    s_bf = None;
  }

⟨function AST_generic.basic_field 261d⟩

⟨function AST_generic.attr 261e⟩
⟨function AST_generic.arg 261f⟩

⟨function AST_generic.fake 200f⟩
```

```
⟨function AST_generic.fake_bracket 200g⟩
⟨function AST_generic.unbracket 25e⟩
(* bugfix: I used to put ";" but now Parse_info.str_of_info prints
 * the string of a fake info
 *)
let sc = Parse_info.fake_info ""

let unhandled_keywordattr (s, t) =
  NamedAttr (t, Id ((s, t), empty_id_info ()), fake_bracket [])

let exprstmt e = s (ExprStmt (e, sc))

let fieldEllipsis t = FieldStmt (exprstmt (Ellipsis t))

let empty_fbody = s (Block (fake_bracket []))

let empty_body = fake_bracket []

⟨function AST_generic.stmt1 262f⟩
```

pfff/lang_GENERIC_base/AST_generic_helpers.ml

```
⟨pfff/lang_GENERIC_base/AST_generic_helpers.ml 274⟩≡
  ⟨pad/r2c copyright 11⟩
  open Common
  open AST_generic
  module M = Map_AST

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Helpers to build or convert AST_generic elements.
   *
   * Very often used helper functions are actually in AST_generic.ml at
   * the end (e.g., AST_generic.basic_entity).
   * This module is for the more rarely used helpers.
   *)

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)

  ⟨constant AST_generic.str_of_ident 24c⟩

  let name_of_entity ent =
    match ent.name with
    | EN (Id (i, pinfo)) | EN (IdQualified ((i, _), pinfo)) -> Some (i, pinfo)
    | EDynamic _ -> None

  ⟨constant AST_generic.gensym_counter 174c⟩
  ⟨function AST_generic.gensym 174d⟩

  let name_of_ids ?(name_typeargs = None) xs =
    match List.rev xs with
    | [] -> failwith "name_of_ids: empty ids"
    | [ x ] -> Id (x, empty_id_info ())
    | x :: xs ->
        let qualif = if xs = [] then None else Some (QDots (List.rev xs)) in
```

```
      IdQualified
        ((x, { name_qualifier = qualif; name_typeargs }), empty_id_info ())
```

⟨*function* AST_generic.expr_to_pattern 261g⟩

⟨*exception* AST_generic.NotAnExpr 262a⟩
⟨*function* AST_generic.pattern_to_expr 262b⟩

⟨*function* AST_generic.expr_to_type 262c⟩

```
(* old: there was a stmt_to_item before *)
(* old: there was a stmt_to_field before *)

(* see also Java_to_generic.entity_to_param *)
(* see also Python_to_generic.expr_to_attribute *)
(* see also Php_generic.list_expr_to_opt *)
(* see also Php_generic.name_of_qualified_ident (also in Java) *)
```

⟨*function* AST_generic.opt_to_empty 262d⟩

⟨*function* AST_generic.opt_to_label_ident 262e⟩

⟨*function* AST_generic.is_boolean_operator 262g⟩

```
let name_or_dynamic_to_expr name idinfo_opt =
  match (name, idinfo_opt) with
  (* assert idinfo = _idinfo below? *)
  | EN (Id (id, idinfo)), None -> N (Id (id, idinfo))
  | EN (Id (id, _idinfo)), Some idinfo -> N (Id (id, idinfo))
  | EN (IdQualified (n, idinfo)), None -> N (IdQualified (n, idinfo))
  | EN (IdQualified (n, _idinfo)), Some idinfo -> N (IdQualified (n, idinfo))
  | EDynamic e, _ -> e
```

⟨*function* AST_generic.vardef_to_assign 97a⟩

⟨*function* AST_generic.funcdef_to_lambda 263a⟩

⟨*function* AST_generic.has_keyword_attr 263b⟩

```
(*****************************************************************************)
(* Abstract position and constness for comparison *)
(*****************************************************************************)

(* update: you should now use AST_generic.equal_any which internally
 * does not care about position information.
 *)

let abstract_for_comparison_visitor recursor =
  let hooks =
    {
      M.default_visitor with
      M.kinfo = (fun (_k, _) i -> { i with Parse_info.token = Parse_info.Ab });
      M.kidinfo =
        (fun (k, _) ii -> k { ii with AST_generic.id_constness = ref None });
    }
  in
  let vout = M.mk_visitor hooks in
  recursor vout

let abstract_for_comparison_any x =
```

```
  abstract_for_comparison_visitor (fun visitor -> visitor.M.vany x)

(*****************************************************************************)
(* Conversion *)
(*****************************************************************************)

module G_ = AST_generic_
module G = AST_generic

(* This module is ugly, but it was written to allow to move AST_generic.ml
 * out of pfff/ and inside semgrep/. However there are many
 * language-specific ASTs that we using AST_generic.ml to factorize
 * the definitions of operators. To break the dependency we had
 * to duplicate that part of AST_generic in pfff/h_program-lang/AST_generic_.ml
 * (note that underscore at the end) and we need those boilerplate functions
 * below to convert them back to AST_generic.
 *
 * alt: use polymorphic variants (e.g., `Plus)
 *)

let (conv_op : AST_generic_.operator -> AST_generic.operator) = function
  | G_.Plus -> G.Plus
  | G_.Minus -> G.Minus
  | G_.Mult -> G.Mult
  | G_.Div -> G.Div
  | G_.Mod -> G.Mod
  | G_.Pow -> G.Pow
  | G_.FloorDiv -> G.FloorDiv
  | G_.MatMult -> G.MatMult
  | G_.LSL -> G.LSL
  | G_.LSR -> G.LSR
  | G_.ASR -> G.ASR
  | G_.BitOr -> G.BitOr
  | G_.BitXor -> G.BitXor
  | G_.BitAnd -> G.BitAnd
  | G_.BitNot -> G.BitNot
  | G_.BitClear -> G.BitClear
  | G_.And -> G.And
  | G_.Or -> G.Or
  | G_.Xor -> G.Xor
  | G_.Not -> G.Not
  | G_.Eq -> G.Eq
  | G_.NotEq -> G.NotEq
  | G_.PhysEq -> G.PhysEq
  | G_.NotPhysEq -> G.NotPhysEq
  | G_.Lt -> G.Lt
  | G_.LtE -> G.LtE
  | G_.Gt -> G.Gt
  | G_.GtE -> G.GtE
  | G_.Cmp -> G.Cmp
  | G_.Concat -> G.Concat
  | G_.Append -> G.Append
  | G_.RegexpMatch -> G.RegexpMatch
  | G_.NotMatch -> G.NotMatch
  | G_.Range -> G.Range
  | G_.RangeInclusive -> G.RangeInclusive
  | G_.NotNullPostfix -> G.NotNullPostfix
  | G_.Length -> G.Length
  | G_.Elvis -> G.Elvis
  | G_.Nullish -> G.Nullish
```

```
    | G_.In -> G.In
    | G_.NotIn -> G.NotIn
    | G_.Is -> G.Is
    | G_.NotIs -> G.NotIs

  let (conv_incr : AST_generic_.incr_decr -> AST_generic.incr_decr) = function
    | G_.Incr -> G.Incr
    | G_.Decr -> G.Decr

  let (conv_prepost : AST_generic_.prefix_postfix -> AST_generic.prefix_postfix) =
    function
    | G_.Prefix -> G.Prefix
    | G_.Postfix -> G.Postfix

  let (conv_incdec :
        AST_generic_.incr_decr * AST_generic_.prefix_postfix ->
        AST_generic.incr_decr * AST_generic.prefix_postfix) =
   fun (x, y) -> (conv_incr x, conv_prepost y)

  let (conv_class_kind :
        AST_generic_.class_kind * Parse_info.t ->
        AST_generic.class_kind * Parse_info.t) =
   fun (c, t) ->
    ( ( match c with
      | G_.Class -> G.Class
      | G_.Interface -> G.Interface
      | G_.Trait -> G.Trait ),
      t )

  let (conv_function_kind :
        AST_generic_.function_kind * Parse_info.t ->
        AST_generic.function_kind * Parse_info.t) =
   fun (c, t) ->
    ( ( match c with
      | G_.Function -> G.Function
      | G_.Method -> G.Method
      | G_.LambdaKind -> G.LambdaKind
      | G_.Arrow -> G.Arrow ),
      t )
```

pfff/lang_GENERIC_base/Visitor_AST.mli

⟨pfff/lang_GENERIC/parsing/Visitor_AST.mli 277⟩≡

```
  open AST_generic

  (* the hooks *)
  ⟨type Visitor_AST.visitor_in 156b⟩
  (* note that internally the visitor uses OCaml.v_ref_do_not_visit *)
  ⟨type Visitor_AST.visitor_out 156c⟩

  ⟨signature Visitor_AST.default_visitor 156d⟩

  ⟨signature Visitor_AST.mk_visitor 157a⟩

  (* poor's man fold *)
  (*
  val do_visit_with_ref:
    ('a list ref -> visitor_in) -> any -> 'a list
```

```
*)

(* Note that ii_of_any relies on Visitor_AST which itself
 * uses OCaml.v_ref_do_not_visit, so no need to worry about
 * tokens inside id_type or id_info.
 *)
val ii_of_any : AST_generic.any -> Parse_info.t list

val range_of_tokens : Parse_info.t list -> Parse_info.t * Parse_info.t

val range_of_any :
  AST_generic.any -> Parse_info.token_location * Parse_info.token_location
```

pfff/lang_GENERIC_base/Map_AST.mli

⟨pfff/lang_GENERIC/parsing/Map_AST.mli 278a⟩≡

```
open AST_generic
```

⟨*type* Map_AST.visitor_in 157f⟩

⟨*type* Map_AST.visitor_out 157g⟩

⟨*signature* Map_AST.default_visitor 157h⟩

⟨*signature* Map_AST.mk_visitor 157i⟩

pfff/lang_GENERIC_base/Meta_AST.mli

⟨*signature* Meta_AST.vof_literal 278b⟩≡ (279c)

```
val vof_literal : AST_generic.literal -> OCaml.v
```

⟨*signature* Meta_AST.vof_type_ 278c⟩≡ (279c)

```
val vof_type_ : AST_generic.type_ -> OCaml.v
```

⟨*signature* Meta_AST.vof_arithmetic_operator 278d⟩≡ (279c)

```
val vof_arithmetic_operator : AST_generic.operator -> OCaml.v
```

⟨*signature* Meta_AST.vof_function_definition 278e⟩≡ (279c)

```
val vof_function_definition : AST_generic.function_definition -> OCaml.v
```

⟨*signature* Meta_AST.vof_class_definition 278f⟩≡ (279c)

```
val vof_class_definition : AST_generic.class_definition -> OCaml.v
```

⟨*signature* Meta_AST.vof_definition 278g⟩≡ (279c)

```
val vof_definition : AST_generic.definition -> OCaml.v
```

⟨*signature* Meta_AST.vof_directive 278h⟩≡ (279c)

```
val vof_directive : AST_generic.directive -> OCaml.v
```

⟨*signature* Meta_AST.vof_expr 279a⟩≡ (279c)
```
val vof_expr : AST_generic.expr -> OCaml.v
```

⟨*signature* Meta_AST.vof_stmt 279b⟩≡ (279c)
```
val vof_stmt : AST_generic.stmt -> OCaml.v
```

⟨pfff/lang_GENERIC/parsing/Meta_AST.mli 279c⟩≡

```
(* Note that you probably should use AST_generic.show_any, which is
 * auto-generated rather than this file to display debug information.
 * We keep Meta_AST.ml just because we use it to generate the JSON
 * from the OCaml.v type of the generic AST in semgrep (and use that
 * JSON in the semgrep Web UI).
 *)
```

⟨*signature* Meta_AST.vof_any 156a⟩

```
(* internals used by other dumpers, e.g., Meta_IL.ml *)
```
⟨*signature* Meta_AST.vof_literal 278b⟩
⟨*signature* Meta_AST.vof_type_ 278c⟩
⟨*signature* Meta_AST.vof_arithmetic_operator 278d⟩
⟨*signature* Meta_AST.vof_function_definition 278e⟩
⟨*signature* Meta_AST.vof_class_definition 278f⟩
⟨*signature* Meta_AST.vof_definition 278g⟩
⟨*signature* Meta_AST.vof_directive 278h⟩
⟨*signature* Meta_AST.vof_expr 279a⟩
⟨*signature* Meta_AST.vof_stmt 279b⟩

```
(* reused in other dumpers *)
val vof_incr_decr : AST_generic.incr_decr -> OCaml.v

val vof_inc_dec : AST_generic.incr_decr * AST_generic.prefix_postfix -> OCaml.v

val vof_prepost : AST_generic.prefix_postfix -> OCaml.v
```

F.8 pfff/lang_GENERIC/parsing/

pfff/lang_GENERIC/parsing/Parse_generic.mli

⟨pfff/lang_GENERIC/parsing/Parse_generic.mli 279d⟩≡

⟨*signature* Parse_generic.parse_with_lang 53a⟩

⟨*signature* Parse_generic.parse_program 54g⟩

⟨*signature* Parse_generic.parse_pattern 55b⟩

pfff/lang_GENERIC/parsing/Parse_generic.ml

⟨pfff/lang_GENERIC/parsing/Parse_generic.ml 279e⟩≡
⟨*pad/r2c copyright* 11⟩
```
open Common

(*****************************************************************************)
```

```
  (* Prelude *)
  (*****************************************************************************)
  (* Wrappers around many languages to transform them in a generic AST
   * (see AST_generic.ml)
  *)

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)
  ⟨function Parse_generic.lang_to_python_parsing_mode 141a⟩

  (*****************************************************************************)
  (* Entry points *)
  (*****************************************************************************)

  ⟨function Parse_generic.parse_with_lang 53b⟩

  ⟨function Parse_generic.parse_program 54h⟩


  ⟨function Parse_generic.parse_pattern 55c⟩
```

pfff/lang_GENERIC/parsing/Test_parsing_generic.mli

⟨pfff/lang_GENERIC/parsing/Test_parsing_generic.mli 280a⟩≡

```
  ⟨signature Test_parsing_generic.lang 50d⟩

  ⟨signature Test_parsing_generic.actions 39f⟩
```

pfff/lang_GENERIC/parsing/Test_parsing_generic.ml

⟨pfff/lang_GENERIC/parsing/Test_parsing_generic.ml 280b⟩≡

```
  open Common

  ⟨constant Test_parsing_generic.lang 50e⟩

  ⟨function Test_parsing_generic.test_parse_generic 39h⟩

  let test_show_generic file =
    let ast = Parse_generic.parse_program file in
    let s = AST_generic.show_program ast in
    pr2 s

  ⟨function Test_parsing_generic.test_dump_generic 40a⟩

  ⟨function Test_parsing_generic.test_dump_pattern_generic 50g⟩


  ⟨function Test_parsing_generic.actions 39g⟩
```

pfff/lang_GENERIC/parsing/Lib_AST.mli

⟨pfff/lang_GENERIC/parsing/Lib_AST.mli 280c⟩≡

```
  (* Note that ii_of_any relies on Visitor_AST which itself
   * uses OCaml.v_ref_do_not_visit, so no need to worry about
```

```
 * tokens inside id_type or id_info.
 *)
```
⟨*signature* Lib_AST.ii_of_any 157c⟩

⟨*signature* Lib_AST.abstract_position_info_any 158a⟩

pfff/lang_GENERIC/parsing/Lib_AST.ml

⟨pfff/lang_GENERIC/parsing/Lib_AST.ml 281a⟩≡
⟨*pad/r2c copyright* 11⟩

```
module V = Visitor_AST
module M = Map_AST

(*****************************************************************************)
(* Extract infos *)
(*****************************************************************************)
```
⟨*function* Lib_AST.extract_info_visitor 157d⟩

⟨*function* Lib_AST.ii_of_any 157e⟩

```
(*****************************************************************************)
(* Abstract position *)
(*****************************************************************************)
```
⟨*function* Lib_AST.abstract_position_visitor 158c⟩
⟨*function* Lib_AST.abstract_position_info_any 158b⟩

F.9 pfff/lang_GENERIC/analyze/

pfff/lang_GENERIC/analyze/Naming_AST.mli

⟨pfff/lang_GENERIC/analyze/Naming_AST.mli 281b⟩≡

⟨*signature* Naming_AST.resolve 175a⟩

/home/pad/pfff/lang_GENERIC/analyze/Naming_AST.ml

⟨*type* Naming_AST.resolved_name 281c⟩≡ (290)
```
(* this includes the "single unique id" (sid) *)
type resolved_name = AST_generic.resolved_name
```

⟨*type* Naming_AST.scope 281d⟩≡ (290)
```
type scope = (string, scope_info) assoc
```

⟨*type* Naming_AST.scopes 281e⟩≡ (290)
```
type scopes = {
  global : scope ref;
  (* function, nested blocks, nested functions (lambdas) *)
  blocks : scope list ref;
  (* useful for python, kind of global scope but for entities *)
  imported : scope ref;
      (* todo?
       * - class? right now we abuse EnclosedVar in resolved_name_kind.
```

```
        * - function? for 'var' in JS
        *)
  }
```

⟨*function* Naming_AST.default_scopes 282a⟩≡ (290)
```
  let default_scopes () = { global = ref []; blocks = ref []; imported = ref [] }
```

⟨*function* Naming_AST.with_new_function_scope 282b⟩≡ (290)
```
  (* because we use a Visitor instead of a clean recursive
   * function passing down an environment, we need to emulate a scoped
   * environment by using save_excursion.
   *)

  let with_new_function_scope params scopes f =
    Common.save_excursion scopes.blocks (params :: !(scopes.blocks)) f
```

⟨*function* Naming_AST._with_new_block_scope 282c⟩≡ (290)
```
  let _with_new_block_scope _params _lang _scopes _f = raise Todo
```

⟨*function* Naming_AST.add_ident_current_scope 282d⟩≡ (290)
```
  let add_ident_current_scope (s, _) resolved scopes =
    match !(scopes.blocks) with
    | [] -> scopes.global := (s, resolved) :: !(scopes.global)
    | xs :: xxs -> scopes.blocks := ((s, resolved) :: xs) :: xxs
```

⟨*function* Naming_AST.add_ident_imported_scope 282e⟩≡ (290)
```
  (* for Python *)
  let add_ident_imported_scope (s, _) resolved scopes =
    scopes.imported := (s, resolved) :: !(scopes.imported)
```

⟨*function* Naming_AST._add_ident_function_scope 282f⟩≡ (290)
```
  (* for JS 'var' *)
  let _add_ident_function_scope _id _resolved _scopes = raise Todo
```

⟨*function* Naming_AST.lookup 282g⟩≡ (290)
```
  let rec lookup s xxs =
    match xxs with
    | [] -> None
    | xs :: xxs -> (
        match List.assoc_opt s xs with
        | None -> lookup s xxs
        | Some res -> Some res )
```

⟨*function* Naming_AST.lookup_scope_opt 282h⟩≡ (290)

⟨*function* Naming_AST.lookup_global_scope 282i⟩≡ (290)
```
  (* for Python, PHP *)
  let lookup_global_scope (s, _) scopes = lookup s [ !(scopes.global) ]
```

⟨*function* Naming_AST.lookup_nonlocal_scope 283a⟩≡ (290)
```
(* for Python, PHP *)
let lookup_nonlocal_scope id scopes =
  let s, tok = id in
  match !(scopes.blocks) with
  | _ :: xxs -> lookup s xxs
  | [] ->
      let _ = error tok "no outerscope" in
      None
```

⟨*type* Naming_AST.context 283b⟩≡ (290)
```
type context =
  | AtToplevel
  | InClass
  (* separate InMethod? InLambda? just look for InFunction::InClass::_ *)
  | InFunction
```

⟨*type* Naming_AST.env 283c⟩≡ (290)
```
type env = {
  ctx : context list ref;
  (* handle locals/params/globals, block vas, enclosed vars (closures).
   * handle also basic typing information now for Java/Go.
   *)
  names : scopes;
  in_lvalue : bool ref;
  in_type : bool ref;
  lang : Lang.t;
}
```

⟨*function* Naming_AST.default_env 283d⟩≡ (290)
```
let default_env lang =
  {
    ctx = ref [ AtToplevel ];
    names = default_scopes ();
    in_lvalue = ref false;
    in_type = ref false;
    lang;
  }
```

⟨*function* Naming_AST.add_constant_env 283e⟩≡ (290)

⟨*function* Naming_AST.with_new_context 283f⟩≡ (290)
```
let with_new_context ctx env f =
  Common.save_excursion env.ctx (ctx :: !(env.ctx)) f
```

⟨*function* Naming_AST.top_context 283g⟩≡ (290)
```
let top_context env =
  match !(env.ctx) with [] -> raise Impossible | x :: _xs -> x
```

⟨*function* Naming_AST.set_resolved 284a⟩≡ (290)
```
let set_resolved env id_info x =
  (* TODO? maybe do it only if we have something better than what the
   * lang-specific resolved found?
   *)
  id_info.id_resolved := Some x.entname;
  (* this is defensive programming against the possibility of introducing
   * cycles in the AST.
   * See tests/python/naming/shadow_name_type.py for a patological case. *)
  if not !(env.in_type) then id_info.id_type := x.enttype
```

⟨*constant* Naming_AST.error_report 284b⟩≡ (290)
```
let error_report = ref false
```

⟨*function* Naming_AST.error 284c⟩≡ (290)
```
let error tok s =
  if !error_report then raise (Parse_info.Other_error (s, tok))
  else logger#error "%s at %s" s (Parse_info.string_of_info tok)
```

⟨*function* Naming_AST.is_local_or_global_ctx 284d⟩≡ (290)
```
let is_resolvable_name_ctx env lang =
  match top_context env with
  | AtToplevel | InFunction -> true
  | InClass -> (
      match lang with
      (* true for Java so that we can type class fields *)
      | Lang.Java
      (* true for JS/TS so that we can resolve class methods *)
      | Lang.Javascript | Lang.Typescript ->
          true
      | _ -> false )
```

⟨*function* Naming_AST.resolved_name_kind 284e⟩≡ (290)
```
let resolved_name_kind env lang =
  match top_context env with
  | AtToplevel -> Global
  | InFunction -> Local
  | InClass -> (
      match lang with
      (* true for Java so that we can type class fields.
       * alt: use a different scope.class?
       *)
      | Lang.Java
      (* true for JS/TS to resolve class methods. *)
      | Lang.Javascript | Lang.Typescript ->
          EnclosedVar
      | _ -> raise Impossible )
```

⟨*function* Naming_AST.params_of_parameters 284f⟩≡ (290)
```
(* !also set the id_info of the parameter as a side effect! *)
let params_of_parameters env xs =
  xs
  |> Common.map_filter (function
       | ParamClassic { pname = Some id; pinfo = id_info; ptype = typ; _ } ->
           let sid = H.gensym () in
```

```
            let resolved = { entname = (Param, sid); enttype = typ } in
            set_resolved env id_info resolved;
            Some (H.str_of_ident id, resolved)
        | _ -> None)
```

⟨*function* Naming_AST.resolve 285⟩≡ (290)

```
  let resolve2 lang prog =
    logger#info "Naming_AST.resolve program";
    let env = default_env lang in

    (* would be better to use a classic recursive-with-environment visit.
     * coupling: we do similar things in Constant_propagation.ml so if you
     * add a feature here, you might want to add a similar thing over there too.
     *)
    let hooks =
      {
        V.default_visitor with
        (* the defs *)
        V.kfunction_definition =
          (fun (k, _v) x ->
            (* todo: add the function as a Global. In fact we should do a first
             * pass for some languages to add all of them first, because
             * Go for example allow the use of forward function reference
             * (no need to declarare prototype and forward decls as in C).
             *)
            let new_params = params_of_parameters env x.fparams in
            with_new_context InFunction env (fun () ->
                with_new_function_scope new_params env.names (fun () ->
                    (* todo: actually we should first go inside x.fparams.ptype
                     * without the new_params (this would also prevent cycle if
                     * a parameter name is the same than type name used in ptype
                     * (see tests/python/naming/shadow_name_type.py) *)
                    k x)));
        V.kclass_definition =
          (fun (k, _v) x ->
            (* todo: we should first process all fields in the class before
             * processing the methods, as some languages may allow forward ref.
             *)
            with_new_context InClass env (fun () -> k x));
        V.kdef =
          (fun (k, v) x ->
            match x with
            | ( ({ name = EN (Id (id, id_info)); _ } as ent),
                VarDef { vinit; vtype } )
            (* note that some languages such as Python do not have VarDef
             * construct
             * todo? should add those somewhere instead of in_lvalue detection? *)
              when is_resolvable_name_ctx env lang ->
                (* Need to visit expressions first so that type is populated, e.g.
                 * if we do var a = 3, then var b = a, we want to propagate the type of a. *)
                (* Note that we are careful to visit each part of the declaration
                 * separately in order to avoid the v_vardef_as_assign_expr
                 * equivalence in Visitor_AST. This is a problem for JS/TS where
                 * there are both explicit and implicit variable declarations, and
                 * it will cause the same identifier to be resolved twice. *)
                v (En ent);
                vinit |> do_option (fun e -> v (E e));
                vtype |> do_option (fun t -> v (T t));
```

```
    (* name resolution *)
    let sid = H.gensym () in
    (* for the type, we use the (optional) type in vtype, or, if we can infer  *)
    (* the type of the expression vinit (literal or id), we use that as a type *)
    (* useful for Go, where you can write var x = 2 without declaring the type *)
    let resolved_type =
      Typing.get_resolved_type lang (vinit, vtype)
    in
    let resolved =
      {
        entname = (resolved_name_kind env lang, sid);
        enttype = resolved_type;
      }
    in
    add_ident_current_scope id resolved env.names;
    set_resolved env id_info resolved
| { name = EN (Id (id, id_info)); _ }, FuncDef _
  when is_resolvable_name_ctx env lang ->
    ( match lang with
    (* We restrict function-name resolution to JS/TS.
     *
     * Note that this causes problems with Python rule/test:
     *
     *     semgrep-rules/python/flask/correctness/same-handler-name.yaml
     *
     * This rule tries to match two different functions using the same
     * meta-variable. This works when the function names are not resolved,
     * but breaks when each function gets a unique sid.
     *
     * Function-name resolution is useful for interprocedural analysis,
     * feature that was requested by JS/TS users, see:
     *
     *     https://github.com/returntocorp/semgrep/issues/2787.
     *)
    | Lang.Javascript | Lang.Typescript ->
        let sid = H.gensym () in
        let resolved =
          untyped_ent (resolved_name_kind env lang, sid)
        in
        (* Previously we tried using add_ident_current_scope, but this
         * shadowed imported function names which are added to the
         * "imported" scope (globals/block scope is looked up first)
         * even when the import statement comes after...
         * This broke the following test:
         *
         *     semgrep-rules/python/django/security/audit/raw-query.py
         *
         * For now we add function names also to the "imported" scope...
         * but do we need a special scope for imported functions?
         *)
        add_ident_imported_scope id resolved env.names;
        set_resolved env id_info resolved
    | ___else___ -> () );
    k x
| { name = EN (Id (id, id_info)); _ }, UseOuterDecl tok ->
    let s = Parse_info.str_of_info tok in
    let flookup =
      match s with
      | "global" -> lookup_global_scope
      | "nonlocal" -> lookup_nonlocal_scope
```

```
          | _ ->
              error tok (spf "unrecognized UseOuterDecl directive: %s" s);
              lookup_global_scope
        in
        ( match flookup id env.names with
        | Some resolved ->
            set_resolved env id_info resolved;
            add_ident_current_scope id resolved env.names
        | None ->
            error tok
              (spf "could not find '%s' for directive %s"
                 (H.str_of_ident id) s) );
        k x
    | _ -> k x);
(* sgrep: the import aliases *)
V.kdir =
  (fun (k, _v) x ->
    ( match x with
    | ImportFrom (_, DottedName xs, id, Some (alias, id_info)) ->
        (* for python *)
        let sid = H.gensym () in
        let resolved = untyped_ent (ImportedEntity (xs @ [ id ]), sid) in
        set_resolved env id_info resolved;
        add_ident_imported_scope alias resolved env.names
    | ImportFrom (_, DottedName xs, id, None) ->
        (* for python *)
        let sid = H.gensym () in
        let resolved = untyped_ent (ImportedEntity (xs @ [ id ]), sid) in
        add_ident_imported_scope id resolved env.names
    | ImportFrom (_, FileName (s, tok), id, None)
      when Lang.is_js lang && fst id <> Ast_js.default_entity ->
        (* for JS we consider import { x } from 'a/b/foo' as foo.x.
         * Note that we guard this code with is_js lang, because Python
         * uses also Filename in 'from ...conf import x'.
         *)
        let sid = H.gensym () in
        let _, b, _ = Common2.dbe_of_filename_noext_ok s in
        let base = (b, tok) in
        let resolved = untyped_ent (ImportedEntity [ base; id ], sid) in
        add_ident_imported_scope id resolved env.names
    | ImportFrom (_, FileName (s, tok), id, Some (alias, id_info))
      when Lang.is_js lang && fst id <> Ast_js.default_entity ->
        (* for JS *)
        let sid = H.gensym () in
        let _, b, _ = Common2.dbe_of_filename_noext_ok s in
        let base = (b, tok) in
        let resolved = untyped_ent (ImportedEntity [ base; id ], sid) in
        set_resolved env id_info resolved;
        add_ident_imported_scope alias resolved env.names
    | ImportAs (_, DottedName xs, Some (alias, id_info)) ->
        (* for python *)
        let sid = H.gensym () in
        let resolved =
          untyped_ent (ImportedModule (DottedName xs), sid)
        in
        set_resolved env id_info resolved;
        add_ident_imported_scope alias resolved env.names
    | ImportAs (_, FileName (s, tok), Some (alias, id_info)) ->
        (* for Go *)
        let sid = H.gensym () in
```

```
        let base = (Filename.basename s, tok) in
        let resolved =
          untyped_ent (ImportedModule (DottedName [ base ]), sid)
        in
        set_resolved env id_info resolved;
        add_ident_imported_scope alias resolved env.names
    | _ -> () );
    k x);
V.kpattern =
  (fun (k, _vout) x ->
    match x with
    | PatId (id, id_info) when is_resolvable_name_ctx env lang ->
        (* todo: in Python it does not necessarily introduce
         * a newvar if the ID was already declared before.
         * Also inside a PatAs(PatId x,b), the 'x' is actually
         * the name of a class, not a newly introduced local.
         *)
        (* mostly copy-paste of VarDef code *)
        let sid = H.gensym () in
        let resolved = untyped_ent (resolved_name_kind env lang, sid) in
        add_ident_current_scope id resolved env.names;
        set_resolved env id_info resolved;
        k x
    | PatVar (_e, Some (id, id_info)) when is_resolvable_name_ctx env lang
      ->
        (* mostly copy-paste of VarDef code *)
        let sid = H.gensym () in
        let resolved = untyped_ent (resolved_name_kind env lang, sid) in
        add_ident_current_scope id resolved env.names;
        set_resolved env id_info resolved;
        k x
    | OtherPat _
    (* This interacts badly with implicit JS/TS declarations. It causes
     * `foo` in `function f({ foo }) { ... }` to be resolved as a global
     * variable, which in turn affects semgrep-rule _react-props-in-state_.
     * This when-clause achieves the previous behavior of leaving `foo`
     * unresolved. *)
    (* TODO: We should fix the AST of JS/TS so those `f({foo})` patterns do
     * not show as regular variables. *)
      when not (Lang.is_js lang) ->
        Common.save_excursion env.in_lvalue true (fun () -> k x)
    | _ -> k x);
(* the uses *)
V.kexpr =
  (fun (k, vout) x ->
    let recurse = ref true in
    ( match x with
    (* Go: This is `x := E`, a single-variable short variable declaration.
     * When this declaration is legal, and that is when the same variable
     * has not yet been declared in the same scope, it *always* introduces
     * a new variable. (Quoting Go' spec, "redeclaration can only appear
     * in a multi-variable short declaration".)
     * See: https://golang.org/ref/spec#Short_variable_declarations *)
    | AssignOp (N (Id (id, id_info)), (Eq, tok), e2)
      when lang = Lang.Go
           && Parse_info.str_of_info tok = ":="
           && is_resolvable_name_ctx env lang ->
        (* Need to visit expressions first so that type is populated *)
        (* If we do var a = 3, then var b = a, we want to propagate the type of a *)
        k x;
```

```
    (* name resolution *)
    let sid = H.gensym () in
    let resolved_type =
      Typing.get_resolved_type lang (Some e2, None)
    in
    let resolved =
      {
        entname = (resolved_name_kind env lang, sid);
        enttype = resolved_type;
      }
    in
    add_ident_current_scope id resolved env.names;
    set_resolved env id_info resolved;
    recurse := false
(* todo: see lrvalue.ml
 * alternative? extra id_info tag?
 *)
| Assign (e1, _, e2) | AssignOp (e1, _, e2) ->
    Common.save_excursion env.in_lvalue true (fun () -> vout (E e1));
    vout (E e2);
    recurse := false
| ArrayAccess (e1, (_, e2, _)) ->
    vout (E e1);
    Common.save_excursion env.in_lvalue false (fun () -> vout (E e2));
    recurse := false
| N (Id (id, id_info)) -> (
    match lookup_scope_opt id env with
    | Some resolved ->
        (* name resolution *)
        set_resolved env id_info resolved
    | None -> (
        match (!(env.in_lvalue), lang) with
        (* first use of a variable can be a VarDef in some languages *)
        (* type propagation not necessary because this does not hold true for Java or Go *)
        | true, (Lang.Python | Lang.Ruby | Lang.PHP)
          when is_resolvable_name_ctx env lang ->
            (* mostly copy-paste of VarDef code *)
            let sid = H.gensym () in
            let resolved =
              untyped_ent (resolved_name_kind env lang, sid)
            in
            add_ident_current_scope id resolved env.names;
            set_resolved env id_info resolved
        | true, (Lang.Javascript | Lang.Typescript)
          when is_resolvable_name_ctx env lang ->
            (* In JS/TS an assignment to a variable that has not been
             * previously declared will implicitly create a property on
             * the *global* object. *)
            let sid = H.gensym () in
            let resolved = untyped_ent (Global, sid) in
            add_ident_global_scope id resolved env.names;
            set_resolved env id_info resolved
            (* hopefully the lang-specific resolved may have resolved that *)
        | _ ->
            (* TODO: this can happen because of in_lvalue bug detection, or
             * for certain entities like functions or classes which are
             * currently tagged
             *)
            let s, tok = id in
            error tok (spf "could not find '%s' in environment" s) ) )
```

```
          | DotAccess (IdSpecial (This, _), _, EN (Id (id, id_info))) -> (
              match lookup_scope_opt id env with
              (* TODO: this is a v0 for doing naming and typing of fields.
               * we should really use a different lookup_scope_class, that
               * would handle shadowing of fields from locals, etc. but it's
               * a start.
               *)
              | Some ({ entname = EnclosedVar, _sid; _ } as resolved) ->
                  set_resolved env id_info resolved
              | _ ->
                  let s, tok = id in
                  error tok (spf "could not find '%s' field in environment" s) )
          | _ -> () );
          if !recurse then k x);
      V.kattr =
        (fun (k, _v) x ->
          ( match x with
          | NamedAttr (_, Id (id, id_info), _args) -> (
              match lookup_scope_opt id env with
              | Some resolved ->
                  (* name resolution *)
                  set_resolved env id_info resolved
              | _ -> () )
          | _ -> () );
          k x);
      V.ktype_ =
        (fun (k, _v) x ->
          let f x =
            ( match x with
            (* TODO: factorize in kname? *)
            | TyN (Id (id, id_info)) -> (
                match lookup_scope_opt id env with
                | Some resolved -> set_resolved env id_info resolved
                | _ -> () )
            | _ -> () );
            k x
          in
          (* when we are inside a type, especially in  (OtherType (OT_Expr)),
           * we don't want set_resolved to set the type on some Id because
           * this could lead to cycle in the AST because of id_type
           * that will reference a type, that could containi an OT_Expr, containing
           * an Id, that could contain the same id_type, and so on.
           * See tests/python/naming/shadow_name_type.py for a patological example
           * See also tests/rust/parsing/misc_recursion.rs for another example.
           *)
          if !(env.in_type) then f x
          else Common.save_excursion env.in_type true (fun () -> f x));
    }
  in
  let visitor = V.mk_visitor hooks in
  visitor (Pr prog);
  ()
```

⟨pfff/lang_GENERIC/analyze/Naming_AST.ml 290⟩≡

```
(* Yoann Padioleau
 *
 * Copyright (C) 2020 r2c
 *
 * This library is free software; you can redistribute it and/or
```

```
 * modify it under the terms of the GNU Lesser General Public License
 * version 2.1 as published by the Free Software Foundation, with the
 * special exception on linking described in file license.txt.
 *
 * This library 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.
 *)
open Common
open AST_generic
module V = Visitor_AST
module H = AST_generic_helpers

let logger = Logging.get_logger [ __MODULE__ ]

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* The goal of this module is to resolve names, a.k.a naming or
 * scope resolution, and to do it in a generic way on the generic AST.
 *
 * In a compiler frontend you often have those phases:
 *  - lexing
 *  - parsing
 *  - naming (the goal of this file)
 *  - typing
 *
 * The goal of naming is to simplify further phases by having each
 * use of an entity clearly linked to its definition. For example,
 * when you see in the AST the use of the identifier 'a', this 'a'
 * could reference a local variable, or a parameter, or a global,
 * or a global defined in another module but imported in the current
 * namespace, or a variable defined in a nested block that "shadows" an
 * enclosing variable with the same name.
 * By resolving once and for all all uses of an entity to its definition,
 * for example by renaming some shadow variables (see AST_generic.gensym),
 * we simpify further phases that don't have to maintain a complex environment
 * to deal with scoping issues (see the essence Of Python paper
 * "Python: The Full Monty" where they show that even complex IDEs still
 * don't correctly handle Python scoping rules and perform wrong renaming
 * refactorings).
 *
 * Resolving names by tagging identifiers is also useful for
 * codemap/efuns to colorize identifiers (locals, params, globals, unknowns)
 * differently.
 *
 * alternatives:
 *  - CURRENT: generic naming and use of a 'ref resolved_name' to annotate
 *    the generic AST. Note that the use of a ref that can be shared with
 *    the lang-specific AST (e.g., ast_go.ml) allows tools like codemap/efuns
 *    to benefit from the generic naming analysis while still caring only
 *    about the lang-specific AST (even though we may want at some point
 *    to have a generic highlighter).
 *  - define a separate type for a named ast, e.g., nast.ml (as done in
 *    hack/skip) instead of modifying refs, with a unique identifier
 *    for each entity. However, this is tedious to
 *    write as both types are almost identical (maybe a functor could help,
 *    or a generic aast type as in recent hack code). Moreover, this is really
 *    useful for complex global analysis (or at least semi-global as in
```

```
*    OCaml where you still need to open many .cmi when you locally type a .ml)
*    such as typing where we want to resolve every use of a global.
*    For sgrep, where we might for quite some time restrict ourselves to
*    local analysis, maybe the ref implementation technique is good enough.
*  - implement a resolve_xxx.ml for each language instead of doing
*    on the generic AST. That is what I was doing previously, which
*    has some advantages (some language-specific constructs that introduce
*    new variables, for example Python comprehensions, are hard to analyze
*    once converted to the generic AST because they are under an
*    Other_xxx category). However, there's potentially lots of code
*    duplication for each language and it's easy for a language to fall
*    behind.
*    A nice compromise might be to do most of the work in naming_ast.ml
*    but still have lang-specific resolve_xxx.ml to tag special
*    constructs that override what naming_ast.ml would do.
*    See set_resolved()
*
* TODO:
*  - generalize the original "resolvers":
*    * resolve_go.ml
*    * resolve_python.ml
*    * ast_js_build.ml
*    * check_variables_cpp.ml
*    * check_variables_php.ml
*  - introduce extra VarDef for languages that do not have them like
*    Python/PHP where the first use is a def (which in turn requires
*    special construct like 'global' or 'nonlocal' to disable this).
*  - go:
*    * handle DShortVars and Foreach local vars, DMethod receiver parameter,
*      and TypeName for new types
*    * in theory if/for/switch with their init declare new scope, as well
*      as Block
*    * should do first pass to get all toplevel decl as you can use
*      forward ref in Go
*  - get rid of the original "resolvers":
*    * resolve_xxx.ml
*    * ast_js_build.ml
*    * check_variables_xxx.ml
*  - get rid of or unify scope_code.ml, scope_php.ml, and
*    ast_generic.resolved_name
*  - resolve also types! in java if you import org.foo.Bar then later
*    you can simply use Bar x; for a type, but we don't currently resolve
*    those.
*
* history:
*  - PHP deadcode detector with global analysis and global code database
*  - local name resolution for PHP and C/C++ in check_variables_cpp.ml and
*    check_variables_php.ml for codemap semantic highlighting of identifiers
*    (mainly local vs params vs globals vs unknown) and for checkModule
*    (scheck ancestor). Use of a ref for C/C++.
*  - graph_code_xxx.ml global name resolution for PHP, then Java,
*    then ML, then ML via cmt, then Clang ASTs, then C, then Javascript
*  - separate named AST (nast.ml) and naming phase for Hack
*  - local name resolution for code highlighting for Javascript, then Python
*    to better colorize identifiers in codemap/efuns, but separate from
*    a variable checker (resolve_xxx.ml instead of check_variables_xxx.ml)
*  - AST generic and its resolved_name ref
*  - simple resolve_python.ml with variable and import resolution
*  - separate resolve_go.ml  with import resolution
*  - try to unify those resolvers in one file, naming_ast.ml
```

```
 *)

(*****************************************************************************)
(* Scope *)
(*****************************************************************************)
```

⟨*type* Naming_AST.resolved_name 281c⟩

```
type scope_info = {
  (* variable kind and sid *)
  entname : resolved_name;
  (* variable type, if known *)
  enttype : type_ option;
}
```

⟨*type* Naming_AST.scope 281d⟩

⟨*type* Naming_AST.scopes 281e⟩

⟨*function* Naming_AST.default_scopes 282a⟩

⟨*function* Naming_AST.with_new_function_scope 282b⟩

⟨*function* Naming_AST._with_new_block_scope 282c⟩

⟨*function* Naming_AST.add_ident_current_scope 282d⟩

⟨*function* Naming_AST.add_ident_imported_scope 282e⟩

```
let add_ident_global_scope (s, _) resolved scopes =
  scopes.global := (s, resolved) :: !(scopes.global)
```

⟨*function* Naming_AST._add_ident_function_scope 282f⟩

```
let untyped_ent name = { entname = name; enttype = None }
```

⟨*function* Naming_AST.lookup 282g⟩

⟨*function* Naming_AST.lookup_scope_opt 282h⟩

⟨*function* Naming_AST.lookup_global_scope 282i⟩

⟨*function* Naming_AST.lookup_nonlocal_scope 283a⟩

```
(*****************************************************************************)
(* Environment *)
(*****************************************************************************)
```

⟨*type* Naming_AST.context 283b⟩

⟨*type* Naming_AST.env 283c⟩

⟨*function* Naming_AST.default_env 283d⟩

```
(*****************************************************************************)
(* Environment Helpers *)
(*****************************************************************************)
```

⟨*function* Naming_AST.add_constant_env 283e⟩

⟨*function* Naming_AST.with_new_context 283f⟩

⟨*function* Naming_AST.top_context 283g⟩

⟨*function* Naming_AST.set_resolved 284a⟩

```
(* accessors *)
let lookup_scope_opt (s, _) env =
  let scopes = env.names in

  let actual_scopes =
    match !(scopes.blocks) with
    | [] -> [ !(scopes.global); !(scopes.imported) ]
    | xs :: xxs -> (
        match env.lang with
        | Lang.Python ->
            if
              !(env.in_lvalue)
              (* just look current scope! no access to nested scopes or global *)
            then [ xs; !(scopes.imported) ]
            else [ xs ] @ xxs @ [ !(scopes.global); !(scopes.imported) ]
        (* | Lang.PHP ->
             (* just look current scope! no access to nested scopes or global *)
             [xs;                            !(scopes.imported)]
        *)
        | _ -> [ xs ] @ xxs @ [ !(scopes.global); !(scopes.imported) ] )
  in
  lookup s actual_scopes

(*****************************************************************************)
(* Error management *)
(*****************************************************************************)
```
⟨*constant* Naming_AST.error_report 284b⟩

⟨*function* Naming_AST.error 284c⟩

```
(*****************************************************************************)
(* Other Helpers *)
(*****************************************************************************)
```
⟨*function* Naming_AST.is_local_or_global_ctx 284d⟩

⟨*function* Naming_AST.resolved_name_kind 284e⟩

⟨*function* Naming_AST.params_of_parameters 284f⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```

⟨*function* Naming_AST.resolve 285⟩
```
let resolve a b =
  Common.profile_code "Naming_ast.resolve" (fun () -> resolve2 a b)
```

pfff/lang_GENERIC/analyze/CFG.ml

⟨pfff/lang_GENERIC/analyze/CFG.ml 294⟩≡
⟨*pad/r2c copyright* 11⟩

pfff/lang_GENERIC/analyze/IL.ml

⟨pfff/lang_GENERIC/analyze/IL.ml 295⟩≡
⟨*pad/r2c copyright* 11⟩
```
module G = AST_generic

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Intermediate Language (IL) for static analysis.
 *
 * Just like for the CST -> AST, the goal of an AST -> IL transformation
 * is to simplify things even more for program analysis purpose.
 *
 * Here are the simplifications done compared to the generic AST:
 *  - intermediate 'instr' type (instr for instruction), for expressions with
 *    side effects and statements without any control flow,
 *    moving Assign/Seq/Call/Conditional out of 'expr' and
 *    moving Assert out of 'stmt'
 *  - new expression type 'exp' for side-effect free expressions
 *  - intermediate 'lvalue' type; expressions are splitted in
 *    lvalue vs regular expressions, moved Dot/Index out of expr
 *
 *  - Assign/Calls are now instructions, not expressions, and no more Seq
 *  - no AssignOp, or Decr/Incr, just Assign
 *  - Lambdas are now instructions (not nested again)
 *
 *  - no For/Foreach/DoWhile/While, converted all in Loop,
 *  - no Foreach, converted in a Loop and 2 new special
 *  - TODO no Switch, converted in Ifs
 *  - TODO no Continue/Break, converted in goto
 *  - less use of expr option (in Return/Assert/...), use Unit in those cases
 *
 *  - no Sgrep constructs
 *  - Naming has been performed, no more ident vs name
 *
 * TODO:
 *   - TODO? have all arguments of Calls be variables?
 *
 *
 * Note that we still want to be close to the original code so that
 * error reported on the IL can be mapped back to error on the original code
 * (source "maps"), or more importantly semantic information computed
 * on the IL (e.g., types, constness) can be mapped back to the generic AST.
 * This is why you will see some 'eorig', 'iorig' fields below and the use of
 * refs such as constness shared with the generic AST.
 *
 * history:
 *  - cst_php.ml (was actually called ast_php.ml)
 *  - ast_php.ml (was called ast_php_simple.ml)
 *  - pil.ml, still for PHP
 *  - IL.ml for AST generic
 *
 * related work:
 *  - CIL, C Intermediate Language, Necula et al, CC'00
 *  - RIL, The Ruby Intermediate Language, Furr et al, DLS'09
 *  - SIL? in Infer? or more a generic AST than a generic IL?
 *  - Rust IL?
 *  - C-- in OCaml? too low-level?
 *  - LLVM IR (but too far away from original code? complicated
```

```
 *    source maps)
 *  - BRIL https://capra.cs.cornell.edu/bril/ used for teaching
 *  - gcc RTL (too low-level? similar to 3-address code?)
 *  - SiMPL language in BAP/BitBlaze dynamic analysis libraries
 *    but probably too close to assembly/bytecode
 *  - Jimpl in Soot/Wala
 *)

(*****************************************************************************)
(* Token (leaf) *)
(*****************************************************************************)
```

⟨*type* IL.tok 177a⟩

```
(* with tarzan *)
```

⟨*type* IL.wrap 177b⟩

```
(* with tarzan *)
```

⟨*type* IL.bracket 177c⟩

```
(* with tarzan *)

(*****************************************************************************)
(* Names *)
(*****************************************************************************)
```

⟨*type* IL.ident 177d⟩

```
(* with tarzan *)
```

⟨*type* IL.name 177e⟩

```
(* with tarzan *)

(*****************************************************************************)
(* Fixme constructs *)
(*****************************************************************************)

(* AST-to-IL translation is still a work in progress. When we encounter some
 * that we cannot handle, we insert a [FixmeExp], [FixmeInstr], or [FixmeStmt].
 *)

type fixme_kind =
  | ToDo (* some construct that we still don't support *)
  | Sgrep_construct (* some Sgrep construct shows up in the code, e.g. '...' *)
  | Impossible (* something we thought impossible happened *)
[@@deriving show]

(*****************************************************************************)
(* Lvalue *)
(*****************************************************************************)

(* An lvalue, represented as in CIL as a pair. *)
```

⟨*type* IL.lval 178a⟩

⟨*type* IL.base 178b⟩

⟨*type* IL.offset 178c⟩

```
(* transpile at some point? *)
```
⟨*type* IL.var_special 178d⟩

```
(*****************************************************************************)
(* Expression *)
(*****************************************************************************)

(* We use 'exp' instead of 'expr' to accentuate the difference
 * with AST_generic.expr.
 * Here 'exp' does not contain any side effect!
 *)
```
⟨*type* IL.exp 178e⟩
⟨*type* IL.exp_kind 179a⟩

⟨*type* IL.composite_kind 179b⟩
```
[@@deriving show { with_path = false }]

(* with tarzan *)
```

⟨*type* IL.argument 179c⟩

```
(* with tarzan *)

(*****************************************************************************)
(* Instruction *)
(*****************************************************************************)

(* Easier type to compute lvalue/rvalue set of a too general 'expr', which
 * is now split in  instr vs exp vs lval.
 *)
```
⟨*type* IL.instr 179d⟩
⟨*type* IL.instr_kind 179e⟩

⟨*type* IL.call_special 179f⟩
```
(* | IntAccess of composite_kind * int (* for tuples/array/list *)
   | StringAccess of string (* for records/hashes *)
*)
```

⟨*type* IL.anonymous_entity 180a⟩
```
[@@deriving show { with_path = false }]

(* with tarzan *)

(*****************************************************************************)
(* Statement *)
(*****************************************************************************)
```
⟨*type* IL.stmt 180b⟩
⟨*type* IL.stmt_kind 180c⟩

⟨*type* IL.other_stmt 180d⟩

⟨*type* IL.label 181a⟩
```
[@@deriving show { with_path = false }]

(* with tarzan *)

(*****************************************************************************)
(* Defs *)
(*****************************************************************************)
```

```
(* See AST_generic.ml *)

(*****************************************************************************)
(* Control-flow graph (CFG) *)
(*****************************************************************************)
(* Similar to controlflow.ml, but with a simpler node_kind.
 * See controlflow.ml for more information. *)
⟨type IL.node 183a⟩
⟨type IL.node_kind 183b⟩
[@@deriving show { with_path = false }]

(* with tarzan *)

⟨type IL.edge 183c⟩

⟨type IL.cfg 183d⟩

⟨type IL.nodei 184c⟩

(*****************************************************************************)
(* Any *)
(*****************************************************************************)
⟨type IL.any 181b⟩
[@@deriving show { with_path = false }]

(* with tarzan *)

(*****************************************************************************)
(* L/Rvalue helpers *)
(*****************************************************************************)
let lval_of_instr_opt x =
  match x.i with
  | Assign (lval, _)
  | AssignAnon (lval, _)
  | Call (Some lval, _, _)
  | CallSpecial (Some lval, _, _) ->
      Some lval
  | Call _ | CallSpecial _ -> None
  | FixmeInstr _ -> None

⟨function IL.lvar_of_instr_opt 191b⟩

⟨function IL.exps_of_instr 191c⟩
let rexps_of_instr x =
  match x.i with
  | Assign (_, exp) -> [ exp ]
  | AssignAnon _ -> []
  | Call (_, e1, args) -> e1 :: args
  | CallSpecial (_, _, args) -> args
  | FixmeInstr _ -> []

(* opti: could use a set *)
let rec lvals_of_exp e =
  match e.e with
  | Lvalue lval -> lval :: lvals_in_lval lval
  | Literal _ -> []
  | Cast (_, e) -> lvals_of_exp e
  | Composite (_, (_, xs, _)) | Operator (_, xs) -> lvals_of_exps xs
  | Record ys -> lvals_of_exps (ys |> List.map snd)
  | FixmeExp _ -> []
```

```
  and lvals_in_lval lval =
    let base_lvals =
      match lval.base with Mem e -> lvals_of_exp e | _else_ -> []
    in
    let offset_lvals =
      match lval.offset with Index e -> lvals_of_exp e | __else_ -> []
    in
    base_lvals @ offset_lvals

  and lvals_of_exps xs = xs |> List.map lvals_of_exp |> List.flatten

  (** The lvals in the RHS of the instruction. *)
  let rlvals_of_instr x =
    let exps = rexps_of_instr x in
    lvals_of_exps exps

  let rvars_of_instr x =
    x |> rlvals_of_instr
    |> Common.map_filter (function
         | { base = Var var; _ } -> Some var
         | ___else___ -> None)

  let rlvals_of_node = function
    | Enter | Exit | TrueNode | FalseNode | NGoto _ | Join -> []
    | NInstr x -> rlvals_of_instr x
    | NCond (_, e) | NReturn (_, e) | NThrow (_, e) -> lvals_of_exp e
    | NOther _ | NTodo _ -> []

  let lval_of_node_opt = function
    | NInstr x -> lval_of_instr_opt x
    | Enter | Exit | TrueNode | FalseNode | NGoto _ | Join | NCond _ | NReturn _
    | NThrow _ | NOther _ | NTodo _ ->
        None

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)
```

⟨*function* IL.str_of_name 177f⟩

⟨*function* IL.find_node 183e⟩

⟨*function* IL.find_exit 184a⟩
⟨*function* IL.find_enter 184b⟩

pfff/lang_GENERIC/analyze/Meta_IL.mli

⟨*signature* Meta_IL.short_string_of_node_kind 299a⟩≡ (299b)

```
  val short_string_of_node_kind: IL.node_kind -> string
```

⟨pfff/lang_GENERIC/analyze/Meta_IL.mli 299b⟩≡

⟨*signature* Meta_IL.vof_any 181c⟩

⟨*signature* Meta_IL.display_cfg 184d⟩

⟨*signature* Meta_IL.short_string_of_node_kind 299a⟩

pfff/lang_GENERIC/analyze/AST_to_IL.mli

⟨pfff/lang_GENERIC/analyze/AST_to_IL.mli 300a⟩≡

```
val function_definition :
  AST_generic.function_definition -> IL.name list * IL.stmt list

⟨signature AST_to_IL.stmt 182⟩

val name_of_entity : AST_generic.entity -> IL.name option
```

/home/pad/pfff/lang_GENERIC/analyze/AST_to_IL.ml

⟨type AST_to_IL.env 300b⟩≡ (302f)

```
type env = {
  (* stmts hidden inside expressions that we want to move out of 'exp',
   * usually simple Instr, but can be also If when handling Conditional expr.
   *)
  stmts : stmt list ref;
}
```

⟨function AST_to_IL.empty_env 300c⟩≡ (302f)

```
let empty_env () = { stmts = ref [] }
```

⟨function AST_to_IL.error 300d⟩≡ (302f)

⟨function AST_to_IL.warning 300e⟩≡ (302f)

⟨function AST_to_IL.error_any 300f⟩≡ (302f)

⟨function AST_to_IL.sgrep_construct 300g⟩≡ (302f)

```
let sgrep_construct any_generic = raise (Fixme (Sgrep_construct, any_generic))
```

⟨function AST_to_IL.todo 300h⟩≡ (302f)

```
let todo any_generic = raise (Fixme (ToDo, any_generic))
```

⟨function AST_to_IL.impossible 300i⟩≡ (302f)

```
let impossible any_generic = raise (Fixme (Impossible, any_generic))
```

⟨function AST_to_IL.fresh_var 300j⟩≡ (302f)

```
let fresh_var _env tok =
  let i = H.gensym () in
  (("_tmp", tok), i)
```

⟨function AST_to_IL._fresh_label 300k⟩≡ (302f)

```
let _fresh_label _env tok =
  let i = H.gensym () in
  (("_label", tok), i)
```

⟨function AST_to_IL.fresh_lval 300l⟩≡ (302f)

```
let fresh_lval env tok =
  let var = fresh_var env tok in
  { base = Var var; offset = NoOffset; constness = ref None }
```

⟨*function* AST_to_IL.lval_of_id_info 301a⟩≡ (302f)
```
let lval_of_id_info _env id id_info =
  let var = var_of_id_info id id_info in
  { base = Var var; offset = NoOffset; constness = id_info.id_constness }
```

⟨*function* AST_to_IL.lval_of_ent 301b⟩≡ (302f)
```
let lval_of_ent env ent =
  match ent.G.name with
  | G.EN (G.Id (id, idinfo)) -> lval_of_id_info env id idinfo
  | G.EN name -> lval env (G.N name)
  | G.EDynamic eorig -> lval env eorig
```

⟨*function* AST_to_IL.label_of_label 301c⟩≡ (302f)
```
(* TODO: should do first pass on body to get all labels and assign
 * a gensym to each.
 *)
let label_of_label _env lbl = (lbl, -1)
```

⟨*function* AST_to_IL.lookup_label 301d⟩≡ (302f)
```
let lookup_label _env lbl = (lbl, -1)
```

⟨*function* AST_to_IL.mk_e 301e⟩≡ (302f)
```
let mk_e e eorig = { e; eorig }
```

⟨*function* AST_to_IL.mk_i 301f⟩≡ (302f)
```
let mk_i i iorig = { i; iorig }
```

⟨*function* AST_to_IL.mk_s 301g⟩≡ (302f)
```
let mk_s s = { s }
```

⟨*function* AST_to_IL.add_instr 301h⟩≡ (302f)
```
let add_instr env instr = Common.push (mk_s (Instr instr)) env.stmts
```

⟨*function* AST_to_IL.add_stmt 301i⟩≡ (302f)
```
let add_stmt env st = Common.push st env.stmts
```

⟨*function* AST_to_IL.add_stmts 301j⟩≡ (302f)
```
let add_stmts env xs = xs |> List.iter (add_stmt env)
```

⟨*function* AST_to_IL.bracket_keep 301k⟩≡ (302f)
```
let bracket_keep f (t1, x, t2) = (t1, f x, t2)
```

⟨*constant* AST_to_IL.expr_orig 301l⟩≡ (302f)
```
(* just to ensure the code after does not call expr directly *)
let expr_orig = expr
```

⟨*function* AST_to_IL.expr 301m⟩≡ (302f)
```
let expr () = ()
```

⟨*function* AST_to_IL.expr_with_pre_stmts 302a⟩≡ (302f)
```
let expr_with_pre_stmts env e =
  ignore (expr ());
  let e = expr_orig env e in
  let xs = List.rev !(env.stmts) in
  env.stmts := [];
  (xs, e)
```

⟨*function* AST_to_IL.expr_with_pre_stmts_opt 302b⟩≡ (302f)
```
let expr_with_pre_stmts_opt env eopt =
  match eopt with
  | None -> ([], expr_opt env None)
  | Some e -> expr_with_pre_stmts env e
```

⟨*function* AST_to_IL.for_var_or_expr_list 302c⟩≡ (302f)
```
let for_var_or_expr_list env xs =
  xs
  |> List.map (function
       | G.ForInitExpr e ->
           let ss, _eIGNORE = expr_with_pre_stmts env e in
           ss
       | G.ForInitVar (ent, vardef) -> (
           (* copy paste of VarDef case in stmt *)
           match vardef with
           | { G.vinit = Some e; vtype = _typTODO } ->
               let ss, e' = expr_with_pre_stmts env e in
               let lv = lval_of_ent env ent in
               ss @ [ mk_s (Instr (mk_i (Assign (lv, e')) e)) ]
           | _ -> [] ))
  |> List.flatten
```

⟨*function* AST_to_IL.stmt 302d⟩≡ (302f)
```
and stmt env st =
  try stmt_aux env st
  with Fixme (kind, any_generic) -> fixme_stmt kind any_generic
```

⟨*function* AST_to_IL.stmt (/home/pad/pfff/lang_GENERIC/analyze/AST_to_IL.ml) 302e⟩≡ (302f)
```
let stmt st =
  let env = empty_env () in
  stmt env st
```

⟨pfff/lang_GENERIC/analyze/AST_to_IL.ml 302f⟩≡
```
(* Yoann Padioleau
 *
 * Copyright (C) 2020 r2c
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * version 2.1 as published by the Free Software Foundation, with the
 * special exception on linking described in file license.txt.
 *
 * This library 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.
```

```
 *)
open Common
open IL
module G = AST_generic
module H = AST_generic_helpers

[@@@warning "-40-42"]

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* AST generic to IL translation.
 *
 * todo:
 *  - a lot ...
 *)

let logger = Logging.get_logger [ __MODULE__ ]

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
⟨type AST_to_IL.env 300b⟩

⟨function AST_to_IL.empty_env 300c⟩

(*****************************************************************************)
(* Error management *)
(*****************************************************************************)

exception Fixme of fixme_kind * G.any

⟨function AST_to_IL.error 300d⟩
⟨function AST_to_IL.warning 300e⟩
⟨function AST_to_IL.error_any 300f⟩

⟨function AST_to_IL.sgrep_construct 300g⟩

⟨function AST_to_IL.todo 300h⟩

⟨function AST_to_IL.impossible 300i⟩

let locate opt_tok s =
  let opt_loc =
    try map_opt Parse_info.string_of_info opt_tok
    with Parse_info.NoTokenLocation _ -> None
  in
  match opt_loc with Some loc -> spf "%s: %s" loc s | None -> s

let log_warning opt_tok msg = logger#warning "%s" (locate opt_tok msg)

let log_error opt_tok msg = logger#error "%s" (locate opt_tok msg)

let log_fixme kind gany =
  let toks = Visitor_AST.ii_of_any gany in
  let opt_tok = Common2.hd_opt toks in
  match kind with
  | ToDo ->
      log_warning opt_tok
        "Unsupported construct(s) may affect the accuracy of dataflow analyses"
```

```
  | Sgrep_construct ->
      log_error opt_tok "Cannot translate Semgrep construct(s) into IL"
  | Impossible ->
      log_error opt_tok "Impossible happened during AST-to-IL translation"

let fixme_exp kind gany eorig =
  log_fixme kind (G.E eorig);
  { e = FixmeExp (kind, gany); eorig }

let fixme_instr kind gany eorig =
  log_fixme kind (G.E eorig);
  { i = FixmeInstr (kind, gany); iorig = eorig }

let fixme_stmt kind gany =
  log_fixme kind gany;
  [ { s = FixmeStmt (kind, gany) } ]

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨function AST_to_IL.fresh_var 300j⟩
⟨function AST_to_IL._fresh_label 300k⟩
⟨function AST_to_IL.fresh_lval 300l⟩

let var_of_id_info id id_info =
  let sid =
    match !(id_info.G.id_resolved) with
    | Some (_resolved, sid) -> sid
    | None ->
        let id_str, id_tok = id in
        let msg = spf "the ident '%s' is not resolved" id_str in
        log_warning (Some id_tok) msg;
        -1
  in
  (id, sid)

⟨function AST_to_IL.lval_of_id_info 301a⟩

let lval_of_id_qualified env name id_info =
  let id, _ = name in
  lval_of_id_info env id id_info

let lval_of_base base = { base; offset = NoOffset; constness = ref None }

⟨function AST_to_IL.label_of_label 301c⟩
⟨function AST_to_IL.lookup_label 301d⟩

⟨function AST_to_IL.mk_e 301e⟩
⟨function AST_to_IL.mk_i 301f⟩
⟨function AST_to_IL.mk_s 301g⟩

⟨function AST_to_IL.add_instr 301h⟩
⟨function AST_to_IL.add_stmt 301i⟩
⟨function AST_to_IL.add_stmts 301j⟩

⟨function AST_to_IL.bracket_keep 301k⟩

let name_of_entity ent =
  match AST_generic_helpers.name_of_entity ent with
  | Some (i, pinfo) ->
```

```
      let name = var_of_id_info i pinfo in
      Some name
  | _____else_____ -> None

(*****************************************************************************)
(* lvalue *)
(*****************************************************************************)
let rec lval env eorig =
  match eorig with
  | G.N n -> name env n
  | G.IdSpecial (G.This, tok) ->
      { base = VarSpecial (This, tok); offset = NoOffset; constness = ref None }
  | G.DotAccess (e1orig, tok, field) -> (
      let base, base_constness = nested_lval env tok e1orig in
      match field with
      | G.EN (G.Id (id, idinfo)) ->
          {
            base;
            offset = Dot (var_of_id_info id idinfo);
            constness = idinfo.id_constness;
          }
      | G.EN name ->
          let attr = expr env (G.N name) in
          { base; offset = Index attr; constness = base_constness }
      | G.EDynamic e2orig ->
          let attr = expr env e2orig in
          { base; offset = Index attr; constness = base_constness } )
  | G.ArrayAccess (e1orig, (_, e2orig, _)) ->
      let tok = G.fake "[]" in
      let base, constness = nested_lval env tok e1orig in
      let e2 = expr env e2orig in
      { base; offset = Index e2; constness }
  | G.DeRef (_, e1orig) ->
      let e1 = expr env e1orig in
      lval_of_base (Mem e1)
  | _ -> todo (G.E eorig)

and name env = function
  | G.Id (("_", tok), _) ->
      (* wildcard *)
      fresh_lval env tok
  | G.Id (id, id_info) ->
      let lval = lval_of_id_info env id id_info in
      lval
  | G.IdQualified (name, id_info) ->
      let lval = lval_of_id_qualified env name id_info in
      lval

and nested_lval env tok eorig =
  let lval = lval env eorig in
  let base =
    match lval.offset with
    | NoOffset -> lval.base
    | _ ->
        let fresh = fresh_lval env tok in
        let lvalexp = mk_e (Lvalue lval) eorig in
        add_instr env (mk_i (Assign (fresh, lvalexp)) eorig);
        fresh.base
  in
  (base, lval.constness)
```

```
(*****************************************************************************)
(* Pattern *)
(*****************************************************************************)
and pattern env pat =
  match pat with
  | G.PatId (id, id_info) -> Left (lval_of_id_info env id id_info)
  | _ -> todo (G.P pat)

and pattern_assign_statements env exp eorig pat =
  let lval =
    match pattern env pat with Left l -> l | Right _ -> todo (G.P pat)
  in
  [ mk_s (Instr (mk_i (Assign (lval, exp)) eorig)) ]

(*****************************************************************************)
(* Assign *)
(*****************************************************************************)
and assign env lhs _tok rhs_exp eorig =
  match lhs with
  | G.N _ | G.DotAccess _ | G.ArrayAccess _ | G.DeRef _ -> (
      try
        let lval = lval env lhs in
        add_instr env (mk_i (Assign (lval, rhs_exp)) eorig);
        mk_e (Lvalue lval) lhs
      with Fixme (kind, any_generic) ->
        add_instr env (fixme_instr kind any_generic eorig);
        fixme_exp kind any_generic lhs )
  | G.Tuple (tok1, lhss, tok2) ->
      (* E1, ..., En = RHS *)
      (* tmp = RHS*)
      let tmp = fresh_var env tok2 in
      let tmp_lval = lval_of_base (Var tmp) in
      add_instr env (mk_i (Assign (tmp_lval, rhs_exp)) eorig);
      (* Ei = tmp[i] *)
      let tup_elems =
        lhss
        |> List.mapi (fun i lhs_i ->
               let index_i = Literal (G.Int (Some i, tok1)) in
               let offset_i = Index { e = index_i; eorig } in
               let lval_i =
                 { base = Var tmp; offset = offset_i; constness = ref None }
               in
               assign env lhs_i tok1 { e = Lvalue lval_i; eorig } eorig)
      in
      (* (E1, ..., En) *)
      mk_e (Composite (CTuple, (tok1, tup_elems, tok2))) eorig
  | _ ->
      add_instr env (fixme_instr ToDo (G.E eorig) eorig);
      fixme_exp ToDo (G.E eorig) lhs

(*****************************************************************************)
(* Expression *)
(*****************************************************************************)
(* less: we could pass in an optional lval that we know the caller want
 * to assign into, which would avoid creating useless fresh_var intermediates.
 *)
and expr_aux env eorig =
  match eorig with
  | G.Call (G.IdSpecial (G.Op op, tok), args) ->
```

```
    let args = arguments env args in
    mk_e (Operator ((op, tok), args)) eorig
| G.Call
    ((G.IdSpecial ((G.This | G.Super | G.Self | G.Parent), tok) as e), args)
  ->
    call_generic env tok e args
| G.Call (G.IdSpecial (G.IncrDecr (incdec, _prepostIGNORE), tok), args) -> (
    (* in theory in expr() we should return each time a list of pre-instr
     * and a list of post-instrs to execute before and after the use
     * of the expression. However this complicates the interface of 'expr()'.
     * Right now, for the pre-instr we agglomerate them instead in env
     * and use them in 'expr_with_pre_instr()' below, but for the post
     * we dont. Anyway, for our static analysis purpose it should not matter.
     * We don't do fancy path-sensitive-evaluation-order-sensitive analysis.
     *)
    match G.unbracket args with
    | [ G.Arg e ] ->
        let lval = lval env e in
        let lvalexp = mk_e (Lvalue lval) e in
        let op =
          ((match incdec with G.Incr -> G.Plus | G.Decr -> G.Minus), tok)
        in
        let one = G.Int (Some 1, tok) in
        let one_exp = mk_e (Literal one) (G.L one) in
        let opexp = mk_e (Operator (op, [ lvalexp; one_exp ])) eorig in
        add_instr env (mk_i (Assign (lval, opexp)) eorig);
        lvalexp
    | _ -> impossible (G.E eorig) )
(* todo: if the xxx_to_generic forgot to generate Eval *)
| G.Call
    ( G.N (G.Id (("eval", tok), { G.id_resolved = { contents = None }; _ })),
      args ) ->
    let lval = fresh_lval env tok in
    let special = (Eval, tok) in
    let args = arguments env args in
    add_instr env (mk_i (CallSpecial (Some lval, special, args)) eorig);
    mk_e (Lvalue lval) eorig
| G.Call (G.IdSpecial spec, args) ->
    let tok = snd spec in
    let lval = fresh_lval env tok in
    let special = call_special env spec in
    let args = arguments env args in
    add_instr env (mk_i (CallSpecial (Some lval, special, args)) eorig);
    mk_e (Lvalue lval) eorig
| G.Call (e, args) ->
    let tok = G.fake "call" in
    call_generic env tok e args
| G.L lit -> mk_e (Literal lit) eorig
| G.N _ | G.DotAccess (_, _, _) | G.ArrayAccess (_, _) | G.DeRef (_, _) ->
    let lval = lval env eorig in
    mk_e (Lvalue lval) eorig
| G.Assign (e1, tok, e2) ->
    let exp = expr env e2 in
    assign env e1 tok exp eorig
| G.AssignOp (e1, (G.Eq, tok), e2) when Parse_info.str_of_info tok = ":=" ->
    (* We encode Go's `:=` as `AssignOp(Eq)`,
     * see "go_to_generic.ml" in Pfff. *)
    let exp = expr env e2 in
    assign env e1 tok exp eorig
| G.AssignOp (e1, op, e2) ->
```

```
    let exp = expr env e2 in
    let lval = lval env e1 in
    let lvalexp = mk_e (Lvalue lval) e1 in
    let opexp = mk_e (Operator (op, [ lvalexp; exp ])) eorig in
    add_instr env (mk_i (Assign (lval, opexp)) eorig);
    lvalexp
| G.Seq xs -> (
    match List.rev xs with
    | [] -> impossible (G.E eorig)
    | last :: xs ->
        let xs = List.rev xs in
        xs
        |> List.iter (fun e ->
               let _eIGNORE = expr env e in
               ());
        expr env last )
| G.Container (kind, xs) ->
    let xs = bracket_keep (List.map (expr env)) xs in
    let kind = composite_kind kind in
    mk_e (Composite (kind, xs)) eorig
| G.Tuple xs ->
    let xs = bracket_keep (List.map (expr env)) xs in
    mk_e (Composite (CTuple, xs)) eorig
| G.Record fields -> record env fields
| G.Lambda def ->
    (* TODO: we should have a use def.f_tok *)
    let tok = G.fake "lambda" in
    let lval = fresh_lval env tok in
    add_instr env (mk_i (AssignAnon (lval, Lambda def)) eorig);
    mk_e (Lvalue lval) eorig
| G.AnonClass def ->
    (* TODO: should use def.ckind *)
    let tok = Common2.fst3 def.G.cbody in
    let lval = fresh_lval env tok in
    add_instr env (mk_i (AssignAnon (lval, AnonClass def)) eorig);
    mk_e (Lvalue lval) eorig
| G.IdSpecial (spec, tok) -> (
    let opt_var_special =
      match spec with
      | G.This -> Some This
      | G.Super -> Some Super
      | G.Self -> Some Self
      | G.Parent -> Some Parent
      | _ -> None
    in
    match opt_var_special with
    | Some var_special ->
        let lval = lval_of_base (VarSpecial (var_special, tok)) in
        mk_e (Lvalue lval) eorig
    | None -> impossible (G.E eorig) )
| G.SliceAccess (_, _) -> todo (G.E eorig)
(* e1 ? e2 : e3 ==>
 *  pre: lval = e1;
 *       if(lval) { lval = e2 } else { lval = e3 }
 *  exp: lval
 *)
| G.Conditional (e1orig, e2orig, e3orig) ->
    let tok = G.fake "conditional" in
    let lval = fresh_lval env tok in
    let lvalexp = mk_e (Lvalue lval) e1orig in
```

```
      (* not sure this is correct *)
      let before = List.rev !(env.stmts) in
      env.stmts := [];
      let e1 = expr env e1orig in
      let ss_for_e1 = List.rev !(env.stmts) in
      env.stmts := [];
      let e2 = expr env e2orig in
      let ss_for_e2 = List.rev !(env.stmts) in
      env.stmts := [];
      let e3 = expr env e3orig in
      let ss_for_e3 = List.rev !(env.stmts) in
      env.stmts := [];

      add_stmts env before;
      add_stmts env ss_for_e1;
      add_stmt env
        (mk_s
           (If
              ( tok,
                e1,
                ss_for_e2 @ [ mk_s (Instr (mk_i (Assign (lval, e2)) e2orig)) ],
                ss_for_e3 @ [ mk_s (Instr (mk_i (Assign (lval, e3)) e3orig)) ]
              )));
      lvalexp
  | G.Xml _ -> todo (G.E eorig)
  | G.Constructor (_, _) | G.LetPattern (_, _) | G.MatchPattern (_, _) ->
      todo (G.E eorig)
  | G.Yield (_, _, _) | G.Await (_, _) -> todo (G.E eorig)
  | G.Cast (typ, e) ->
      let e = expr env e in
      mk_e (Cast (typ, e)) eorig
  | G.Ref (_, _) -> todo (G.E eorig)
  | G.Ellipsis _
  | G.TypedMetavar (_, _, _)
  | G.DisjExpr (_, _)
  | G.DeepEllipsis _ | G.DotAccessEllipsis _ ->
      sgrep_construct (G.E eorig)
  | G.OtherExpr (_, _) -> todo (G.E eorig)

and expr env eorig =
  try expr_aux env eorig
  with Fixme (kind, any_generic) -> fixme_exp kind any_generic eorig

and expr_opt env = function
  | None ->
      let void = G.Unit (G.fake "void") in
      mk_e (Literal void) (G.L void)
  | Some e -> expr env e

and call_generic env tok e args =
  let eorig = G.Call (e, args) in
  let e = expr env e in
  (* In theory, instrs in args could have side effect on the value in 'e',
   * but we will agglomerate all those instrs in the environment and
   * the caller will call them in sequence (see expr_with_pre_instr).
   * In theory, we should not execute those instrs before getting the
   * value in 'e' in the caller, but for our static analysis purpose
   * we should not care about those edge cases. That would require
   * to return in expr multiple arguments and thread things around; Not
```

```
   * worth it.
   *)
  let args = arguments env args in
  let lval = fresh_lval env tok in
  add_instr env (mk_i (Call (Some lval, e, args)) eorig);
  mk_e (Lvalue lval) eorig

and call_special _env (x, tok) =
  ( ( match x with
    | G.Op _ | G.IncrDecr _ ->
        impossible (G.E (G.IdSpecial (x, tok)))
        (* should be intercepted before *)
    | G.This | G.Super | G.Self | G.Parent ->
        impossible (G.E (G.IdSpecial (x, tok)))
        (* should be intercepted before *)
    | G.Eval -> Eval
    | G.Typeof -> Typeof
    | G.Instanceof -> Instanceof
    | G.Sizeof -> Sizeof
    | G.New -> New
    | G.ConcatString _kindopt -> Concat
    | G.Spread -> Spread
    | G.EncodedString _ | G.Defined | G.HashSplat | G.ForOf | G.NextArrayIndex
    | G.InterpolatedElement ->
        todo (G.E (G.IdSpecial (x, tok))) ),
    tok )

and composite_kind = function
  | G.Array -> CArray
  | G.List -> CList
  | G.Dict -> CDict
  | G.Set -> CSet

(* TODO: dependency of order between arguments for instr? *)
and arguments env xs = xs |> G.unbracket |> List.map (argument env)

and argument env arg =
  match arg with
  | G.Arg e -> expr env e
  | G.ArgKwd (_, e) ->
      (* TODO: Handle the keyword/label somehow (when relevant). *)
      expr env e
  | _ -> todo (G.Ar arg)

and record env ((_tok, origfields, _) as record_def) =
  let eorig = G.Record record_def in
  let fields =
    origfields
    |> List.map (function
         | G.FieldStmt
             {
               s =
                 G.DefStmt
                   ({ G.name = G.EN (G.Id (id, _)); tparams = []; _ }, def_kind);
               _;
             } ->
             let fdeforig =
               match def_kind with
               (* TODO: Consider what to do with vtype. *)
               | G.VarDef { G.vinit = Some fdeforig; _ }
```

```
              | G.FieldDefColon { G.vinit = Some fdeforig; _ } ->
                  fdeforig
              | ___else___ -> todo (G.E eorig)
            in
            let field_def = expr env fdeforig in
            (id, field_def)
        | G.FieldStmt _ -> todo (G.E eorig)
        | G.FieldSpread _ -> todo (G.E eorig))
  in
  mk_e (Record fields) eorig

(*****************************************************************************)
(* Exprs and instrs *)
(*****************************************************************************)
```

⟨*function* AST_to_IL.lval_of_ent 301b⟩

⟨*constant* AST_to_IL.expr_orig 301l⟩
⟨*function* AST_to_IL.expr 301m⟩

⟨*function* AST_to_IL.expr_with_pre_stmts 302a⟩

⟨*function* AST_to_IL.expr_with_pre_stmts_opt 302b⟩

⟨*function* AST_to_IL.for_var_or_expr_list 302c⟩

```
(*****************************************************************************)
(* Parameters *)
(*****************************************************************************)

let parameters _env params =
  params
  |> List.filter_map (function
       | G.ParamClassic { pname = Some i; pinfo; _ } ->
           Some (var_of_id_info i pinfo)
       | ___else___ -> None (* TODO *))

(*****************************************************************************)
(* Statement *)
(*****************************************************************************)
let rec stmt_aux env st =
  match st.G.s with
  | G.ExprStmt (e, _) ->
      (* optimize? pass context to expr when no need for return value? *)
      let ss, _eIGNORE = expr_with_pre_stmts env e in
      ss
  | G.DefStmt (ent, G.VarDef { G.vinit = Some e; vtype = _typTODO }) ->
      let ss, e' = expr_with_pre_stmts env e in
      let lv = lval_of_ent env ent in
      ss @ [ mk_s (Instr (mk_i (Assign (lv, e')) e)) ]
  | G.DefStmt def -> [ mk_s (MiscStmt (DefStmt def)) ]
  | G.DirectiveStmt dir -> [ mk_s (MiscStmt (DirectiveStmt dir)) ]
  | G.Block xs -> xs |> G.unbracket |> List.map (stmt env) |> List.flatten
  | G.If (tok, e, st1, st2) ->
      let ss, e' = expr_with_pre_stmts env e in
      let st1 = stmt env st1 in
      let st2 =
        List.map (stmt env) (st2 |> Common.opt_to_list) |> List.flatten
      in
      ss @ [ mk_s (If (tok, e', st1, st2)) ]
```

```
| G.Switch (_, _, _) -> todo (G.S st)
| G.While (tok, e, st) ->
    let ss, e' = expr_with_pre_stmts env e in
    let st = stmt env st in
    ss @ [ mk_s (Loop (tok, e', st @ ss)) ]
| G.DoWhile (tok, st, e) ->
    let st = stmt env st in
    let ss, e' = expr_with_pre_stmts env e in
    st @ ss @ [ mk_s (Loop (tok, e', st @ ss)) ]
| G.For (tok, G.ForEach (pat, tok2, e), st) ->
    let ss, e' = expr_with_pre_stmts env e in
    let st = stmt env st in

    let next_lval = fresh_lval env tok2 in
    let hasnext_lval = fresh_lval env tok2 in
    let hasnext_call =
      mk_s
        (Instr
           (mk_i
              (CallSpecial (Some hasnext_lval, (ForeachHasNext, tok2), [ e' ]))
              e))
    in
    let next_call =
      mk_s
        (Instr
           (mk_i
              (CallSpecial (Some next_lval, (ForeachNext, tok2), [ e' ]))
              e))
    in
    (* same semantic? or need to take Ref? or pass lval
     * directly in next_call instead of using intermediate next_lval?
     *)
    let assign =
      pattern_assign_statements env (mk_e (Lvalue next_lval) e) e pat
    in
    let cond = mk_e (Lvalue hasnext_lval) e in

    (ss @ [ hasnext_call ])
    @ [
        mk_s
          (Loop
             ( tok,
               cond,
               [ next_call ] @ assign @ st @ [ (* ss @ ?*) hasnext_call ] ));
      ]
| G.For (tok, G.ForClassic (xs, eopt1, eopt2), st) ->
    let ss1 = for_var_or_expr_list env xs in
    let st = stmt env st in
    let ss2, cond =
      match eopt1 with
      | None ->
          let vtrue = G.Bool (true, tok) in
          ([], mk_e (Literal vtrue) (G.L vtrue))
      | Some e -> expr_with_pre_stmts env e
    in
    let next =
      match eopt2 with
      | None -> []
      | Some e ->
          let ss, _eIGNORE = expr_with_pre_stmts env e in
```

```
          ss
      in
      ss1 @ ss2 @ [ mk_s (Loop (tok, cond, st @ next @ ss2)) ]
  | G.For (_, G.ForEllipsis _, _) -> sgrep_construct (G.S st)
  | G.For (tok, G.ForIn (xs, e), st) ->
      let ss1 = for_var_or_expr_list env xs in
      let st = stmt env st in
      let ss2, cond = expr_with_pre_stmts env (List.nth e 0) (* TODO list *) in
      ss1 @ ss2 @ [ mk_s (Loop (tok, cond, st @ ss2)) ]
  (* TODO: repeat env work of controlflow_build.ml *)
  | G.Continue _ | G.Break _ -> todo (G.S st)
  | G.Label (lbl, st) ->
      let lbl = label_of_label env lbl in
      let st = stmt env st in
      [ mk_s (Label lbl) ] @ st
  | G.Goto (tok, lbl) ->
      let lbl = lookup_label env lbl in
      [ mk_s (Goto (tok, lbl)) ]
  | G.Return (tok, eopt, _) ->
      let ss, e = expr_with_pre_stmts_opt env eopt in
      ss @ [ mk_s (Return (tok, e)) ]
  | G.Assert (tok, e, eopt, _) ->
      let ss1, e' = expr_with_pre_stmts env e in
      let ss2, eopt' = expr_with_pre_stmts_opt env eopt in
      let special = (Assert, tok) in
      (* less: wrong e? would not be able to match on Assert, or
       * need add sorig:
       *)
      ss1 @ ss2
      @ [ mk_s (Instr (mk_i (CallSpecial (None, special, [ e'; eopt' ])) e)) ]
  | G.Throw (tok, e, _) ->
      let ss, e = expr_with_pre_stmts env e in
      ss @ [ mk_s (Throw (tok, e)) ]
  | G.Try (_tok, try_st, catches, opt_finally) ->
      let try_stmt = stmt env try_st in
      let catches_stmt_rev =
        List.fold_left
          (fun acc (ctok, pattern, catch_st) ->
            (* TODO: Handle pattern properly. *)
            let name = fresh_var env ctok in
            let todo_pattern = fixme_stmt ToDo (G.P pattern) in
            let catch_stmt = stmt env catch_st in
            (name, todo_pattern @ catch_stmt) :: acc)
          [] catches
      in
      let finally_stmt =
        match opt_finally with
        | None -> []
        | Some (_tok, finally_st) -> stmt env finally_st
      in
      [ mk_s (Try (try_stmt, List.rev catches_stmt_rev, finally_stmt)) ]
  | G.WithUsingResource (_, stmt1, stmt2) ->
      let stmt1 = stmt env stmt1 in
      let stmt2 = stmt env stmt2 in
      stmt1 @ stmt2
  | G.DisjStmt _ -> sgrep_construct (G.S st)
  | G.OtherStmt _ | G.OtherStmtWithStmt _ -> todo (G.S st)
```

⟨*function* AST_to_IL.stmt 302d⟩

```
(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)

let function_definition def =
  let env = empty_env () in
  let params = parameters env def.G.fparams in
  let body = stmt env def.G.fbody in
  (params, body)
```

⟨*function* AST_to_IL.stmt (/home/pad/pfff/lang_GENERIC/analyze/AST_to_IL.ml) 302e⟩

pfff/lang_GENERIC/analyze/CFG_build.mli

⟨pfff/lang_GENERIC/analyze/CFG_build.mli 314a⟩≡

⟨*signature* CFG_build.cfg_of_stmts 185⟩

/home/pad/pfff/lang_GENERIC/analyze/CFG_build.ml

⟨*type* CFG_build.state 314b⟩≡ (314e)

```
(* Information passed recursively in stmt or stmt_list below.
 * The graph g is mutable, so most of the work is done by side effects on it.
 * No need to return a new state.
 *)
type state = {
  g : F.cfg;
  (* When there is a 'return' we need to know the exit node to link to *)
  exiti : F.nodei;
  (* Attaches labels to nodes. *)
  labels : (label_key, F.nodei) Hashtbl.t;
  (* Gotos pending to be resolved. *)
  gotos : (nodei * label_key) list ref;
}
```

⟨*function* CFG_build.add_arc 314c⟩≡ (314e)

```
let add_arc (starti, nodei) g = g#add_arc ((starti, nodei), F.Direct)
```

⟨*function* CFG_build.add_arc_opt 314d⟩≡ (314e)

```
let add_arc_opt (starti_opt, nodei) g =
  starti_opt
  |> Common.do_option (fun starti -> g#add_arc ((starti, nodei), F.Direct))
```

⟨pfff/lang_GENERIC/analyze/CFG_build.ml 314e⟩≡

```
(* Yoann Padioleau
 *
 * Copyright (C) 2009, 2010, 2011 Facebook
 * Copyright (C) 2020 r2c
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * version 2.1 as published by the Free Software Foundation, with the
 * special exception on linking described in file license.txt.
 *
 * This library 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.
 *)
open IL
module F = IL (* to be even more similar to controlflow_build.ml *)

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Control-flow graph generation for the IL.
 *
 * This is mostly a copy-paste (with less cases) of controlflow_build.ml
 *
 * TODO:
 *  - factorize at some point with controlflow_build.ml?
 *  - remove controlflow.ml? now that we have the Il, maybe better to
 *    do any kind of cfg-based analysis on the IL rather than the generic AST.
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

(* Like IL.label but without the token attached to the ident, this is to allow
 * the label to be used as a key in a map or hash table.
 *)
type label_key = string * G.sid

⟨type CFG_build.state 314b⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function CFG_build.add_arc 314c⟩

⟨function CFG_build.add_arc_opt 314d⟩

let key_of_label ((str, _tok), sid) : label_key = (str, sid)

let add_pending_goto state gotoi label =
  state.gotos := (gotoi, key_of_label label) :: !(state.gotos)

let label_node state labels nodei =
  labels
  |> List.iter (fun label ->
         Hashtbl.add state.labels (key_of_label label) nodei)

let resolve_gotos state =
  !(state.gotos)
  |> List.iter (fun (srci, label_key) ->
         match Hashtbl.find_opt state.labels label_key with
         | None ->
             Common.pr2
             @@ Common.spf "Could not resolve label: %s" (fst label_key)
         | Some dsti -> state.g |> add_arc (srci, dsti));
  state.gotos := []

(*****************************************************************************)
```

```
(* Algorithm *)
(*****************************************************************************)

(* The CFG building algorithm works by iteratively visiting the
 * statements in the AST of a function. At each statement,
 * the cfg_stmt function is called, and passed the index of the;
 * previous node (if there is one), and returns a cfg_stmt_result.
 *
 * Function cfg_stmt_list is the one responsible for labeling nodes.
 * We do everything in one pass by collecting the list of gotos and
 * resolving them at the end. Alternatively, we could do it in two
 * passes, with the first pass doing the labeling work.
 *
 * history:
 * - ver1: old code was returning a nodei, but break has no end, so
 *   cfg_stmt should return a nodei option.
 * - ver2: old code was taking a nodei, but should also take a nodei
 *   option. There can be deadcode in the function.
 * - ver3: In order to handle labels/gotos the now return either a
 *   label or a pair nodei * nodei option (entry and exit).
 *
 * subtle: try/throw. The current algo is not very precise, but
 * it's probably good enough for many analysis.
 *)

type cfg_stmt_result =
  (* A label for a label statement. *)
  | CfgLabel of label
  (* The fist (entry) and last (exit) node of the created CFG.
   * Last node is optional; it is None when the execution will not
   * continue (return), or when it may continue with a different
   * statement than the subsequent one (goto).
   *)
  | CfgFirstLast of F.nodei * F.nodei option

let rec cfg_stmt : state -> F.nodei option -> stmt -> cfg_stmt_result =
 fun state previ stmt ->
  match stmt.s with
  | Instr x ->
      let newi = state.g#add_node { F.n = F.NInstr x } in
      state.g |> add_arc_opt (previ, newi);
      CfgFirstLast (newi, Some newi)
  | If (tok, e, st1, st2) -> (
      (* previ -> newi --->  newfakethen -> ... -> finalthen --> lasti -> <rest>
       *               |                                    |
       *               |->  newfakeelse -> ... -> finalelse -|
       *
       * The lasti can be a Join when there is no return in either branch.
       *)
      let newi = state.g#add_node { F.n = F.NCond (tok, e) } in
      state.g |> add_arc_opt (previ, newi);

      let newfakethen = state.g#add_node { F.n = F.TrueNode } in
      let newfakeelse = state.g#add_node { F.n = F.FalseNode } in
      state.g |> add_arc (newi, newfakethen);
      state.g |> add_arc (newi, newfakeelse);

      let finalthen = cfg_stmt_list state (Some newfakethen) st1 in
      let finalelse = cfg_stmt_list state (Some newfakeelse) st2 in
```

```
    match (finalthen, finalelse) with
    | None, None ->
        (* probably a return in both branches *)
        CfgFirstLast (newi, None)
    | Some nodei, None | None, Some nodei -> CfgFirstLast (newi, Some nodei)
    | Some n1, Some n2 ->
        let lasti = state.g#add_node { F.n = F.Join } in
        state.g |> add_arc (n1, lasti);
        state.g |> add_arc (n2, lasti);
        CfgFirstLast (newi, Some lasti) )
| Loop (tok, e, st) ->
    (* previ -> newi ---> newfakethen -> ... -> finalthen -
     *             |---|-----------------------------------|
     *                 |-> newfakelse
     *)
    let newi = state.g#add_node { F.n = NCond (tok, e) } in
    state.g |> add_arc_opt (previ, newi);

    let newfakethen = state.g#add_node { F.n = F.TrueNode } in
    let newfakeelse = state.g#add_node { F.n = F.FalseNode } in
    state.g |> add_arc (newi, newfakethen);
    state.g |> add_arc (newi, newfakeelse);

    let finalthen = cfg_stmt_list state (Some newfakethen) st in
    state.g |> add_arc_opt (finalthen, newi);
    CfgFirstLast (newi, Some newfakeelse)
| Label label -> CfgLabel label
| Goto (tok, label) ->
    let newi = state.g#add_node { F.n = F.NGoto (tok, label) } in
    state.g |> add_arc_opt (previ, newi);
    add_pending_goto state newi label;
    CfgFirstLast (newi, None)
| Return (tok, e) ->
    let newi = state.g#add_node { F.n = F.NReturn (tok, e) } in
    state.g |> add_arc_opt (previ, newi);
    state.g |> add_arc (newi, state.exiti);
    (* the next statement if there is one will not be linked to
     * this new node *)
    CfgFirstLast (newi, None)
| Try (try_st, catches, finally_st) ->
    (* TODO: This is not a proper CFG for try-catch-finally... but
     * it's probably "good enough" for now! *)
    (* previ -> newi -> try --> catch1 -|
     *                      |->  ...   -|
     *                      |-> catchN -|
     *                      |-----------|-> newfakefinally -> finally
     *)
    let newi = state.g#add_node { F.n = F.TrueNode } in
    state.g |> add_arc_opt (previ, newi);
    let finaltry = cfg_stmt_list state (Some newi) try_st in
    let newfakefinally = state.g#add_node { F.n = F.TrueNode } in
    state.g |> add_arc_opt (finaltry, newfakefinally);
    catches
    |> List.iter (fun (_, catch_st) ->
           let finalcatch = cfg_stmt_list state finaltry catch_st in
           state.g |> add_arc_opt (finalcatch, newfakefinally));
    let finalfinally = cfg_stmt_list state (Some newfakefinally) finally_st in
    CfgFirstLast (newi, finalfinally)
| Throw (_, _) -> cfg_todo state previ stmt
| MiscStmt x ->
```

```
      let newi = state.g#add_node { F.n = F.NOther x } in
      state.g |> add_arc_opt (previ, newi);
      CfgFirstLast (newi, Some newi)
  | FixmeStmt _ -> cfg_todo state previ stmt

and cfg_todo state previ stmt =
  let newi = state.g#add_node { F.n = F.NTodo stmt } in
  state.g |> add_arc_opt (previ, newi);
  CfgFirstLast (newi, Some newi)

and cfg_stmt_list state previ xs =
  let lasti_opt =
    xs
    |> List.fold_left
         (fun (previ, labels) stmt ->
           (* We don't create special nodes for labels in the CFG; instead,
            * we assign them to the entry nodes of the labeled statements.
            *)
           match cfg_stmt state previ stmt with
           | CfgFirstLast (firsti, lasti) ->
               label_node state labels firsti;
               (lasti, [])
           | CfgLabel label -> (previ, label :: labels))
         (previ, [])
    |> fst
  in
  lasti_opt

(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)

let (cfg_of_stmts : stmt list -> F.cfg) =
 fun xs ->
  (* yes, I sometimes use objects, and even mutable objects in OCaml ... *)
  let g = new Ograph_extended.ograph_mutable in

  let enteri = g#add_node { F.n = F.Enter } in
  let exiti = g#add_node { F.n = F.Exit } in

  let newi = enteri in

  let state = { g; exiti; labels = Hashtbl.create 2; gotos = ref [] } in
  let last_node_opt = cfg_stmt_list state (Some newi) xs in
  (* Must wait until all nodes have been labeled before resolving gotos. *)
  resolve_gotos state;
  (* maybe the body does not contain a single 'return', so by default
   * connect last stmt to the exit node
   *)
  g |> add_arc_opt (last_node_opt, exiti);
  g
```

pfff/lang_GENERIC/analyze/Dataflow.mli

⟨pfff/lang_GENERIC/analyze/Dataflow.mli 318⟩≡
  ⟨*type* Dataflow.nodei 186a⟩

  ⟨*type* Dataflow.var 186b⟩

⟨*module* Dataflow.VarMap 186c⟩
⟨*module* Dataflow.VarSet 186d⟩

```
(* Return value of a dataflow analysis.
 * The array is indexed by nodei.
 *)
```
⟨*type* Dataflow.mapping 186e⟩
⟨*type* Dataflow.inout 186f⟩
⟨*type* Dataflow.env 186g⟩

⟨*signature* Dataflow.empty_env 186h⟩
⟨*signature* Dataflow.empty_inout 186i⟩

⟨*type* Dataflow.transfn 187a⟩

⟨*signature* Dataflow.varmap_union 188a⟩
⟨*signature* Dataflow.varmap_diff 188b⟩

```
(* common/useful 'a for mapping: a set of nodes (via their indices),
 * used for example in the reaching analysis.
 *)
```
⟨*module* Dataflow.NodeiSet 187d⟩
⟨*signature* Dataflow.union_env 188c⟩
⟨*signature* Dataflow.diff_env 188d⟩

⟨*signature* Dataflow.add_var_and_nodei_to_env 188e⟩
⟨*signature* Dataflow.add_vars_and_nodei_to_env 188f⟩

⟨*signature* Dataflow.ns_to_str 188g⟩

```
(* we use now a functor so we can reuse the same code for dataflow on
 * the IL (IL.cfg) or generic AST (Controlflow.flow)
 *)
```
⟨*module type* Dataflow.Flow 187b⟩
⟨*functor signature* Dataflow.Make 187c⟩

/home/pad/pfff/lang_GENERIC/analyze/Dataflow.ml

⟨*function* Dataflow.empty_env 319a⟩≡ (320c)
```
let empty_env () = VarMap.empty
```

⟨*function* Dataflow.empty_inout 319b⟩≡ (320c)
```
let empty_inout () = { in_env = empty_env (); out_env = empty_env () }
```

⟨*function* Dataflow.eq_env 319c⟩≡ (320c)
```
(* the environment is polymorphic, so we require to pass an eq for 'a *)
let eq_env eq env1 env2 = VarMap.equal eq env1 env2
```

⟨*function* Dataflow.eq_inout 319d⟩≡ (320c)
```
let eq_inout eq io1 io2 =
  let eqe = eq_env eq in
  eqe io1.in_env io2.in_env && eqe io1.out_env io2.out_env
```

⟨*function* Dataflow.csv_append 319e⟩≡ (320c)
```
let csv_append s v = if String.length s = 0 then v else s ^ "," ^ v
```

⟨*function* Dataflow.array_fold_left_idx 320a⟩≡ (320c)
```
let array_fold_left_idx f =
  let idx = ref 0 in
  Array.fold_left (fun v e ->
      let r = f v !idx e in
      incr idx;
      r)
```

⟨*function* Dataflow.ns_to_str 320b⟩≡ (320c)
```
let ns_to_str ns =
  "{" ^ NodeiSet.fold (fun n s -> csv_append s (string_of_int n)) ns "" ^ "}"
```

⟨pfff/lang_GENERIC/analyze/Dataflow.ml 320c⟩≡
```
(* Iain Proctor, Yoann Padioleau, Jiao Li
 *
 * Copyright (C) 2009-2010 Facebook
 * Copyright (C) 2019 r2c
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * version 2.1 as published by the Free Software Foundation, with the
 * special exception on linking described in file license.txt.
 *
 * This library 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.
 *)
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Dataflow analysis "framework".
 *
 * The goal of a dataflow analysis is to store information about each
 * variable at each program point, that is each node in a CFG
 * (e.g. whether a variable is "live" at a program point).
 * As you may want different kinds of information, the types below
 * are polymorphic. But each take as a key a variable name.
 *
 * todo:
 *  - could use a functor, so would not have all those 'a?
 *  - do we need other kind of information than variable environment?
 *    Dataflow analysis talks only about variables? for the belief analysis
 *    we actually want expressions instead.
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

(* I was using directly Controlflow.xxx before, but now that we have both
 * Controlflow.flow and Il.cfg, we need to functorize things.
 *)
module type Flow = sig
  type node
```

```
  type edge

  type flow = (node, edge) Ograph_extended.ograph_mutable

  val short_string_of_node : node -> string
end
```

⟨*type* Dataflow.nodei 186a⟩

⟨*type* Dataflow.var 186b⟩

```
(* convenient aliases *)
module VarMap = Map.Make (String)
module VarSet = Set.Make (String)
module NodeiSet = Set.Make (Int)

(* The final dataflow result; a map from each program point to a map containing
 * information for each variables.
 *
 * opti: this used to be a 'NodeiMap.t' instead of an 'array' but 'nodei'
 * are always int and array gives a 6x speedup according to Iain
 * so let's use array.
 *)
```
⟨*type* Dataflow.mapping 186e⟩

```
(* the In and Out sets, as in Appel Modern Compiler in ML book *)
```
⟨*type* Dataflow.inout 186f⟩
⟨*type* Dataflow.env 186g⟩

⟨*function* Dataflow.empty_env 319a⟩
⟨*function* Dataflow.empty_inout 319b⟩

```
(*****************************************************************************)
(* Equality *)
(*****************************************************************************)
```

⟨*function* Dataflow.eq_env 319c⟩

⟨*function* Dataflow.eq_inout 319d⟩

```
(*****************************************************************************)
(* Env manipulation *)
(*****************************************************************************)

let (varmap_union : ('a -> 'a -> 'a) -> 'a env -> 'a env -> 'a env) =
 fun union_op env1 env2 ->
  let acc = env1 in
  VarMap.fold
    (fun var x acc ->
      let x' = try union_op x (VarMap.find var acc) with Not_found -> x in
      VarMap.add var x' acc)
    env2 acc

let (varmap_diff :
      ('a -> 'a -> 'a) -> ('a -> bool) -> 'a env -> 'a env -> 'a env) =
 fun diff_op is_empty env1 env2 ->
  let acc = env1 in
  VarMap.fold
    (fun var x acc ->
      try
```

```
        let diff = diff_op (VarMap.find var acc) x in
        if is_empty diff then VarMap.remove var acc else VarMap.add var diff acc
      with Not_found -> acc)
    env2 acc

(* useful helpers when the environment maps to a set of Nodes, e.g.,
 * for reaching definitions.
 *)
let (union_env : NodeiSet.t env -> NodeiSet.t env -> NodeiSet.t env) =
 fun env1 env2 ->
  let acc = env1 in
  VarMap.fold
    (fun var set acc ->
      let set2 =
        try NodeiSet.union set (VarMap.find var acc) with Not_found -> set
      in
      VarMap.add var set2 acc)
    env2 acc

let (diff_env : NodeiSet.t env -> NodeiSet.t env -> NodeiSet.t env) =
 fun env1 env2 ->
  let acc = env1 in
  VarMap.fold
    (fun var set acc ->
      try
        let diff = NodeiSet.diff (VarMap.find var acc) set in
        if NodeiSet.is_empty diff then VarMap.remove var acc
        else VarMap.add var diff acc
      with Not_found -> acc)
    env2 acc

let (add_var_and_nodei_to_env :
      var -> nodei -> NodeiSet.t env -> NodeiSet.t env) =
 fun var ni env ->
  let set =
    try NodeiSet.add ni (VarMap.find var env)
    with Not_found -> NodeiSet.singleton ni
  in
  VarMap.add var set env

let (add_vars_and_nodei_to_env :
      VarSet.t -> nodei -> NodeiSet.t env -> NodeiSet.t env) =
 fun varset ni env ->
  let acc = env in
  VarSet.fold (fun var acc -> add_var_and_nodei_to_env var ni acc) varset acc

(*****************************************************************************)
(* Debugging support *)
(*****************************************************************************)
```

⟨*function* Dataflow.csv_append 319e⟩

⟨*function* Dataflow.array_fold_left_idx 320a⟩

⟨*function* Dataflow.ns_to_str 320b⟩

```
let (env_to_str : ('a -> string) -> 'a env -> string) =
 fun val2str env ->
  VarMap.fold (fun dn v s -> s ^ dn ^ ":" ^ val2str v ^ " ") env ""
```

```
let (inout_to_str : ('a -> string) -> 'a inout -> string) =
 fun val2str inout ->
  spf "IN= %15s  OUT = %15s"
    (env_to_str val2str inout.in_env)
    (env_to_str val2str inout.out_env)

(*****************************************************************************)
(* Main generic entry point *)
(*****************************************************************************)
```

⟨*type* Dataflow.transfn 187a⟩

```
module Make (F : Flow) = struct
  let mapping_to_str (fl : F.flow) val2str mapping =
    array_fold_left_idx
      (fun s ni v ->
        s
        ^ spf "%2d <- %7s: %15s %s\n" ni
            ((fl#predecessors ni)#fold
               (fun s (ni, _) -> csv_append s (string_of_int ni))
               "")
            (F.short_string_of_node (fl#nodes#find ni))
            (inout_to_str val2str v))
      "" mapping

  let (display_mapping : F.flow -> 'a mapping -> ('a -> string) -> unit) =
   fun flow mapping string_of_val ->
    pr (mapping_to_str flow string_of_val mapping)

  let rec fixpoint_worker eq mapping trans flow succs workset =
    if NodeiSet.is_empty workset then mapping
    else
      let ni = NodeiSet.choose workset in
      let work' = NodeiSet.remove ni workset in
      let old = mapping.(ni) in
      let new_ = trans mapping ni in
      let work'' =
        if eq_inout eq old new_ then work'
        else (
          mapping.(ni) <- new_;
          NodeiSet.union work' (succs flow ni) )
      in
      fixpoint_worker eq mapping trans flow succs work''

  let forward_succs (f : F.flow) n =
    (f#successors n)#fold (fun s (ni, _) -> NodeiSet.add ni s) NodeiSet.empty

  let backward_succs (f : F.flow) n =
    (f#predecessors n)#fold (fun s (ni, _) -> NodeiSet.add ni s) NodeiSet.empty

  let (fixpoint :
        eq:('a -> 'a -> bool) ->
        init:'a mapping ->
        trans:'a transfn ->
        flow:F.flow ->
        forward:bool ->
        'a mapping) =
   fun ~eq ~init ~trans ~flow ~forward ->
    let succs = if forward then forward_succs else backward_succs in
    let work =
```

```
      flow#nodes#fold (fun s (ni, _) -> NodeiSet.add ni s) NodeiSet.empty
    in
    fixpoint_worker eq init trans flow succs work

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)

  let new_node_array (f : F.flow) v =
    let nb_nodes = f#nb_nodes in
    let max_nodei = ref (-1) in

    f#nodes#tolist
    |> List.iter (fun (ni, _nod) ->
           (* actually there are some del_node done in cfg_build, for
            * switch, so sometimes ni is >= len
            *
            * old:
            * if ni >= nb_nodes
            * then pr2 "the CFG nodei is bigger than the number of nodes"
            *)
           if ni > !max_nodei then max_nodei := ni);
    assert (!max_nodei + 1 >= nb_nodes);
    Array.make (!max_nodei + 1) v
end

(*
module F = Controlflow
module X1 = Make (struct
  type node = F.node
  type edge = F.edge
  type flow = (node, edge) Ograph_extended.ograph_mutable
  let short_string_of_node = F.short_string_of_node
end
)
*)
```

pfff/lang_GENERIC/analyze/Dataflow_tainting.mli

⟨pfff/lang_GENERIC/analyze/Dataflow_tainting.mli 324a⟩≡

```
⟨type Dataflow_tainting.mapping 189a⟩

type fun_env = (Dataflow.var, unit) Hashtbl.t
(** Set of "tainted" functions in the overall program.
  * Note that here [Dataflow.var] is a string of the form "<source name>:<sid>". *)

⟨type Dataflow_tainting.config 193e⟩

⟨signature Dataflow_tainting.fixpoint 189b⟩
```

pfff/lang_GENERIC/analyze/Dataflow_tainting.ml

⟨function Dataflow_tainting.str_of_name 324b⟩≡ (325b)

```
let str_of_name ((s, _tok), sid) = spf "%s:%d" s sid
```

⟨*function* Dataflow_tainting.option_to_varmap 325a⟩≡ (325b)
```
let option_to_varmap = function
  | None -> VarMap.empty
  | Some lvar -> VarMap.singleton (str_of_name lvar) ()
```

⟨pfff/lang_GENERIC/analyze/Dataflow_tainting.ml 325b⟩≡
```
⟨pad/r2c copyright 11⟩
open Common
open IL
module F = IL
module D = Dataflow
module VarMap = Dataflow.VarMap

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Tainting dataflow analysis.
 *
 * This is a very rudimentary tainting analysis. Just intraprocedural,
 * very coarse grained (taint whole array/object).
 * This is step1 for taint tracking support in semgrep.
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

⟨type Dataflow_tainting.mapping 189a⟩

(* Tracks tainted functions. *)
type fun_env = (Dataflow.var, unit) Hashtbl.t

⟨type Dataflow_tainting.config 193e⟩

⟨module Dataflow.Make(Il) 189e⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Dataflow_tainting.str_of_name 324b⟩

⟨function Dataflow_tainting.option_to_varmap 325a⟩

(*****************************************************************************)
(* Tainted *)
(*****************************************************************************)

let sanitized config instr =
  match instr.i with
  | Call (_, { e = Lvalue { base = Var (("sanitize", _), _); _ }; _ }, []) ->
      true
  | ___else___ -> config.is_sanitizer instr

let rec tainted config fun_env env exp =
  (* We call `tainted` recursively on each subexpression, so each subexpression
   * is checked against `pattern-sources`. For example, if `location.href` were
   * a source, this would infer that `"aa" + location.href + "bb"` is tainted.
   * Also note that any arbitrary expression can be source! *)
```

```
    let is_tainted = tainted config fun_env env in
    let go_into = function
      | Lvalue { base = Var var; _ } ->
          VarMap.mem (str_of_name var) env
          || Hashtbl.mem fun_env (str_of_name var)
      | Lvalue { base = VarSpecial (This, _); offset = Dot fld; _ } ->
          Hashtbl.mem fun_env (str_of_name fld)
      | Lvalue _ | Literal _ | FixmeExp _ -> false
      | Composite (_, (_, es, _)) | Operator (_, es) -> List.exists is_tainted es
      | Record fields -> List.exists (fun (_, e) -> is_tainted e) fields
      | Cast (_, e) -> is_tainted e
    in
    config.is_source_exp exp || go_into exp.e

  let tainted_instr config fun_env env instr =
    let is_tainted = tainted config fun_env env in
    let tainted_args = function
      | Assign (_, e) -> is_tainted e
      | AssignAnon _ -> false (* TODO *)
      | Call (_, { e = Lvalue { base = Var (("source", _), _); _ }; _ }, []) ->
          true
      | Call (_, e, args) -> is_tainted e || List.exists is_tainted args
      | CallSpecial (_, _, args) -> List.exists is_tainted args
      | FixmeInstr _ -> false
    in
    (not (sanitized config instr))
    && (config.is_source instr || tainted_args instr.i)

  (*****************************************************************************)
  (* Transfer *)
  (*****************************************************************************)
  (* Not sure we can use the Gen/Kill framework here.
  *)

  ⟨constant Dataflow_tainting.union 189c⟩
  ⟨constant Dataflow_tainting.diff 189d⟩

  ⟨function Dataflow_tainting.transfer 190⟩

  (*****************************************************************************)
  (* Entry point *)
  (*****************************************************************************)

  ⟨function Dataflow_tainting.fixpoint 191a⟩
```

pfff/lang_GENERIC/analyze/Test_analyze_generic.mli

⟨pfff/lang_GENERIC/analyze/Test_analyze_generic.mli 326a⟩≡

```
  ⟨signature Test_analyze_generic.actions 181d⟩
```

pfff/lang_GENERIC/analyze/Test_analyze_generic.ml

⟨function Test_analyze_generic.test_cfg_generic 326b⟩≡ (327)

```
  let test_cfg_generic file =
    let ast = Parse_target.parse_program file in
    ast
```

```
    |> List.iter (fun item ->
           match item.s with
           | DefStmt (_ent, FuncDef def) -> (
               try
                 let flow = Controlflow_build.cfg_of_func def in
                 Controlflow.display_flow flow
               with Controlflow_build.Error err ->
                 Controlflow_build.report_error err )
           | _ -> ())
```

⟨pfff/lang_GENERIC/analyze/Test_analyze_generic.ml 327⟩≡

```
  open Common
  open AST_generic
  module V = Visitor_AST

  let test_typing_generic file =
    let ast = Parse_target.parse_program file in
    let lang = List.hd (Lang.langs_of_filename file) in
    Naming_AST.resolve lang ast;

    let v =
      V.mk_visitor
        {
          V.default_visitor with
          V.kfunction_definition =
            (fun (_k, _) def ->
              let s = AST_generic.show_any (S def.fbody) in
              pr2 s;
              pr2 "==>";

              let xs = AST_to_IL.stmt def.fbody in
              let s = IL.show_any (IL.Ss xs) in
              pr2 s);
        }
    in
    v (Pr ast)

  ⟨function Test_analyze_generic.test_cfg_generic 326b⟩

  module F = Controlflow

  module DataflowX = Dataflow.Make (struct
    type node = F.node

    type edge = F.edge

    type flow = (node, edge) Ograph_extended.ograph_mutable

    let short_string_of_node = F.short_string_of_node
  end)

  ⟨function Test_analyze_generic.test_dfg_generic 188h⟩

  ⟨function Test_analyze_generic.test_naming_generic 175b⟩

  let test_constant_propagation file =
    let ast = Parse_target.parse_program file in
    let lang = List.hd (Lang.langs_of_filename file) in
    Naming_AST.resolve lang ast;
```

```
  Constant_propagation.propagate_basic lang ast;
  let s = AST_generic.show_any (AST_generic.Pr ast) in
  pr2 s
```

⟨*function* Test_analyze_generic.test_il_generic 181f⟩

⟨*function* Test_analyze_generic.test_cfg_il 184e⟩

```
module F2 = IL

module DataflowY = Dataflow.Make (struct
  type node = F2.node

  type edge = F2.edge

  type flow = (node, edge) Ograph_extended.ograph_mutable

  let short_string_of_node n = Display_IL.short_string_of_node_kind n.F2.n
end)
```

⟨*function* Test_analyze_generic.test_dfg_tainting 192⟩

```
let test_dfg_constness file =
  let ast = Parse_target.parse_program file in
  let lang = List.hd (Lang.langs_of_filename file) in
  Naming_AST.resolve lang ast;
  let v =
    V.mk_visitor
      {
        V.default_visitor with
        V.kfunction_definition =
          (fun (_k, _) def ->
            let inputs, xs = AST_to_IL.function_definition def in
            let flow = CFG_build.cfg_of_stmts xs in
            pr2 "Constness";
            let mapping = Dataflow_constness.fixpoint inputs flow in
            Dataflow_constness.update_constness flow mapping;
            DataflowY.display_mapping flow mapping
              Dataflow_constness.string_of_constness;
            let s = AST_generic.show_any (S def.fbody) in
            pr2 s);
      }
  in
  v (Pr ast)
```

⟨*function* Test_analyze_generic.actions 181e⟩

F.10 pfff/lang_python/parsing/

pfff/lang_python/parsing/AST_python.ml

⟨*function* AST_python.context_of_expr 328⟩≡ (329)

```
  let context_of_expr = function
    | Attribute (_, _, _, ctx) -> Some ctx
    | Subscript (_, _, ctx) -> Some ctx
    | Name (_, ctx, _)   -> Some ctx
    | List (_, ctx)         -> Some ctx
    | Tuple (_, ctx)        -> Some ctx
```

```
    | _                     -> None
```

⟨pfff/lang_python/parsing/AST_python.ml 329⟩≡

```
  (* Yoann Padioleau
   *
   * Copyright (C) 2010 Facebook
   * Copyright (C) 2011-2015 Tomohiro Matsuyama
   * Copyright (C) 2019 r2c
   *
   * This library is free software; you can redistribute it and/or
   * modify it under the terms of the GNU Lesser General Public License
   * version 2.1 as published by the Free Software Foundation, with the
   * special exception on linking described in file license.txt.
   *
   * This library 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.
  *)

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Abstract Syntax Tree for Python3.
   *
   * Most of the code in this file derives from code from
   * Tomohiro Matsuyama in ocaml-pythonlib, which itself derives from
   * the official grammar definition of Python.
   *
   * reference: http://docs.python.org/3/library/AST.html
   *
   * See also:
   *  - http://trevorjim.com/python-is-not-context-free/
   *  - https://github.com/gvanrossum/pegen a WIP to write the Python grammar
   *    using a PEG parser
   *
   * Note that this AST supports partly Python2 syntax with the special
   * print and exec statements. It does not support the special tuple
   * parameters syntax though.
   *
   * related work:
   *  - https://github.com/m2ym/ocaml-pythonlib
   *    The original code. The repo was also forked by jeremy buisson
   *    who added a very basic simplifier but remains mostly the same.
   *  - Pyre-check
   *    typechecker and taint-tracker for Python, written in OCaml from facebook
   *  - https://github.com/mattgreen/hython
   *    Python3 interpreter written in Haskell
   *  - libCST (a concrete syntax tree, better for program transformation)
   *    by Instagram
   *
   * history:
   *  - 2019 port to the pfff infrastructure.
   *  - 2019 modified to support types, and many other Python 3 features
   *    (see the python3: tag in this file)
   *  - 2020 backport print and exec statements, to parse some python2 code.
   *
   * todo:
   *  - could use records for all the XxxDef, but what matters now is
   *    AST_generic.ml, which uses records at least.
```

```
*)

(*****************************************************************************)
(* Names *)
(*****************************************************************************)
(* ------------------------------------------------------------------------- *)
(* Token/info *)
(* ------------------------------------------------------------------------- *)

⟨type AST_python.tok 127a⟩
[@@deriving show]
⟨type AST_python.wrap 127b⟩
[@@deriving show] (* with tarzan *)
⟨type AST_python.bracket 127c⟩
[@@deriving show] (* with tarzan *)
(* ------------------------------------------------------------------------- *)
(* Name *)
(* ------------------------------------------------------------------------- *)
⟨type AST_python.name 127d⟩
[@@deriving show] (* with tarzan *)

⟨type AST_python.dotted_name 127f⟩
[@@deriving show] (* with tarzan *)

⟨type AST_python.module_name 127g⟩
[@@deriving show]

⟨type AST_python.resolved_name 128a⟩
[@@deriving show { with_path = false }] (* with tarzan *)

(*****************************************************************************)
(* Expression *)
(*****************************************************************************)
⟨type AST_python.expr 128b⟩

⟨type AST_python.number 129a⟩

(* less: could reuse AST_generic.arithmetic_operator *)
⟨type AST_python.boolop 129b⟩

(* the % operator can also be used for strings! "foo %s" % name *)
⟨type AST_python.operator 129c⟩

⟨type AST_python.unaryop 129d⟩

⟨type AST_python.cmpop 129e⟩

(* usually a Str or a simple expr.
 * TODO: should also handle format specifier, they are skipped for now
 * during parsing
*)
⟨type AST_python.interpolated 129f⟩

⟨type AST_python.list_or_comprehension 129g⟩

⟨type AST_python.comprehension 129h⟩
⟨type AST_python.for_if 130a⟩

⟨type AST_python.dictorset_elt 130b⟩
```

```
(* AugLoad and AugStore are not used *)
⟨type AST_python.expr_context 130c⟩

⟨type AST_python.slice 130d⟩

⟨type AST_python.parameters 130f⟩
⟨type AST_python.parameter 130e⟩

⟨type AST_python.argument 130g⟩

(*****************************************************************************)
(* Type *)
(*****************************************************************************)
(* python3: type annotations!
 * see https://docs.python.org/3/library/typing.html for the semantic
 * and https://www.python.org/dev/peps/pep-3107/ (function annotations)
 * for https://www.python.org/dev/peps/pep-0526/ (variable annotations)
 * for its syntax.
*)
⟨type AST_python.type_ 132c⟩

(* used in inheritance, to allow default value for metaclass *)
⟨type AST_python.type_parent 132d⟩

(*****************************************************************************)
(* Pattern *)
(*****************************************************************************)
⟨type AST_python.pattern 132e⟩

(* python2? *)
and param_pattern =
  | PatternName of name
  | PatternTuple of param_pattern list
[@@deriving show { with_path = false }]  (* with tarzan *)

(*****************************************************************************)
(* Statement *)
(*****************************************************************************)
⟨type AST_python.stmt 131⟩


⟨type AST_python.excepthandler 132a⟩

(*****************************************************************************)
(* Definitions *)
(*****************************************************************************)

(* ------------------------------------------------------------------------ *)
(* Decorators (a.k.a annotations) *)
(* ------------------------------------------------------------------------ *)
⟨type AST_python.decorator 132f⟩

(* ------------------------------------------------------------------------ *)
(* Function (or method) definition *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* Variable definition *)
(* ------------------------------------------------------------------------ *)
```

```
  (* ------------------------------------------------------------------------ *)
  (* Class definition *)
  (* ------------------------------------------------------------------------ *)

  (*****************************************************************************)
  (* Module *)
  (*****************************************************************************)
  (* ------------------------------------------------------------------------ *)
  (* Module import/export *)
  (* ------------------------------------------------------------------------ *)
  ⟨type AST_python.alias 132b⟩
  [@@deriving show { with_path = false }]  (* with tarzan *)

  (*****************************************************************************)
  (* Toplevel *)
  (*****************************************************************************)
  ⟨type AST_python.program 132g⟩
  [@@deriving show]   (* with tarzan *)

  (*****************************************************************************)
  (* Any *)
  (*****************************************************************************)
  ⟨type AST_python.any 132h⟩
  [@@deriving show { with_path = false }] (* with tarzan *)

  (*****************************************************************************)
  (* Wrappers *)
  (*****************************************************************************)
  ⟨constant AST_python.str_of_name 127e⟩

  (*****************************************************************************)
  (* Accessors *)
  (*****************************************************************************)
  ⟨function AST_python.context_of_expr 328⟩
```

/home/pad/pfff/lang_python/parsing/Lexer_python.mll

```
⟨pfff/lang_python/parsing/Lexer_python.mll 332⟩≡
  {
  (* Yoann Padioleau
   *
   * Copyright (C) 2010 Facebook
   * Copyright (C) 2011-2015 Tomohiro Matsuyama
   * Copyright (C) 2019 r2c
   *
   * This library is free software; you can redistribute it and/or
   * modify it under the terms of the GNU Lesser General Public License
   * version 2.1 as published by the Free Software Foundation, with the
   * special exception on linking described in file license.txt.
   *
   * This library 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.
   *)
  open Common

  open Lexing
```

```
open Parser_python
module PI = Parse_info
module Flag = Flag_parsing_python

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* The Python lexer.
 *
 * original src:
 *  - https://github.com/m2ym/ocaml-pythonlib/blob/master/src/lexer.mll
 *  - some code stolen from pyre-check python lexer (itself started
 *    from ocaml-pythonlib)
 * reference:
 *    - http://docs.python.org/release/2.5.4/ref/ref.html
 * old src:
 *  - http://inst.eecs.berkeley.edu/~cs164/sp10/python-grammar.html
 *    which was itself from the python reference manual
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

(* shortcuts *)
let tok = Lexing.lexeme
let tokinfo = Parse_info.tokinfo
let error = Parse_info.lexical_error

let unescaped s =
  let buf = Buffer.create (String.length s) in
  let escape = ref false in
  let unescapechar c =
    if !escape then begin
      match c with
      | '\r' -> ()
      | '\n' -> escape := false
      | _ -> begin
          escape := false;
          (* TODO http://docs.python.org/reference/lexical_analysis.html#string-literals *)
          Buffer.add_char
            buf
            (match c with
             | '\\' -> '\\'
             | '\'' -> '\''
             | '"' -> '"'
             | 'a' -> Char.chr 7
             | 'b' -> '\b'
             | 'f' -> Char.chr 12
             | 'n' -> '\n'
             | 'r' -> '\r'
             | 't' -> '\t'
             | 'v' -> Char.chr 11
             | _ -> (Buffer.add_char buf '\\'; c))
        end
    end else if c = '\\' then
      escape := true
    else
      Buffer.add_char buf c
  in
```

```
    String.iter unescapechar s;
    Buffer.contents buf

(* ------------------------------------------------------------------------ *)
(* Lexer state *)
(* ------------------------------------------------------------------------ *)

(* this is to return space/comment tokens *)
type state_mode =
  | STATE_TOKEN
  | STATE_OFFSET
  | STATE_UNDERSCORE_TOKEN

  (* below the 'string' contains the prefix, e.g. "f", but
   * can also be "fr" in which case we must treat \ differently.
   *)
  | STATE_IN_FSTRING_SINGLE of string
  | STATE_IN_FSTRING_DOUBLE of string
  | STATE_IN_FSTRING_TRIPLE_SINGLE of string
  | STATE_IN_FSTRING_TRIPLE_DOUBLE of string

type lexer_state = {
  mutable curr_offset : int;
  offset_stack : int Stack.t;
  mutable nl_ignore : int;

  mode: state_mode list ref;
}

let create () =
  let stack = Stack.create () in
  Stack.push 0 stack;
  { curr_offset = 0;
    offset_stack = stack;
    nl_ignore = 0;
    mode = ref [STATE_TOKEN];
  }

let ignore_nl t =
  t.nl_ignore <- succ t.nl_ignore

and aware_nl t =
  t.nl_ignore <- pred t.nl_ignore

(* stack mode management *)
let top_mode state =
  match !(state.mode) with
  | [] -> failwith "Lexer_python.top_mode: empty stack"
  | x::_ -> x
let push_mode state mode = Common.push mode state.mode
let pop_mode state = ignore(Common2.pop2 state.mode)
let set_mode state mode = begin pop_mode state; push_mode state mode end

let pr_mode mode = pr2 (match mode with
  | STATE_TOKEN -> "token"
  | STATE_OFFSET -> "offset"
  | STATE_UNDERSCORE_TOKEN -> "_token"
  | STATE_IN_FSTRING_SINGLE _ -> "f'"
  | STATE_IN_FSTRING_DOUBLE _ -> "f\""
  | STATE_IN_FSTRING_TRIPLE_SINGLE _ -> "f'''"
```

```
  | STATE_IN_FSTRING_TRIPLE_DOUBLE _ -> "f\"\"\""
)

let pr_state state = List.iter pr_mode !(state.mode)

(* This used to be 8, but tests/python/parsing/eof_comment.py was not parsing.
 * This maybe should be a command-line parameter? Or we should fix
 * eof_comment.py in another way?
 *)
let space_per_tab = 8
}

(*****************************************************************************)
(* Regexp aliases *)
(*****************************************************************************)

(* epsilon *)
let e = ""

let newline = ('\n' | "\r\n" | '\x0C')
let whitespace = [' ' '\t']
let comment = '#' [^ '\n' '\r']*

let digit = ['0'-'9']
let octdigit = ['0'-'7']
(* python3-ext: underscore in numbers *)
let digipart = digit (('_'? digit)* )

let hexdigit = ['0'-'9' 'a'-'f' 'A'-'F']

let longintpostfix = ['l' 'L']

(* python3-ext: underscore in numbers (src: Pyre-check) *)
let integer =
  ('0' ['b' 'B'] ('_'? ['0'-'1'])+) | (* Binary. *)
  ('0' ['o' 'O'] ('_'? ['0'-'7'])+) | (* Octal. *)
  ('0' ['x' 'X'] ('_'? hexdigit)+) | (* Hexadecimal. *)
  (['1' - '9'] ('_'? digit)* | '0' ('_'? '0')* ) |  (* Decimal. *)
  ('0' digit+) (* Valid before python 3.6 *)


let intpart = digipart
let fraction = '.' digipart
let pointfloat = intpart? fraction | intpart '.'
let exponent = ['e' 'E'] ['+' '-']? digipart
let exponentfloat = (intpart | pointfloat) exponent
let floatnumber = pointfloat | exponentfloat

let imagnumber = (floatnumber | intpart) ['j' 'J']

let kind = 'b' | 'B'
let rawprefix = 'r' | 'R'
let encoding = 'u' | 'U' | rawprefix
(* (encoding encoding) for python2 legacy support *)
let stringprefix = (encoding | kind | (encoding kind) | (kind encoding) | (encoding encoding))?

(* Per https://www.python.org/dev/peps/pep-0498/, 'f' can be combined with 'r' but
 * not 'u' or 'b'
 *)
let fstringspecifier = 'f' | 'F'
```

```
let fstringprefix = fstringspecifier | (fstringspecifier rawprefix) | (rawprefix fstringspecifier)

let escapeseq = '\\' _
(* for raw fstring *)
let escapeseq2 = '\\' [^ '{']

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

let nonidchar = [^ 'a'-'z' 'A'-'Z' '0'-'9' '_']

(*****************************************************************************)
(* Rule initial *)
(*****************************************************************************)

rule token python2 state = parse
  | e {
        let curr_offset = state.curr_offset in
        let last_offset = Stack.top state.offset_stack in
        match () with
        | _ when curr_offset < last_offset ->
            ignore (Stack.pop state.offset_stack);
            DEDENT (tokinfo lexbuf)
        | _ when curr_offset > last_offset ->
            Stack.push curr_offset state.offset_stack;
            INDENT (tokinfo lexbuf)
        (* curr_offset = last_offset *)
        | _ -> _token python2 state lexbuf
      }

(* this is just used to adjuste the state *)
and offset state = parse
  | e { "" }
  | ' '
     {
       state.curr_offset <- state.curr_offset + 1;
       " " ^ offset state lexbuf
     }
  | '\t'
     {
       state.curr_offset <- state.curr_offset + space_per_tab;
       "\t" ^ offset state lexbuf
     }

and _token python2 state = parse

  (* ---------------------------------------------------------------------- *)
  (* spacing/comments *)
  (* ---------------------------------------------------------------------- *)
  | ((whitespace* comment? newline)* whitespace* comment?) newline
      {
        let info = tokinfo lexbuf in
        if state.nl_ignore <= 0 then begin
          state.curr_offset <- 0;
          let s = offset state lexbuf in
          NEWLINE (Parse_info.tok_add_s s info)
        end else begin
         set_mode state STATE_UNDERSCORE_TOKEN;
         TCommentSpace info
        end
      }
```

```
| '\\' newline whitespace*
    {
        let info = tokinfo lexbuf in
        let pos = lexbuf.lex_curr_p in
        lexbuf.lex_curr_p <-
          { pos with
              pos_bol = pos.pos_cnum;
              pos_lnum = pos.pos_lnum + 1 };
        set_mode state STATE_UNDERSCORE_TOKEN;
        TCommentSpace info
    }

| whitespace+
    {
      let info = tokinfo lexbuf in
      set_mode state STATE_UNDERSCORE_TOKEN;
      TCommentSpace info
    }

(* ------------------------------------------------------------------------ *)
(* symbols *)
(* ------------------------------------------------------------------------ *)
(* symbols *)
| ":="    { COLONEQ (tokinfo lexbuf) }
| "+="    { ADDEQ (tokinfo lexbuf) }
| "-="    { SUBEQ (tokinfo lexbuf) }
| "*="    { MULTEQ (tokinfo lexbuf) }
| "/="    { DIVEQ (tokinfo lexbuf) }
| "%="    { MODEQ (tokinfo lexbuf) }
| "**="   { POWEQ (tokinfo lexbuf) }
| "//="   { FDIVEQ (tokinfo lexbuf) }
| "&="    { ANDEQ (tokinfo lexbuf) }
| "|="    { OREQ (tokinfo lexbuf) }
| "^="    { XOREQ (tokinfo lexbuf) }
| "<<="   { LSHEQ (tokinfo lexbuf) }
| ">>="   { RSHEQ (tokinfo lexbuf) }

| "=="    { EQUAL (tokinfo lexbuf) }
| "!="    { NOTEQ (tokinfo lexbuf) }
(* != equivalent to <>? *)
| "<>"    { NOTEQ (tokinfo lexbuf) }
| "<="    { LEQ (tokinfo lexbuf) }
| ">="    { GEQ (tokinfo lexbuf) }
| '<'     { LT (tokinfo lexbuf) }
| '>'     { GT (tokinfo lexbuf) }

| '='     { EQ (tokinfo lexbuf) }

| "**"    { POW (tokinfo lexbuf) }
| "//"    { FDIV (tokinfo lexbuf) }

| '+'     { ADD (tokinfo lexbuf) }
| '-'     { SUB (tokinfo lexbuf) }
| '*'     { MULT (tokinfo lexbuf) }
| '/'     { DIV (tokinfo lexbuf) }
| '%'     { MOD (tokinfo lexbuf) }

| '|'     { BITOR (tokinfo lexbuf) }
| '&'     { BITAND (tokinfo lexbuf) }
| '^'     { BITXOR (tokinfo lexbuf) }
```

```
| '~'     { BITNOT (tokinfo lexbuf) }

| "<<"    { LSHIFT (tokinfo lexbuf) }
| ">>"    { RSHIFT (tokinfo lexbuf) }

| '('     { ignore_nl state; LPAREN (tokinfo lexbuf) }
| ')'     { aware_nl state; RPAREN (tokinfo lexbuf) }
| '['     { ignore_nl state; LBRACK (tokinfo lexbuf) }
| ']'     { aware_nl state; RBRACK (tokinfo lexbuf) }
| '{'     {
    ignore_nl state;
    push_mode state STATE_UNDERSCORE_TOKEN;
    LBRACE (tokinfo lexbuf)
   }
| '}'     {
    aware_nl state;
    pop_mode state;
    RBRACE (tokinfo lexbuf)
   }

| ':'     { COLON (tokinfo lexbuf) }
| ';'     { SEMICOL (tokinfo lexbuf) }
| '.'     { DOT (tokinfo lexbuf) }
| ','     { COMMA (tokinfo lexbuf) }
| '`'     { BACKQUOTE (tokinfo lexbuf) }
| '@'     { AT (tokinfo lexbuf) }
(* part of python3 and also sgrep-ext: *)
| "..."   { ELLIPSES (tokinfo lexbuf) }
(* sgrep-ext: *)
| "<..."  { Flag_parsing.sgrep_guard (LDots (tokinfo lexbuf)) }
| "...>"  { Flag_parsing.sgrep_guard (RDots (tokinfo lexbuf)) }

| "!" { if !(state.mode) |> List.exists (function
          | STATE_IN_FSTRING_SINGLE _
          | STATE_IN_FSTRING_DOUBLE _
          | STATE_IN_FSTRING_TRIPLE_SINGLE _
          | STATE_IN_FSTRING_TRIPLE_DOUBLE _ -> true
          | _ -> false)
        then BANG (tokinfo lexbuf)
        else begin
           error (spf "unrecognized symbols: %s" (tok lexbuf)) lexbuf;
           TUnknown (tokinfo lexbuf)
        end
    }

(* ---------------------------------------------------------------------- *)
(* Keywords and ident *)
(* ---------------------------------------------------------------------- *)

(* keywords *)
| identifier as id
    { match id with

      | "if"       -> IF (tokinfo lexbuf)
      | "elif"     -> ELIF (tokinfo lexbuf)
      | "else"     -> ELSE (tokinfo lexbuf)

      | "for"      -> FOR (tokinfo lexbuf)
      | "while"    -> WHILE (tokinfo lexbuf)
```

```
      | "return"   -> RETURN (tokinfo lexbuf)
      | "break"    -> BREAK (tokinfo lexbuf)
      | "continue" -> CONTINUE (tokinfo lexbuf)
      | "pass"     -> PASS (tokinfo lexbuf)

      | "raise"    -> RAISE (tokinfo lexbuf)
      | "try"      -> TRY (tokinfo lexbuf)
      | "except"   -> EXCEPT (tokinfo lexbuf)
      | "finally"  -> FINALLY (tokinfo lexbuf)

      | "def"      -> DEF (tokinfo lexbuf)
      | "class"    -> CLASS (tokinfo lexbuf)
      | "lambda"   -> LAMBDA (tokinfo lexbuf)

      | "and"      -> AND (tokinfo lexbuf)
      | "or"       -> OR (tokinfo lexbuf)
      | "not"      -> NOT (tokinfo lexbuf)

      | "import"   -> IMPORT (tokinfo lexbuf)
      | "from"     -> FROM (tokinfo lexbuf)

      | "as"       -> AS (tokinfo lexbuf)
      | "in"       -> IN (tokinfo lexbuf)
      | "is"       -> IS (tokinfo lexbuf)
      | "assert"   -> ASSERT (tokinfo lexbuf)
      | "del"      -> DEL (tokinfo lexbuf)
      | "global"   -> GLOBAL (tokinfo lexbuf)
      | "with"     -> WITH (tokinfo lexbuf)
      | "yield"    -> YIELD (tokinfo lexbuf)

      (* python3-ext:
       * coupling: if python3 has more special keywords, you may need to
       * modify the heuristic in parse_python.ml that fallbacks to python2
       * in case of a parse error.
       *)
      | "None"    -> NONE (tokinfo lexbuf)
      | "True"   when not python2 -> TRUE (tokinfo lexbuf)
      | "False"  when not python2 -> FALSE (tokinfo lexbuf)

      | "async"    when not python2 -> ASYNC (tokinfo lexbuf)
      | "await"    when not python2 -> AWAIT (tokinfo lexbuf)
      | "nonlocal" when not python2 -> NONLOCAL (tokinfo lexbuf)

      (* python2: *)
      | "print" when python2 -> PRINT (tokinfo lexbuf)
      | "exec" when python2 -> EXEC (tokinfo lexbuf)
      (* python3-ext: no more: print, exec *)

      | _          -> NAME (id, (tokinfo lexbuf))
  }

(* sgrep-ext: *)
| '$' identifier
  { Flag_parsing.sgrep_guard (NAME (tok lexbuf, tokinfo lexbuf)) }
(* sgrep-ext: *)
| '$' "..." ['A'-'Z''_']['A'-'Z''_''0'-'9']*
   { Flag_parsing.sgrep_guard (NAME (tok lexbuf, tokinfo lexbuf)) }


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

```
  (* Constant *)
  (* ------------------------------------------------------------------------ *)

  (* literals *)
  | integer as n longintpostfix
      { LONGINT (int_of_string_opt n, tokinfo lexbuf) }
  | integer as n
      { INT (int_of_string_opt n, tokinfo lexbuf) }

  | floatnumber as n
      { FLOAT (float_of_string_opt n, tokinfo lexbuf) }

  | imagnumber as n
      { IMAG (n, tokinfo lexbuf) }

  (* ------------------------------------------------------------------------ *)
  (* Strings *)
  (* ------------------------------------------------------------------------ *)
  | fstringprefix as pre "'"  {
       push_mode state (STATE_IN_FSTRING_SINGLE pre);
       FSTRING_START (tokinfo lexbuf)
    }
  | fstringprefix as pre '"'  {
       push_mode state (STATE_IN_FSTRING_DOUBLE pre);
       FSTRING_START (tokinfo lexbuf)
    }
  | fstringprefix as pre "'''" {
       push_mode state (STATE_IN_FSTRING_TRIPLE_SINGLE pre);
       FSTRING_START (tokinfo lexbuf)
     }
  | fstringprefix as pre "\"\"\"" {
       push_mode state (STATE_IN_FSTRING_TRIPLE_DOUBLE pre);
       FSTRING_START (tokinfo lexbuf)
     }

  | stringprefix as pre '\''
      { sq_shortstrlit state (tokinfo lexbuf) pre lexbuf }
  | stringprefix as pre '"'
      { dq_shortstrlit state (tokinfo lexbuf) pre lexbuf }
  | stringprefix as pre "'''"
      { sq_longstrlit state (tokinfo lexbuf) pre lexbuf }
  | stringprefix as pre "\"\"\""
      { dq_longstrlit state (tokinfo lexbuf) pre lexbuf }

  (* ------------------------------------------------------------------------ *)
  (* eof *)
  (* ------------------------------------------------------------------------ *)
  | (whitespace* comment?) eof { EOF (tokinfo lexbuf) }

  | _ { error (spf "unrecognized symbol: %s" (tok lexbuf)) lexbuf;
        TUnknown (tokinfo lexbuf)
      }

(*****************************************************************************)
(* Rules on strings *)
(*****************************************************************************)

and sq_shortstrlit state pos pre = parse
  | (([^ '\\' '\r' '\n' '\''] | escapeseq)* as s) '\''
     {
```

```
       let full_str = Lexing.lexeme lexbuf in
       STR (unescaped s, pre, PI.tok_add_s full_str pos) }
 | eof { error "EOF in string" lexbuf; EOF (tokinfo lexbuf) }
 | _  { error "unrecognized symbol in string" lexbuf; TUnknown(tokinfo lexbuf)}

(* because here we're using 'shortest', do not put a rule for '| _ { ... }' *)
and sq_longstrlit state pos pre = shortest
| (([^ '\\'] | escapeseq)* as s) "'''"
    {
      let full_str = Lexing.lexeme lexbuf in
      STR (unescaped s, pre, PI.tok_add_s full_str pos)
    }

and dq_shortstrlit state pos pre = parse
  | (([^ '\\' '\r' '\n' '\"'] | escapeseq)* as s) '"'
     {
       let full_str = Lexing.lexeme lexbuf in
       STR (unescaped s, pre, PI.tok_add_s full_str pos) }
 | eof { error "EOF in string" lexbuf; EOF (tokinfo lexbuf) }
 | _  { error "unrecognized symbol in string" lexbuf; TUnknown(tokinfo lexbuf)}

and dq_longstrlit state pos pre = shortest
  | (([^ '\\'] | escapeseq)* as s) "\"\"\""
      {
        let full_str = Lexing.lexeme lexbuf in
        STR (unescaped s, pre, PI.tok_add_s full_str pos) }

(*****************************************************************************)
(* Rules on interpolated strings *)
(*****************************************************************************)

and fstring_single state pre = parse
 | "'" { pop_mode state; FSTRING_END (tokinfo lexbuf) }
 | "{{" { FSTRING_STRING (tok lexbuf, tokinfo lexbuf)}
 | '{' {
    ignore_nl state;
    push_mode state STATE_UNDERSCORE_TOKEN;
    FSTRING_LBRACE (tokinfo lexbuf)
   }
 (* using escapeseq2 here! for raw-fstring *)
 | ([^ '\\' '\r' '\n' '\'' '{'] | escapeseq2)*
    { FSTRING_STRING (tok lexbuf, tokinfo lexbuf)}
 (* ugly: for raw-fstring *)
 | "\\{" { if pre =~ ".*[rR]" then Parse_info.yyback 1 lexbuf;
      FSTRING_STRING (tok lexbuf, tokinfo lexbuf)
    }

 | eof { error "EOF in string" lexbuf; EOF (tokinfo lexbuf) }
 | _  { error "unrecognized symbol in string" lexbuf; TUnknown(tokinfo lexbuf)}

and fstring_double state pre = parse
 | '"' { pop_mode state; FSTRING_END (tokinfo lexbuf) }
 | "{{" { FSTRING_STRING (tok lexbuf, tokinfo lexbuf)}
 | '{' {
    ignore_nl state;
    push_mode state STATE_UNDERSCORE_TOKEN;
    FSTRING_LBRACE (tokinfo lexbuf)
   }
 | ([^ '\\' '\r' '\n' '\"' '{'] | escapeseq2)*
    { FSTRING_STRING (tok lexbuf, tokinfo lexbuf)}
```

```
  (* ugly: for raw-fstring *)
  | "\\{" { if pre =~ ".*[rR]" then Parse_info.yyback 1 lexbuf;
       FSTRING_STRING (tok lexbuf, tokinfo lexbuf)
     }

  | eof { error "EOF in string" lexbuf; EOF (tokinfo lexbuf) }
  | _  { error "unrecognized symbol in string" lexbuf; TUnknown(tokinfo lexbuf)}

 and fstring_triple_single state pre = parse
  | "'''" { pop_mode state; FSTRING_END (tokinfo lexbuf) }
  | "{{" { FSTRING_STRING (tok lexbuf, tokinfo lexbuf)}
  | '{' {
     ignore_nl state;
     push_mode state STATE_UNDERSCORE_TOKEN;
     FSTRING_LBRACE (tokinfo lexbuf)
    }
  | '\'' { FSTRING_STRING (tok lexbuf, tokinfo lexbuf) }

  | ([^ '\\' '{' '\''] | escapeseq2)*
     { FSTRING_STRING (tok lexbuf, tokinfo lexbuf)}
  (* ugly: for raw-fstring *)
  | "\\{" { if pre =~ ".*[rR]" then Parse_info.yyback 1 lexbuf;
       FSTRING_STRING (tok lexbuf, tokinfo lexbuf)
     }

  | eof { error "EOF in string" lexbuf; EOF (tokinfo lexbuf) }
  | _  { error "unrecognized symbol in string" lexbuf; TUnknown(tokinfo lexbuf)}

 and fstring_triple_double state pre = parse
  | "\"\"\"" { pop_mode state; FSTRING_END (tokinfo lexbuf) }
  | "{{" { FSTRING_STRING (tok lexbuf, tokinfo lexbuf)}
  | '{' {
     ignore_nl state;
     push_mode state STATE_UNDERSCORE_TOKEN;
     FSTRING_LBRACE (tokinfo lexbuf)
    }
  | '"' { FSTRING_STRING (tok lexbuf, tokinfo lexbuf) }

  | ([^ '\\' '{' '"'] | escapeseq2)*
     { FSTRING_STRING (tok lexbuf, tokinfo lexbuf)}
  (* ugly: for raw-fstring *)
  | "\\{" { if pre =~ ".*[rR]" then Parse_info.yyback 1 lexbuf;
       FSTRING_STRING (tok lexbuf, tokinfo lexbuf)
     }

  | eof { error "EOF in string" lexbuf; EOF (tokinfo lexbuf) }
  | _  { error "unrecognized symbol in string" lexbuf; TUnknown(tokinfo lexbuf)}
```

/home/pad/pfff/lang_python/parsing/Parser_python.mly

⟨pfff/lang_python/parsing/Parser_python.mly 342⟩≡

```
  %{
  (* Yoann Padioleau
   *
   * Copyright (C) 2010 Facebook
   * Copyright (C) 2011-2015 Tomohiro Matsuyama
   * Copyright (C) 2019-2020 r2c
   *
   * This library is free software; you can redistribute it and/or
```

```
 * modify it under the terms of the GNU Lesser General Public License
 * version 2.1 as published by the Free Software Foundation, with the
 * special exception on linking described in file license.txt.
 *
 * This library 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.
 *)

(* This file contains a grammar for Python 3
 * (which is mostly a superset of Python 2).
 *
 * original src:
 *  https://github.com/m2ym/ocaml-pythonlib/blob/master/src/python2_parser.mly
 * reference:
 *  - https://docs.python.org/3/reference/grammar.html
 *  - http://docs.python.org/release/2.5.2/ref/grammar.txt
 * old src:
 *  - http://inst.eecs.berkeley.edu/~cs164/sp10/python-grammar.html
 *)
open Common
open AST_python
module PI = Parse_info

(* intermediate helper type *)
type single_or_tuple =
  | Single of expr
  | Tup of expr list

let cons e = function
  | Single e' -> Tup (e::[e'])
  | Tup l -> Tup (e::l)

let tuple_expr = function
  | Single e -> e
  | Tup l -> Tuple (CompList (PI.fake_bracket l), Load)

let to_list = function
  | Single e -> [e]
  | Tup l -> l

(* TODO: TypedExpr? ExprStar? then can appear as lvalue
 * CompForIf though is not an lvalue.
*)
let rec set_expr_ctx ctx = function
  | Name (id, _, x) ->
      Name (id, ctx, x)
  | Attribute (value, t, attr, _) ->
      Attribute (value, t, attr, ctx)
  | Subscript (value, slice, _) ->
      Subscript (value, slice, ctx)

  | List (CompList (t1, elts, t2), _) ->
      List (CompList ((t1, List.map (set_expr_ctx ctx) elts, t2)), ctx)
  | Tuple (CompList (t1, elts, t2), _) ->
      Tuple (CompList ((t1, List.map (set_expr_ctx ctx) elts, t2)), ctx)

  | e -> e
```

```
let expr_store = set_expr_ctx Store
and expr_del = set_expr_ctx Del

let tuple_expr_store l =
  let e = tuple_expr l in
    match AST_python.context_of_expr e with
    | Some Param -> e
    | _ -> expr_store e

let mk_str ii =
  let s = Parse_info.str_of_info ii in
  Str (s, ii)

%}

(*************************************************************************)
(* Tokens *)
(*************************************************************************)
%token <AST_python.tok> TUnknown  (* unrecognized token *)
%token <AST_python.tok> EOF

(*-----------------------------------------*)
(* The space/comment tokens *)
(*-----------------------------------------*)
(* coupling: Token_helpers.is_comment *)
%token <AST_python.tok> TCommentSpace TComment
(* see the extra token below NEWLINE instead of TCommentNewline *)

(*-----------------------------------------*)
(* The normal tokens *)
(*-----------------------------------------*)

(* tokens with "values" *)
%token <string * AST_python.tok> NAME
%token <int option    * AST_python.tok> INT LONGINT
%token <float option  * AST_python.tok> FLOAT
%token <string * AST_python.tok> IMAG
%token <string * string * AST_python.tok> STR

(*-----------------------------------------*)
(* Keyword tokens *)
(*-----------------------------------------*)
%token <AST_python.tok>
 IF ELSE ELIF
 WHILE FOR
 RETURN CONTINUE BREAK PASS
 DEF LAMBDA CLASS GLOBAL
 TRY FINALLY EXCEPT RAISE
 AND NOT OR
 IMPORT FROM AS
 DEL IN IS WITH YIELD
 ASSERT
 NONE TRUE FALSE
 ASYNC AWAIT
 NONLOCAL
 (* python2: *)
 PRINT EXEC

(*-----------------------------------------*)
(* Punctuation tokens *)
```

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

(* syntax *)
%token <AST_python.tok>
 LPAREN "("     RPAREN ")"
 LBRACK "["     RBRACK "]"
 LBRACE "{"     RBRACE "}"
 COLON  ":"
 SEMICOL ";"
 DOT     "."
 COMMA   ","
 BACKQUOTE "`"
 AT        "@"
 ELLIPSES  "..."
 LDots "<..." RDots "...>"

(* operators *)
%token <AST_python.tok>
  ADD            (* + *)  SUB            (* - *)
  MULT "*"       (* * *)  DIV "/"        (* / *)
  MOD            (* % *)
  POW  "**"      (* ** *)  FDIV           (* // *)
  BITOR          (* | *)  BITAND         (* & *)  BITXOR         (* ^ *)
  BITNOT         (* ~ *)  LSHIFT         (* << *)  RSHIFT         (* >> *)

%token <AST_python.tok>
  EQ "="             (* = *)
  COLONEQ ":="   (* := *)
  ADDEQ          (* += *) SUBEQ          (* -= *)
  MULTEQ         (* *= *) DIVEQ          (* /= *)
  MODEQ          (* %= *)
  POWEQ          (* **= *) FDIVEQ         (* //= *)
  ANDEQ          (* &= *) OREQ           (* |= *) XOREQ          (* ^= *)
  LSHEQ          (* <<= *) RSHEQ          (* >>= *)

  EQUAL          (* == *) NOTEQ          (* !=, <> *)
  LT             (* < *) GT             (* > *)
  LEQ            (* <= *) GEQ            (* >= *)

(*----------------------------------------*)
(* Extra tokens: *)
(*----------------------------------------*)
(* fstrings *)
%token <AST_python.tok> FSTRING_START FSTRING_END
%token <AST_python.tok> FSTRING_LBRACE
%token <string * AST_python.tok> FSTRING_STRING
%token <AST_python.tok> BANG

(* layout *)
%token <AST_python.tok> INDENT DEDENT
%token <AST_python.tok> NEWLINE

(*****************************************************************************)
(* Rules type declaration *)
(*****************************************************************************)
%start <AST_python.program> main
%start <AST_python.any> sgrep_spatch_pattern
%%

(*****************************************************************************)
```

```
(* Macros *)
(*****************************************************************************)

list_sep(X,Sep):
 | X                       { [$1] }
 | list_sep(X,Sep) Sep X  { $1 @ [$3] }

(* list separated by Sep and possibly terminated by trailing Sep.
 * This has to be recursive on the right, otherwise s/r conflict.
 *)
list_sep_term(X,Sep):
 | X                        { [$1] }
 | X Sep                    { [$1] }
 | X Sep list_sep_term(X,Sep)  { $1::$3 }

list_comma(X): list_sep_term(X, ",") { $1 }

tuple(X):
 | X               { Single $1 }
 | X ","           { Tup [$1] }
 | X "," tuple(X)  { cons $1 $3 }

(*****************************************************************************)
(* Toplevel *)
(*****************************************************************************)

main:
 | file_input EOF { $1 }
 (* Handles trailing indentation.
  * Note that I couldn't figure out why the lexer spits this out,
  * but I didn't want to mess with its state machine too much,
  * so I just match against the relavent output here.
  *)
 | file_input INDENT NEWLINE DEDENT NEWLINE EOF { $1 }

file_input: nl_or_stmt* { List.flatten $1 }

nl_or_stmt:
 | NEWLINE { [] }
 | stmt    { $1 }

sgrep_spatch_pattern:
 | stmt NEWLINE? EOF   {
   match $1 with
   | [ExprStmt x] -> Expr x
   | [x] -> Stmt x
   | xs -> Stmts xs
 }
 | stmt stmt+ NEWLINE? EOF { Stmts ($1 @ (List.flatten $2)) }

(*****************************************************************************)
(* Import *)
(*****************************************************************************)
(* In Python, imports can actually appear not just at the toplevel *)
import_stmt:
  | import_name { $1 }
  | import_from { $1 }


import_name: IMPORT list_sep(dotted_as_name, ",")
```

```
  { $2 |> List.map (fun (v1, v2) -> let dots = None in
         ImportAs ($1, (v1, dots), v2))   }

dotted_as_name:
  | dotted_name         { $1, None }
  | dotted_name AS NAME { $1, Some $3 }

dotted_name:
  | NAME { [$1] }
  | NAME "." dotted_name { $1::$3 }


import_from:
  | FROM name_and_level IMPORT "*"
      { [ImportAll ($1, $2, $4)] }
  | FROM name_and_level IMPORT "(" list_comma(import_as_name) ")"
      { [ImportFrom ($1, $2, $5)] }
  | FROM name_and_level IMPORT list_comma(import_as_name)
      { [ImportFrom ($1, $2, $4)] }

name_and_level:
  | dot_level dotted_name {
    match $1 with
    | [] -> $2, None
    | dl -> $2, Some dl
  }
  | "." dot_level         { [("",$1(*TODO*))], Some ($1 :: $2) }
  | "..." dot_level         { [("",$1(*TODO*))], Some ($1:: $2) }

dot_level:
  | (*empty *) { [] }
  | "." dot_level  { $1::$2 }
  | "..." dot_level { $1::$2 }

import_as_name:
  | NAME         { $1, None }
  | NAME AS NAME { $1, Some $3 }

(************************************************************************)
(* Variable definition *)
(************************************************************************)

namedexpr_test:
  | test { $1 }
  | test COLONEQ test { NamedExpr ($1, $2, $3) }

expr_stmt:
  | tuple(test_or_star_expr)
      { ExprStmt (tuple_expr $1) }
  (* typing-ext: *)
  | tuple(test_or_star_expr) ":" test
      { ExprStmt (TypedExpr (tuple_expr $1, $3)) }
  | tuple(test_or_star_expr) ":" test "=" test
      { Assign ([TypedExpr (tuple_expr_store $1, $3)], $4, $5) }

  | tuple(test_or_star_expr) augassign yield_expr
      { AugAssign (tuple_expr_store $1, $2, $3) }
  | tuple(test_or_star_expr) augassign tuple(test)
      { AugAssign (tuple_expr_store $1, $2, tuple_expr $3) }
  | tuple(test_or_star_expr) "=" expr_stmt_rhs_list
```

```
      { Assign ((tuple_expr_store $1)::(fst $3), $2, snd $3) }

namedexpr_or_star_expr:
  | namedexpr_test { $1 }
  | star_expr      { $1 }

test_or_star_expr:
  | test      { $1 }
  | star_expr { $1 }

expr_or_star_expr:
  | expr      { $1 }
  | star_expr { $1 }

exprlist: tuple(expr_or_star_expr) { $1 }

expr_stmt_rhs_list:
  | expr_stmt_rhs                          { [], $1 }
  | expr_stmt_rhs "=" expr_stmt_rhs_list  { (expr_store $1)::(fst $3), snd $3 }

expr_stmt_rhs:
  | yield_expr               { $1 }
  | tuple(test_or_star_expr) { tuple_expr $1 }

augassign:
  | ADDEQ   { Add, $1 }
  | SUBEQ   { Sub, $1 }
  | MULTEQ  { Mult, $1 }
  | DIVEQ   { Div, $1 }
  | POWEQ   { Pow, $1 }
  | MODEQ   { Mod, $1 }
  | LSHEQ   { LShift, $1 }
  | RSHEQ   { RShift, $1 }
  | OREQ    { BitOr, $1 }
  | XOREQ   { BitXor, $1 }
  | ANDEQ   { BitAnd, $1 }
  | FDIVEQ  { FloorDiv, $1 }

(*************************************************************************)
(* Function definition *)
(*************************************************************************)
(* this rule is referenced in compound_stmt shown later *)
funcdef: DEF NAME parameters return_type? ":" suite
    { FunctionDef ($1, $2, $3, $4, $6, []) }

async_funcdef: ASYNC DEF NAME parameters return_type? ":" suite
    { FunctionDef ($2, $3, $4, $5, $7, [] (* TODO $1 *)) }

(* typing-ext: *)
return_type:
  | SUB GT test     { $3 }

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

parameters: "(" typedargslist ")" { $2 }

(* typing-ext: *)
typedargslist:
```

```
  | (*empty*)                       { [] }
  | typed_parameter                     { [$1] }
  | typed_parameter "," typedargslist { $1::$3 }

(* the original grammar enforces more restrictions on the order between
   * Param, ParamStar, and ParamPow, but each language version relaxed it *)
typed_parameter:
  | tfpdef_or_fpdef { ParamPattern (fst $1, snd $1) }
  (* TODO check default args come after variable args later *)
  | tfpdef "=" test { ParamDefault ($1, $3) }
  | "*" tfpdef      { ParamStar ($1, $2) }
  | "*"             { ParamSingleStar $1 }
  (* python3-ext: https://www.python.org/dev/peps/pep-0570/ *)
  | "/"             { ParamSlash $1 }
  | "**" tfpdef     { ParamPow ($1, $2) }
  (* sgrep-ext: *)
  | "..."           { Flag_parsing.sgrep_guard (ParamEllipsis $1) }

tfpdef:
  | NAME            { $1, None }
  (* typing-ext: *)
  | NAME ":" test   { $1, Some $3 }

tfpdef_or_fpdef:
  | tfpdef          { PatternName (fst $1), snd $1 }
  (* python2-ext:
   * Note that this allows mixed typed and pattern parameters,
   * which are actually exclusive between Python 2 and 3
   *)
  | "(" fplist ")"  { PatternTuple $2, None }


(* without types, as in lambda *)
varargslist:
  | (*empty*)               { [] }
  | parameter                   { [$1] }
  | parameter "," varargslist { $1::$3 }

(* python3-ext: can be in any order, ParamStar before or after Classic *)
parameter:
  | fpdef           { ParamPattern ($1, None) }
  | NAME "=" test   { ParamDefault (($1, None), $3) }
  | "*" NAME        { ParamStar ($1, ($2, None)) }
  (* python3-ext: https://www.python.org/dev/peps/pep-0570/ *)
  | "/"             { ParamSlash $1 }
  | "**" NAME       { ParamPow ($1, ($2, None)) }

fpdef:
  | NAME          { PatternName $1 }
  | "(" fplist ")" { PatternTuple $2 }

fplist:
  | fpdef            { [$1] }
  | fpdef "," fplist { $1::$3 }

(*****************************************************************************)
(* Class definition *)
(*****************************************************************************)

classdef: CLASS NAME arglist_paren_opt ":" suite
```

```
  { ClassDef ($1, $2, $3, $5, []) }

arglist_paren_opt:
 | (* empty *) { [] }
 | "(" ")"     { [] }
 (* python3-ext: was expr_list before *)
 | "(" list_comma(argument) ")" { $2 }

(*************************************************************************)
(* Annotations *)
(*************************************************************************)

decorator: "@" decorator_name arglist_paren2_opt NEWLINE
    { $1, $2, $3 }

decorator_name:
  | NAME                    { [$1] }
  | decorator_name "." NAME { $1 @ [$3] }

arglist_paren2_opt:
 | (* empty *) { None }
 | "(" ")"     { Some ($1, [], $2) }
 (* python3-ext: was expr_list before *)
 | "(" list_comma(argument) ")" { Some ($1, $2, $3) }

(*************************************************************************)
(* Statement *)
(*************************************************************************)

stmt:
  | simple_stmt { $1 }
  | compound_stmt { [$1] }

simple_stmt:
  | small_stmt NEWLINE { $1 }
  | small_stmt ";" NEWLINE { $1 }
  | small_stmt ";" simple_stmt { $1 @ $3 }

small_stmt:
  | expr_stmt   { [$1] }
  | del_stmt    { [$1] }
  | pass_stmt   { [$1] }
  | flow_stmt   { [$1] }
  | import_stmt { $1 }
  | global_stmt { [$1] }
  | nonlocal_stmt { [$1] }
  | assert_stmt { [$1] }
  (* python2: *)
  | print_stmt { [$1] }
  | exec_stmt { [$1] }

(* for expr_stmt see above *)

(* python2: *)
print_stmt:
  | PRINT                     { Print ($1, None, [], true) }
  | PRINT test print_testlist { Print ($1, None, $2::(fst $3), snd $3) }
  | PRINT RSHIFT test { Print ($1, Some $3, [], true) }
  | PRINT RSHIFT test "," test print_testlist
      { Print ($1, Some $3, $5::(fst $6), snd $6) }
```

```
print_testlist:
  | (* empty *)  { [], true }
  | ","          { [], false }
  | "," test print_testlist { $2::(fst $3), snd $3 }

exec_stmt:
  | EXEC expr { Exec ($1, $2, None, None) }
  | EXEC expr IN test { Exec ($1, $2, Some $4, None) }
  | EXEC expr IN test "," test { Exec ($1, $2, Some $4, Some $6) }


del_stmt: DEL exprlist { Delete ($1, List.map expr_del (to_list $2)) }

pass_stmt: PASS { Pass $1 }


flow_stmt:
  | break_stmt    { $1 }
  | continue_stmt { $1 }
  | return_stmt   { $1 }
  | raise_stmt    { $1 }
  | yield_stmt    { $1 }

break_stmt:    BREAK    { Break $1  }
continue_stmt: CONTINUE { Continue $1 }

return_stmt:
  | RETURN          { Return ($1, None) }
  | RETURN tuple(test) { Return ($1, Some (tuple_expr $2)) }

yield_stmt: yield_expr { ExprStmt ($1) }

raise_stmt:
  | RAISE                          { Raise ($1, None) }
  | RAISE test                     { Raise ($1, Some ($2, None)) }
  (* python3-ext: *)
  | RAISE test FROM test           { Raise ($1, Some ($2, Some $4)) }
  (* python2-ext: *)
  | RAISE test "," test            { RaisePython2 ($1, $2, Some $4, None) }
  | RAISE test "," test "," test   { RaisePython2 ($1, $2, Some $4, Some $6) }


global_stmt: GLOBAL list_sep(NAME, ",") { Global ($1, $2) }

(* python3-ext: *)
nonlocal_stmt: NONLOCAL list_sep(NAME, ",") { NonLocal ($1, $2) }

assert_stmt:
  | ASSERT test          { Assert ($1, $2, None) }
  | ASSERT test "," test { Assert ($1, $2, Some $4) }



compound_stmt:
  | if_stmt     { $1 }
  | while_stmt  { $1 }
  | for_stmt    { $1 }
  | try_stmt    { $1 }
  | with_stmt   { $1 }
```

```
  | funcdef     { $1 }
  | classdef    { $1 }
  | decorated   { $1 }
  (* Note that there is no async_funcdef above. To avoid conflict
     * with async_stmt below, Python enforces the def to be decorated.
     *)
  | async_stmt  { $1 }

decorated:
  | decorator+ classdef {
     match $2 with
     | ClassDef (t, a, b, c, d) -> ClassDef (t, a, b, c, $1 @ d)
     | _ -> raise Impossible
   }
  | decorator+ funcdef {
     match $2 with
     | FunctionDef (t, a, b, c, d, e) -> FunctionDef (t, a, b, c, d, $1 @ e)
     | _ -> raise Impossible
   }
  | decorator+ async_funcdef {
     match $2 with
     | FunctionDef (t, a, b, c, d, e) -> FunctionDef (t, a, b, c, d, $1 @ e)
     | _ -> raise Impossible
  }

(* this is always preceded by a ":" *)
suite:
  | simple_stmt { $1 }
  | NEWLINE INDENT stmt* DEDENT { List.flatten $3 }


if_stmt: IF namedexpr_test ":" suite elif_stmt_list { If ($1, $2, $4, $5) }

elif_stmt_list:
  | (*empty *)                                  { None }
  | ELIF namedexpr_test ":" suite elif_stmt_list { Some [If ($1, $2, $4, $5)] }
  | ELSE ":" suite                              { Some ($3) }


while_stmt:
  | WHILE namedexpr_test ":" suite                { While ($1, $2, $4, []) }
  | WHILE namedexpr_test ":" suite ELSE ":" suite { While ($1, $2, $4, $7) }


for_stmt:
  | FOR exprlist IN tuple(test) ":" suite
      { For ($1, tuple_expr_store $2, $3, tuple_expr $4, $6, []) }
  | FOR exprlist IN tuple(test) ":" suite ELSE ":" suite
      { For ($1, tuple_expr_store $2, $3, tuple_expr $4, $6, $9) }


try_stmt:
  | TRY ":" suite excepthandler+
      { TryExcept ($1, $3, $4, []) }
  | TRY ":" suite excepthandler+ ELSE ":" suite
      { TryExcept ($1, $3, $4, $7) }
  | TRY ":" suite excepthandler+ ELSE ":" suite FINALLY ":" suite
      { TryFinally ($1, [TryExcept ($1, $3, $4, $7)], $8, $10) }
  | TRY ":" suite excepthandler+ FINALLY ":" suite
```

```
      { TryFinally ($1, [TryExcept ($1, $3, $4, [])], $5, $7) }
  | TRY ":" suite FINALLY ":" suite
      { TryFinally ($1, $3, $4, $6) }

excepthandler:
  | EXCEPT              ":" suite { ExceptHandler ($1, None, None, $3) }
  | EXCEPT test         ":" suite { ExceptHandler ($1, Some $2, None, $4) }
  | EXCEPT test AS NAME ":" suite { ExceptHandler ($1, Some $2, Some $4, $6)}
  (* python2: *)
  | EXCEPT test "," NAME ":" suite { ExceptHandler ($1, Some $2, Some $4, $6) }

with_stmt: WITH with_inner { $2 $1 }

with_inner:
  | test         ":" suite      { fun t -> With (t, $1, None, $3) }
  | test AS expr ":" suite      { fun t -> With (t, $1, Some $3, $5) }
  | test         "," with_inner { fun t -> With (t, $1, None, [$3 t]) }
  | test AS expr "," with_inner { fun t -> With (t, $1, Some $3, [$5 t]) }

(* python3-ext: *)
async_stmt:
  | ASYNC funcdef   { Async ($1, $2) }
  | ASYNC with_stmt { Async ($1, $2) }
  | ASYNC for_stmt  { Async ($1, $2) }

(************************************************************************)
(* Expressions *)
(************************************************************************)

expr:
  | xor_expr            { $1 }
  | expr BITOR xor_expr { BinOp ($1, (BitOr,$2), $3) }


xor_expr:
  | and_expr                  { $1 }
  | xor_expr BITXOR and_expr { BinOp ($1, (BitXor,$2), $3) }

and_expr:
  | shift_expr                 { $1 }
  | shift_expr BITAND and_expr { BinOp ($1, (BitAnd,$2), $3) }


shift_expr:
  | arith_expr                   { $1 }
  | shift_expr LSHIFT arith_expr { BinOp ($1, (LShift,$2), $3) }
  | shift_expr RSHIFT arith_expr { BinOp ($1, (RShift,$2), $3) }

arith_expr:
  | term               { $1 }
  | arith_expr ADD term { BinOp ($1, (Add,$2), $3) }
  | arith_expr SUB term { BinOp ($1, (Sub,$2), $3) }


term:
  | factor             { $1 }
  | factor term_op term { BinOp ($1, $2, $3) }

term_op:
  | "*"     { Mult, $1 }
```

```
  | DIV     { Div, $1 }
  | MOD     { Mod, $1 }
  | FDIV    { FloorDiv, $1 }
  | "@"     { MatMult, $1 }

factor:
  | ADD factor    { UnaryOp ((UAdd,$1), $2) }
  | SUB factor    { UnaryOp ((USub,$1), $2) }
  | BITNOT factor { UnaryOp ((Invert,$1), $2) }
  | power         { $1 }

power:
  | atom_expr            { $1 }
  | atom_expr "**" factor { BinOp ($1, (Pow,$2), $3) }

(*----------------------------*)
(* Atom expr *)
(*----------------------------*)

atom_expr:
  | atom_and_trailers        { $1 }
  | AWAIT atom_and_trailers  { Await ($1, $2) }

atom_and_trailers:
  | atom { $1 }

  | atom_and_trailers "("          ")" { Call ($1, ($2,[],$3)) }
  | atom_and_trailers "(" list_comma(argument) ")" { Call ($1, ($2,$3,$4)) }

  | atom_and_trailers "[" list_comma(subscript)   "]"
      { match $3 with
          (* TODO test* => Index (Tuple (elts)) *)
        | [s] -> Subscript ($1, ($2, [s], $4), Load)
        | l -> Subscript ($1, ($2, l, $4), Load) }

  | atom_and_trailers "." NAME { Attribute ($1, $2, $3, Load) }

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

atom:
  | NAME        { Name ($1, Load, ref NotResolved) }

  | INT         { Num (Int ($1)) }
  | LONGINT     { Num (LongInt ($1)) }
  | FLOAT       { Num (Float ($1)) }
  | IMAG        { Num (Imag ($1)) }

  | TRUE        { Bool (true, $1) }
  | FALSE       { Bool (false, $1) }

  | NONE        { None_ $1 }

  | string+ {
     match $1 with
     | [] ->  raise Common.Impossible
     | [x] -> x
     | xs -> ConcatenatedString xs
   }
```

```
  | atom_tuple  { $1 }
  | atom_list   { $1 }
  | atom_dict   { $1 }

  | atom_repr   { $1 }

  (* typing-ext: sgrep-ext: *)
  | "..."              { Ellipsis $1 }
  | "<..." test "...>" { Flag_parsing.sgrep_guard (DeepEllipsis ($1, $2, $3)) }

atom_repr: "`" testlist1 "`" { Repr ($1, tuple_expr $2, $3) }

testlist1:
  | test                 { Single $1 }
  | test "," testlist1 { cons $1 $3 }

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

string:
  | STR { let (s, pre, tok) = $1 in
          if pre = "" then Str (s, tok) else EncodedStr ((s, tok), pre) }
  | FSTRING_START interpolated* FSTRING_END { InterpolatedString $2 }

interpolated:
  | FSTRING_STRING { Str $1 }
  | FSTRING_LBRACE interpolant fstring_print_spec "}" { InterpolatedString ($2::$3) }

fstring_print_spec:
  |     fstring_format_clause { $1 }
  | "=" fstring_format_clause { mk_str $1::$2 }

fstring_format_clause:
  | (*empty*) { [] }
  | fstring_format_delimeter format_specifier { mk_str $1::$2 }

fstring_format_delimeter:
  | ":" { $1 }
  | BANG { $1 }

interpolant:
  (* Note that the f-string mini-language at
   * https://docs.python.org/3/library/string.html#format-string-syntax
   * simply states that this is parsed as an "expression"; we are now
   * left trying to determine to which grammar rule an "expression"
   * corresponds. However, testing with the cpython interpreter indicates
   * that the testlist rule is what applies, as strings like:
   *   f"{not True, 1}"
   * are parsed by the interpreter.
   *
   * "interpolant" is the "value" inside one of:
   * f"{value}"
   * f"{value:format}"
   * f"{value!format}"
   *)
  | tuple(test) { tuple_expr $1 }

(* todo: maybe need another lexing state when ":" inside FSTRING_LBRACE*)
```

```
format_specifier: format_token+ { $1 }

(* see "Format Specification Mini-Language" at
 * https://docs.python.org/3/library/string.html#format-string-syntax
 *)
format_token:
  | INT   { mk_str (snd $1) }
  | FLOAT { mk_str (snd $1) }
  | NAME  { mk_str (snd $1) }
  | MOD { mk_str $1 } (* for dates *)
  | LT { mk_str $1 } | GT { mk_str $1 }
  | BITXOR { mk_str $1 }
  | ADD  { mk_str $1 } | SUB { mk_str $1 }
  | DIV { mk_str $1 }
  | "."   { mk_str $1 }
  | "=" { mk_str $1 }
  | "," { mk_str $1 }
  | ":" { mk_str $1 }

  | "{" test "}" { $2 }


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

atom_tuple:
  | "("               ")"         { Tuple (CompList ($1, [], $2), Load) }
  | "(" testlist_comp_or_expr ")" { $2 }
  | "(" yield_expr    ")"         { $2 }

atom_list:
  | "["               "]" { List (CompList ($1, [], $2), Load) }
  | "[" testlist_comp "]" { List ($2 ($1, $3), Load) }

atom_dict:
  | "{"                "}" { DictOrSet (CompList ($1, [], $2)) }
  | "{" dictorsetmaker "}" { DictOrSet ($2 ($1, $3)) }

dictorsetmaker:
  | dictorset_elem comp_for { fun _ -> CompForIf ($1, $2) }
  | list_comma(dictorset_elem)     { fun (t1, t2) -> CompList (t1, $1, t2) }

dictorset_elem:
  | test ":" test { KeyVal ($1, $3) }
  | test           { Key $1 }
  | star_expr      { Key $1 }
  (* python3-ext: *)
  | "**" expr        { PowInline $2 }

(*----------------------------*)
(* Array access *)
(*----------------------------*)

subscript:
  | test { Index ($1) }
  | test? ":" test? { Slice ($1, $3, None) }
  | test? ":" test? ":" test? { Slice ($1, $3, $5) }

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

```
(* test *)
(*----------------------------*)

test:
  | or_test                      { $1 }
  | or_test IF or_test ELSE test { IfExp ($3, $1, $5) }
  | lambdadef                    { $1 }


or_test:
  | and_test                           { $1 }
  | and_test OR list_sep(and_test, OR) { BoolOp ((Or,$2), $1::$3) }

and_test:
  | not_test                             { $1 }
  | not_test AND list_sep(not_test, AND) { BoolOp ((And,$2), $1::$3) }


not_test:
  | NOT not_test { UnaryOp ((Not,$1), $2) }
  | comparison   { $1 }

comparison:
  | expr                         { $1 }
  | expr comp_op comparison_list { Compare ($1, ($2)::(fst $3), snd $3) }

comparison_list:
  | expr                         { [], [$1] }
  | expr comp_op comparison_list { ($2)::(fst $3), $1::(snd $3) }

comp_op:
  | EQUAL   { Eq, $1 }
  | NOTEQ   { NotEq, $1 }
  | LT      { Lt, $1 }
  | LEQ     { LtE, $1 }
  | GT      { Gt, $1 }
  | GEQ     { GtE, $1 }
  | IS      { Is, $1 }
  | IS NOT  { IsNot, $1 }
  | IN      { In, $1 }
  | NOT IN  { NotIn, $1 }

(*----------------------------*)
(* Advanced features *)
(*----------------------------*)

(* python3-ext: *)
star_expr: "*" expr { ExprStar $2 }


yield_expr:
  | YIELD           { Yield ($1, None, false) }
  | YIELD FROM test { Yield ($1, Some $3, true) }
  | YIELD tuple(test)  { Yield ($1, Some (tuple_expr $2), false) }

lambdadef: LAMBDA varargslist ":" test { Lambda ($1, $2, $3, $4) }

(*----------------------------*)
(* Comprehensions *)
(*----------------------------*)
```

```
testlist_comp:
  | namedexpr_or_star_expr listcomp_for  { fun _ -> CompForIf ($1, $2) }
  | tuple(namedexpr_or_star_expr)
      { fun (t1, t2) -> CompList (t1, to_list $1, t2) }

(* mostly equivalent to testlist_comp, but transform a single expression
 * in parenthesis, e.g., (1) in a regular expr, not a tuple *)
testlist_comp_or_expr:
  | namedexpr_or_star_expr comp_for  { Tuple (CompForIf ($1, $2), Load) }
  | tuple(namedexpr_or_star_expr)    {
    match $1 with
    | Single e -> e
    | Tup l -> Tuple (CompList (PI.fake_bracket l), Load)
   }

(* supports comp_for when used generically -- not inside atom_list
 * Note that the division here is necessary to solve a shift-reduce conflict between:
 *   foo(x for x in bar, baz)
 * in Python 3, and
 *   foo = [x for x in 1, 2]
 * in Python 2
 *)
comp_for:
 | sync_comp_for       { $1 }
 | ASYNC sync_comp_for { (* TODO *) $2 }

sync_comp_for:
  | FOR exprlist IN or_test
    { [CompFor (tuple_expr_store $2, $4)] }
  | FOR exprlist IN or_test comp_iter
    { [CompFor (tuple_expr_store $2, $4)] @ $5 }


(* support mixed python2 / python3 comp_for only when used inside atom_list *)
listcomp_for:
 | listsync_comp_for       { $1 }
 | ASYNC listsync_comp_for { (* TODO *) $2 }

list_for:
  or_test "," list_for_rest {
    List (CompList (PI.fake_bracket ($1::$3)), Load)
  }

list_for_rest:
  | or_test                   { [$1] }
  | or_test "," list_for_rest { $1::$3 }

listsync_comp_for:
  | sync_comp_for { $1 }
  (* python2-ext: [x for x in 1, 2] *)
  | FOR exprlist IN list_for { [CompFor (tuple_expr_store $2, $4) ] }

(* /comp_for *)

comp_iter:
  | comp_for { $1 }
  | comp_if  { $1 }

comp_if:
```

```
  | IF test_nocond           { [CompIf ($2)] }
  | IF test_nocond comp_iter { [CompIf ($2)] @ $3 }

test_nocond:
  | or_test          { $1 }
  | lambdadef_nocond { $1 }

lambdadef_nocond: LAMBDA varargslist ":" test_nocond
    { Lambda ($1, $2, $3, $4) }


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

(* python3-ext: can be any order, ArgStar before or after ArgKwd *)
argument:
  | test           { Arg $1 }
  | test comp_for  { ArgComp ($1, $2) }

  (* python3-ext: *)
  | test COLONEQ test  { Arg (NamedExpr ($1, $2, $3)) }
  | "*" test           { ArgStar ($1, $2) }
  | "**" test          { ArgPow  ($1, $2) }

  (* sgrep-ext: difficult to move in atom without s/r conflict so restricted
     * to argument for now *)
  | NAME ":" test
    { Flag_parsing.sgrep_guard (Arg (TypedMetavar ($1, $2, $3))) }

  | test "=" test
      { match $1 with
        | Name (id, _, _) -> ArgKwd (id, $3)
        | _ -> raise Parsing.Parse_error
      }
```

pfff/lang_python/parsing/Parse_python.mli

⟨*signature* Parse_python.program_of_string 359a⟩≡ (359c)

```
  (* to help write test code *)
  val program_of_string: string -> AST_python.program
```

⟨*signature* Parse_python.tokens 359b⟩≡ (359c)

```
  (* internal *)
  val tokens: parsing_mode -> Common.filename -> Parser_python.token list
```

⟨pfff/lang_python/parsing/Parse_python.mli 359c⟩≡

⟨*type* Parse_python.program_and_tokens 136a⟩

⟨*type* Parse_python.parsing_mode 136b⟩

⟨*signature* Parse_python.parse 136c⟩

⟨*signature* Parse_python.parse_program 136d⟩

```
  (* other parsers *)
```

⟨*signature* Parse_python.any_of_string 139b⟩

```
(* for sgrep via fuzzy AST *)
(*
val parse_fuzzy:
  Common.filename -> Ast_fuzzy.trees * Parser_python.token list
*)
```

⟨*signature* Parse_python.program_of_string 359a⟩

⟨*signature* Parse_python.tokens 359b⟩

pfff/lang_python/parsing/Parse_python.ml

⟨pfff/lang_python/parsing/Parse_python.ml 360⟩≡

```
(* Yoann Padioleau
 *
 * Copyright (C) 2010 Facebook
 * Copyright (C) 2019 r2c
 *
 * 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.
 *
 * 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.
*)
open Common

module Flag = Flag_parsing
module TH   = Token_helpers_python
module PI = Parse_info
module Lexer = Lexer_python
module T = Parser_python

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

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* Parse_python.program_and_tokens 136a⟩

⟨*type* Parse_python.parsing_mode 136b⟩

```
(*****************************************************************************)
(* Error diagnostic  *)
(*****************************************************************************)
```

⟨*function* Parse_python.error_msg_tok 134a⟩

```
(*****************************************************************************)
(* Lexing only *)
(*****************************************************************************)
```

⟨*function* Parse_python.tokens2 134b⟩

```
[@@profiling]
```

```
⟨function Parse_python.tokens 135a⟩

(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)

⟨function Parse_python.parse_basic 136e⟩


⟨function Parse_python.parse 137a⟩

⟨function Parse_python.parse_program 137b⟩

(*****************************************************************************)
(* Sub parsers *)
(*****************************************************************************)

let (program_of_string: string -> AST_python.program) = fun s ->
  Common2.with_tmp_file ~str:s ~ext:"py" (fun file ->
    parse_program file
  )

⟨function Parse_python.any_of_string 139c⟩


(*****************************************************************************)
(* Fuzzy parsing *)
(*****************************************************************************)

(*
let parse_fuzzy file =
  let toks = tokens file in
  let trees = Parse_fuzzy.mk_trees { Parse_fuzzy.
     tokf = TH.info_of_tok;
     kind = TH.token_kind_of_tok;
  } toks
  in
  trees, toks
*)
```

pfff/lang_python/parsing/Lib_parsing_python.mli

```
⟨pfff/lang_python/parsing/Lib_parsing_python.mli 361a⟩≡
  ⟨signature Lib_parsing_python.find_source_files_of_dir_or_files 140d⟩

  ⟨signature Lib_parsing_python.ii_of_any 140e⟩
```

pfff/lang_python/parsing/Lib_parsing_python.ml

```
⟨pfff/lang_python/parsing/Lib_parsing_python.ml 361b⟩≡
  (* Yoann Padioleau
   *
   * Copyright (C) 2010 Facebook
   *
   * This library is free software; you can redistribute it and/or
   * modify it under the terms of the GNU Lesser General Public License
   * version 2.1 as published by the Free Software Foundation, with the
```

```
 * special exception on linking described in file license.txt.
 *
 * This library 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.
*)

module V = Visitor_python

(*****************************************************************************)
(* Filenames *)
(*****************************************************************************)

⟨function Lib_parsing_python.find_source_files_of_dir_or_files 140f⟩

(*****************************************************************************)
(* Extract infos *)
(*****************************************************************************)

⟨function Lib_parsing_python.extract_info_visitor 140g⟩

⟨function Lib_parsing_python.ii_of_any 140h⟩
```

pfff/lang_python/parsing/Parsing_hacks_python.mli

⟨pfff/lang_python/parsing/Parsing_hacks_python.mli 362a⟩≡

```
⟨signature Parsing_hacks_python.fix_tokens 137c⟩
```

pfff/lang_python/parsing/Parsing_hacks_python.ml

⟨pfff/lang_python/parsing/Parsing_hacks_python.ml 362b⟩≡

```
⟨pad/r2c copyright 11⟩

module T = Parser_python

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* The goal for this module is to add extra "closing" tokens
 * for the grammar to remain simple. The closing tokens
 * are usually one NEWLINE and a few DEDENTs.
 *
 * This file:
 *  class A:
 *     def f():
 *        bar()
 *
 * is tokenized as (skipping the TCommentSpace):
 *  [class; A; :; NEWLINE; INDENT;
 *   def; f; (; ); :; NEWLINE; IDENT;
 *   bar; (; ); NEWLINE; #1
 *   DEDENT; DEDENT; #2
 *   NEWLINE; EOF
 *
 * Instead if bar() is replaced by '...', many .pyi files do not have
 * the trailing NEWLINE which causes some missing DEDENT which would
```

```
 * cause parsing errors. This is why for those files we must
 * insert the NEWLINE at #1, and matching DEDENT at #2.
 *
 * alt:
 *  - could insert those closing tokens during error recovery
 *  - could look at state.offset_stack when encounters EOF in the lexer
 *    and also pop and create the DEDENT.
*)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Parsing_hacks_python.add_dedent_aux 137d⟩

⟨function Parsing_hacks_python.add_dedent 138a⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨function Parsing_hacks_python.fix_tokens 138b⟩
```

pfff/lang_python/parsing/Token_helpers_python.mli

⟨pfff/lang_python/parsing/Token_helpers_python.mli 363a⟩≡

```
⟨signature Token_helpers_python.is_eof 135b⟩
⟨signature Token_helpers_python.is_comment 135c⟩

⟨signature Token_helpers_python.info_of_tok 135g⟩
⟨signature Token_helpers_python.visitor_info_of_tok 135f⟩
```

pfff/lang_python/parsing/Token_helpers_python.ml

⟨function Token_helpers_python.visitor_info_of_tok 363b⟩≡ (365)

```
let visitor_info_of_tok f = function

  | TUnknown ii -> TUnknown (f ii)
  | EOF ii -> EOF (f ii)
  | TCommentSpace ii -> TCommentSpace (f ii)
  | TComment ii -> TComment (f ii)

  | FSTRING_START ii -> FSTRING_START (f ii)
  | FSTRING_END ii -> FSTRING_END (f ii)
  | FSTRING_LBRACE ii -> FSTRING_LBRACE (f ii)
  | FSTRING_STRING (x, ii) -> FSTRING_STRING (x, f ii)
  | BANG ii -> BANG (f ii)

  | NAME (x, ii) -> NAME (x, f ii)
  | INT (x, ii) -> INT (x, f ii)
  | LONGINT (x, ii) -> LONGINT (x, f ii)
  | FLOAT (x, ii) -> FLOAT (x, f ii)
  | IMAG (x, ii) -> IMAG (x, f ii)
  | STR (x, pre, ii) -> STR (x, pre, f ii)

  | NONE ii -> NONE (f ii)
  | TRUE ii -> TRUE (f ii)
```

```
| FALSE ii -> FALSE (f ii)
| ASYNC ii -> ASYNC (f ii)
| AWAIT ii -> AWAIT (f ii)
| NONLOCAL ii -> NONLOCAL (f ii)

| ELLIPSES ii -> ELLIPSES (f ii)
| LDots ii -> LDots (f ii)
| RDots ii -> RDots (f ii)

| AND ii -> AND (f ii)
| AS ii -> AS (f ii)
| ASSERT ii -> ASSERT (f ii)
| BREAK ii -> BREAK (f ii)
| CLASS ii -> CLASS (f ii)
| CONTINUE ii -> CONTINUE (f ii)
| DEF ii -> DEF (f ii)
| DEL ii -> DEL (f ii)
| ELIF ii -> ELIF (f ii)
| ELSE ii -> ELSE (f ii)
| EXCEPT ii -> EXCEPT (f ii)
| FINALLY ii -> FINALLY (f ii)
| FOR ii -> FOR (f ii)
| FROM ii -> FROM (f ii)
| GLOBAL ii -> GLOBAL (f ii)
| IF ii -> IF (f ii)
| IMPORT ii -> IMPORT (f ii)
| IN ii -> IN (f ii)
| IS ii -> IS (f ii)
| LAMBDA ii -> LAMBDA (f ii)
| NOT ii -> NOT (f ii)
| OR ii -> OR (f ii)
| PASS ii -> PASS (f ii)
| RAISE ii -> RAISE (f ii)
| RETURN ii -> RETURN (f ii)
| TRY ii -> TRY (f ii)
| WHILE ii -> WHILE (f ii)
| WITH ii -> WITH (f ii)
| YIELD ii -> YIELD (f ii)
| PRINT ii -> PRINT (f ii)
| EXEC ii -> EXEC (f ii)

| LPAREN ii -> LPAREN (f ii)
| RPAREN ii -> RPAREN (f ii)
| LBRACK ii -> LBRACK (f ii)
| RBRACK ii -> RBRACK (f ii)
| LBRACE ii -> LBRACE (f ii)
| RBRACE ii -> RBRACE (f ii)
| COLON ii -> COLON (f ii)
| SEMICOL ii -> SEMICOL (f ii)
| DOT ii -> DOT (f ii)
| COMMA ii -> COMMA (f ii)
| BACKQUOTE ii -> BACKQUOTE (f ii)
| AT ii -> AT (f ii)
| ADD ii -> ADD (f ii)
| SUB ii -> SUB (f ii)
| MULT ii -> MULT (f ii)
| DIV ii -> DIV (f ii)
| MOD ii -> MOD (f ii)
| POW ii -> POW (f ii)
| FDIV ii -> FDIV (f ii)
```

```
    | BITOR ii -> BITOR (f ii)
    | BITAND ii -> BITAND (f ii)
    | BITXOR ii -> BITXOR (f ii)
    | BITNOT ii -> BITNOT (f ii)
    | LSHIFT ii -> LSHIFT (f ii)
    | RSHIFT ii -> RSHIFT (f ii)
    | EQ ii -> EQ (f ii)
    | COLONEQ ii -> COLONEQ (f ii)
    | ADDEQ ii -> ADDEQ (f ii)
    | SUBEQ ii -> SUBEQ (f ii)
    | MULTEQ ii -> MULTEQ (f ii)
    | DIVEQ ii -> DIVEQ (f ii)
    | MODEQ ii -> MODEQ (f ii)
    | POWEQ ii -> POWEQ (f ii)
    | FDIVEQ ii -> FDIVEQ (f ii)
    | ANDEQ ii -> ANDEQ (f ii)
    | OREQ ii -> OREQ (f ii)
    | XOREQ ii -> XOREQ (f ii)
    | LSHEQ ii -> LSHEQ (f ii)
    | RSHEQ ii -> RSHEQ (f ii)
    | EQUAL ii -> EQUAL (f ii)
    | NOTEQ ii -> NOTEQ (f ii)
    | LT ii -> LT (f ii)
    | GT ii -> GT (f ii)
    | LEQ ii -> LEQ (f ii)
    | GEQ ii -> GEQ (f ii)

    | INDENT ii -> INDENT (f ii)
    | DEDENT ii -> DEDENT (f ii)
    | NEWLINE ii -> NEWLINE (f ii)
```

⟨pfff/lang_python/parsing/Token_helpers_python.ml 365⟩≡

```
  (* Yoann Padioleau
   *
   * Copyright (C) 2010 Facebook
   * Copyright (C) 2019 Yoann Padioleau
   *
   * This library is free software; you can redistribute it and/or
   * modify it under the terms of the GNU Lesser General Public License
   * version 2.1 as published by the Free Software Foundation, with the
   * special exception on linking described in file license.txt.
   *
   * This library 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.
  *)
  open Parser_python

  (*****************************************************************************)
  (* Token Helpers *)
  (*****************************************************************************)
  ⟨function Token_helpers_python.is_eof 135d⟩

  ⟨function Token_helpers_python.is_comment 135e⟩

  (*****************************************************************************)
  (* Visitors *)
  (*****************************************************************************)
```

⟨*function* Token_helpers_python.visitor_info_of_tok 363b⟩

⟨*function* Token_helpers_python.info_of_tok 135h⟩

pfff/lang_python/parsing/Test_parsing_python.mli

⟨pfff/lang_python/parsing/Test_parsing_python.mli 366a⟩≡

⟨*signature* Test_parsing_python.test_tokens_python 135i⟩

⟨*signature* Test_parsing_python.actions 138c⟩

pfff/lang_python/parsing/Test_parsing_python.ml

⟨pfff/lang_python/parsing/Test_parsing_python.ml 366b⟩≡
```
open Common

module Flag = Flag_parsing

(*****************************************************************************)
(* Subsystem testing *)
(*****************************************************************************)
```

⟨*function* Test_parsing_python.test_tokens_python 135j⟩

⟨*function* Test_parsing_python.test_parse_python_common 138d⟩

⟨*function* Test_parsing_python.test_dump_python 133⟩

```
(*****************************************************************************)
(* Main entry for Arg *)
(*****************************************************************************)
```

⟨*function* Test_parsing_python.actions 139a⟩

pfff/lang_python/parsing/Unit_parsing_python.mli

⟨pfff/lang_python/parsing/Unit_parsing_python.mli 366c⟩≡

⟨*signature* Unit_parsing_python.unittest 214c⟩

pfff/lang_python/parsing/Unit_parsing_python.ml

⟨pfff/lang_python/parsing/Unit_parsing_python.ml 366d⟩≡
```
open Common
open OUnit

(*****************************************************************************)
(* Unit tests *)
(*****************************************************************************)
```

⟨*constant* Unit_parsing_python.unittest 214d⟩

pfff/lang_python/parsing/Visitor_python.mli

⟨pfff/lang_python/parsing/Visitor_python.mli 367a⟩≡
```
open AST_python

(* hooks *)
```
⟨*type* Visitor_python.visitor_in 139d⟩
⟨*type* Visitor_python.visitor_out 140a⟩

⟨*signature* Visitor_python.default_visitor 140b⟩

⟨*signature* Visitor_python.mk_visitor 140c⟩

pfff/lang_python/parsing/Meta_AST_python.mli

⟨pfff/lang_python/parsing/Meta_AST_python.mli 367b⟩≡

⟨*signature* Meta_AST_python.vof_program 140i⟩

⟨*signature* Meta_AST_python.vof_any 140j⟩

F.11 pfff/lang_python/analyze/

pfff/lang_python/analyze/Python_to_generic.mli

⟨pfff/lang_python/analyze/Python_to_generic.mli 367c⟩≡

⟨*signature* Python_to_generic.program 159a⟩

⟨*signature* Python_to_generic.any 159b⟩

```
(* exception Error of string * Parse_info.info *)
(* may raise Error *)
```

pfff/lang_python/analyze/Python_to_generic.ml

⟨pfff/lang_python/analyze/Python_to_generic.ml 367d⟩≡
```
⟨pad/r2c copyright 11⟩
open Common
open AST_python
module G = AST_generic
module H = AST_generic_helpers

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* AST_python to AST_generic.
 *
 * See AST_generic.ml for more information.
 *
 * TODO: intercept Call to eval and transform in special Eval?
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*constant* `Python_to_generic.id` 159c⟩
⟨*constant* `Python_to_generic.option` 159d⟩
⟨*constant* `Python_to_generic.list` 159e⟩
⟨*function* `Python_to_generic.vref` 159f⟩

⟨*constant* `Python_to_generic.string` 159g⟩
⟨*constant* `Python_to_generic.bool` 159h⟩

⟨*function* `Python_to_generic.fake` 159i⟩
⟨*function* `Python_to_generic.fake_bracket` 159j⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```

⟨*function* `Python_to_generic.info` 160a⟩

⟨*constant* `Python_to_generic.wrap` 160b⟩

⟨*function* `Python_to_generic.bracket` 160c⟩

⟨*function* `Python_to_generic.name` 160d⟩

⟨*function* `Python_to_generic.dotted_name` 160e⟩

⟨*function* `Python_to_generic.module_name` 160f⟩

⟨*function* `Python_to_generic.resolved_name` 160g⟩

⟨*function* `Python_to_generic.expr_context` 161a⟩

⟨*function* `Python_to_generic.expr` 161d⟩

⟨*function* `Python_to_generic.argument` 164a⟩

⟨*function* `Python_to_generic.for_if` 164b⟩

⟨*function* `Python_to_generic.dictorset_elt` 164c⟩

⟨*function* `Python_to_generic.number` 165a⟩

⟨*function* `Python_to_generic.boolop` 165b⟩

⟨*function* `Python_to_generic.operator` 165c⟩

⟨*function* `Python_to_generic.unarop` 165d⟩

⟨*function* `Python_to_generic.cmpop` 165e⟩

⟨*function* `Python_to_generic.comprehension` 166a⟩

⟨*function* `Python_to_generic.comprehension2` 166b⟩

⟨*function* `Python_to_generic.slice` 166c⟩

```
and param_pattern = function
  | PatternName n -> G.PatId (name n, G.empty_id_info ())
  | PatternTuple t ->
      let t = list param_pattern t in
      G.PatTuple (G.fake_bracket t)
```

⟨*function* Python_to_generic.parameters 166e⟩

⟨*function* Python_to_generic.type_ 166d⟩

⟨*function* Python_to_generic.type_parent 167a⟩

⟨*function* Python_to_generic.list_stmt1 167b⟩

⟨*function* Python_to_generic.stmt_aux 168⟩
```
and ident_and_id_info x =
  let x = name x in
  (x, G.empty_id_info ())
```

⟨*function* Python_to_generic.stmt 171a⟩

⟨*function* Python_to_generic.pattern 171b⟩

⟨*function* Python_to_generic.excepthandler 171c⟩

⟨*function* Python_to_generic.expr_to_attribute 171d⟩

⟨*function* Python_to_generic.decorator 171e⟩

⟨*function* Python_to_generic.alias 172⟩

⟨*function* Python_to_generic.program 161b⟩

⟨*function* Python_to_generic.any 161c⟩

F.12 semgrep/cli

semgrep/cli/Main.ml

⟨semgrep/CLI/Main.ml 369⟩≡
```
(*
 * The author disclaims copyright to this source code.  In place of
 * a legal notice, here is a blessing:
 *
 *    May you do good and not evil.
 *    May you find forgiveness for yourself and forgive others.
 *    May you share freely, never taking more than you give.
 *)
open Common
module Flag = Flag_semgrep
module PI = Parse_info
module S = Scope_code
module E = Error_code
module MR = Mini_rule
module R = Rule
module J = JSON
module FT = File_type
module RP = Report

(*****************************************************************************)
(* Purpose *)
(*****************************************************************************)
(* A semantic grep.
```

```
 * See https://semgrep.dev/ for more information.
 *
 * Right now there is:
 *  - good support for: Python, Java, Go, Ruby,
 *    Javascript (and JSX), Typescript (and TSX), JSON
 *  - partial support for: PHP, C, OCaml, Lua, C#, YAML
 *  - almost support for: Rust, R, Kotlin.
 *
 * opti: git grep foo | xargs semgrep -e 'foo(...)'
 *
 * related:
 *  - Structural Search and Replace (SSR) in Jetbrains IDE
 *    http://www.jetbrains.com/idea/documentation/ssr.html
 *    http://tv.jetbrains.net/videocontent/intellij-idea-static-analysis-custom-rules-with-structural-search-re
 *  - gogrep: https://github.com/mvdan/gogrep/
 *  - ruleguard: https://github.com/quasilyte/go-ruleguard
 *    (use gogrep internally)
 *  - phpgrep: https://github.com/quasilyte/phpgrep
 *    https://github.com/VKCOM/noverify/blob/master/docs/dynamic-rules.md
 *    https://speakerdeck.com/quasilyte/phpgrep-syntax-aware-code-search
 *  - rubocop pattern
 *    https://github.com/marcandre/rubocop/blob/master/manual/node_pattern.md
 *  - astpath, using XPATH on ASTs https://github.com/hchasestevens/astpath
 *  - ack http://beyondgrep.com/
 *  - cgrep http://awgn.github.io/cgrep/
 *  - hound https://codeascraft.com/2015/01/27/announcing-hound-a-lightning-fast-code-search-tool/
 *  - many grep-based linters (in Zulip, autodesk, bento, etc.)
 *
 * See also codequery for more structural queries.
 * See also old information at https://github.com/facebook/pfff/wiki/Sgrep.
 *)

(*****************************************************************************)
(* Flags *)
(*****************************************************************************)

(* ------------------------------------------------------------------------- *)
(* debugging/profiling/logging flags *)
(* ------------------------------------------------------------------------- *)

(* You can set those environment variables to enable debugging/profiling
 * instead of using -debug or -profile. This is useful when you don't call
 * directly semgrep-core but instead use the semgrep Python wrapper.
 *)
let env_debug = "SEMGREP_CORE_DEBUG"

let env_profile = "SEMGREP_CORE_PROFILE"

let env_extra = "SEMGREP_CORE_EXTRA"

let logger = Logging.get_logger [ __MODULE__ ]

let log_config_file = ref "log_config.json"

(* see also verbose/... flags in Flag_semgrep.ml *)
(* to test things *)
let test = ref false
```

⟨*constant* Main_semgrep_core.verbose 46b⟩
⟨*constant* Main_semgrep_core.debug 211f⟩

```
let profile = ref false

(* report matching times per file *)
let report_time = ref false

⟨constant Main_semgrep_core.error_recovery 224d⟩
(* related: Flag_semgrep.debug_matching *)
let fail_fast = ref false

(* used for -json -profile *)
let profile_start = ref 0.

(* there are a few other debugging flags in Flag_semgrep.ml
 * (e.g., debug_matching)
 *)
(* ------------------------------------------------------------------------ *)
(* main flags *)
(* ------------------------------------------------------------------------ *)

⟨constant Main_semgrep_core.pattern_string 42b⟩
⟨constant Main_semgrep_core.pattern_file 43h⟩
⟨constant Main_semgrep_core.rules_file 43k⟩
⟨constant Main_semgrep_core.tainting_rules_file 193a⟩

(* -config *)
let config_file = ref ""

⟨constant Main_semgrep_core.equivalences_file 102b⟩

(* todo: infer from basename argv(0) ? *)
⟨constant Main_semgrep_core.lang 41b⟩

⟨constant Main_semgrep_core.excludes 119c⟩
⟨constant Main_semgrep_core.includes 119d⟩
⟨constant Main_semgrep_core.exclude_dirs 119e⟩
⟨constant Main_semgrep_core.include_dirs 119f⟩

type output_format = Text | Json

⟨constant Main_semgrep_core.output_format_json 113d⟩

⟨constant Main_semgrep_core.match_format 111a⟩

⟨constant Main_semgrep_core.mvars 115a⟩

⟨constant Main_semgrep_core.layer_file 124b⟩

⟨constant Main_semgrep_core.keys 41d⟩
⟨constant Main_semgrep_core.supported_langs 41c⟩

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

let timeout = ref 0. (* in seconds; 0 or less means no timeout *)

let max_memory = ref 0 (* in MB *)

(* arbitrary limit *)
let max_match_per_file = ref 10_000
```

⟨*constant* Main_semgrep_core.ncores 117a⟩

```
(* ---------------------------------------------------------------------- *)
(* optional optimizations *)
(* ---------------------------------------------------------------------- *)
(* see Flag_semgrep.ml *)

(* ---------------------------------------------------------------------- *)
(* flags used by the semgrep-python wrapper *)
(* ---------------------------------------------------------------------- *)

(* path to cache (given by semgrep-python) *)
let use_parsing_cache = ref ""

(* take the list of files in a file (given by semgrep-python) *)
let target_file = ref ""
```

⟨*constant* Main_semgrep_core.action 48a⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

let version =
  spf "semgrep-core version: %s, pfff: %s" Version.version Config_pfff.version
```

⟨*function* Main_semgrep_core.set_gc 117e⟩

⟨*function* Main_semgrep_core.map 117c⟩

⟨*constant* Main_semgrep_core._matching_tokens 124c⟩

⟨*function* Main_semgrep_core.print_match 111d⟩
⟨*function* Main_semgrep_core.gen_layer 124e⟩

⟨*function* Main_semgrep_core.unsupported_language_message 41e⟩

```
let lang_of_string s =
  match Lang.lang_of_string_opt s with
  | Some x -> x
  | None -> failwith (unsupported_language_message s)

(* when called from semgrep-python, error messages in semgrep-core or
 * certain profiling statistics may refer to rule id that are generated
 * by semgrep-python, making it hard to know what the problem is.
 * At least we can save this generated rule file to help debugging.
 *)
let save_rules_file_in_tmp () =
  let tmp = Filename.temp_file "semgrep_core_rule-" ".yaml" in
  pr2 (spf "saving rules file for debugging in: %s" tmp);
  Common.write_file ~file:tmp (Common.read_file !rules_file)

(*****************************************************************************)
(* xLang *)
(*****************************************************************************)

(* coupling: Parse_mini_rule.parse_languages *)
let xlang_of_string s =
  match s with
```

```
  | "none" | "regex" -> R.LNone
  | "generic" -> R.LGeneric
  | _ ->
      let lang = lang_of_string s in
      R.L (lang, [])

let xlang_files_of_dirs_or_files xlang files_or_dirs =
  match xlang with
  | R.LNone | R.LGeneric ->
      (* TODO: assert is_file ? spacegrep filter files? *)
      files_or_dirs
  | R.L (lang, _) -> Lang.files_of_dirs_or_files lang files_or_dirs

(*****************************************************************************)
(* Caching *)
(*****************************************************************************)

let filemtime file = (Unix.stat file).Unix.st_mtime

(* The function below is mostly a copy-paste of Common.cache_computation.
 * This function is slightly more flexible because we can put the cache file
 * anywhere thanks to the argument 'cache_file_of_file'.
 * We also try to be a bit more type-safe by using the version tag above.
 * TODO: merge in pfff/commons/Common.ml at some point
 *)
let cache_computation file cache_file_of_file f =
  if !use_parsing_cache = "" then f ()
  else if not (Sys.file_exists file) then (
    pr2 ("WARNING: cache_computation: can't find file " ^ file);
    pr2 "defaulting to calling the function";
    f () )
  else
    Common.profile_code "Main.cache_computation" (fun () ->
        let file_cache = cache_file_of_file file in
        if Sys.file_exists file_cache && filemtime file_cache >= filemtime file
        then (
          logger#info "using cache: %s" file_cache;
          let version, file2, res = Common2.get_value file_cache in
          if version <> Version.version then
            failwith
              (spf "Version mismatch! Clean the cache file %s" file_cache);
          if file <> file2 then
            failwith
              (spf
                 "Not the same file! Md5sum collision! Clean the cache file %s"
                 file_cache);

          res )
        else
          let res = f () in
          Common2.write_value (Version.version, file, res) file_cache;
          res)

let cache_file_of_file filename =
  let dir = !use_parsing_cache in
  if not (Sys.file_exists dir) then Unix.mkdir dir 0o700;
  (* hopefully there will be no collision *)
  let md5 = Digest.string filename in
  Filename.concat dir (spf "%s.ast_cache" (Digest.to_hex md5))
```

```
(*****************************************************************************)
(* Timeout *)
(*****************************************************************************)

(* subtle: You have to make sure that Timeout is not intercepted, so
 * avoid exn handler such as try (...) with _ -> otherwise Timeout will
 * not bubble up enough. In such case, add a case before such as
 * with Timeout -> raise Timeout | _ -> ...
 *)
let timeout_function file f =
  let timeout = !timeout in
  if timeout <= 0. then f ()
  else
    Common.timeout_function_float ~verbose:false timeout (fun () ->
        try f ()
        with Timeout ->
          logger#info "raised Timeout in timeout_function for %s" file;
          raise Timeout)

(*
   Fail gracefully if memory becomes insufficient.

   It raises Out_of_memory if we're over the memory limit at the end of a
   major GC cycle.

   See https://discuss.ocaml.org/t/todays-trick-memory-limits-with-gc-alarms/4431
   for detailed explanations.
*)
let run_with_memory_limit limit_mb f =
  if limit_mb = 0 then f ()
  else if limit_mb < 0 then
    invalid_arg (spf "run_with_memory_limit: negative argument %i" limit_mb)
  else
    let limit = limit_mb * 1024 * 1024 in
    let limit_memory () =
      let mem = (Gc.quick_stat ()).Gc.heap_words in
      if mem > limit / (Sys.word_size / 8) then (
        logger#info "maxout allocated memory: %d" (mem * (Sys.word_size / 8));
        raise Out_of_memory )
    in
    let alarm = Gc.create_alarm limit_memory in
    Fun.protect f ~finally:(fun () ->
        Gc.delete_alarm alarm;
        Gc.compact ())

(* Certain patterns may be too general and match too many times on big files.
 * This does not cause a Timeout during parsing or matching, but returning
 * a huge number of matches can stress print_matches_and_errors_json
 * and anyway is probably a sign that the pattern should be rewritten.
 * This puts also lots of stress on the semgrep Python wrapper which has
 * to do lots of range intersections with all those matches.
 *)
let filter_files_with_too_many_matches_and_transform_as_timeout matches =
  let per_files =
    matches
    |> List.map (fun m -> (m.Pattern_match.file, m))
    |> Common.group_assoc_bykey_eff
  in
  let offending_files =
    per_files
```

```
    |> List.filter_map (fun (file, xs) ->
           if List.length xs > !max_match_per_file then Some file else None)
    |> Common.hashset_of_list
  in
  let new_matches =
    matches
    |> Common.exclude (fun m ->
           Hashtbl.mem offending_files m.Pattern_match.file)
  in
  let new_errors =
    offending_files |> Common.hashset_to_list
    |> List.map (fun file ->
           (* logging useful info for rule writers *)
           logger#info "too many matches on %s, generating exn for it" file;
           let biggest_offending_rule =
             let matches = List.assoc file per_files in
             matches
             |> List.map (fun m ->
                    let rule_id = m.Pattern_match.rule_id in
                    ( ( rule_id.Pattern_match.id,
                        rule_id.Pattern_match.pattern_string ),
                      m ))
             |> Common.group_assoc_bykey_eff
             |> List.map (fun (k, xs) -> (k, List.length xs))
             |> Common.sort_by_val_highfirst |> List.hd
             (* nosemgrep *)
           in
           let (id, pat), cnt = biggest_offending_rule in
           logger#info
             "most offending rule: id = %s, matches = %d, pattern = %s" id cnt
             pat;

           (* todo: we should maybe use a new error: TooManyMatches of int * string*)
           let loc = Parse_info.first_loc_of_file file in
           Error_code.mk_error_loc loc (Error_code.TooManyMatches pat))
  in
  (new_matches, new_errors)
  [@@profiling "Main.filter_too_many_matches"]

(*****************************************************************************)
(* Parsing *)
(*****************************************************************************)
```

⟨*function* Main_semgrep_core.parse_generic 52b⟩

⟨*function* Main_semgrep_core.parse_equivalences 102d⟩

⟨*type* Main_semgrep_core.ast 43a⟩

⟨*function* Main_semgrep_core.create_ast 52a⟩

⟨*type* Main_semgrep_core.pattern 43b⟩

⟨*function* Main_semgrep_core.parse_pattern 55a⟩

```
(*****************************************************************************)
(* Iteration helpers *)
(*****************************************************************************)
```
⟨*function* Main_semgrep_core.filter_files 119g⟩

⟨*function* Main_semgrep_core.get_final_files 119h⟩

⟨*function* Main_semgrep_core.iter_generic_ast_of_files_and_get_matches_and_exn_to_errors 44b⟩

⟨*function* Main_semgrep_core.print_matches_and_errors 113a⟩
⟨*function* Main_semgrep_core.format_output_exception 57f⟩

```
(*****************************************************************************)
(* Semgrep -rules_file *)
(*****************************************************************************)
(* This is the main function used by the semgrep python wrapper right now.
 * It takes a language, a set of mini rules (rules with a single pattern,
 * no formula) and a set of files or dirs and recursively process those
 * files or dirs.
 *)
```
⟨*function* Main_semgrep_core.semgrep_with_rules 44a⟩

```
let semgrep_with_rules_file lang rules_file files_or_dirs =
  try
    ⟨Main_semgrep_core.semgrep_with_rules() if verbose 46a⟩
    let timed_rules =
      Common.with_time (fun () -> Parse_mini_rule.parse rules_file)
    in
    semgrep_with_rules lang timed_rules files_or_dirs;
    if !profile then save_rules_file_in_tmp ()
  with exn ->
    logger#debug "exn before exit %s" (Common.exn_to_s exn);
    (* if !Flag.debug then save_rules_file_in_tmp (); *)
    let json = JSON_report.json_of_exn exn in
    let s = J.string_of_json json in
    pr s;
    exit 2

(*****************************************************************************)
(* Semgrep -config *)
(*****************************************************************************)

let semgrep_with_real_rules (rules, rule_parse_time) files_or_dirs =
  (* todo: at some point we should infer the lang from the rules and
   * apply different rules with different languages and different files
   * automatically, like the semgrep python wrapper.
   *)
  let xlang = xlang_of_string !lang in
  let files = xlang_files_of_dirs_or_files xlang files_or_dirs in
  logger#info "processing %d files" (List.length files);

  let file_results =
    files
    |> iter_files_and_get_matches_and_exn_to_errors (fun file ->
           let rules =
             rules
             |> List.filter (fun r ->
                    match (r.R.languages, xlang) with
                    | R.L (x, xs), R.L (lang, _) -> List.mem lang (x :: xs)
                    | R.LNone, R.LNone | R.LGeneric, R.LGeneric -> true
                    | _ -> false)
           in
           let hook str env matched_tokens =
             if !output_format = Text then
               let xs = Lazy.force matched_tokens in
```

```
              print_match ~str !mvars env Metavariable.ii_of_mval xs
          in
          let lazy_ast_and_errors =
            lazy
              ( match xlang with
              | R.L (lang, _) -> parse_generic lang file
              | R.LNone | R.LGeneric ->
                  failwith "requesting generic AST for LNone|LGeneric" )
          in
          let res =
            Semgrep.check hook Config_semgrep.default_config rules
              (file, xlang, lazy_ast_and_errors)
          in
          RP.add_file file res)
  in
  let res = RP.make_rule_result file_results !report_time rule_parse_time in
  logger#info "found %d matches and %d errors" (List.length res.matches)
    (List.length res.errors);
  let matches, new_errors =
    filter_files_with_too_many_matches_and_transform_as_timeout res.matches
  in
  let errors = new_errors @ res.errors in
  let res = { RP.matches; errors; rule_profiling = res.RP.rule_profiling } in
  (* note: uncomment the following and use semgrep-core -stat_matches
   * to debug too-many-matches issues.
   * Common2.write_value matches "/tmp/debug_matches";
   *)
  match !output_format with
  | Json ->
      let flds = JSON_report.json_fields_of_matches_and_errors files res in
      let flds =
        if !profile then (
          let json = JSON_report.json_of_profile_info !profile_start in
          (* so we don't get also the profile output of Common.main_boilerplate*)
          Common.profile := Common.ProfNone;
          flds @ [ ("profiling", json) ] )
        else flds
      in
      let s = J.string_of_json (J.Object flds) in
      logger#info "size of returned JSON string: %d" (String.length s);
      pr s
  | Text ->
      (* the match has already been printed above. We just print errors here *)
      (* pr (spf "number of errors: %d" (List.length errs)); *)
      errors |> List.iter (fun err -> pr (E.string_of_error err))

let semgrep_with_real_rules_file rules_file files_or_dirs =
  try
    logger#info "Parsing %s" rules_file;
    let timed_rules =
      Common.with_time (fun () -> Parse_rule.parse rules_file)
    in
    semgrep_with_real_rules timed_rules files_or_dirs
  with exn when !output_format = Json ->
    logger#debug "exn before exit %s" (Common.exn_to_s exn);
    let json = JSON_report.json_of_exn exn in
    let s = J.string_of_json json in
    pr s;
    exit 2
```

```
(*****************************************************************************)
(* Semgrep -e/-f *)
(*****************************************************************************)

let rule_of_pattern lang pattern_string pattern =
  {
    MR.id = "-e/-f";
    pattern_string;
    pattern;
    message = "";
    severity = MR.Error;
    languages = [ lang ];
  }

⟨function Main_semgrep_core.sgrep_ast 43c⟩

⟨function Main_semgrep_core.semgrep_with_one_pattern 42c⟩

(*****************************************************************************)
(* Semgrep -tainting_rules_file *)
(*****************************************************************************)

module TR = Tainting_rule

⟨function Main_semgrep_core.tainting_with_rules 193f⟩

(*****************************************************************************)
(* Checker *)
(*****************************************************************************)
⟨function Main_semgrep_core.read_all 122d⟩

⟨function Main_semgrep_core.validate_pattern 123b⟩

(* See also Check_rule.check_files *)

(*****************************************************************************)
(* Dumpers *)
(*****************************************************************************)

(* used for the Dump AST in semgrep.live *)
⟨function Main_semgrep_core.json_of_v 121b⟩

⟨function Main_semgrep_core.dump_v_to_format 51b⟩

⟨function Main_semgrep_core.dump_pattern 50b⟩

⟨function Main_semgrep_core.dump_ast 51a⟩
let dump_v1_json file =
  match Lang.langs_of_filename file with
  | lang :: _ ->
      E.try_with_print_exn_and_reraise file (fun () ->
          let { Parse_target.ast; errors; _ } =
            Parse_target.parse_and_resolve_name_use_pfff_or_treesitter lang file
          in
          let v1 = AST_generic_to_v1.program ast in
          let s = AST_generic_v1_j.string_of_program v1 in
          pr s;
          if errors <> [] then pr2 (spf "WARNING: fail to fully parse %s" file))
  | [] -> failwith (spf "unsupported language for %s" file)
```

⟨*function* Main_semgrep_core.dump_ext_of_lang 49b⟩

⟨*function* Main_semgrep_core.dump_equivalences 105b⟩

⟨*function* Main_semgrep_core.dump_tainting_rules 198c⟩

```
let dump_rule file =
  let rules = Parse_rule.parse file in
  rules |> List.iter (fun r -> pr (Rule.show r))

(*****************************************************************************)
(* Experiments *)
(*****************************************************************************)
(* See Experiments.ml now *)

(*****************************************************************************)
(* The options *)
(*****************************************************************************)
```

⟨*function* Main_semgrep_core.all_actions 48d⟩

⟨*function* Main_semgrep_core.options 46d⟩

```
(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)
```

⟨*function* Main_semgrep_core.main 40c⟩

```
(*****************************************************************************)
```
⟨*toplevel* Main_semgrep_core._1 41a⟩

F.13 semgrep/core/

semgrep/core/Rule.ml

⟨semgrep/core/Rule.ml 379⟩≡
```
  ⟨pad/r2c copyright 11⟩
  module MV = Metavariable

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Data structure representing a semgrep rule.
   *
   * See also Mini_rule.ml where formula and many other features disappears.
   *
   * TODO:
   *  - parse equivalences
   *)

  (*****************************************************************************)
  (* Extended languages and patterns *)
  (*****************************************************************************)

  (* less: merge with xpattern_kind? *)
  type xlang =
    (* for "real" semgrep (the first language is used to parse the pattern) *)
```

```
  | L of Lang.t * Lang.t list
  (* for pattern-regex (less: rename LRegex? *)
  | LNone
  (* for spacegrep *)
  | LGeneric
[@@deriving show]

type regexp = Regexp_engine.Pcre_engine.t [@@deriving show, eq]

type xpattern = {
  pat : xpattern_kind;
  (* two patterns may have different indentation, we don't care. We can
   * rely on the equality on pat, which will do the right thing (e.g., abstract
   * away line position).
   * TODO: right now we have some false positives because
   * for example in Python assert(...) and assert ... are considered equal
   * AST-wise, but it might be a bug!.
   *)
  pstr : string; [@equal fun _ _ -> true]
  (* unique id, incremented via a gensym()-like function in mk_pat() *)
  pid : pattern_id; [@equal fun _ _ -> true]
}

and xpattern_kind =
  | Sem of Pattern.t * Lang.t (* language used for parsing the pattern *)
  | Spacegrep of Spacegrep.Pattern_AST.t
  | Regexp of regexp

(* used in the engine for rule->mini_rule and match_result gymnastic *)
and pattern_id = int [@@deriving show, eq]

let count = ref 0

let mk_xpat pat pstr =
  incr count;
  { pat; pstr; pid = !count }

(*****************************************************************************)
(* Formula (patterns boolean composition) *)
(*****************************************************************************)

(* Classic boolean-logic/set operators with text range set semantic.
 * The main complication is the handling of metavariables and especially
 * negation in the presence of metavariables.
 * TODO: add tok (Parse_info.t) for good metachecking error locations.
 *)
type formula =
  | Leaf of leaf
  | And of formula list
  | Or of formula list
  (* There are restrictions on where a Not can appear in a formula. It
   * should always be inside an And to be intersected with "positive" formula.
   *)
  | Not of formula

(* todo: try to remove this at some point, but difficult. See
 * https://github.com/returntocorp/semgrep/issues/1218
 *)
and inside = Inside
```

```
and leaf =
  (* Turn out, pattern: and pattern-inside: are slightly different so
   * we need to keep the information around.
   * (see tests/OTHER/rules/inside.yaml)
   * The same is true for pattern-not and pattern-not-inside
   * (see tests/OTHER/rules/negation_exact.yaml)
   *)
  | P of xpattern (* a leaf pattern *) * inside option
  | MetavarCond of metavar_cond

and metavar_cond =
  | CondGeneric of AST_generic.expr (* see Eval_generic.ml *)
  (* todo: at some point we should remove CondRegexp and have just
   * CondGeneric, but for now there are some
   * differences between using the matched text region of a metavariable
   * (which we use for MetavarRegexp) and using its actual value
   * (which we use for MetavarComparison), which translate to different
   * calls in Eval_generic.ml
   * update: this is also useful to keep separate from CondGeneric for
   * the "regexpizer" optimizer (see Analyze_rule.ml).
   *)
  | CondRegexp of MV.mvar * regexp
[@@deriving show, eq]

(*****************************************************************************)
(* Old Formula style *)
(*****************************************************************************)

(* Unorthodox original pattern compositions.
 * See also the JSON schema in rule_schema.yaml
 *)
type formula_old =
  (* pattern: *)
  | Pat of xpattern
  (* pattern-not: *)
  | PatNot of xpattern
  | PatExtra of extra
  (* pattern-inside: *)
  | PatInside of xpattern
  (* pattern-not-inside: *)
  | PatNotInside of xpattern
  (* pattern-either: Or *)
  | PatEither of formula_old list
  (* patterns: And *)
  | Patterns of formula_old list

(* extra conditions, usually on metavariable content *)
and extra =
  | MetavarRegexp of MV.mvar * regexp
  | MetavarComparison of metavariable_comparison
  | PatWherePython of string

(* arbitrary code, dangerous! *)

(* See also matching/eval_generic.ml *)
and metavariable_comparison = {
  metavariable : MV.mvar;
  comparison : AST_generic.expr;
  (* see Eval_generic.ml *)
  strip : bool option;
```

```
  base : int option;
}
[@@deriving show, eq]

(* pattern formula *)
type pformula = New of formula | Old of formula_old [@@deriving show, eq]

(*****************************************************************************)
(* The rule *)
(*****************************************************************************)

type rule = {
  (* mandatory fields *)
  id : string;
  formula : pformula;
  message : string;
  severity : Mini_rule.severity;
  languages : xlang;
  file : string;
  (* for metachecking error location *)

  (* optional fields *)
  equivalences : string list option;
  (* TODO: parse them *)
  fix : string option;
  fix_regexp : (regexp * int option * string) option;
  paths : paths option;
  (* ex: [("owasp", "A1: Injection")] but can be anything *)
  metadata : JSON.t option;
}

and paths = {
  (* not regexp but globs *)
  include_ : string list;
  exclude : string list;
}
[@@deriving show]

(* alias *)
type t = rule [@@deriving show]

type rules = rule list [@@deriving show]

(*****************************************************************************)
(* Visitor *)
(*****************************************************************************)
(* currently used in Check_rule.ml metachecker *)
let rec visit_new_formula f formula =
  match formula with
  | Leaf (P (p, _)) -> f p
  | Leaf (MetavarCond _) -> ()
  | Not x -> visit_new_formula f x
  | Or xs | And xs -> xs |> List.iter (visit_new_formula f)

(*****************************************************************************)
(* Converter *)
(*****************************************************************************)

let convert_extra x =
  match x with
```

```
   | MetavarRegexp (mvar, re) -> CondRegexp (mvar, re)
   | MetavarComparison comp -> (
       match comp with
       (* do we care about strip and base? should not Eval_generic handle it?
        * base I think can be handled automatically, and for strip the user
        * should instead use a more complex condition that converts
        * the string into a number (e.g., "1234" in 1234).
        *)
       | { metavariable = _; comparison = x; strip = _TODO1; base = _TODO2 } ->
           CondGeneric x )
   | _ ->
       (*
   logger#debug "convert_extra: %s" s;
   Parse_rule.parse_metavar_cond s
 *)
       failwith (Common.spf "convert_extra: TODO: %s" (show_extra x))

 let (convert_formula_old : formula_old -> formula) =
  fun e ->
   let rec aux e =
     match e with
     | Pat x -> Leaf (P (x, None))
     | PatInside x -> Leaf (P (x, Some Inside))
     | PatNot x -> Not (Leaf (P (x, None)))
     | PatNotInside x -> Not (Leaf (P (x, Some Inside)))
     | PatEither xs ->
         let xs = List.map aux xs in
         Or xs
     | Patterns xs ->
         let xs = List.map aux xs in
         And xs
     | PatExtra x ->
         let e = convert_extra x in
         Leaf (MetavarCond e)
   in
   aux e

 let formula_of_rule r =
   match r.formula with New f -> f | Old oldf -> convert_formula_old oldf
```

semgrep/core/Config_semgrep.ml

⟨semgrep/core/Config_semgrep.ml 383⟩≡
  ⟨*pad/r2c copyright* 11⟩

```
 (*****************************************************************************)
 (* Prelude *)
 (*****************************************************************************)
 (* Semgrep engine configuration.
  *
  * The goal of this module is to gather in one place all the possible
  * ways to configure the semgrep matching engine. At some point, we
  * may let the user enable/disable certain features on a per-rule or even
  * per-pattern basis. For example, constant propagation may be too powerful
  * sometimes and prevent people to find certain code.
  *
  * Note that each feature in this file will change the matching results;
  * for non-functinal settings such as optimizations (e.g., using a
```

```
 * cache) use instead Flag_semgrep.ml
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
type t = {
  constant_propagation : bool;
  (* !experimental: a bit hacky, and may introduce big perf regressions! *)
  (* should be used with DeepEllipsis; do it implicitely has issues *)
  go_deeper_expr : bool;
  (* this ultimately should go away once '...' works on the CFG *)
  go_deeper_stmt : bool;
      (* TODO: equivalences:
       *   - const_let_to_var
       *   - require_to_import (but need pass config to Js_to_generic)
       *)
}

(*****************************************************************************)
(* Default config *)
(*****************************************************************************)

let default_config =
  { constant_propagation = true; go_deeper_expr = true; go_deeper_stmt = true }
```

semgrep/core/Flag_semgrep.ml

⟨semgrep/core/Flag_semgrep.ml 384⟩≡

```
  ⟨constant Flag_semgrep.verbose 46c⟩

  (* debugging flags *)

  ⟨constant Flag_semgrep.debug 212a⟩
  ⟨constant Flag_semgrep.debug_with_full_position 212b⟩

  (* we usually try first with the pfff parser and then with the tree-sitter
   * parser if pfff fails. Here you can force to only use tree-sitter.
   *)
  let tree_sitter_only = ref false

  let pfff_only = ref false

  (* optimization flags *)

  ⟨constant Flag_semgrep.go_deeper_expr 89c⟩
  ⟨constant Flag_semgrep.go_deeper_stmt 92d⟩
  ⟨constant Flag_semgrep.go_really_deeper_stmt 92e⟩

  (* look if identifiers in pattern intersect with file using simple regexps *)
  let filter_irrelevant_patterns = ref false

  (* similar to filter_irrelevant_patterns, but use the whole rule to extract
   * the regexp *)
  let filter_irrelevant_rules = ref false

  (* check for identifiers before attempting to match a stmt or stmt list *)
  let use_bloom_filter = ref true
```

```
  let set_instead_of_bloom_filter = ref false

  (* opt = optimization *)
  let with_opt_cache = ref true

  (* Improves performance on some patterns, degrades performance on others. *)
  let max_cache = ref false

  (* Disabling this lets us measure the effectiveness of our GC tuning. *)
  let gc_tuning = ref true

  ⟨constant Flag_semgrep.equivalence_mode 104a⟩
```

semgrep/core/Metavariable.ml

```
⟨semgrep/core/Metavariable.ml 385⟩≡
  ⟨pad/r2c copyright 11⟩
  open Common
  module G = AST_generic
  module H = AST_generic_helpers

  (* Provide hash_* and hash_fold_* for the core ocaml types *)
  open Ppx_hash_lib.Std.Hash.Builtin

  let debug = false

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

  (*****************************************************************************)
  (* Types *)
  (*****************************************************************************)

  (* less: could want to remember the position in the pattern of the metavar
   * for error reporting on pattern itself? so use a 'string AST_generic.wrap'?
   *)
  ⟨type Metavars_generic.mvar 33d⟩

  (* 'mvalue' below used to be just an alias to AST_generic.any, but it is more
   * precise to have a type just for the metavariable values; we do not
   * need all the AST_generic.any cases (however this forces us to
   * define a few boilerplate functions like mvalue_to_any below).
   *
   * AST_generic.any is already (ab)used for many things: for representing
   * a semgrep pattern, for being able to dump any AST constructs,
   * for poor's man overloading for visiting, mapping, so there's no
   * need to add an extra thing. It would probably be better to also
   * define our own Pattern.t with just the valid cases, but we don't
   * want code in pfff to depend on semgrep/core/Pattern.ml, hence the
   * use of AST_generic.any for patterns.
   *)
  type mvalue =
    (* TODO: get rid of Id, N generalize it *)
    | Id of AST_generic.ident * AST_generic.id_info option
    | N of AST_generic.name
    | E of AST_generic.expr
```

```
  | S of AST_generic.stmt
  | Ss of AST_generic.stmt list
  | T of AST_generic.type_
  | P of AST_generic.pattern
  | Args of AST_generic.argument list
[@@deriving show, eq, hash]

(* we sometimes need to convert to an any to be able to use
 * Lib_AST.ii_of_any, or Lib_AST.abstract_position_info_any
 *)
let mvalue_to_any = function
  | E e -> G.E e
  | S s -> G.S s
  (* bugfix: do not return G.I id. We need the id_info because
   * it can be used to check if two metavars are equal and have the same
   * sid (single unique id).
   *)
  | Id (id, Some idinfo) -> G.E (G.N (G.Id (id, idinfo)))
  | Id (id, None) -> G.E (G.N (G.Id (id, G.empty_id_info ())))
  | N x -> G.E (G.N x)
  | Ss x -> G.Ss x
  | Args x -> G.Args x
  | T x -> G.T x
  | P x -> G.P x

let str_of_any any =
  if !Flag_semgrep.debug_with_full_position then
    Meta_parse_info._current_precision :=
      {
        Meta_parse_info.default_dumper_precision with
        Meta_parse_info.full_info = true;
      };
  let s = AST_generic.show_any any in
  s

let ii_of_mval x = x |> mvalue_to_any |> Visitor_AST.ii_of_any

let str_of_mval x = x |> mvalue_to_any |> str_of_any
```

⟨*type* Metavars_generic.metavars_binding 36⟩

⟨*constant* Metavars_generic.metavar_regexp_string 33e⟩

⟨*function* Metavars_generic.is_metavar_name 33f⟩

```
(* $...XXX multivariadic metavariables. Note that I initially chose
 * $X... but this leads to parsing conflicts in Javascript.
 *)
let metavar_ellipsis_regexp_string = "^\\(\\$\\.\\.\\.[A-Z_][A-Z_0-9]*\\)$"

let is_metavar_ellipsis s = s =~ metavar_ellipsis_regexp_string

module Structural = struct
  let equal_mvalue = AST_utils.with_structural_equal equal_mvalue

  let equal_bindings = AST_utils.with_structural_equal equal_bindings
end

module Referential = struct
  let equal_mvalue = AST_utils.with_referential_equal equal_mvalue
```

```
    let equal_bindings = AST_utils.with_referential_equal equal_bindings

    let hash_bindings = hash_bindings
  end
```

semgrep/core/Target.ml

⟨*type* `Target.t` 387a⟩≡ (387b)
```
  type t = AST_generic.program
```

⟨`semgrep/core/Target.ml` 387b⟩≡
```
  ⟨pad/r2c copyright 11⟩
  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* A target file.
   *
   * See AST_generic.program for more information. This module just
   * defines an alias to have better naming convetion to differentiate
   * a target program from a pattern program.
   *)

  (*****************************************************************************)
  (* Types *)
  (*****************************************************************************)

  ⟨type Target.t 387a⟩
```

semgrep/core/Pattern.ml

⟨`semgrep/core/Pattern.ml` 387c⟩≡
```
  ⟨pad/r2c copyright 11⟩
  open Common

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* A semgrep pattern.
   *
   * A pattern is represented essentially as an AST_generic.any
   * where special constructs are now allowed (e.g., Ellipsis),
   * where certain identifiers are metavariables (e.g., $X), or
   * where certain strings are ellipsis or regular expressions
   * (e.g., "=~/foo/").
   *
   * See also Metavariable.ml.
   *)

  (*****************************************************************************)
  (* Types *)
  (*****************************************************************************)

  ⟨type Pattern.t 33a⟩
```

```
  let regexp_regexp_string = "^=~/\\(.*\\)/\\([mi]?\\)$"

  let is_regexp_string s = s =~ regexp_regexp_string

  let is_special_string_literal str = str = "..." || is_regexp_string str

  let is_js lang = match lang with Some x -> Lang.is_js x | None -> true

  let is_special_identifier ?lang str =
    Metavariable.is_metavar_name str
    (* emma: a hack because my regexp skills are not great *)
    || (String.length str > 4 && Str.first_chars str 4 = "$...")
    (* in JS field names can be regexps *)
    || (is_js lang && is_regexp_string str)
    (* ugly hack that we then need to handle also here *)
    || str = AST_generic.special_multivardef_pattern
    (* ugly: because ast_js_build introduce some extra "!default" ids *)
    || (is_js lang && str = Ast_js.default_entity)
    (* parser_java.mly inserts some implicit this *)
    || (lang = Some Lang.Java && str = "this")
    || (* TODO: PHP converts some Eval in __builtin *)
    (lang = Some Lang.PHP && str =~ "__builtin__*")
```

semgrep/core/Mini_rule.ml

⟨semgrep/core/Mini_rule.ml 388⟩≡

```
  (* Yoann Padioleau
   *
   * Copyright (C) 2011 Facebook
   * Copyright (C) 2019 r2c
   *
   * 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.
   *
   * 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.
   *)

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* The goal of this module is to make it easy to add lint rules by using
   * semgrep patterns. You just have to store the patterns in a special
   * YAML file and add the corresponding warnings you want the linter to raise.
   *
   * update: if you need advanced patterns with boolean logic (which used
   * to be partially provided by the hacky OK error keyword), use
   * instead Rule.ml (or the semgrep python wrapper). Rule.ml also uses a YAML
   * file but it has more features, e.g. some pattern-either fields,
   * pattern-inside, metavariable-comparison, etc.
   *)

  (*****************************************************************************)
  (* Types *)
  (*****************************************************************************)
```

⟨*type* Rule.pattern 34c⟩

⟨*type* Rule.rule 34b⟩

⟨*type* Rule.rules 34d⟩

```
(* TODO? just reuse Error_code.severity *)
```
⟨*type* Rule.severity 34e⟩
```
[@@deriving eq, show]
```

⟨*type* Rule.t 34f⟩

semgrep/core/Tainting_rule.ml

⟨semgrep/core/Tainting_rule.ml 389a⟩≡
⟨*pad/r2c copyright* 11⟩
```
module R = Mini_rule

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* This is a spin-off of Rule.ml but specialized for tainting analysis.
 *
 * At some point we may want tainting to be integrated and queryable
 * directly from regular semgrep rules, but for now it's simpler
 * to have a specialized type and format.
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* Tainting_rule.pattern 196c⟩

```
(* less: could extend Rule.t *)
```
⟨*type* Tainting_rule.rule 196d⟩

⟨*type* Tainting_rule.rules 196e⟩

⟨*type* Tainting_rule.t 197a⟩

⟨*function* Tainting_rule.rule_of_tainting_rule 197b⟩

semgrep/core/Equivalence.ml

⟨semgrep/core/Equivalence.ml 389b⟩≡
⟨*pad/r2c copyright* 11⟩

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* The goal of this module is to let the user defines "code equivalences"
 * (a.k.a. isomorphisms).
 *
 * There are lots of equivalences between code and more than one way
 * to perform an operation (TMTOWTDI). For example, when looking for
 * $X == $X, we may want this pattern to also match code like a != a,
```

```
 * because this is equivalent to !(a == a).
 *
 * One of the great idea in Coccinelle was to simply reuse the same
 * ideas and machinery to match code to also apply code equivalences!
 * One can write simply $X != $Y <==> !($X == $Y) in a config file
 * and have the engine handles this Equivalence.
 *
 * alternatives:
 *  - macros/templates over the yaml rule file to generate some pattern-either,
 *    but this may be a bit hacky and add yet another layer
 *    (sgrep-core -> sgrep-python -> sgrep-yaml-generator)
 *    As Matt said, adding a templating language, on top of a markup language
 *    on top of a domain specific language, on top of a programming language
 *    is hard to grasp.
 *    update: maybe with jsonnet it's not too bad
 *  - pfff/lang_GENERIC/analize/normalize_ast.ml but this requires
 *    to know OCaml (we go back to the argument of sgrep vs AST visitors)
 *
 * Note that some code equivalences are handled directly in the engine
 * in Generic_vs_generic.ml or normalize_ast.ml because they are too
 * difficult to encode otherwise (e.g., the less-is-ok).
 * todo: we should give them name too, so they can be disabled too.
 *
 * related work:
 *  - standard.iso in coccinelle
 *    https://github.com/coccinelle/coccinelle/blob/master/standard.iso
 *  - paper on semantic equivalences search recently at PLDI
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* Equivalence.pattern 35b⟩

⟨*type* Equivalence.equivalence_kind 35a⟩

⟨*type* Equivalence.equivalence 34g⟩

⟨*type* Equivalence.equivalences 35c⟩

⟨*type* Equivalence.t 35d⟩

semgrep/core/Pattern_match.ml

⟨semgrep/core/Pattern_match.ml 390⟩≡
⟨*pad/r2c copyright* 11⟩

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Type to represent a pattern match.
 *
 * old: used to be called Match_result.t
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

```
  (* We use 'eq' below to possibly remove redundant equivalent matches. Indeed,
   * Generic_vs_generic sometimes return multiple times the same match,
   * sometimes because of some bugs we didn't fix, sometimes it's normal
   * because of the way '...' operate. TODO: add an example of such situation.
   *
   * Note that you should not ignore the rule id when comparing 2 matches!
   * One match can come from a pattern-not: in which case
   * even if it returns the same match than a similar match coming
   * from a pattern:, we should not merge them!
   *)

  ⟨type Match_result.t 35e⟩
```

semgrep/core/Rule_match.ml

```
⟨type Rule_match.t 391a⟩≡                                                (391b)
  type t = Pattern_match.t
```

```
⟨semgrep/core/Rule_match.ml 391b⟩≡
  ⟨pad/r2c copyright 11⟩

  ⟨type Rule_match.t 391a⟩
```

F.14 semgrep/engine/

semgrep/engine/Convert_rule.mli

```
⟨semgrep/engine/Convert_rule.mli 391c⟩≡

  val convert_formula_old: Rule.formula_old -> Rule.formula
```

semgrep/engine/Convert_rule.ml

```
⟨semgrep/engine/Convert_rule.ml 391d⟩≡
  ⟨pad/r2c copyright 11⟩
  open Common

  open Rule

  let _logger = Logging.get_logger [__MODULE__]

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* The goal of this module is to convert rules in whatever format to
   * the latest format. Its main goal is to maintain backward compatibility.
   * For actual optimizations or complex transformations on rules, see
   * Transform_rule.ml.
   *
  *)

  let convert_extra x =
```

```
    match x with
    | MetavarRegexp (mvar, re) ->
        CondRegexp (mvar, re)
    | MetavarComparison comp ->
        (match comp with
         (* do we care about strip and base? should not Eval_generic handle it?
          * base I think can be handled automatically, and for strip the user
          * should instead use a more complex condition that converts
          * the string into a number (e.g., "1234" in 1234).
         *)
         | { metavariable = _; comparison = s; strip = _TODO1; base = _TODO2 } ->
             CondGeneric (Parse_rule.parse_metavar_cond s)
        )
    | _ ->
  (*
    logger#debug "convert_extra: %s" s;
    Parse_rule.parse_metavar_cond s
  *)
        failwith (spf "convert_extra: TODO: %s" (Rule.show_extra x))

  (*****************************************************************************)
  (* Entry points *)
  (*****************************************************************************)

  let (convert_formula_old: formula_old -> formula) = fun e ->
    let rec aux e =
      match e with
      | Pat x | PatInside x -> P x
      | PatNot x | PatNotInside x -> Not (P x)
      | PatEither xs ->
          let xs = List.map aux xs in
          Or xs
      | Patterns xs ->
          let xs = List.map aux xs in
          And xs
      | PatExtra x ->
          let e = convert_extra x in
          MetavarCond e
    in
    aux e
```

semgrep/engine/Transform_rule.mli

⟨semgrep/engine/Transform_rule.mli 392a⟩≡

semgrep/engine/Transform_rule.ml

⟨semgrep/engine/Transform_rule.ml 392b⟩≡
  ⟨*pad/r2c copyright* 11⟩

```
  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* The goal of this module is to transform rules mainly for optimization
   * purpose, a bit like a database engine can "compile" SQL queries
   * in the best execution plan.
```

```
 *
 * See also Optimizing/Analyze_rule.ml
 *
 *)

(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)
```

semgrep/engine/Semgrep.mli

⟨semgrep/engine/Semgrep.mli 393a⟩≡

```
(*
   Return matches, errors, match time.
*)
val check :
  (string -> Metavariable.bindings -> Parse_info.t list Lazy.t -> unit) ->
  Config_semgrep.t ->
  Rule.rules ->
  Common.filename * Rule.xlang * (Target.t * Error_code.error list) Lazy.t ->
  Report.times Report.match_result
```

semgrep/engine/Semgrep.ml

⟨semgrep/engine/Semgrep.ml 393b⟩≡

```
⟨pad/r2c copyright 11⟩
open Common
module R = Rule
module MR = Mini_rule
module PM = Pattern_match
module G = AST_generic
module PI = Parse_info
module MV = Metavariable
module RP = Report

let logger = Logging.get_logger [ __MODULE__ ]

let debug_timeout = ref false

let debug_matches = ref false

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* The core engine of Semgrep.
 *
 * This module implements the boolean composition of patterns.
 * See Semgrep_generic.ml for the code to handle a single pattern and
 * the visitor/matching engine.
 *
 * Thus, we can decompose the engine in 3 main components:
 *  - composing matching results using boolean/set logic (this file)
 *  - visiting code (=~ Semgrep_generic.ml)
 *  - matching code (=~ Generic_vs_generic.ml)
 *
```

```
 * There are also "preprocessing" work before that:
 *  - parsing (lexing, parsing) rules, code, patterns
 *  - normalizing (convert to a generic AST)
 *  - naming (but bugs probably)
 *  - SEMI typing (propagating type decls at least and small inference)
 *  - SEMI analyzing (dataflow constant propagation)
 *    but could do much more: deep static analysis using Datalog?
 *
 * TODO
 *  - associate the metavariable-regexp to the appropriate pattern
 *  - pattern-where-python? use pycaml? works for dlint rule?
 *    right now only 4 rules are using pattern-where-python
 *
 * LATER (if really decide to rewrite the python wrapper in OCaml):
 *  - paths
 *  - autofix
 *  - ...
 *
 * FUTURE WORK:
 * Right now we just analyze one file at a time. Later we could
 * maybe take a list of files and do some global analysis for:
 *     * caller/callee in different files
 *       which can be useful to understand keyword arguments
 *     * inheritance awareness, because right now we can't match
 *       code that inherits indirectly form a class mentioned in a pattern
 * There are different options for such global analysis:
 *  - generate a giant file a la CIL, but scale?
 *    (there is a recent LLVM project that does the same)
 *  - do it via a 2 passes process. 1st pass iterates over all files, report
 *    already matches, record semantic information (e.g., inheritance tree,
 *    call graph, etc.) as it goes, and let the matching engine report
 *    todo_second_pass if for example is_children returned a Maybe.
 *    Then in 2nd pass just process the files that were marked as todo.
 *  - use LSP, so don't even need 2 pass and can even work when passing
 *    a single file or subdir to semgrep
 *
 * Note that we opted here for simple patterns with simple extensions
 * to the grammar (metavar, ellipsis) with simple (but powerful) logic
 * compositions of patterns.
 * Coccinelle instead opted for very complex patterns and using CTL to
 * hold of that together.
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
(* Id of a single pattern in a formula. This will be used to generate
 * mini rules with this id, and later when we evaluate the formula, find
 * the matching results corresponding to this id.
 *)
type pattern_id = R.pattern_id

(* range with metavars *)
type range_with_mvars = {
  r : Range.t;
  mvars : Metavariable.bindings;
  (* subtle but the pattern:/pattern-inside: and
   * pattern-not:/pattern-not-inside: are actually different, so we need
   * to keep the information around during the evaluation.
   * Note that this useful only for few tests in semgrep-rules/ so we
```

```
   * probably want to simplify things later and remove the difference between
   * xxx and xxx-inside.
   * TODO: in fact, if we do a proper intersection of ranges, where we
   * propery intersect (not just filter one or the other), and also merge
   * metavariables, this will clean lots of things, and remove the need
   * to keep around the Inside. AND will be commutative again!
   *)
  inside : R.inside option;
  origin : Pattern_match.t;
}
[@@deriving show]

type ranges = range_with_mvars list [@@deriving show]

(* !This hash table uses the Hashtbl.find_all property! *)
type id_to_match_results = (pattern_id, Pattern_match.t) Hashtbl.t

type env = {
  config : Config_semgrep.t;
  pattern_matches : id_to_match_results;
  (* unused for now, but could be passed down for Range.content_at_range *)
  file : Common.filename;
}

(*****************************************************************************)
(* Range_with_mvars *)
(*****************************************************************************)

let included_in config rv1 rv2 =
  Range.( $<=$ ) rv1.r rv2.r
  && rv1.mvars
     |> List.for_all (fun (mvar, mval1) ->
            match List.assoc_opt mvar rv2.mvars with
            | None -> true
            | Some mval2 ->
                Matching_generic.equal_ast_binded_code config mval1 mval2)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

let (xpatterns_in_formula : R.formula -> R.xpattern list) =
 fun e ->
  let res = ref [] in
  e |> R.visit_new_formula (fun xpat -> Common.push xpat res);
  !res

let partition_xpatterns xs =
  xs
  |> Common.partition_either3 (fun xpat ->
         let id = xpat.R.pid in
         let str = xpat.R.pstr in
         match xpat.R.pat with
         | R.Sem (x, _lang) -> Left3 (x, id, str)
         | R.Spacegrep x -> Middle3 (x, id, str)
         | R.Regexp x -> Right3 (x, id, str))

let (group_matches_per_pattern_id : Pattern_match.t list -> id_to_match_results)
    =
 fun xs ->
```

```
  let h = Hashtbl.create 101 in
  xs
  |> List.iter (fun m ->
         let id = int_of_string m.PM.rule_id.id in
         Hashtbl.add h id m);
  h

let (range_to_pattern_match_adjusted :
      Rule.t -> range_with_mvars -> Pattern_match.t) =
 fun r range ->
  let m = range.origin in
  let rule_id = m.rule_id in
  (* adjust the rule id *)
  let rule_id =
    {
      rule_id with
      Pattern_match.id = r.R.id;
      message = r.R.message (* keep pattern_str which can be useful to debug *);
    }
  in
  { m with rule_id }

let (match_result_to_range : Pattern_match.t -> range_with_mvars) =
 fun m ->
  let { Pattern_match.range_loc = start_loc, end_loc; env = mvars; _ } = m in
  let r = Range.range_of_token_locations start_loc end_loc in
  { r; mvars; origin = m; inside = None }

(* return list of "positive" x list of Not x list of Conds *)
let (split_and :
      R.formula list -> R.formula list * R.formula list * R.metavar_cond list) =
 fun xs ->
  xs
  |> Common.partition_either3 (fun e ->
         match e with
         | R.Not f -> Middle3 f
         | R.Leaf (R.MetavarCond c) -> Right3 c
         | _ -> Left3 e)

let lazy_force x = Lazy.force x [@@profiling]

(*****************************************************************************)
(* Adapters *)
(*****************************************************************************)

let (mini_rule_of_pattern : R.t -> Pattern.t * Rule.pattern_id * string -> MR.t)
    =
 fun r (pattern, id, pstr) ->
  {
    MR.id = string_of_int id;
    pattern;
    (* parts that are not really needed I think in this context, since
     * we just care about the matching result.
     *)
    message = "";
    severity = MR.Error;
    languages =
      ( match r.R.languages with
      | R.L (x, xs) -> x :: xs
      | R.LNone | R.LGeneric -> raise Impossible );
```

```
    (* useful for debugging timeout *)
    pattern_string = pstr;
  }

(* this will be adjusted later in range_to_pattern_match_adjusted *)
let fake_rule_id (id, str) =
  { PM.id = string_of_int id; pattern_string = str; message = "" }

(* todo: same, we should not need that *)
let hmemo = Hashtbl.create 101

let line_col_of_charpos file charpos =
  let conv =
    Common.memoized hmemo file (fun () -> PI.full_charpos_to_pos_large file)
  in
  conv charpos

(* todo: same, we should not need that *)
let info_of_token_location loc =
  { PI.token = PI.OriginTok loc; transfo = PI.NoTransfo }

(* for spacegrep *)
let lexing_pos_to_loc file x str =
  (* almost like Spacegrep.Semgrep.semgrep_pos() *)
  let line = x.Lexing.pos_lnum in
  let charpos = x.Lexing.pos_cnum in
  (* bugfix: not +1 here, Parse_info.column is 0-based.
   * JSON_report.json_range does the adjust_column + 1.
   *)
  let column = x.Lexing.pos_cnum - x.Lexing.pos_bol in
  { PI.str; charpos; file; line; column }

let mval_of_spacegrep_string str t =
  let literal =
    match int_of_string_opt str with
    | Some i -> G.Int (Some i, t)
    (* TODO? could try float_of_string_opt? *)
    | None -> G.String (str, t)
  in
  MV.E (G.L literal)

(*****************************************************************************)
(* Logic on ranges *)
(*****************************************************************************)

(* when we know x <= y, are the ranges also in the good Inside direction *)
let inside_compatible x y =
  match (x.inside, y.inside) with
  | Some R.Inside, Some R.Inside -> true
  | None, None -> true
  | None, Some R.Inside -> true
  (* if we do x=pattern-inside:[1-2] /\ y=pattern:[1-3]
   * we don't want this x to survive.
   * See tests/OTHER/rules/and_inside.yaml
   *)
  | Some R.Inside, None -> false

let intersect_ranges config xs ys =
  if !debug_matches then
    logger#info "intersect_range:\n\t%s\nvs\n\t%s" (show_ranges xs)
```

```
      (show_ranges ys);
  let surviving_xs =
    xs
    |> List.filter (fun x ->
           ys
           |> List.exists (fun y ->
                  included_in config x y && inside_compatible x y))
  in
  let surviving_ys =
    ys
    |> List.filter (fun y ->
           xs
           |> List.exists (fun x ->
                  included_in config y x && inside_compatible y x))
  in
  surviving_xs @ surviving_ys
  [@@profiling]

let difference_ranges config pos neg =
  let surviving_pos =
    pos
    |> List.filter (fun x ->
           not
             ( neg
             |> List.exists (fun y ->
                    (* pattern-not vs pattern-not-inside, the difference matters.
                     * This fixed 10 mismatches in semgrep-rules.
                     *)
                    match y.inside with
                    (* pattern-not-inside: *)
                    | Some R.Inside -> included_in config x y
                    (* pattern-not: we require the ranges to be equal *)
                    | None -> included_in config x y && included_in config y x)
             ))
  in
  surviving_pos
  [@@profiling]

let filter_ranges xs cond =
  xs
  |> List.filter (fun r ->
         let bindings = r.mvars in
         match cond with
         | R.CondGeneric e ->
             let env = Eval_generic.bindings_to_env bindings in
             Eval_generic.eval_bool env e
         (* todo: would be nice to have CondRegexp also work on
          * eval'ed bindings.
          * We could also use re.match(), to be close to python, but really
          * Eval_generic must do something special here with the metavariable
          * which may not always be a string. The regexp is really done on
          * the text representation of the metavar content.
          *)
         | R.CondRegexp (mvar, (re_str, _re)) ->
             let fk = Parse_info.fake_info "" in
             let fki = AST_generic.empty_id_info () in
             let e =
               (* old: spf "semgrep_re_match(%s, \"%s\")" mvar re_str
                * but too many possible escaping problems, so easier to build
                * an expression manually.
```

```
               *)
              G.Call
                ( G.DotAccess
                    ( G.N (G.Id (("re", fk), fki)),
                      fk,
                      EN (Id (("match", fk), fki)) ),
                  ( fk,
                    [
                      G.Arg (G.N (G.Id ((mvar, fk), fki)));
                      G.Arg (G.L (G.String (re_str, fk)));
                    ],
                    fk ) )
            in

            let env =
              Eval_generic.bindings_to_env_with_just_strings bindings
            in
            Eval_generic.eval_bool env e)
  [@@profiling]

(*****************************************************************************)
(* Debugging semgrep *)
(*****************************************************************************)

let debug_semgrep config mini_rules equivalences file lang ast =
  (* process one mini rule at a time *)
  logger#info "DEBUG SEMGREP MODE!";
  mini_rules
  |> List.map (fun mr ->
         logger#debug "Checking mini rule with pattern %s" mr.MR.pattern_string;
         let res =
           Semgrep_generic.check
             ~hook:(fun _ _ -> ())
             config [ mr ] equivalences (file, lang, ast)
         in
         if !debug_matches then (
           let json = res |> List.map JSON_report.match_to_json in
           let json_uniq = Common.uniq_by ( = ) json in
           let res_uniq =
             Common.uniq_by (AST_utils.with_structural_equal PM.equal) res
           in
           logger#debug
             "Found %d mini rule matches (uniq = %d) (json_uniq = %d)"
             (List.length res) (List.length res_uniq) (List.length json_uniq);
           res |> List.iter (fun m -> logger#debug "match = %s" (PM.show m)) );
         res)
  |> List.flatten

(*****************************************************************************)
(* Evaluating spacegrep *)
(*****************************************************************************)
let matches_of_spacegrep spacegreps file =
  (* coupling: mostly copypaste of Spacegrep_main.run_all *)
  (*
     We inspect the first 4096 bytes to guess whether the file type.
     This saves time on large files, by reading typically just one
     block from the file system.
   *)
  let peek_length = 4096 in
  let partial_doc_src = Spacegrep.Src_file.of_file ~max_len:peek_length file in
```

```
let doc_type = Spacegrep.File_type.classify partial_doc_src in
match doc_type with
| Minified | Binary ->
    logger#info "ignoring gibberish file: %s\n%!" file;
    ([], 0.0, 0.0)
| _ ->
    let src =
      if
        Spacegrep.Src_file.length partial_doc_src < peek_length
        (* it's actually complete, no need to re-input the file *)
      then partial_doc_src
      else Spacegrep.Src_file.of_file file
    in
    let doc, parse_time =
      Common.with_time (fun () -> Spacegrep.Parse_doc.of_src src)
    in
    (* pr (Spacegrep.Doc_AST.show doc); *)
    let res, match_time =
      Common.with_time (fun () ->
          spacegreps
          |> List.map (fun (pat, id, pstr) ->
                 let matches =
                   Spacegrep.Match.search ~case_sensitive:true src pat doc
                 in
                 matches
                 |> List.map (fun m ->
                        let (pos1, _), (_pos2, _) =
                          m.Spacegrep.Match.region
                        in
                        let { Spacegrep.Match.value = str; _ } =
                          m.Spacegrep.Match.capture
                        in
                        let env =
                          m.Spacegrep.Match.named_captures
                          |> List.map (fun (s, capture) ->
                                 let mvar = "$" ^ s in
                                 let {
                                   Spacegrep.Match.value = str;
                                   loc = pos, _;
                                 } =
                                   capture
                                 in
                                 let loc = lexing_pos_to_loc file pos str in
                                 let t = info_of_token_location loc in
                                 let mval = mval_of_spacegrep_string str t in
                                 (mvar, mval))
                        in

                        let loc = lexing_pos_to_loc file pos1 str in
                        (* this will be adjusted later *)
                        let rule_id = fake_rule_id (id, pstr) in
                        {
                          PM.rule_id;
                          file;
                          range_loc = (loc, loc);
                          env;
                          tokens = lazy [ info_of_token_location loc ];
                        }))
          |> List.flatten)
    in
```

```
      (res, parse_time, match_time)
  [@@profiling]

(*****************************************************************************)
(* Evaluating regexps *)
(*****************************************************************************)

let matches_of_regexs regexps lazy_content file =
  let big_str, parse_time =
    Common.with_time (fun () -> Lazy.force lazy_content)
  in
  let res, match_time =
    Common.with_time (fun () ->
        regexps
        |> List.map (fun ((s, re), id, _pstr) ->
               let subs =
                 try Pcre.exec_all ~rex:re big_str with Not_found -> [||]
               in
               subs |> Array.to_list
               |> List.map (fun sub ->
                      let charpos, _ = Pcre.get_substring_ofs sub 0 in
                      let str = Pcre.get_substring sub 0 in

                      let line, column = line_col_of_charpos file charpos in
                      let loc = { PI.str; charpos; file; line; column } in
                      (* this will be re-adjusted later *)
                      let rule_id = fake_rule_id (id, s) in
                      {
                        PM.rule_id;
                        file;
                        range_loc = (loc, loc);
                        tokens = lazy [ info_of_token_location loc ];
                        env = [];
                      }))
        |> List.flatten)
  in
  (res, parse_time, match_time)
  [@@profiling]

(*****************************************************************************)
(* Evaluating xpatterns *)
(*****************************************************************************)

let matches_of_xpatterns config orig_rule
    (file, xlang, lazy_ast_and_errors, lazy_content) xpatterns =
  (* Right now you can only mix semgrep/regexps and spacegrep/regexps, but
   * in theory we could mix all of them together.This is why below
   * I don't match over xlang and instead assume we could have the 3 different
   * kind of patterns at the same time.
   *)
  let patterns, spacegreps, regexps = partition_xpatterns xpatterns in

  (* semgrep *)
  let semgrep_res =
    match xlang with
    | R.L (lang, _) ->
        let (ast, errors), parse_time =
          Common.with_time (fun () -> lazy_force lazy_ast_and_errors)
        in
        let (matches, errors), match_time =
```

```
          Common.with_time (fun () ->
              let mini_rules =
                patterns |> List.map (mini_rule_of_pattern orig_rule)
              in
              let equivalences =
                (* TODO *)
                []
              in
              (* debugging path *)
              if !debug_timeout || !debug_matches then
                ( debug_semgrep config mini_rules equivalences file lang ast,
                  errors ) (* regular path *)
              else
                ( Semgrep_generic.check
                    ~hook:(fun _ _ -> ())
                    config mini_rules equivalences (file, lang, ast),
                  errors ))
        in
        { RP.matches; errors; profiling = { RP.parse_time; match_time } }
    | _ -> RP.empty_semgrep_result
  in

  (* spacegrep *)
  let spacegrep_matches, spacegrep_parse_time, spacegrep_match_time =
    if spacegreps = [] then ([], 0.0, 0.0)
    else matches_of_spacegrep spacegreps file
  in

  (* regexps *)
  let regexp_matches, regexp_parse_time, regexp_match_time =
    if regexps = [] then ([], 0.0, 0.0)
    else matches_of_regexs regexps lazy_content file
  in

  (* final result *)
  {
    RP.matches = semgrep_res.matches @ regexp_matches @ spacegrep_matches;
    errors = semgrep_res.errors;
    profiling =
      {
        RP.parse_time =
          semgrep_res.profiling.parse_time +. regexp_parse_time
          +. spacegrep_parse_time;
        match_time =
          semgrep_res.profiling.match_time +. regexp_match_time
          +. spacegrep_match_time;
      };
  }
  [@@profiling]

(*****************************************************************************)
(* Formula evaluation *)
(*****************************************************************************)
(* TODO: use Set instead of list? *)
let rec (evaluate_formula : env -> R.formula -> range_with_mvars list) =
 fun env e ->
  match e with
  | R.Leaf (R.P (xpat, inside)) ->
      let id = xpat.R.pid in
      let match_results =
```

```
      try Hashtbl.find_all env.pattern_matches id with Not_found -> []
    in
    match_results
    |> List.map match_result_to_range
    |> List.map (fun r -> { r with inside })
| R.Or xs -> xs |> List.map (evaluate_formula env) |> List.flatten
| R.And xs -> (
    let pos, neg, conds = split_and xs in

    (* we now treat pattern: and pattern-inside: differently. We first
     * process the pattern: and then the pattern-inside.
     * This fixed only one mismatch in semgrep-rules.
     * old:
     *  (match pos with
     *  | [] -> failwith "empty And; no positive terms in And"
     *  | start::pos ->
     *     let res = evaluate_formula env start in
     *    let res = pos |> List.fold_left (fun acc x ->
     *      intersect_ranges acc (evaluate_formula env x)
     * ...
     *)

    (* let's start with the positive ranges *)
    let posrs = List.map (evaluate_formula env) pos in
    (* subtle: we need to process and intersect the pattern-inside after
     * (see tests/OTHER/rules/inside.yaml).
     * TODO: this is ugly; AND should be commutative, so we should just
     * merge ranges, not just filter one or the other.
     * update: however we have some tests that rely on pattern-inside:
     * being special, see tests/OTHER/rules/and_inside.yaml.
     *)
    let posrs, posrs_inside =
      posrs
      |> Common.partition_either (fun xs ->
             match xs with
             (* todo? should we double check they are all inside? *)
             | { inside = Some R.Inside; _ } :: _ -> Right xs
             | _ -> Left xs)
    in
    match posrs @ posrs_inside with
    | [] -> failwith "empty And; no positive terms in And"
    | posr :: posrs ->
        let res = posr in
        let res =
          posrs
          |> List.fold_left
               (fun acc r -> intersect_ranges env.config acc r)
               res
        in

        (* let's remove the negative ranges *)
        let res =
          neg
          |> List.fold_left
               (fun acc x ->
                 difference_ranges env.config acc (evaluate_formula env x))
               res
        in
        (* let's apply additional filters.
         * TODO: Note that some metavariable-regexp may be part of an
```

```
           * AND where not all patterns define the metavar, e.g.,
           *   pattern-inside: def $FUNC() ...
           *   pattern: return $X
           *   metavariable-regexp: $FUNC regex: (foo|bar)
           * in which case the order in which we do the operation matters
           * (at this point intersect_range will have filtered the
           *  range of the pattern_inside).
           * alternative solutions?
           *  - bind closer metavariable-regexp with the relevant pattern
           *  - propagate metavariables when intersecting ranges
           *  - distribute filter_range in intersect_range?
           * See https://github.com/returntocorp/semgrep/issues/2664
           *)
          let res =
            conds |> List.fold_left (fun acc cond -> filter_ranges acc cond) res
          in
          res )
  | R.Not _ -> failwith "Invalid Not; you can only negate inside an And"
  | R.Leaf (R.MetavarCond _) ->
      failwith "Invalid MetavarCond; you can MetavarCond only inside an And"
 [@@profiling]

(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)

let check hook config rules file_and_more =
  let file, xlang, lazy_ast_and_errors = file_and_more in
  logger#info "checking %s with %d rules" file (List.length rules);
  if !Common.profile = Common.ProfAll then (
    logger#info "forcing eval of ast outside of rules, for better profile";
    lazy_force lazy_ast_and_errors |> ignore );
  let lazy_content = lazy (Common.read_file file) in
  rules
  |> List.map (fun r ->
         Common.profile_code (spf "real_rule:%s" r.R.id) (fun () ->
             let formula = Rule.formula_of_rule r in
             let relevant_rule =
               if !Flag_semgrep.filter_irrelevant_rules then (
                 match Analyze_rule.regexp_prefilter_of_rule r with
                 | None -> true
                 | Some (re, f) ->
                     let content = Lazy.force lazy_content in
                     logger#info "looking for %s in %s" re file;
                     f content )
               else true
             in
             if not relevant_rule then (
               logger#info "skipping rule %s for %s" r.R.id file;
               RP.empty_semgrep_result )
             else
               let xpatterns = xpatterns_in_formula formula in
               let res =
                 matches_of_xpatterns config r
                   (file, xlang, lazy_ast_and_errors, lazy_content)
                   xpatterns
               in
               logger#info "found %d matches" (List.length res.matches);
               (* match results per minirule id which is the same than pattern_id in
                * the formula *)
```

```
                let pattern_matches_per_id =
                  group_matches_per_pattern_id res.matches
                in
                let env =
                  { config; pattern_matches = pattern_matches_per_id; file }
                in
                logger#info "evaluating the formula";
                let final_ranges = evaluate_formula env formula in
                logger#info "found %d final ranges" (List.length final_ranges);

                {
                  matches =
                    final_ranges
                    |> List.map (range_to_pattern_match_adjusted r)
                    (* dedup similar findings (we do that also in Semgrep_generic.ml,
                     * but different mini-rules matches can now become the same match)
                     *)
                    |> Common.uniq_by (AST_utils.with_structural_equal PM.equal)
                    |> before_return (fun v ->
                           v
                           |> List.iter (fun (m : Pattern_match.t) ->
                                  let str = spf "with rule %s" r.R.id in
                                  hook str m.env m.tokens));
                  errors = res.errors;
                  profiling = res.profiling;
                }))
    |> RP.collate_semgrep_results
    [@@profiling]
```

F.15 semgrep/finding/

semgrep/finding/Files_finder.mli

⟨semgrep/finding/Files_finder.mli 405a⟩≡

```
  ⟨signature Files_finder.files_of_dirs_or_files 119i⟩
```

semgrep/finding/Files_finder.ml

⟨semgrep/finding/Files_finder.ml 405b⟩≡

```
  ⟨pad/r2c copyright 11⟩

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

  (*****************************************************************************)
  (* Main entry point *)
  (*****************************************************************************)
  ⟨function Files_finder.files_of_dirs_or_files 119j⟩
```

semgrep/finding/Files_filter.mli

⟨semgrep/finding/Files_filter.mli 405c⟩≡

```
(* This uses an external "globbing" library whose syntax is similar
 * to UNIX globbing (as in .gitignore file for example).
 * See https://dune.readthedocs.io/en/stable/concepts.html#glob
 * for more information on its syntax.
 *)
type glob
```

⟨*type* `Files_filter.filters` 119k⟩

⟨*exception* `Files_filter.GlobSyntaxError` 120b⟩
⟨*signature* `Files_filter.mk_filters` 120c⟩

⟨*signature* `Files_filter.filter` 120d⟩

semgrep/finding/Files_filter.ml

⟨`semgrep/finding/Files_filter.ml` 406⟩≡
⟨*pad/r2c copyright* 11⟩

```
module Glob = Dune_glob__Glob

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Filter files.
 *
 * In theory we should use find ... | grep ... | xargs sgrep ...
 * which would be more the UNIX spirit, but on huge codebase
 * xargs fails.
 *
 * We just copy the options in GNU grep.
 *
 * todo?
 *  - also process .gitignore as in ripgrep?
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* `Files_filter.glob` 120a⟩

⟨*type* `Files_filter.filters` 119k⟩

⟨*exception* `Files_filter.GlobSyntaxError` 120b⟩

```
(*****************************************************************************)
(* Parsing *)
(*****************************************************************************)
```

⟨*function* `Files_filter.mk_filters` 120e⟩

```
(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)
```

⟨*function* `Files_filter.filter` 120f⟩

semgrep/finding/Unit_files.mli

⟨semgrep/finding/Unit_files.mli 407a⟩≡
  ⟨*signature* Unit_files.unittest 120g⟩

semgrep/finding/Unit_files.ml

⟨semgrep/finding/Unit_files.ml 407b⟩≡
```
open OUnit
```

  ⟨*constant* Unit_files.unittest 121a⟩

F.16 semgrep/metachecking/

semgrep/parsing/Check_pattern.mli

⟨semgrep/parsing/Check_pattern.mli 407c⟩≡

  ⟨*signature* Check_semgrep.check_pattern 121c⟩

semgrep/parsing/Check_pattern.ml

⟨semgrep/parsing/Check_pattern.ml 407d⟩≡

  ⟨*constant* Check_semgrep.lang_has_no_dollar_ids 122a⟩

  ⟨*function* Check_semgrep.check_pattern_metavars 122b⟩

  ⟨*function* Check_semgrep.check_pattern 122c⟩

semgrep/parsing/Check_rule.mli

⟨semgrep/metachecking/Check_rule.mli 407e⟩≡

```
  (* will populate Error_code.errors *)
  val check : Rule.t -> unit

  val check_files :
    (Common.filename -> Rule.t list) -> Common.filename list -> unit

  val stat_files :
    (Common.filename -> Rule.t list) -> Common.filename list -> unit
```

semgrep/parsing/Check_pattern.ml

⟨semgrep/metachecking/Check_rule.ml 407f⟩≡
  ⟨*pad/r2c copyright* 11⟩
```
  open Common
  module FT = File_type
  open Rule
  module R = Rule
  module E = Error_code
```

```
let logger = Logging.get_logger [ __MODULE__ ]

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Checking the checker (metachecking).
 *
 * The goal of this module is to detect bugs, performance issues, or
 * to suggest the use of certain features in semgrep rules.
 *
 * See also semgrep-rules/meta/ for semgrep meta rules expressed in
 * semgrep itself!
 *
 * When to use semgrep/yaml (or spacegrep) to express meta rules and
 * when to use OCaml (this file)?
 * When you need to express meta rules on the yaml structure,
 * then semgrep/yaml is fine, but sometimes you need to express
 * meta-rules by inspecting the pattern content, in which case
 * you have to use OCaml (a bit like in templating languages).
 *
 * TODO infra:
 *  - use our new position-aware yaml parser to parse a rule,
 *    so we can give error messages on patterns with the right location.
 * TODO rules:
 *  - detect if scope of metavariable-regexp is wrong and should be put
 *    in a AND with the relevant pattern. If used with an AND of OR,
 *    make sure all ORs define the metavar.
 *    see https://github.com/returntocorp/semgrep/issues/2664
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
type env = Rule.t

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

(* TODO: use a Parse_info.t when we switch to YAML with position info *)
let error (env : env) s =
  let loc = Parse_info.first_loc_of_file env.file in
  let s = spf "%s (in ruleid: %s)" s env.id in
  let check_id = "semgrep-metacheck-rule" in
  let err = E.mk_error_loc loc (E.SemgrepMatchFound (check_id, s)) in
  pr2 (E.string_of_error err)

(*****************************************************************************)
(* Formula *)
(*****************************************************************************)

let show_formula pf =
  match pf with Leaf (P (x, _)) -> x.pstr | _ -> R.show_formula pf

let equal_formula x y = AST_utils.with_structural_equal R.equal_formula x y

let check_formula env lang f =
  (* check duplicated patterns, essentially:
   *  $K: $PAT
```

```
   *  ...
   *  $K2: $PAT
   * but at the same level!
   *
   * See also now semgrep-rules/meta/identical_pattern.sgrep :)
   *)
  let rec find_dupe f =
    match f with
    | Leaf (P _) -> ()
    | Leaf (MetavarCond _) -> ()
    | Not f -> find_dupe f
    | Or xs | And xs ->
        let rec aux xs =
          match xs with
          | [] -> ()
          | x :: xs ->
              (* todo: for Pat, we could also check if exist PatNot
               * in which case intersection will always be empty
               *)
              if xs |> List.exists (equal_formula x) then
                error env (spf "Duplicate pattern %s" (show_formula x));
              if xs |> List.exists (equal_formula (Not x)) then
                error env (spf "Unsatisfiable patterns %s" (show_formula x));
              aux xs
        in
        (* breadth *)
        aux xs;
        (* depth *)
        xs |> List.iter find_dupe
  in
  find_dupe f;

  (* call Check_pattern subchecker *)
  f
  |> visit_new_formula (fun { pat; pstr = _pat_str; pid = _ } ->
         match (pat, lang) with
         | Sem (semgrep_pat, _lang), L (lang, _rest) ->
             Check_pattern.check lang semgrep_pat
         | Spacegrep _spacegrep_pat, LGeneric -> ()
         | Regexp _, _ -> ()
         | _ -> raise Impossible);
  ()

(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)

let check r =
  (* less: maybe we could also have formula_old specific checks *)
  let f = Rule.formula_of_rule r in
  check_formula r r.languages f;
  ()

(* We parse the parsing function fparser (Parser_rule.parse) to avoid
 * circular dependencies.
 * Similar to Test_parsing.test_parse_rules.
 *)
let check_files fparser xs =
  let fullxs =
    xs
```

```
    |> File_type.files_of_dirs_or_files (function
         | FT.Config (FT.Yaml (*FT.Json |*) | FT.Jsonnet) -> true
         | _ -> false)
    |> Skip_code.filter_files_if_skip_list ~root:xs
  in
  fullxs
  |> List.iter (fun file ->
         logger#info "processing %s" file;
         let rs = fparser file in
         rs |> List.iter check)

let stat_files fparser xs =
  let fullxs =
    xs
    |> File_type.files_of_dirs_or_files (function
         | FT.Config (FT.Yaml (*FT.Json |*) | FT.Jsonnet) -> true
         | _ -> false)
    |> Skip_code.filter_files_if_skip_list ~root:xs
  in
  let good = ref 0 in
  let bad = ref 0 in
  fullxs
  |> List.iter (fun file ->
         logger#info "processing %s" file;
         let rs = fparser file in
         rs
         |> List.iter (fun r ->
                let res = Analyze_rule.regexp_prefilter_of_rule r in
                match res with
                | None ->
                    incr bad;
                    pr2 (spf "PB: no regexp prefilter for rule %s:%s" file r.id)
                | Some (s, _f) ->
                    incr good;
                    pr2 (spf "regexp: %s" s)));
  pr2 (spf "good = %d, no regexp found = %d" !good !bad)
```

F.17 semgrep/parsing/

semgrep/parsing/Parse_target.mli

⟨semgrep/parsing/Parse_target.mli 410⟩≡

```
type parsing_result = {
  ast : AST_generic.program;
  errors : Error_code.error list;
  stat : Parse_info.parsing_stat;
}

(* This uses either the pfff or tree-sitter parsers.
 * This also resolve names and propagate constants.
 *)
val parse_and_resolve_name_use_pfff_or_treesitter :
  Lang.t -> Common.filename -> parsing_result

(* used only for testing purpose *)
val just_parse_with_lang : Lang.t -> Common.filename -> parsing_result
```

```
  val parse_program : Common.filename -> AST_generic.program

  (* used by Parse_pattern *)
  val lang_to_python_parsing_mode : Lang.t -> Parse_python.parsing_mode
```

semgrep/parsing/Parse_target.ml

⟨semgrep/parsing/Parse_target.ml 411⟩≡
⟨*pad/r2c copyright* 11⟩

```
  open Common
  module Flag = Flag_semgrep
  module PI = Parse_info
  module E = Error_code

  let logger = Logging.get_logger [ __MODULE__ ]

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Mostly a wrapper around pfff Parse_generic, but which can also use
   * tree-sitter parsers when possible.
   *)

  (*****************************************************************************)
  (* Types *)
  (*****************************************************************************)

  type parsing_result = {
    ast : AST_generic.program;
    errors : Error_code.error list;
    stat : Parse_info.parsing_stat;
  }

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)

  type 'ast parser =
    | Pfff of (Common.filename -> 'ast * Parse_info.parsing_stat)
    | TreeSitter of (Common.filename -> 'ast Tree_sitter_run.Parsing_result.t)

  type 'ast internal_result =
    | Ok of ('ast * Parse_info.parsing_stat)
    | Partial of 'ast * Error_code.error list * Parse_info.parsing_stat
    | Error of exn

  let error_of_tree_sitter_error (err : Tree_sitter_run.Tree_sitter_error.t) =
    let start = err.start_pos in
    let loc =
      {
        PI.str = err.substring;
        charpos = 0;
        (* fake *)
        line = start.row + 1;
        column = start.column;
        file = err.file.name;
```

```
    }
  in
  let info = { PI.token = PI.OriginTok loc; transfo = PI.NoTransfo } in
  PI.Parsing_error info

let stat_of_tree_sitter_stat file (stat : Tree_sitter_run.Parsing_result.stat) =
  {
    Parse_info.filename = file;
    total_line_count = stat.total_line_count;
    error_line_count = stat.error_line_count;
    have_timeout = false;
    commentized = 0;
    problematic_lines = [];
  }

let (run_parser : 'ast parser -> Common.filename -> 'ast internal_result) =
 fun parser file ->
  match parser with
  | Pfff f ->
      Common.save_excursion Flag_parsing.show_parsing_error false (fun () ->
          logger#info "trying to parse with Pfff parser %s" file;
          try
            let res = f file in
            Ok res
          with
          | Timeout -> raise Timeout
          | exn ->
              logger#debug "exn (%s) with Pfff parser" (Common.exn_to_s exn);
              Error exn)
  | TreeSitter f -> (
      logger#info "trying to parse with TreeSitter parser %s" file;
      try
        let res = f file in
        let stat = stat_of_tree_sitter_stat file res.stat in
        match (res.program, res.errors) with
        | None, [] -> raise Impossible
        | Some ast, [] -> Ok (ast, stat)
        | None, ts_error :: _xs ->
            let exn = error_of_tree_sitter_error ts_error in
            logger#info "non-recoverable error (%s) with TreeSitter parser"
              (Common.exn_to_s exn);
            Error exn
        | Some ast, x :: _xs ->
            (* let's just return the first one for now; the following one
             * may be due to cascading effect of the first error *)
            let exn = error_of_tree_sitter_error x in
            logger#info "partial error (%s) with TreeSitter parser"
              (Common.exn_to_s exn);
            let err = E.exn_to_error file exn in
            Partial (ast, [ err ], stat)
      with
      | Timeout -> raise Timeout
      | exn ->
          logger#debug "exn (%s) with TreeSitter parser" (Common.exn_to_s exn);
          Error exn )

let rec (run_either :
          Common.filename -> 'ast parser list -> 'ast internal_result) =
 fun file xs ->
  match xs with
```

```
  | [] -> Error (Failure (spf "no parser found for %s" file))
  | p :: xs -> (
      let res = run_parser p file in
      match res with
      | Ok ast -> Ok ast
      | Partial (ast, errs, stat) -> (
          let res = run_either file xs in
          match res with
          | Ok res -> Ok res
          | Error exn2 ->
              logger#debug "exn again (%s) but return Partial"
                (Common.exn_to_s exn2);
              (* prefer a Partial to an Error *)
              Partial (ast, errs, stat)
          | Partial _ ->
              logger#debug "Partial again but return first Partial";
              Partial (ast, errs, stat) )
      | Error exn -> (
          let res = run_either file xs in
          match res with
          | Ok res -> Ok res
          | Partial (ast, errs, stat) ->
              logger#debug "Got now a Partial, better than exn (%s)"
                (Common.exn_to_s exn);
              Partial (ast, errs, stat)
          | Error exn2 ->
              logger#debug "exn again (%s) but return original exn (%s)"
                (Common.exn_to_s exn2) (Common.exn_to_s exn);
              (* prefer the first error *)
              Error exn ) )

let (run :
      Common.filename ->
      'ast parser list ->
      ('ast -> AST_generic.program) ->
      parsing_result) =
 fun file xs fconvert ->
  let xs =
    match () with
    | _ when !Flag.tree_sitter_only ->
        xs |> Common.exclude (function Pfff _ -> true | _ -> false)
    | _ when !Flag.pfff_only ->
        xs |> Common.exclude (function TreeSitter _ -> true | _ -> false)
    | _ -> xs
  in
  match run_either file xs with
  | Ok (ast, stat) -> { ast = fconvert ast; errors = []; stat }
  | Partial (ast, errs, stat) -> { ast = fconvert ast; errors = errs; stat }
  | Error exn -> raise exn

let throw_tokens f file =
  let res = f file in
  (res.PI.ast, res.PI.stat)

let lang_to_python_parsing_mode = function
  | Lang.Python -> Parse_python.Python
  | Lang.Python2 -> Parse_python.Python2
  | Lang.Python3 -> Parse_python.Python3
  | s -> failwith (spf "not a python language:%s" (Lang.string_of_lang s))
```

```
let just_parse_with_lang lang file =
  match lang with
  | Lang.Ruby ->
      (* for Ruby we start with the tree-sitter parser because the pfff parser
       * is not great and some of the token positions may be wrong.
       *)
      run file
        [
          TreeSitter Parse_ruby_tree_sitter.parse;
          (* right now the parser is verbose and the token positions
           * may be wrong, but better than nothing. *)
          Pfff (throw_tokens Parse_ruby.parse);
        ]
        Ruby_to_generic.program
  | Lang.Java ->
      run file
        [
          (* we used to start with the pfff one; it was quite good and faster
           * than tree-sitter (because we need to wrap tree-sitter inside
           * an invoke because of a segfault/memory-leak), but when both parsers
           * fail, it's better to give the tree-sitter parsing error now.
           *)
          TreeSitter Parse_java_tree_sitter.parse;
          Pfff (throw_tokens Parse_java.parse);
        ]
        Java_to_generic.program
  | Lang.Go ->
      run file
        [
          TreeSitter Parse_go_tree_sitter.parse;
          Pfff (throw_tokens Parse_go.parse);
        ]
        Go_to_generic.program
  | Lang.Javascript ->
      (* we start directly with tree-sitter here, because
       * the pfff parser is slow on minified files due to its (slow) error
       * recovery strategy.
       *)
      run file
        [
          TreeSitter Parse_javascript_tree_sitter.parse;
          Pfff (throw_tokens Parse_js.parse);
        ]
        Js_to_generic.program
  | Lang.Typescript ->
      run file
        [ TreeSitter (Parse_typescript_tree_sitter.parse ?dialect:None) ]
        Js_to_generic.program
  (* there is no pfff parsers for C#/Kotlin/... so let's go directly to
   *  tree-sitter, and there's no ast_xxx.ml either so we directly generate
   * a generic AST (no xxx_to_generic here)
   *)
  | Lang.Csharp ->
      run file [ TreeSitter Parse_csharp_tree_sitter.parse ] (fun x -> x)
  | Lang.Kotlin ->
      run file [ TreeSitter Parse_kotlin_tree_sitter.parse ] (fun x -> x)
  | Lang.Lua -> run file [ TreeSitter Parse_lua_tree_sitter.parse ] (fun x -> x)
  | Lang.Rust ->
      run file [ TreeSitter Parse_rust_tree_sitter.parse ] (fun x -> x)
  | Lang.C ->
```

```
      run file
        [
          (* this internally uses the CST for c++ *)
          Pfff (throw_tokens Parse_c.parse);
          TreeSitter Parse_c_tree_sitter.parse;
        ]
        C_to_generic.program
  (* use pfff *)
  | Lang.Python | Lang.Python2 | Lang.Python3 ->
      let parsing_mode = lang_to_python_parsing_mode lang in
      run file
        [ Pfff (throw_tokens (Parse_python.parse ~parsing_mode)) ]
        (* old: Resolve_python.resolve ast;
         * switched to call Naming_AST.ml to correct def and use tagger
         *)
        Python_to_generic.program
  | Lang.JSON ->
      run file
        [
          Pfff
            (fun file ->
              (Parse_json.parse_program file, Parse_info.correct_stat file));
        ]
        Json_to_generic.program
  | Lang.Cplusplus -> failwith "TODO"
  | Lang.OCaml ->
      run file
        [
          Pfff (throw_tokens Parse_ml.parse);
          (* TODO TreeSitter Parse_ocaml_tree_sitter.parse; *)
        ]
        Ml_to_generic.program
  | Lang.PHP ->
      run file
        [ Pfff (throw_tokens Parse_php.parse) ]
        (fun cst ->
          let ast = Ast_php_build.program cst in
          Php_to_generic.program ast)
  | Lang.R -> failwith "No R parser yet; improve the one in tree-sitter"
  | Lang.Yaml ->
      {
        ast = Yaml_to_generic.program file;
        errors = [];
        stat = Parse_info.default_stat file;
      }

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

let parse_and_resolve_name_use_pfff_or_treesitter lang file =
  let { ast; errors; stat } = just_parse_with_lang lang file in

  (* to be deterministic, reset the gensym; anyway right now semgrep is
   * used only for local per-file analysis, so no need to have a unique ID
   * among a set of files in a project like codegraph.
   *)
  AST_generic_helpers.gensym_counter := 0;
  Naming_AST.resolve lang ast;
  Constant_propagation.propagate_basic lang ast;
```

```
    Constant_propagation.propagate_dataflow ast;
    if !Flag.use_bloom_filter then Bloom_annotation.annotate_program ast;

    logger#info "Parse_target.parse_and_resolve_name_use_pfff_or_treesitter done";
    { ast; errors; stat }

  (*****************************************************************************)
  (* For testing purpose *)
  (*****************************************************************************)
  (* was in pfff/.../Parse_generic.ml before *)
  let parse_program file =
    let lang = List.hd (Lang.langs_of_filename file) in
    let res = just_parse_with_lang lang file in
    res.ast
```

semgrep/parsing/Parse_pattern.mli

⟨semgrep/parsing/Parse_pattern.mli 416a⟩≡

```
  val parse_pattern : Lang.t -> ?print_errors:bool -> string -> Pattern.t

  val dump_tree_sitter_pattern_cst : Lang.t -> Common.filename -> unit
```

semgrep/parsing/Parse_pattern.ml

⟨semgrep/parsing/Parse_pattern.ml 416b⟩≡

```
  ⟨pad/r2c copyright 11⟩
  open Common

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Mostly a wrapper around pfff Parse_generic.
  *)

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)

  let dump_and_print_errors dumper (res : 'a Tree_sitter_run.Parsing_result.t) =
    ( match res.program with
    | Some cst -> dumper cst
    | None -> failwith "unknown error from tree-sitter parser" );
    res.errors
    |> List.iter (fun err ->
           pr2 (Tree_sitter_run.Tree_sitter_error.to_string ~color:true err))

  let extract_pattern_from_tree_sitter_result
      (res : 'a Tree_sitter_run.Parsing_result.t) (print_errors : bool) =
    match (res.Tree_sitter_run.Parsing_result.program, res.errors) with
    | None, _ -> failwith "no pattern found"
    | Some x, [] -> x
    | Some _, _ :: _ ->
        if print_errors then
          res.errors
          |> List.iter (fun err ->
```

```
              pr2 (Tree_sitter_run.Tree_sitter_error.to_string ~color:true err));
      failwith "error parsing the pattern"

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
let parse_pattern lang ?(print_errors = false) str =
  let any =
    match lang with
    | Lang.Csharp ->
        let res = Parse_csharp_tree_sitter.parse_pattern str in
        extract_pattern_from_tree_sitter_result res print_errors
    | Lang.Lua ->
        let res = Parse_lua_tree_sitter.parse_pattern str in
        extract_pattern_from_tree_sitter_result res print_errors
    | Lang.Rust ->
        let res = Parse_rust_tree_sitter.parse_pattern str in
        extract_pattern_from_tree_sitter_result res print_errors
    | Lang.Kotlin ->
        let res = Parse_kotlin_tree_sitter.parse_pattern str in
        extract_pattern_from_tree_sitter_result res print_errors
    (* use pfff *)
    | Lang.Python | Lang.Python2 | Lang.Python3 ->
        let parsing_mode = Parse_target.lang_to_python_parsing_mode lang in
        let any = Parse_python.any_of_string ~parsing_mode str in
        Python_to_generic.any any
    (* abusing JS parser so no need extend tree-sitter grammar*)
    | Lang.Typescript | Lang.Javascript ->
        let any = Parse_js.any_of_string str in
        Js_to_generic.any any
    | Lang.JSON ->
        let any = Parse_json.any_of_string str in
        Json_to_generic.any any
    | Lang.C ->
        let any = Parse_c.any_of_string str in
        C_to_generic.any any
    | Lang.Java ->
        let any = Parse_java.any_of_string str in
        Java_to_generic.any any
    | Lang.Go ->
        let any = Parse_go.any_of_string str in
        Go_to_generic.any any
    | Lang.OCaml ->
        let any = Parse_ml.any_of_string str in
        Ml_to_generic.any any
    | Lang.Ruby ->
        let any = Parse_ruby.any_of_string str in
        Ruby_to_generic.any any
    | Lang.PHP ->
        let any_cst = Parse_php.any_of_string str in
        let any = Ast_php_build.any any_cst in
        Php_to_generic.any any
    | Lang.Cplusplus -> failwith "No C++ generic parser yet"
    | Lang.R -> failwith "No R generic parser yet"
    | Lang.Yaml -> Yaml_to_generic.any str
  in

  Caching.prepare_pattern any;
  Check_pattern.check lang any;
  any
```

```
  let dump_tree_sitter_pattern_cst lang file =
    match lang with
    | Lang.Csharp ->
        Tree_sitter_c_sharp.Parse.file file
        |> dump_and_print_errors Tree_sitter_c_sharp.CST.dump_tree
    | Lang.Lua ->
        Tree_sitter_lua.Parse.file file
        |> dump_and_print_errors Tree_sitter_lua.CST.dump_tree
    | Lang.Rust ->
        Tree_sitter_rust.Parse.file file
        |> dump_and_print_errors Tree_sitter_rust.CST.dump_tree
    | Lang.Kotlin ->
        Tree_sitter_kotlin.Parse.file file
        |> dump_and_print_errors Tree_sitter_kotlin.CST.dump_tree
    | _ -> ()
```

semgrep/parsing/Parse_rule.mli

⟨semgrep/parsing/Parse_rule.mli 418a⟩≡

```
  val parse : Common.filename -> Rule.rules

  (* used also by Convert_rule.ml *)
  val parse_metavar_cond : string -> AST_generic.expr
```

semgrep/parsing/Parse_rule.ml

⟨semgrep/parsing/Parse_rule.ml 418b⟩≡

```
  ⟨pad/r2c copyright 11⟩
  open Common
  module J = JSON
  module FT = File_type
  module R = Rule
  module E = Parse_mini_rule
  module H = Parse_mini_rule

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Parsing a Semgrep rule, including complex pattern formulas.
   *
   * See also the JSON schema in rule_schema.yaml
   *
   * TODO:
   *  - use the new position-aware YAML parser to get position information (for
   *    precise error location) by using an AST_generic expression instead of
   *    JSON.t (at the same time, in the long term we want
   *    to use JSON and jsonnet, so we might get anyway a line location
   *    in a generated file, so maybe better to give error location by
   *    describing the line and what is wrong with it?).
   *  - Move the H.xxx here and get rid of Parse_mini_rule.ml
   *)

  (*****************************************************************************)
  (* Helpers *)
```

```
(*****************************************************************************)

(* less: could use a hash to accelerate things *)
let rec find_fields flds xs =
  match flds with
  | [] -> ([], xs)
  | fld :: flds ->
      let fld_match = List.assoc_opt fld xs in
      let xs = List.remove_assoc fld xs in
      let matches, rest = find_fields flds xs in
      ((fld, fld_match) :: matches, rest)

let error s = raise (E.InvalidYamlException s)

let rec yaml_to_json = function
  | `Null -> J.Null
  | `Bool b -> J.Bool b
  | `Float f ->
      (* or use J.Int? *)
      J.Float f
  | `String s -> J.String s
  | `A xs -> J.Array (xs |> List.map yaml_to_json)
  | `O xs -> J.Object (xs |> List.map (fun (k, v) -> (k, yaml_to_json v)))

(*****************************************************************************)
(* Sub parsers basic types *)
(*****************************************************************************)
let parse_string ctx = function
  | J.String s -> s
  | x ->
      pr2_gen x;
      error (spf "parse_string for %s" ctx)

let parse_strings ctx = function
  | J.Array xs -> List.map (fun t -> parse_string ctx t) xs
  | x ->
      pr2_gen x;
      error (spf "parse_strings for %s" ctx)

let parse_bool ctx = function
  | J.String "true" -> true
  | J.String "false" -> false
  | J.Bool b -> b
  | x ->
      pr2_gen x;
      error (spf "parse_bool for %s" ctx)

let parse_int ctx = function
  | J.String s -> (
      try int_of_string s
      with Failure _ -> error (spf "parse_int  for %s" ctx) )
  | J.Float f ->
      let i = int_of_float f in
      if float_of_int i = f then i
      else (
        pr2_gen f;
        error "not an int" )
  | x ->
      pr2_gen x;
      error (spf "parse_int for %s" ctx)
```

```
(*****************************************************************************)
(* Sub parsers extra *)
(*****************************************************************************)

let parse_metavar_cond s =
  try
    let lang = Lang.Python in
    (* todo? use lang in env? *)
    match Parse_pattern.parse_pattern lang ~print_errors:false s with
    | AST_generic.E e -> e
    | _ -> error "not an expression"
  with exn -> raise exn

let parse_regexp s =
  try (s, Pcre.regexp s)
  with Pcre.Error _ as exn ->
    failwith
      (spf "failing to parse regexp %s, error = %s" s (Common.exn_to_s exn))

let parse_extra _env x =
  match x with
  | "metavariable-regex", J.Object xs -> (
      match find_fields [ "metavariable"; "regex" ] xs with
      | ( [
            ("metavariable", Some (J.String metavar));
            ("regex", Some (J.String regexp));
          ],
          [] ) ->
          R.MetavarRegexp (metavar, parse_regexp regexp)
      | x ->
          pr2_gen x;
          error "wrong parse_extra fields" )
  | "metavariable-comparison", J.Object xs -> (
      match
        find_fields [ "metavariable"; "comparison"; "strip"; "base" ] xs
      with
      | ( [
            ("metavariable", Some (J.String metavariable));
            ("comparison", Some (J.String comparison));
            ("strip", strip_opt);
            ("base", base_opt);
          ],
          [] ) ->
          let comparison = parse_metavar_cond comparison in
          R.MetavarComparison
            {
              R.metavariable;
              comparison;
              strip = Common.map_opt (parse_bool "strip") strip_opt;
              base = Common.map_opt (parse_int "base") base_opt;
            }
      | x ->
          pr2_gen x;
          error "wrong parse_extra fields" )
  | "pattern-where-python", J.String s -> R.PatWherePython s
  | x ->
      pr2_gen x;
      error "wrong parse_extra fields"
```

```
let parse_fix_regex = function
  | J.Object xs -> (
      match find_fields [ "regex"; "replacement"; "count" ] xs with
      | ( [
            ("regex", Some (J.String regex));
            ("replacement", Some (J.String replacement));
            ("count", count_opt);
          ],
          [] ) ->
          ( parse_regexp regex,
            Common.map_opt (parse_int "count") count_opt,
            replacement )
      | x ->
          pr2_gen x;
          error "parse_fix_regex" )
  | x ->
      pr2_gen x;
      error "parse_fix_regex"

let parse_equivalences = function
  | J.Array xs ->
      xs
      |> List.map (function
           | J.Object [ ("equivalence", J.String s) ] -> s
           | x ->
               pr2_gen x;
               error "parse_equivalence")
  | x ->
      pr2_gen x;
      error "parse_equivalences"

let parse_paths = function
  | J.Object xs -> (
      match find_fields [ "include"; "exclude" ] xs with
      | [ ("include", inc_opt); ("exclude", exc_opt) ], [] ->
          {
            R.include_ =
              ( match inc_opt with
              | None -> []
              | Some xs -> parse_strings "include" xs );
            exclude =
              ( match exc_opt with
              | None -> []
              | Some xs -> parse_strings "exclude" xs );
          }
      | x ->
          pr2_gen x;
          error "parse_paths" )
  | x ->
      pr2_gen x;
      error "parse_paths"

(*****************************************************************************)
(* Sub parsers patterns and formulas *)
(*****************************************************************************)

type _env = string * R.xlang

let parse_pattern (id, lang) s =
  match lang with
```

```
  | R.L (lang, _) -> R.mk_xpat (Sem (H.parse_pattern ~id ~lang s, lang)) s
  | R.LNone -> failwith "you should not use real pattern with language = none"
  | R.LGeneric ->
      let src = Spacegrep.Src_file.of_string s in
      let ast = Spacegrep.Parse_pattern.of_src src in
      R.mk_xpat (Spacegrep ast) s

let rec parse_formula_old env (x : string * J.t) : R.formula_old =
  match x with
  | "pattern", J.String pattern_string ->
      let pattern = parse_pattern env pattern_string in
      R.Pat pattern
  | "pattern-not", J.String pattern_string ->
      let pattern = parse_pattern env pattern_string in
      R.PatNot pattern
  | "pattern-inside", J.String pattern_string ->
      let pattern = parse_pattern env pattern_string in
      R.PatInside pattern
  | "pattern-not-inside", J.String pattern_string ->
      let pattern = parse_pattern env pattern_string in
      R.PatNotInside pattern
  | "pattern-either", J.Array xs ->
      R.PatEither
        (List.map
           (fun x ->
             match x with
             | J.Object [ x ] -> parse_formula_old env x
             | x ->
                 pr2_gen x;
                 error "wrong parse_formula fields")
           xs)
  | "patterns", J.Array xs ->
      R.Patterns
        (List.map
           (fun x ->
             match x with
             | J.Object [ x ] -> parse_formula_old env x
             | x ->
                 pr2_gen x;
                 error "wrong parse_formula fields")
           xs)
  | "pattern-regex", J.String s ->
      let xpat = R.mk_xpat (Regexp (parse_regexp s)) s in
      R.Pat xpat
  | x ->
      let extra = parse_extra env x in
      R.PatExtra extra

let rec parse_formula_new env (x : J.t) : R.formula =
  match x with
  | J.String s -> R.Leaf (R.P (parse_pattern env s, None))
  | J.Object xs -> (
      match xs with
      | [ ("and", J.Array xs) ] ->
          let xs = xs |> List.map (parse_formula_new env) in
          R.And xs
      | [ ("or", J.Array xs) ] ->
          let xs = xs |> List.map (parse_formula_new env) in
          R.Or xs
      | [ ("not", v) ] ->
```

```
          let f = parse_formula_new env v in
          R.Not f
      | [ ("inside", J.String s) ] ->
          R.Leaf (R.P (parse_pattern env s, Some Inside))
      | [ ("regex", J.String s) ] ->
          let xpat = R.mk_xpat (R.Regexp (parse_regexp s)) s in
          R.Leaf (R.P (xpat, None))
      | [ ("where", J.String s) ] ->
          R.Leaf (R.MetavarCond (R.CondGeneric (parse_metavar_cond s)))
      | [ ("metavariable_regex", J.Array [ J.String mvar; J.String re ]) ] ->
          R.Leaf (R.MetavarCond (R.CondRegexp (mvar, parse_regexp re)))
      | _ ->
          pr2_gen x;
          error "parse_formula_new" )
  | _ ->
      pr2_gen x;
      error "parse_formula_new"

let parse_formula env (x : string * J.t) : R.pformula =
  match x with
  | "match", v -> R.New (parse_formula_new env v)
  | _ -> R.Old (parse_formula_old env x)

let parse_languages ~id langs =
  match langs with
  | [ J.String ("none" | "regex") ] -> R.LNone
  | [ J.String "generic" ] -> R.LGeneric
  | xs -> (
      let languages =
        xs
        |> List.map (function
             | J.String s -> (
                 match Lang.lang_of_string_opt s with
                 | None ->
                     raise
                       (E.InvalidLanguageException
                          (id, spf "unsupported language: %s" s))
                 | Some l -> l )
             | _ ->
                 raise
                   (E.InvalidRuleException
                      (id, spf "expecting a string for languages")))
      in
      match languages with
      | [] ->
          raise (E.InvalidRuleException (id, "we need at least one language"))
      | x :: xs -> R.L (x, xs) )

(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)

let top_fields =
  [
    "id";
    "languages";
    "message";
    "severity";
    (* pattern* handled specialy via parse_formula *)
```

```
    (* optional *)
    "metadata";
    "fix";
    "fix-regex";
    "paths";
    "equivalences";
  ]

let parse_json file json =
  match json with
  | J.Object [ ("rules", J.Array xs) ] ->
      xs
      |> List.map (fun v ->
             match v with
             | J.Object xs -> (
                 match find_fields top_fields xs with
                 (* coupling: the order of the fields below must match the
                  * order in top_fields. *)
                 | ( [
                       ("id", Some (J.String id));
                       ("languages", Some (J.Array langs));
                       ("message", Some (J.String message));
                       ("severity", Some (J.String sev));
                       ("metadata", metadata_opt);
                       ("fix", fix_opt);
                       ("fix-regex", fix_regex_opt);
                       ("paths", paths_opt);
                       ("equivalences", equivs_opt);
                     ],
                     rest ) ->
                     let languages = parse_languages ~id langs in
                     let formula =
                       match rest with
                       | [ x ] -> parse_formula (id, languages) x
                       | x ->
                           pr2_gen x;
                           error "wrong rule fields"
                     in
                     {
                       R.id;
                       formula;
                       message;
                       languages;
                       file;
                       severity = H.parse_severity ~id sev;
                       (* optional fields *)
                       metadata = metadata_opt;
                       fix = Common.map_opt (parse_string "fix") fix_opt;
                       fix_regexp = Common.map_opt parse_fix_regex fix_regex_opt;
                       paths = Common.map_opt parse_paths paths_opt;
                       equivalences =
                         Common.map_opt parse_equivalences equivs_opt;
                     }
                 | x ->
                     pr2_gen x;
                     error "wrong rule fields" )
             | x ->
                 pr2_gen x;
                 error "wrong rule fields")
  | _ -> error "missing rules entry as top-level key"
```

```
  let parse file =
    let json =
      match FT.file_type_of_file file with
      | FT.Config FT.Yaml -> (
          let str = Common.read_file file in
          let yaml_res = Yaml.of_string str in
          match yaml_res with
          | Result.Ok v -> yaml_to_json v
          | Result.Error (`Msg s) -> raise (E.UnparsableYamlException s) )
      | FT.Config FT.Json -> J.load_json file
      | FT.Config FT.Jsonnet ->
          Common2.with_tmp_file ~str:"parse_rule" ~ext:"json" (fun tmpfile ->
              let cmd = spf "jsonnet %s -o %s" file tmpfile in
              let n = Sys.command cmd in
              if n <> 0 then failwith (spf "error executing %s" cmd);
              J.load_json tmpfile)
      | _ ->
          failwith
            (spf "wrong rule format, only JSON/YAML/JSONNET are valid:%s:" file)
    in
    parse_json file json
```

semgrep/parsing/Parse_mini_rule.mli

⟨semgrep/parsing/Parse_mini_rule.mli 425a⟩≡

```
  ⟨signature Parse_rules.parse 56a⟩

  ⟨exception Parse_rules.InvalidRuleException 57a⟩
  ⟨exception Parse_rules.InvalidLanguageException 57b⟩
  ⟨exception Parse_rules.InvalidPatternException 57c⟩
  ⟨exception Parse_rules.UnparsableYamlException 57d⟩
  ⟨exception Parse_rules.InvalidYamlException 57e⟩

  (* internals used by other parsers (e.g., Parse_tainting_rules.ml) *)

  ⟨signature Parse_rules.parse_languages 57g⟩
  ⟨signature Parse_rules.parse_severity 57h⟩
  ⟨signature Parse_rules.parse_pattern 57i⟩
```

semgrep/parsing/Parse_mini_rule.ml

⟨semgrep/parsing/Parse_mini_rule.ml 425b⟩≡

```
  (* Yoann Padioleau
   *
   * Copyright (C) 2011 Facebook
   * Copyright (C) 2019 r2c
   *
   * 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.
   *
   * 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.
 *)
open Common
module R = Mini_rule

⟨exception Parse_rules.InvalidRuleException 57a⟩
⟨exception Parse_rules.InvalidLanguageException 57b⟩
⟨exception Parse_rules.InvalidPatternException 57c⟩
⟨exception Parse_rules.UnparsableYamlException 57d⟩
⟨exception Parse_rules.InvalidYamlException 57e⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Parse_rules.parse_severity 57j⟩

⟨function Parse_rules.parse_pattern 57k⟩

⟨function Parse_rules.parse_languages 58a⟩

(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)

⟨function Parse_rules.parse 56b⟩

(*
      let sgrep_string = Common.matched1 s in
      let title, msg = match group with
        | title :: description -> title, Common2.unlines description
        | _ -> failwith ("sgrep_lint: expected \"[title]\\n\\n[description]\"")
      in
      Parse_generic.parse_pattern !lang sgrep_string,
      title,
      (* yes ocaml regexps are not that good ... *)
      (if msg =~ "^\\([A-Z]+\\):\\(\\(.\\|\n\\)*\\)"
       then
         let (error_kind, rest_msg) = Common.matched2 msg in
         (match error_kind with
         | _ -> failwith ("sgrep_lint: wrong format: " ^ msg)
         )
        else failwith ("sgrep_lint: wrong format: " ^ msg)
      )
    else raise Impossible
  )
*)
```

semgrep/parsing/Parse_tainting_rules.mli

⟨semgrep/parsing/Parse_tainting_rules.mli 426a⟩≡

```
⟨signature Parse_tainting_rules.parse 194a⟩
```

semgrep/parsing/Parse_tainting_rules.ml

⟨semgrep/parsing/Parse_tainting_rules.ml 426b⟩≡

```
⟨pad/r2c copyright 11⟩
```

```
open Common
module R = Tainting_rule
open Parse_mini_rule (* for the exns *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Parse_tainting_rules.parse_patterns 194b⟩

(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)

⟨function Parse_tainting_rules.parse 194c⟩
```

semgrep/parsing/Parse_equivalences.mli

⟨semgrep/parsing/Parse_equivalences.mli 427a⟩≡

```
⟨signature Parse_equivalences.parse 58b⟩
```

semgrep/parsing/Parse_equivalences.ml

⟨semgrep/parsing/Parse_equivalences.ml 427b⟩≡

```
⟨pad/r2c copyright 11⟩
open Common
module Eq = Equivalence

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨function Parse_equivalences.error 58c⟩

(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)

⟨function Parse_equivalences.parse 58d⟩
```

F.18 semgrep/matching/

semgrep/matching/Unit_matcher.mli

⟨semgrep/matching/Unit_matcher.mli 427c⟩≡

```
⟨signature Unit_matcher.unittest 99f⟩
```

semgrep/matching/Unit_matcher.ml

⟨semgrep/matching/Unit_matcher.ml 427d⟩≡

```
open Common
open OUnit

(*****************************************************************************)
```

```
(* Semgrep Unit tests *)
(*****************************************************************************)
(* TODO:
 *  - we could add unit tests for the range returned by match_sts_sts
 *  - we could add unit tests for the code dealing with equivalences
 *)

⟨function Unit_matcher.unittest 100⟩
```

semgrep/matching/Generic_vs_generic.mli

⟨semgrep/matching/Generic_vs_generic.mli 428a⟩≡

```
(* entry points, used in the sgrep_generic visitors *)
⟨signature Generic_vs_generic.m_expr 65b⟩
⟨signature Generic_vs_generic.m_stmt 65c⟩
⟨signature Generic_vs_generic.m_stmts_deep 92c⟩

val m_type_ : (AST_generic.type_, AST_generic.type_) Matching_generic.matcher

val m_pattern :
  (AST_generic.pattern, AST_generic.pattern) Matching_generic.matcher

val m_attribute :
  (AST_generic.attribute, AST_generic.attribute) Matching_generic.matcher

val m_partial :
  (AST_generic.partial, AST_generic.partial) Matching_generic.matcher

val m_field : (AST_generic.field, AST_generic.field) Matching_generic.matcher

⟨signature Generic_vs_generic.m_any 99e⟩
```

semgrep/matching/Generic_vs_generic.ml

⟨semgrep/matching/Generic_vs_generic.ml 428b⟩≡

```
⟨pad/r2c copyright 11⟩
open Common

(* A is the pattern, and B the concrete source code. For now
 * we both use the same module but they may differ later
 * as the expressivity of the pattern language grows.
 *
 * subtle: use 'b' to report errors, because 'a' is the sgrep pattern and it
 * has no file information usually.
 *)
module A = AST_generic
module B = AST_generic
module MV = Metavariable
module AST = AST_generic
module Flag = Flag_semgrep
module Config = Config_semgrep
module H = AST_generic_helpers

(* optimisations *)
module CK = Caching.Cache_key
module Env = Metavariable_capture
module F = Bloom_filter
```

```
open Matching_generic

let logger = Logging.get_logger [ __MODULE__ ]

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* AST generic vs AST generic code matcher.
 *
 * This module allows to match some AST elements against other AST elements in
 * a flexible way, providing a kind of grep but at a syntactical level.
 *
 * Most of the boilerplate code was generated by
 *    $ pfff/meta/gen_code -matcher_gen_all
 * using OCaml pad-style reflection (see commons/OCaml.ml) on
 * h_program-lang/AST_generic.ml.
 *
 * See pfff/matcher/fuzzy_vs_fuzzy.ml for another approach.
 *
 * There are four main features allowing a "pattern" to match some "code":
 *  - metavariables can match anything (see metavar: tag in this file)
 *  - '...' can match any sequence (see dots: tag)
 *  - simple constructs match complex constructs having more details
 *    (e.g., the absence of attribute in a pattern will still match functions
 *     having many attributes) (see less-is-ok: tag)
 *  - the underlying AST uses some normalization (!= is transformed in !(..=))
 *    to support certain code equivalences (see equivalence: tag)
 *  - we do not care about differences in spaces/indentations/comments.
 *    we work at the AST-level.
 *
 * alternatives:
 *  - would it be simpler to work on a simpler AST, like a Term language,
 *    or even a Node/Leaf? or Ast_fuzzy? the "less-is-ok" would be
 *    difficult with that approach, because you need to know that some
 *    parts of the AST are attributes/annotations that can be skipped.
 *    In the same way, code equivalences like name resolution on the AST
 *    would be more difficult with an untyped-general tree.
 *)

(*****************************************************************************)
(* Extra Helpers *)
(*****************************************************************************)

let env_add_matched_stmt rightmost_stmt (tin : tin) =
  [ extend_stmts_match_span rightmost_stmt tin ]

⟨function Generic_vs_generic.m_string_xhp_text 99c⟩

(* less: could be made more general by taking is_dots function parameter *)
let has_ellipsis_and_filter_ellipsis xs =
  let has_ellipsis = ref false in
  let ys =
    xs
    |> Common.exclude (function
         | A.Ellipsis _ ->
             has_ellipsis := true;
             true
         | _ -> false)
  in
  (!has_ellipsis, ys)
```

```
let has_xml_ellipsis_and_filter_ellipsis xs =
  let has_ellipsis = ref false in
  let ys =
    xs
    |> Common.exclude (function
         | A.XmlEllipsis _ ->
             has_ellipsis := true;
             true
         | _ -> false)
  in
  (!has_ellipsis, ys)

let has_case_ellipsis_and_filter_ellipsis xs =
  let has_ellipsis = ref false in
  let ys =
    xs
    |> Common.exclude (function
         | A.CaseEllipsis _ ->
             has_ellipsis := true;
             true
         | _ -> false)
  in
  (!has_ellipsis, ys)

let rec obj_and_method_calls_of_expr = function
  | B.Call (B.DotAccess (e, tok, fld), args) ->
      let o, xs = obj_and_method_calls_of_expr e in
      (o, (fld, tok, args) :: xs)
  | o -> (o, [])

let rec expr_of_obj_and_method_calls (obj, xs) =
  match xs with
  | [] -> obj
  | (fld, tok, args) :: xs ->
      let e = expr_of_obj_and_method_calls (obj, xs) in
      B.Call (B.DotAccess (e, tok, fld), args)

let rec all_suffix_of_list xs =
  xs :: (match xs with [] -> [] | _x :: xs -> all_suffix_of_list xs)

let _ =
  Common2.example
    (all_suffix_of_list [ 1; 2; 3 ] = [ [ 1; 2; 3 ]; [ 2; 3 ]; [ 3 ]; [] ])

(*****************************************************************************)
(* Optimisations (caching, bloom filter) *)
(*****************************************************************************)

(* Getters and setters that were left abstract in the cache implementation. *)
let cache_access : tin Caching.Cache.access =
  {
    get_span_field = (fun tin -> tin.stmts_match_span);
    set_span_field = (fun tin x -> { tin with stmts_match_span = x });
    get_mv_field = (fun tin -> tin.mv);
    set_mv_field = (fun tin mv -> { tin with mv });
  }

let stmts_may_match pattern_stmts (stmts : AST_generic.stmt list) =
  if not !Flag.use_bloom_filter then F.Maybe
```

```
  else
    let pattern_list =
      Bloom_annotation.list_of_pattern_strings (Ss pattern_stmts)
    in
    let pat_in_stmt pat (stmt : AST_generic.stmt) =
      match stmt.s_bf with None -> F.Maybe | Some bf -> F.mem pat bf
    in
    let rec pattern_in_any_stmt pat stmts acc =
      match stmts with
      | [] -> acc
      | stmt :: rest -> (
          match acc with
          | F.No -> pattern_in_any_stmt pat rest (pat_in_stmt pat stmt)
          | F.Maybe -> acc )
    in
    let patterns_all_in_stmts acc x =
      match acc with
      | F.No -> Bloom_filter.No
      | Maybe -> pattern_in_any_stmt x stmts F.No
    in
    List.fold_left patterns_all_in_stmts F.Maybe pattern_list
  [@@profiling]

(*****************************************************************************)
(* Name *)
(*****************************************************************************)
```

⟨*function* Generic_vs_generic.m_ident 70a⟩

⟨*function* Generic_vs_generic.m_dotted_name 226b⟩

⟨*function* Generic_vs_generic.make_dotted 109a⟩

```
(* similar to m_list_prefix but binding $X to the whole list *)
let rec m_dotted_name_prefix_ok a b =
  match (a, b) with
  | [], [] -> return ()
  | [ (s, t) ], [ x ] when MV.is_metavar_name s -> envf (s, t) (MV.Id (x, None))
  | [ (s, t) ], _ :: _ when MV.is_metavar_name s ->
      (* TODO: should we bind it instead to a MV.N IdQualified?
       * but it is actually just the qualifier part; the last id
       * is not here (even though make_dottd will not care about that)
       *)
      envf (s, t) (MV.E (make_dotted b))
  | xa :: aas, xb :: bbs ->
      let* () = m_ident xa xb in
      m_dotted_name_prefix_ok aas bbs
  (* prefix is ok *)
  | [], _ -> return ()
  | _ :: _, _ -> fail ()
```

⟨*function* Generic_vs_generic.m_module_name_prefix 95c⟩

⟨*function* Generic_vs_generic.m_module_name 227c⟩

⟨*function* Generic_vs_generic.m_sid 106b⟩

⟨*function* Generic_vs_generic.m_resolved_name_kind 227d⟩

⟨*function* Generic_vs_generic._m_resolved_name 106c⟩

```
(* start of recursive need *)
```
⟨*function* Generic_vs_generic.m_name 226c⟩

⟨*function* Generic_vs_generic.m_name_info 230a⟩
```
and m_qualifier a b =
  match (a, b) with
  | A.QDots a, B.QDots b -> m_dotted_name a b
  | A.QTop a, B.QTop b -> m_tok a b
  | A.QExpr (a1, a2), B.QExpr (b1, b2) -> m_expr a1 b1 >>= fun () -> m_tok a2 b2
  | A.QDots _, _ | A.QTop _, _ | A.QExpr _, _ -> fail ()

and m_type_option_with_hook idb taopt tbopt =
  match (taopt, tbopt) with
  | Some ta, Some tb -> m_type_ ta tb
  | Some ta, None -> (
      match !Hooks.get_type idb with
      | Some tb -> m_type_ ta tb
      | None -> fail () )
  (* less-is-ok:, like m_option_none_can_match_some *)
  | None, _ -> return ()
```

⟨*function* Generic_vs_generic.m_ident_and_id_info_add_in_env_Expr 78a⟩
```
and m_ident_and_empty_id_info a1 b1 =
  let empty = AST.empty_id_info () in
  m_ident_and_id_info (a1, empty) (b1, empty)
```

⟨*function* Generic_vs_generic.m_id_info 106a⟩

```
(*****************************************************************************)
(* Expression *)
(*****************************************************************************)

(* possibly go deeper when someone wants that a pattern like
 *   'bar();'
 * match also an expression statement like
 *   'x = bar();'.
 *
 * This is very hacky.
 *
 * alternatives:
 *  - force the user to use 'if(... <expr> ...)' (isaac, jmelton)
 *  - do as in coccinelle and use 'if(<... <expr> ...>)'
 *  - CURRENT: impicitely go deep without requiring an extra syntax
 *
 * todo? we could restrict ourselves to only a few forms? see SubAST_generic.ml
 *   - x = <expr>,
 *   - <call>(<exprs).
 *)
(* experimental! *)
```
⟨*function* Generic_vs_generic.m_expr_deep 89d⟩

```
(* coupling: if you add special sgrep hooks here, you should probably
 * also add them in m_pattern
 *)
```
⟨*function* Generic_vs_generic.m_expr 70b⟩

⟨*function* Generic_vs_generic.m_field_ident 72c⟩

⟨*function* Generic_vs_generic.m_label_ident 229d⟩

⟨*function* Generic_vs_generic.m_literal 71⟩
```
and m_wrap_m_int_opt (a1, a2) (b1, b2) =
  match (a1, b1) with
  (* iso: semantic equivalence of value! 0x8 can match 8 *)
  | Some i1, Some i2 -> if i1 =|= i2 then return () else fail ()
  (* if the integers (or floats) were too large or were using
   * a syntax OCaml int_of_string could not parse,
   * we default to a string comparison *)
  | _ ->
      let a1 = Parse_info.str_of_info a2 in
      (* bugfix: not that with constant propagation, some integers don't have
       * a real token associated with them, so b2 may be a FakeTok, but
       * Parse_info.str_of_info does not raise an exn anymore on a FakeTok
       *)
      let b1 = Parse_info.str_of_info b2 in
      m_wrap m_string (a1, a2) (b1, b2)

and m_wrap_m_float_opt (a1, a2) (b1, b2) =
  match (a1, b1) with
  (* iso: semantic equivalence of value! 0x8 can match 8 *)
  | Some i1, Some i2 when i1 = i2 -> return ()
  | _ ->
      let a1 = Parse_info.str_of_info a2 in
      let b1 = Parse_info.str_of_info b2 in
      m_wrap m_string (a1, a2) (b1, b2)

and m_literal_constness a b =
  match b with
  | B.Lit b1 -> m_literal a b1
  | B.Cst B.Cstr -> (
      match a with A.String ("...", _) -> return () | ___else___ -> fail () )
  | B.Cst _ | B.NotCst -> fail ()
```

⟨*function* Generic_vs_generic.m_action 229e⟩

⟨*function* Generic_vs_generic.m_arithmetic_operator 229f⟩

⟨*function* Generic_vs_generic.m_special 229g⟩

```
(* fstring pattern should match only fstring *)
and m_concat_string_kind a b =
  match (a, b) with
  | A.FString, B.FString -> return ()
  | A.FString, _ -> fail ()
  (* less-is-more: *)
  | _ -> return ()

and m_container_set_or_dict_unordered_elements (a1, a2) (b1, b2) =
  match ((a1, a2), (b1, b2)) with
  (* those rules should be applied only for python? *)
  | ((A.Dict | A.Set), []), ((A.Dict | A.Set), []) -> return ()
  | ((A.Dict | A.Set), [ A.Ellipsis _ ]), ((A.Dict | A.Set), _) -> return ()
  | (A.Set, a2), (B.Set, b2) ->
      let has_ellipsis, a2 = has_ellipsis_and_filter_ellipsis a2 in
      m_list_in_any_order ~less_is_ok:has_ellipsis m_expr a2 b2
  | (A.Dict, a2), (B.Dict, b2) ->
      let has_ellipsis, a2 = has_ellipsis_and_filter_ellipsis a2 in
      m_list_in_any_order ~less_is_ok:has_ellipsis m_expr a2 b2
  | _, _ ->
```

```
      (* less: could return fail () *)
      m_container_operator a1 b1 >>= fun () -> m_list m_expr a2 b2
```

⟨*function* Generic_vs_generic.m_container_operator 72a⟩

⟨*function* Generic_vs_generic.m_container_ordered_elements 82a⟩

⟨*function* Generic_vs_generic.m_other_expr_operator 229a⟩

```
and m_compatible_type typed_mvar t e =
  match (t, e) with
  (* for Python literal checking *)
  | ( A.OtherType (A.OT_Expr, [ A.E (A.N (A.Id (("int", _tok), _idinfo))) ]),
      B.L (B.Int _) ) ->
      envf typed_mvar (MV.E e)
  | ( A.OtherType (A.OT_Expr, [ A.E (A.N (A.Id (("float", _tok), _idinfo))) ]),
      B.L (B.Float _) ) ->
      envf typed_mvar (MV.E e)
  | ( A.OtherType (A.OT_Expr, [ A.E (A.N (A.Id (("str", _tok), _idinfo))) ]),
      B.L (B.String _) ) ->
      envf typed_mvar (MV.E e)
  (* for java literals *)
  | A.TyBuiltin ("int", _), B.L (B.Int _) -> envf typed_mvar (MV.E e)
  | A.TyBuiltin ("float", _), B.L (B.Float _) -> envf typed_mvar (MV.E e)
  | A.TyN (A.Id (("String", _), _)), B.L (B.String _) ->
      envf typed_mvar (MV.E e)
  (* for C specific literals *)
  | A.TyPointer (_, TyBuiltin ("char", _)), B.L (B.String _) ->
      envf typed_mvar (MV.E e)
  | A.TyPointer (_, _), B.L (B.Null _) -> envf typed_mvar (MV.E e)
  (* for go literals *)
  | A.TyN (Id (("int", _), _)), B.L (B.Int _) -> envf typed_mvar (MV.E e)
  | A.TyN (Id (("float", _), _)), B.L (B.Float _) -> envf typed_mvar (MV.E e)
  | A.TyN (Id (("string", _), _)), B.L (B.String _) -> envf typed_mvar (MV.E e)
  (* for C strings to match metavariable pointer types *)
  | A.TyPointer (t1, A.TyN (A.Id ((_, tok), _id_info))), B.L (B.String _) ->
      m_type_ t (A.TyPointer (t1, TyBuiltin ("char", tok))) >>= fun () ->
      envf typed_mvar (MV.E e)
  (* for matching ids *)
  | ta, B.N (B.Id (idb, ({ B.id_type = tb; _ } as id_infob))) ->
      (* NOTE: Name values must be represented with MV.Id! *)
      m_type_option_with_hook idb (Some ta) !tb >>= fun () ->
      envf typed_mvar (MV.Id (idb, Some id_infob))
  | ( ta,
      ( B.N (B.IdQualified ((idb, _), { B.id_type = tb; _ }))
      | B.DotAccess
          (IdSpecial (This, _), _, EN (Id (idb, { B.id_type = tb; _ }))) ) ) ->
      m_type_option_with_hook idb (Some ta) !tb >>= fun () ->
      envf typed_mvar (MV.E e)
  | _ -> fail ()

(*----------------------------------------------------------------------------*)
(* XML *)
(*----------------------------------------------------------------------------*)
```

⟨*function* Generic_vs_generic.m_xml 230b⟩

```
and m_xml_kind a b =
  match (a, b) with
  (* iso: allow a Classic to match a Singleton, and vice versa *)
  | A.XmlClassic (a0, a1, a2, _), B.XmlSingleton (b0, b1, b2)
  | A.XmlSingleton (a0, a1, a2), B.XmlClassic (b0, b1, b2, _) ->
```

```
      let* () = m_tok a0 b0 in
      let* () = m_ident a1 b1 in
      let* () = m_tok a2 b2 in
      return ()
  | A.XmlClassic (a0, a1, a2, a3), B.XmlClassic (b0, b1, b2, b3) ->
      let* () = m_tok a0 b0 in
      let* () = m_ident a1 b1 in
      let* () = m_tok a2 b2 in
      let* () = m_tok a3 b3 in
      return ()
  | A.XmlSingleton (a0, a1, a2), B.XmlSingleton (b0, b1, b2) ->
      let* () = m_tok a0 b0 in
      let* () = m_ident a1 b1 in
      let* () = m_tok a2 b2 in
      return ()
  | A.XmlFragment (a1, a2), B.XmlFragment (b1, b2) ->
      let* () = m_tok a1 b1 in
      let* () = m_tok a2 b2 in
      return ()
  | A.XmlClassic _, _ | A.XmlSingleton _, _ | A.XmlFragment _, _ -> fail ()
```

⟨*function* Generic_vs_generic.m_attrs 87c⟩

⟨*function* Generic_vs_generic.m_bodies 94f⟩

⟨*function* Generic_vs_generic.m_list__m_body 95a⟩

⟨*function* Generic_vs_generic.m_xml_attr 230c⟩

⟨*function* Generic_vs_generic.m_xml_attr_value 99d⟩

⟨*function* Generic_vs_generic.m_body 99b⟩

```
(*---------------------------------------------------------------------------*)
(* Arguments list iso *)
(*---------------------------------------------------------------------------*)
```

⟨*function* Generic_vs_generic.m_arguments 82f⟩

```
(* less: factorize in m_list_and_dots? but also has unordered for kwd args *)
```
⟨*function* Generic_vs_generic.m_list__m_argument 83a⟩

```
(* special case m_arguments when inside a Call(Special(Concat,_), ...)
 * less: factorize with m_list_with_dots? hard because of the special
 * call to Normalize_generic below.
 *)
```
⟨*function* Generic_vs_generic.m_arguments_concat 85e⟩

⟨*function* Generic_vs_generic.m_argument 72b⟩

⟨*function* Generic_vs_generic.m_other_argument_operator 229b⟩

```
(*****************************************************************************)
(* Type *)
(*****************************************************************************)
```

⟨*function* Generic_vs_generic.m_type_ 75c⟩

⟨*function* Generic_vs_generic.m_ident_and_type_ 231b⟩

⟨*function* Generic_vs_generic.m_type_arguments 231c⟩

⟨*function* Generic_vs_generic.m_type_argument 232a⟩
```
and m_wildcard (a1, a2) (b1, b2) =
  let* () = m_wrap m_bool a1 b1 in
  m_type_ a2 b2
```

⟨*function* Generic_vs_generic.m_other_type_operator 232b⟩
⟨*function* Generic_vs_generic.m_other_type_argument_operator 232c⟩
```
(*****************************************************************************)
(* Attribute *)
(*****************************************************************************)
```

⟨*function* Generic_vs_generic.m_keyword_attribute 98d⟩

⟨*function* Generic_vs_generic.m_attribute 76a⟩

⟨*function* Generic_vs_generic.m_other_attribute_operator 232d⟩

```
(*****************************************************************************)
(* Statement list *)
(*****************************************************************************)
(* possibly go deeper when someone wants that a pattern like
 *   ...
 *   bar();
 * to match also calls to bar() deeply as in
 *   foo();
 *   if(true)
 *      bar();
 *
 * When combined with the deep expr, this even allows to match code like
 *  if(true)
 *     x = bar();
 *
 * This is currently very hacky. We just flatten the list of all substmts.
 *
 * alternatives:
 *   - do it the right way by having '...' work on control-flow paths as in
 *     coccinelle
 *
 * todo? we could restrict ourselves to only a few forms?
 *)
(* experimental! *)
and m_stmts_deep ~less_is_ok (xsa : A.stmt list) (xsb : A.stmt list) tin =
  (* shares the cache with m_list__m_stmt *)
  match (tin.cache, xsa, xsb) with
  | Some cache, a :: _, _ :: _ when a.s_use_cache ->
      let tin = { tin with mv = Env.update_min_env tin.mv a } in
      Caching.Cache.match_stmt_list ~access:cache_access ~cache
        ~function_id:CK.Match_deep ~list_kind:CK.Original ~less_is_ok
        ~compute:(m_stmts_deep_uncached ~less_is_ok)
        ~pattern:xsa ~target:xsb tin
  | _ -> m_stmts_deep_uncached ~less_is_ok xsa xsb tin
```

⟨*function* Generic_vs_generic.m_stmts_deep 91⟩
```
and m_list__m_stmt ~list_kind xsa xsb tin =
  (* shares the cache with m_stmts_deep *)
  match (tin.cache, xsa, xsb) with
  | Some cache, a :: _, _ :: _ when a.s_use_cache ->
      let tin = { tin with mv = Env.update_min_env tin.mv a } in
```

```
      Caching.Cache.match_stmt_list ~access:cache_access ~cache
        ~function_id:CK.Match_list ~list_kind ~less_is_ok:true
        ~compute:(m_list__m_stmt_uncached ~list_kind)
        ~pattern:xsa ~target:xsb tin
  | _ -> m_list__m_stmt_uncached ~list_kind xsa xsb tin

(* TODO: factorize with m_list_and_dots less_is_ok = true *)
⟨function Generic_vs_generic.m_list__m_stmt 83c⟩

(*****************************************************************************)
(* Statement *)
(*****************************************************************************)

⟨function Generic_vs_generic.m_stmt 73a⟩
and m_case_clauses a b =
  let _has_ellipsis, a = has_case_ellipsis_and_filter_ellipsis a in
  (* todo? always implicit ...?
   * todo? do in any order? In theory the order of the cases matter, but
   * in a semgrep context, people probably don't want to find
   * specific cases in a specific order.
   *)
  m_list_in_any_order ~less_is_ok:true m_case_and_body a b

⟨function Generic_vs_generic.m_for_header 233a⟩
and m_block a b =
  match (a.s, b.s) with
  | A.Block _, B.Block _ -> m_stmt a b
  | A.Block (_, [ a_stmt ], _), _ -> m_stmt a_stmt b
  | _, B.Block (_, [ b_stmt ], _) -> m_stmt a b_stmt
  | _, _ -> m_stmt a b

⟨function Generic_vs_generic.m_for_var_or_expr 233b⟩

⟨function Generic_vs_generic.m_label 234a⟩

⟨function Generic_vs_generic.m_catch 234b⟩

⟨function Generic_vs_generic.m_finally 234c⟩

⟨function Generic_vs_generic.m_case_and_body 234d⟩

⟨function Generic_vs_generic.m_case 234e⟩

⟨function Generic_vs_generic.m_other_stmt_operator 234f⟩

⟨function Generic_vs_generic.m_other_stmt_with_stmt_operator 234g⟩

(*****************************************************************************)
(* Pattern *)
(*****************************************************************************)

⟨function Generic_vs_generic.m_pattern 76c⟩

⟨function Generic_vs_generic.m_field_pattern 235a⟩

⟨function Generic_vs_generic.m_other_pattern_operator 235b⟩

(*****************************************************************************)
(* Definitions *)
(*****************************************************************************)
```

⟨*function* Generic_vs_generic.m_definition 73c⟩

⟨*function* Generic_vs_generic.m_entity 78b⟩

⟨*function* Generic_vs_generic.m_definition_kind 73d⟩

⟨*function* Generic_vs_generic.m_type_parameter_constraint 235d⟩
```
and m_other_type_parameter_operator = m_other_xxx
```

⟨*function* Generic_vs_generic.m_type_parameter_constraints 236a⟩

⟨*function* Generic_vs_generic.m_type_parameter 236b⟩

```
(* ------------------------------------------------------------------------ *)
(* Function (or method) definition *)
(* ------------------------------------------------------------------------ *)

(* iso: we don't care if it's a Function or Arrow *)
and m_function_kind _ _ = return ()
```

⟨*function* Generic_vs_generic.m_function_definition 74a⟩

⟨*function* Generic_vs_generic.m_parameters 82e⟩

⟨*function* Generic_vs_generic.m_parameter 74b⟩

⟨*function* Generic_vs_generic.m_parameter_classic 74c⟩

⟨*function* Generic_vs_generic.m_other_parameter_operator 236d⟩

```
(* ------------------------------------------------------------------------ *)
(* Variable definition *)
(* ------------------------------------------------------------------------ *)
```

⟨*function* Generic_vs_generic.m_variable_definition 74d⟩

```
(* ------------------------------------------------------------------------ *)
(* Field definition and use *)
(* ------------------------------------------------------------------------ *)

(* As opposed to statements, the order of fields should not matter.
 *
 * We actually filter the '...' and use a less-is-ok approach.
 * Indeed '...' are not really useful, and in some patological cases they
 * were actually leading to the use a tons of memory. Indeed, in certain
 * files containing a long list of fields (like 3000 static fields),
 * the classic use of >|> to handle Ellipsis variations were stressing
 * a lot the engine. Simpler to just filter them.
 *)
```
⟨*function* Generic_vs_generic.m_fields 75a⟩

```
(* less: mix of m_list_and_dots and m_list_unordered_keys, hard to factorize *)
```
⟨*function* Generic_vs_generic.m_list__m_field 84b⟩

⟨*function* Generic_vs_generic.m_field 75b⟩

```
(* ------------------------------------------------------------------------ *)
(* Type definition *)
(* ------------------------------------------------------------------------ *)
```

⟨*function* Generic_vs_generic.m_type_definition 236f⟩

⟨*function* Generic_vs_generic.m_type_definition_kind 236g⟩

⟨*function* Generic_vs_generic.m_or_type 237a⟩

⟨*function* Generic_vs_generic.m_other_type_kind_operator 237b⟩

⟨*function* Generic_vs_generic.m_other_or_type_element_operator 237c⟩

⟨*function* Generic_vs_generic.m_list__m_type_ 99a⟩
```
and m_list__m_type_any_order (xsa : A.type_ list) (xsb : A.type_ list) =
  (* TODO? filter existing ellipsis?
   * let _has_ellipsis, xsb = has_ellipsis_and_filter_ellipsis xsb in *)
  (* always implicit ... *)
  m_list_in_any_order ~less_is_ok:true m_type_ xsa xsb

(* ------------------------------------------------------------------------ *)
(* Class definition *)
(* ------------------------------------------------------------------------ *)
(* TODO: there are a few remaining m_list m_type_ we could transform
 * to use instead m_list__m_type_, for Exception, TyTuple and OrConstructor
 * but maybe quite different from list of types in inheritance
 *)
```

⟨*function* Generic_vs_generic.m_class_definition 74e⟩

⟨*function* Generic_vs_generic.m_class_kind 237d⟩

```
(* ------------------------------------------------------------------------ *)
(* Module definition *)
(* ------------------------------------------------------------------------ *)
```

⟨*function* Generic_vs_generic.m_module_definition 237e⟩

⟨*function* Generic_vs_generic.m_module_definition_kind 238a⟩

⟨*function* Generic_vs_generic.m_other_module_operator 238b⟩

```
(* ------------------------------------------------------------------------ *)
(* Macro definition *)
(* ------------------------------------------------------------------------ *)
```

⟨*function* Generic_vs_generic.m_macro_definition 238c⟩

```
(*****************************************************************************)
(* Directives (Module import/export, macros) *)
(*****************************************************************************)
```

⟨*function* Generic_vs_generic.m_directive 97d⟩

```
(* less-is-ok: a few of these below with the use of m_module_name_prefix and
 * m_option_none_can_match_some.
 * todo? not sure it makes sense to always allow m_module_name_prefix below
 *)
```
⟨*function* Generic_vs_generic.m_directive_basic 76b⟩

⟨*function* Generic_vs_generic.m_other_directive_operator 238e⟩

```
  (*****************************************************************************)
  (* Toplevel *)
  (*****************************************************************************)
```

⟨*function* Generic_vs_generic.m_item 238g⟩

⟨*function* Generic_vs_generic.m_program 238f⟩

```
  (*****************************************************************************)
  (* Any *)
  (*****************************************************************************)
  and m_partial a b =
    match (a, b) with
    | A.PartialDef a1, B.PartialDef b1 -> m_definition a1 b1
    | A.PartialIf (a1, a2), B.PartialIf (b1, b2) ->
        let* () = m_tok a1 b1 in
        m_expr a2 b2
    | A.PartialTry (a1, a2), B.PartialTry (b1, b2) ->
        let* () = m_tok a1 b1 in
        m_stmt a2 b2
    | A.PartialFinally (a1, a2), B.PartialFinally (b1, b2) ->
        let* () = m_tok a1 b1 in
        m_stmt a2 b2
    | A.PartialCatch a1, B.PartialCatch b1 -> m_catch a1 b1
    | A.PartialSingleField (a1, a2, a3), B.PartialSingleField (b1, b2, b3) ->
        let* () = m_ident a1 b1 in
        let* () = m_tok a2 b2 in
        m_expr a3 b3
    | A.PartialLambdaOrFuncDef a1, B.PartialLambdaOrFuncDef b1 ->
        m_function_definition a1 b1
    | A.PartialDef _, _
    | A.PartialIf _, _
    | A.PartialTry _, _
    | A.PartialCatch _, _
    | A.PartialFinally _, _
    | A.PartialSingleField _, _
    | A.PartialLambdaOrFuncDef _, _ ->
        fail ()
```

⟨*function* Generic_vs_generic.m_any 238h⟩

semgrep/matching/Matching_generic.mli

⟨semgrep/matching/Matching_generic.mli 440⟩≡

⟨*type* Matching_generic.tin 66a⟩
⟨*type* Matching_generic.tout 66b⟩

⟨*type* Matching_generic.matcher 65d⟩

```
  (* monadic combinators *)
```
⟨*signature* Matching_generic.monadic_bind 66c⟩
⟨*signature* Matching_generic.TODOOPERATOR2 67c⟩
⟨*signature* Matching_generic.TODOOPERATOR3 67e⟩

⟨*signature* Matching_generic.return 66d⟩
⟨*signature* Matching_generic.fail 66e⟩

```
  val or_list : ('a, 'b) matcher -> 'a -> 'b list -> tin -> tout
```

```
(* shortcut for >>=, since OCaml 4.08 you can define those "extended-let" *)
val ( let* ) : (tin -> tout) -> (unit -> tin -> tout) -> tin -> tout
```

⟨*signature* Matching_generic.empty_environment 63c⟩

```
val add_mv_capture : Metavariable.mvar -> Metavariable.mvalue -> tin -> tin

val get_mv_capture : Metavariable.mvar -> tin -> Metavariable.mvalue option

(* Update the matching list of statements by providing a new matching
   statement. *)
val extend_stmts_match_span : AST_generic.stmt -> tin -> tin
```

⟨*signature* Matching_generic.envf 79c⟩

```
val if_config :
  (Config_semgrep.t -> bool) ->
  then_:(tin -> tout) ->
  else_:(tin -> tout) ->
  tin ->
  tout
```

⟨*signature* Matching_generic.check_and_add_metavar_binding 79g⟩

```
(* helpers *)
```
⟨*signature* Matching_generic.has_ellipsis_stmts 92a⟩
```
val inits_and_rest_of_list : 'a list -> ('a list * 'a list) list
```

⟨*signature* Matching_generic.all_elem_and_rest_of_list 86d⟩
```
val lazy_rest_of_list : 'a Lazy.t -> 'a
```

⟨*signature* Matching_generic.is_regexp_string 88d⟩
```
type regexp = Re.re
```

⟨*signature* Matching_generic.regexp_of_regexp_string 88f⟩

```
val equal_ast_binded_code :
  Config_semgrep.t -> Metavariable.mvalue -> Metavariable.mvalue -> bool
```

```
(* generic matchers *)
```
⟨*signature* Matching_generic.m_option 68d⟩
⟨*signature* Matching_generic.m_option_ellipsis_ok 84g⟩
⟨*signature* Matching_generic.m_option_none_can_match_some 94d⟩

⟨*signature* Matching_generic.m_ref 68f⟩

⟨*signature* Matching_generic.m_list 68h⟩
⟨*signature* Matching_generic.m_list_prefix 96a⟩
⟨*signature* Matching_generic.m_list_with_dots 82b⟩
```
val m_list_in_any_order :
  less_is_ok:bool -> ('a, 'b) matcher -> ('a list, 'b list) matcher
```

```
(* use = *)
val m_eq : ('a, 'a) matcher
```

⟨*signature* Matching_generic.m_bool 67g⟩
⟨*signature* Matching_generic.m_int 67h⟩
⟨*signature* Matching_generic.m_string 67i⟩
⟨*signature* Matching_generic.string_is_prefix 96c⟩

```
⟨signature Matching_generic.m_string_prefix 96e⟩
val m_string_ellipsis_or_regexp_or_default :
  ?m_string_for_default:(string, string) matcher ->
  string ->
  string ->
  tin ->
  tout

⟨signature Matching_generic.m_info 69c⟩
⟨signature Matching_generic.m_tok 69a⟩
⟨signature Matching_generic.m_wrap 69e⟩
⟨signature Matching_generic.m_bracket 69g⟩
val m_tuple3 :
  ('a -> 'b -> tin -> tout) ->
  ('c -> 'd -> tin -> tout) ->
  ('e -> 'f -> tin -> tout) ->
  'a * 'c * 'e ->
  'b * 'd * 'f ->
  tin ->
  tout

⟨signature Matching_generic.m_other_xxx 227a⟩
```

semgrep/matching/Matching_generic.ml

```
⟨semgrep/matching/Matching_generic.ml 442⟩≡
  ⟨pad/r2c copyright 11⟩
  open Common
  module A = AST_generic
  module B = AST_generic
  module MV = Metavariable
  module H = AST_generic_helpers
  module AST = AST_generic
  module Flag = Flag_semgrep
  module Env = Metavariable_capture

  let logger = Logging.get_logger [ __MODULE__ ]

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Helper types and functions for Generic_vs_generic.ml.
   * See Generic_vs_generic.ml top comment for more information.
   *
   * todo:
   *  - use m_list_in_any_order at some point for:
   *     * m_list__m_field
   *     * m_list__m_attribute
   *     * m_list__m_xml_attr
   *     * m_list__m_argument (harder)
   *)

  (*****************************************************************************)
  (* Types *)
  (*****************************************************************************)

  (* ------------------------------------------------------------------------*)
  (* Combinators history *)
  (* ------------------------------------------------------------------------*)
```

```
(*
 * version0:
 *   type ('a, 'b) matcher = 'a -> 'b -> bool
 *
 *   This just lets you know if you matched something.
 *
 * version1:
 *   type ('a, 'b) matcher = 'a -> 'b -> unit -> ('a, 'b) option
 *
 *   The Maybe monad.
 *
 * version2:
 *   type ('a, 'b) matcher = 'a -> 'b -> binding -> binding list
 *
 *   Why not returning a binding option ? because we need sometimes
 *   to return multiple possible bindings for one matching code.
 *   For instance with the pattern do 'f(..., $X, ...)', $X could be binded
 *   to different parts of the code.
 *
 *   Note that the empty list means a match failure.
 *
 * version3:
 *   type ('a, 'b) matcher = 'a -> 'b -> tin -> ('a,'b) tout
 *
 * version4: back to simpler
 *   type ('a, 'b) matcher = 'a -> 'b -> tin -> tout
 *)
```

⟨*type* Matching_generic.tin 66a⟩
⟨*type* Matching_generic.tout 66b⟩

⟨*type* Matching_generic.matcher 65d⟩

```
(*****************************************************************************)
(* Globals *)
(*****************************************************************************)

(*****************************************************************************)
(* Debugging *)
(*****************************************************************************)
```
⟨*function* Matching_generic.str_of_any 212d⟩

```
(*****************************************************************************)
(* Monadic operators *)
(*****************************************************************************)
(* The >>= combinator below allow you to configure the matching process
 * anyway you want. Essentially this combinator takes a matcher,
 * another matcher, and returns a matcher that combines the 2
 * matcher arguments.
 *
 * In the case of a simple boolean matcher, you just need to write:
 *
 *   let (>>=) m1 m2 = fun tin ->
 *    match m1 tin with
 *    | None -> None
 *    | Some x ->
 *        m2 x tin
 *
 * For more context, this tutorial on monads in OCaml can be useful:
 * https://www.cs.cornell.edu/courses/cs3110/2019sp/textbook/ads/ex_maybe_monad.html
```

```
 *)
```

⟨*function* Matching_generic.monadic_bind 66h⟩

⟨*function* Matching_generic.monadic_or 67d⟩

⟨*function* Matching_generic.monadic_if_fail 67f⟩

```
let if_config f ~then_ ~else_ tin =
  if f tin.config then then_ tin else else_ tin
```

⟨*function* Matching_generic.return 67a⟩

⟨*function* Matching_generic.fail 67b⟩

```
let or_list m a bs =
  let rec aux xs = match xs with [] -> fail | b :: bs -> m a b >||> aux bs in
  aux bs

(* Since OCaml 4.08 you can define your own let operators!
 * alt: use ppx_let, but you need to write it as let%bind (uglier)
 * You can use the ppx future_syntax to support older version of OCaml, but
 * then you can not use other PPX rewriters (which we do).
 *)
let ( let* ) o f = o >>= f

(*****************************************************************************)
(* Environment *)
(*****************************************************************************)

let add_mv_capture key value (env : tin) =
  { env with mv = Env.add_capture key value env.mv }

let get_mv_capture key (env : tin) = Env.get_capture key env.mv

let extend_stmts_match_span rightmost_stmt (env : tin) =
  let stmts_match_span =
    Stmts_match_span.extend rightmost_stmt env.stmts_match_span
  in
  { env with stmts_match_span }
```

⟨*function* Matching_generic.equal_ast_binded_code 80b⟩

⟨*function* Matching_generic.check_and_add_metavar_binding 80a⟩

⟨*function* Matching_generic.envf 79d⟩

⟨*function* Matching_generic.empty_environment 63d⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*function* Matching_generic.has_ellipsis_stmts 92b⟩

```
let rec inits_and_rest_of_list = function
  | [] -> failwith "inits_1 requires a non-empty list"
  | [ e ] -> [ ([ e ], []) ]
  | e :: l ->
      ([ e ], l)
```

```
      :: List.map (fun (l, rest) -> (e :: l, rest)) (inits_and_rest_of_list l)

let _ =
  Common2.example
    ( inits_and_rest_of_list [ 'a'; 'b'; 'c' ]
    = [
        ([ 'a' ], [ 'b'; 'c' ]); ([ 'a'; 'b' ], [ 'c' ]); ([ 'a'; 'b'; 'c' ], []);
      ] )
```

⟨*function* Matching_generic.all_elem_and_rest_of_list 87a⟩

⟨*toplevel* Matching_generic._1 87b⟩

```
(* Since all_elem_and_rest_of_list computes the rest of list lazily,
 * we want to still keep track of how much time we're spending on
  * computing the rest of the list *)
let lazy_rest_of_list v =
  Common.profile_code "Matching_generic.eval_rest_of_list" (fun () ->
      Lazy.force v)
```

⟨*function* Matching_generic.return_bis 66f⟩
⟨*function* Matching_generic.fail_bis 66g⟩

⟨*constant* Matching_generic.regexp_regexp_string 88c⟩
⟨*function* Matching_generic.is_regexp_string 88e⟩

```
(* TODO: move in Core/Regexp_engine.ml! *)
type regexp = Re.re (* old: Str.regexp *)
```

⟨*function* Matching_generic.regexp_of_regexp_string 88g⟩

```
(*****************************************************************************)
(* Generic matchers *)
(*****************************************************************************)

(* ---------------------------------------------------------------------- *)
(* stdlib: option *)
(* ---------------------------------------------------------------------- *)
```

⟨*function* Matching_generic.m_option 68e⟩

⟨*function* Matching_generic.m_option_ellipsis_ok 85a⟩

⟨*function* Matching_generic.m_option_none_can_match_some 94e⟩

```
(* ---------------------------------------------------------------------- *)
(* stdlib: ref *)
(* ---------------------------------------------------------------------- *)
```

⟨*function* Matching_generic.m_ref 68g⟩

```
(* ---------------------------------------------------------------------- *)
(* stdlib: list *)
(* ---------------------------------------------------------------------- *)
```

⟨*function* Matching_generic.m_list 68i⟩

⟨*function* Matching_generic.m_list_prefix 96b⟩

⟨*function* Matching_generic.m_list_with_dots 82c⟩

```
(* todo? opti? try to go faster to the one with split_when?
```

```
 * need reflect tin so we can call the matcher and query whether there
 * was a match. Maybe a <??> monadic operator?
 *)
let rec m_list_in_any_order ~less_is_ok f xsa xsb =
  match (xsa, xsb) with
  | [], [] -> return ()
  (* less-is-ok: empty list can sometimes match non-empty list *)
  | [], _ :: _ -> if less_is_ok then return () else fail ()
  | a :: xsa, xsb ->
      let candidates = all_elem_and_rest_of_list xsb in
      (* less: could use a fold *)
      let rec aux xs =
        match xs with
        | [] -> fail ()
        | (b, xsb) :: xs ->
            f a b
            >>= (fun () ->
                  m_list_in_any_order ~less_is_ok f xsa (lazy_rest_of_list xsb))
            >||> aux xs
      in
      aux candidates

(* ---------------------------------------------------------------------- *)
(* stdlib: bool/int/string/... *)
(* ---------------------------------------------------------------------- *)

let m_eq a b = if a = b then return () else fail ()

⟨function Matching_generic.m_bool 68a⟩

⟨function Matching_generic.m_int 68b⟩

⟨function Matching_generic.m_string 68c⟩

⟨function Matching_generic.string_is_prefix 96d⟩

⟨function Matching_generic.m_string_prefix 96f⟩

let m_string_ellipsis_or_regexp_or_default ?(m_string_for_default = m_string) a
    b =
  match a with
  (* dots: '...' on string *)
  | "..." -> return ()
  | _ when Pattern.is_regexp_string a ->
      let f = regexp_matcher_of_regexp_string a in
      if f b then return () else fail ()
  | _ -> m_string_for_default a b

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

⟨function Matching_generic.m_info 69d⟩

⟨function Matching_generic.m_tok 69b⟩

⟨function Matching_generic.m_wrap 69f⟩

⟨function Matching_generic.m_bracket 69h⟩
```

```
  let m_tuple3 m_a m_b m_c (a1, b1, c1) (a2, b2, c2) =
    (m_a a1 a2 >>= fun () -> m_b b1 b2) >>= fun () -> m_c c1 c2

  (* ---------------------------------------------------------------------- *)
  (* Misc *)
  (* ---------------------------------------------------------------------- *)
```

⟨*function* Matching_generic.m_other_xxx 227b⟩

semgrep/matching/Apply_equivalences.mli

⟨semgrep/matching/Apply_equivalences.mli 447a⟩≡

⟨*signature* Apply_equivalences.apply 102a⟩

semgrep/matching/Apply_equivalences.ml

⟨semgrep/matching/Apply_equivalences.ml 447b⟩≡

```
  ⟨pad/r2c copyright 11⟩
  open Common
  open AST_generic
  module H = AST_generic_helpers
  module Flag = Flag_semgrep
  module MV = Metavariable
  module M = Map_AST
  module Eq = Equivalence
  module Env = Metavariable_capture

  (*****************************************************************************)
  (* Matchers for code equivalence mode *)
  (*****************************************************************************)
```

⟨*function* Apply_equivalences.match_e_e_for_equivalences 104b⟩

```
  (*****************************************************************************)
  (* Substituters *)
  (*****************************************************************************)
```

⟨*function* Apply_equivalences.subst_e 104c⟩

⟨*function* Apply_equivalences.apply 103e⟩

semgrep/matching/Semgrep_generic.mli

⟨semgrep/matching/Semgrep_generic.mli 447c⟩≡

⟨*signature* Semgrep_generic.check 37a⟩

```
  val last_matched_rule : Mini_rule.t option ref
```

⟨*type* Semgrep_generic.matcher 63a⟩

```
  (* used by tainting *)
```

⟨*signature* Semgrep_generic.match_e_e 62b⟩

⟨*signature* Semgrep_generic.match_any_any 63e⟩

semgrep/matching/Semgrep_generic.ml

⟨semgrep/matching/Semgrep_generic.ml 448⟩≡

```
(* Yoann Padioleau
 *
 * Copyright (C) 2011 Facebook
 * Copyright (C) 2019, 2020 r2c
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * version 2.1 as published by the Free Software Foundation, with the
 * special exception on linking described in file license.txt.
 *
 * This library 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.
 *)
open AST_generic
module V = Visitor_AST
module AST = AST_generic
module Err = Error_code
module PI = Parse_info
module R = Mini_rule (* TODO: rename to MR *)

module Eq = Equivalence
module PM = Pattern_match
module GG = Generic_vs_generic
module MV = Metavariable
module Flag = Flag_semgrep
module MG = Matching_generic

let logger = Logging.get_logger [ __MODULE__ ]

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Main matching engine behind sgrep. This module implements mainly
 * the expr/stmt visitor, while generic_vs_generic does the matching.
 *
 * history: this file was split in sgrep_generic.ml for -e/-f and
 * sgrep_lint_generic.ml for -rules_file. The -e/-f returns results as
 * it goes and takes a single pattern while -rules_file applies a list
 * of patterns and return a result just at the end. We have now factorized
 * the two files because of many bugs and discrepancies between the
 * two operating modes. It was easy to forget to add a new feature in
 * one of the file. Now -rules_file and -e/-f work mostly the same.
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

⟨type Semgrep_generic.matcher 63a⟩

(*****************************************************************************)
(* Debugging *)
(*****************************************************************************)

(* This is used to let the user know which rule the engine was using when
```

```
 * a Timeout or OutOfMemory exn occured.
 *)
let (last_matched_rule : Mini_rule.t option ref) = ref None

let set_last_matched_rule rule f =
  last_matched_rule := Some rule;
  (* note that if this raise an exn, last_matched_rule will not be
   * reset to None and that's what we want!
   *)
  let res = f () in
  last_matched_rule := None;
  res

(*****************************************************************************)
(* Matchers *)
(*****************************************************************************)
```

⟨*function* Semgrep_generic.match_e_e 63b⟩

⟨*function* Semgrep_generic.match_st_st 64a⟩

⟨*function* Semgrep_generic.match_sts_sts 65a⟩

⟨*function* Semgrep_generic.match_any_any 63f⟩

```
let match_t_t rule a b env =
  Common.profile_code ("rule:" ^ rule.R.id) (fun () ->
      set_last_matched_rule rule (fun () -> GG.m_type_ a b env))
  [@@profiling]

let match_p_p rule a b env =
  Common.profile_code ("rule:" ^ rule.R.id) (fun () ->
      set_last_matched_rule rule (fun () -> GG.m_pattern a b env))
  [@@profiling]

let match_partial_partial rule a b env =
  Common.profile_code ("rule:" ^ rule.R.id) (fun () ->
      set_last_matched_rule rule (fun () -> GG.m_partial a b env))
  [@@profiling]

let match_at_at rule a b env =
  Common.profile_code ("rule:" ^ rule.R.id) (fun () ->
      set_last_matched_rule rule (fun () -> GG.m_attribute a b env))
  [@@profiling]

let match_fld_fld rule a b env =
  Common.profile_code ("rule:" ^ rule.R.id) (fun () ->
      set_last_matched_rule rule (fun () -> GG.m_field a b env))
  [@@profiling]

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

let (rule_id_of_mini_rule : Mini_rule.t -> Pattern_match.rule_id) =
 fun mr ->
  {
    PM.id = mr.Mini_rule.id;
    message = mr.Mini_rule.message;
    pattern_string = mr.Mini_rule.pattern_string;
```

```
  }

let match_rules_and_recurse config (file, hook, matches) rules matcher k any x =
  rules
  |> List.iter (fun (pattern, rule, cache) ->
         let env = MG.empty_environment cache config in
         let matches_with_env = matcher rule pattern x env in
         if matches_with_env <> [] then
           (* Found a match *)
           matches_with_env
           |> List.iter (fun (env : MG.tin) ->
                  let env = env.mv.full_env in
                  let range_loc = V.range_of_any (any x) in
                  let tokens = lazy (V.ii_of_any (any x)) in
                  let rule_id = rule_id_of_mini_rule rule in
                  Common.push
                    { PM.rule_id; file; env; range_loc; tokens }
                    matches;
                  hook env tokens));
  (* try the rules on substatements and subexpressions *)
  k x

let must_analyze_statement_bloom_opti_failed pattern_strs
    (st : AST_generic.stmt) =
  (* if it's empty, meaning we were not able to extract any useful specific
   * identifiers or strings from the pattern, then the pattern is too general
   * and we must analyze the stmt
   *)
  match st.s_bf with
  (* No bloom filter, expected if -bloom_filter is not used *)
  | None -> true
  (* only when the Bloom_filter says No we can skip the stmt *)
  | Some bf -> Bloom_filter.is_subset pattern_strs bf = Bloom_filter.Maybe

(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)

⟨function Semgrep_generic.check2 60⟩

(* TODO: cant use [@@profile] because it does not handle yet label params *)
let check ~hook config a b c =
  Common.profile_code "Semgrep_generic.check" (fun () ->
      check2 ~hook config a b c)
```

semgrep/matching/SubAST_generic.mli

⟨signature SubAST_generic.subexprs_of_expr 450a⟩≡ (450c)

```
val subexprs_of_expr : AST_generic.expr -> AST_generic.expr list
```

⟨signature SubAST_generic.flatten_substmts_of_stmts 450b⟩≡ (450c)

```
val flatten_substmts_of_stmts :
  AST_generic.stmt list -> (AST_generic.stmt list * AST_generic.stmt) option
```

⟨semgrep/matching/SubAST_generic.mli 450c⟩≡

⟨*signature* SubAST_generic.subexprs_of_expr 450a⟩

⟨*signature* SubAST_generic.flatten_substmts_of_stmts 450b⟩

semgrep/matching/SubAST_generic.ml

⟨*function* SubAST_generic.do_visit_with_ref 451a⟩≡ (451c)
```
(* TODO: move in pfff at some point *)
let do_visit_with_ref mk_hooks any =
  let res = ref [] in
  let hooks = mk_hooks res in
  let vout = V.mk_visitor hooks in
  vout any;
  List.rev !res
```

⟨*function* SubAST_generic.lambdas_in_expr 451b⟩≡ (451c)
```
let lambdas_in_expr e =
  do_visit_with_ref
    (fun aref ->
      {
        V.default_visitor with
        V.kexpr =
          (fun (k, _) e ->
            match e with Lambda def -> Common.push def aref | _ -> k e);
      })
    (E e)
  [@@profiling]
```

⟨semgrep/matching/SubAST_generic.ml 451c⟩≡
⟨*pad/r2c copyright* 11⟩
```
open AST_generic
module V = Visitor_AST

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Various helper functions to extract subparts of AST elements.
 *
 *)

let go_really_deeper_stmt = ref true

(*****************************************************************************)
(* Sub-expressions and sub-statements *)
(*****************************************************************************)

let subexprs_of_any_list xs =
  xs |> List.fold_left (fun x -> function E e -> e :: x | _ -> x) []

(* used for really deep statement matching *)
let subexprs_of_stmt st =
  match st.s with
  (* 1 *)
  | ExprStmt (e, _)
  | If (_, e, _, _)
  | While (_, e, _)
```

```
  | DoWhile (_, _, e)
  | DefStmt (_, VarDef { vinit = Some e; _ })
  | DefStmt (_, FieldDefColon { vinit = Some e; _ })
  | For (_, ForEach (_, _, e), _)
  | Continue (_, LDynamic e, _)
  | Break (_, LDynamic e, _)
  | Throw (_, e, _) ->
      [ e ]
  (* opt *)
  | Switch (_, eopt, _) | Return (_, eopt, _) | OtherStmtWithStmt (_, eopt, _)
    ->
      Common.opt_to_list eopt
  (* n *)
  | For (_, ForClassic (xs, eopt1, eopt2), _) ->
      ( xs
      |> Common.map_filter (function
           | ForInitExpr e -> Some e
           | ForInitVar (_, vdef) -> vdef.vinit) )
      @ Common.opt_to_list eopt1 @ Common.opt_to_list eopt2
  | Assert (_, e1, e2opt, _) -> e1 :: Common.opt_to_list e2opt
  | For (_, ForIn (_, es), _) -> es
  | OtherStmt (_op, xs) -> subexprs_of_any_list xs
  (* 0 *)
  | DirectiveStmt _ | Block _
  | For (_, ForEllipsis _, _)
  | Continue _ | Break _ | Label _ | Goto _ | Try _ | DisjStmt _ | DefStmt _
  | WithUsingResource _ ->
      []
```

⟨*function* SubAST_generic.subexprs_of_expr 89e⟩

⟨*function* SubAST_generic.subexprs_of_stmt 92f⟩

⟨*function* SubAST_generic.substmts_of_stmt 92g⟩

```
(*****************************************************************************)
(* Visitors  *)
(*****************************************************************************)
```

⟨*function* SubAST_generic.do_visit_with_ref 451a⟩

⟨*function* SubAST_generic.lambdas_in_expr 451b⟩

```
(* opti: using memoization speed things up a bit too
 * (but again, this is still slow when called many many times).
 * todo? note that this is not the optimal memoization we can do because
 * using Hashtbl where the key is a full expression can be slow (hashing
 * huge expressions still takes some time). It would be better to
 * return a unique identifier to each expression to remove the hashing cost.
 *)
let hmemo = Hashtbl.create 101

let lambdas_in_expr_memo a =
  Common.memoized hmemo a (fun () -> lambdas_in_expr a)
  [@@profiling]

(*****************************************************************************)
(* Really substmts_of_stmts *)
(*****************************************************************************)
```

⟨*function* SubAST_generic.flatten_substmts_of_stmts 93⟩

semgrep/matching/Normalize_generic.mli

⟨semgrep/matching/Normalize_generic.mli 453a⟩≡

```
  ⟨signature Normalize_generic.normalize_import_opt 98a⟩

  ⟨signature Normalize_generic.constant_propagation_and_evaluate_literal 108a⟩
```

semgrep/matching/Normalize_generic.ml

⟨semgrep/matching/Normalize_generic.ml 453b⟩≡

```
  ⟨pad/r2c copyright 11⟩
  open Common (* >>= *)

  open AST_generic

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Various helper functions to normalize AST elements.
   *
   * TODO: merge with pfff/.../normalize_ast.ml at some point
   *
   *)

  (*****************************************************************************)
  (* Entry points *)
  (*****************************************************************************)

  ⟨function Normalize_generic.full_module_name 98b⟩

  ⟨function Normalize_generic.normalize_import_opt 98c⟩

  ⟨function Normalize_generic.eval 108b⟩

  ⟨constant Normalize_generic.constant_propagation_and_evaluate_literal 108c⟩
```

F.19 semgrep/reporting/

semgrep/reporting/JSON_report.mli

⟨signature JSON_report.match_to_error 453c⟩≡ (453d)

```
  (* internal *)
  val match_to_error : Pattern_match.t -> unit
```

⟨semgrep/reporting/JSON_report.mli 453d⟩≡

```
  ⟨signature JSON_report.match_to_json 113b⟩

  (* takes the starting time of the program *)
  val json_of_profile_info : float -> JSON.t
```

```
  val json_of_exn : exn -> JSON.t

  val json_fields_of_matches_and_errors :
    Common.filename list -> Report.rule_result -> (string * JSON.t) list
```

⟨*signature* JSON_report.match_to_error 453c⟩

semgrep/reporting/JSON_report.ml

⟨*function* JSON_report.string_of_resolved 454a⟩≡ (454d)

```
  let string_of_resolved = function
    | Global -> "Global"
    | Local -> "Local"
    | Param -> "Param"
    | EnclosedVar -> "EnclosedVar"
    | ImportedEntity _ -> "ImportedEntity"
    | ImportedModule _ -> "ImportedModule"
    | TypeName -> "TypeName"
    | Macro -> "Macro"
    | EnumConstant -> "EnumConstant"
```

⟨*function* JSON_report.error 454b⟩≡ (454d)

```
  (* this is used only in the testing code, to reuse the
   * Error_code.compare_actual_to_expected
   *)
  let error loc (rule : Pattern_match.rule_id) =
    E.error_loc loc (E.SemgrepMatchFound (rule.id, rule.message))
```

⟨*function* JSON_report.match_to_error 454c⟩≡ (454d)

```
  let match_to_error x =
    let min_loc, _max_loc = x.range_loc in
    error min_loc x.rule_id
```

⟨semgrep/reporting/JSON_report.ml 454d⟩≡

```
  ⟨pad/r2c copyright 11⟩
  open Common
  open AST_generic
  module V = Visitor_AST
  module PI = Parse_info
  module R = Mini_rule (* TODO: use MR instead *)

  module E = Error_code
  module J = JSON
  module MV = Metavariable
  module RP = Report
  open Pattern_match

  (*****************************************************************************)
  (* Unique ID *)
  (*****************************************************************************)
  ⟨function JSON_report.string_of_resolved 454a⟩

  ⟨function JSON_report.unique_id 114c⟩

  (*****************************************************************************)
```

```
(* JSON *)
(****************************************************************************)

⟨function JSON_report.json_range 114a⟩

⟨function JSON_report.range_of_any 113e⟩

⟨function JSON_report.json_metavar 114b⟩

⟨function JSON_report.match_to_json 113c⟩

let json_time_of_profiling_data profiling_data =
  [
    ( "time",
      J.Object
        [
          ( "targets",
            J.Array
              (List.map
                 (fun { RP.file = target; parse_time; match_time; run_time } ->
                   J.Object
                     [
                       ("path", J.String target);
                       ("parse_time", J.Float parse_time);
                       ("match_time", J.Float match_time);
                       ("run_time", J.Float run_time);
                     ])
                 profiling_data.RP.file_times) );
          ("rule_parse_time", J.Float profiling_data.RP.rule_parse_time);
        ] );
  ]

let json_fields_of_matches_and_errors files res =
  let matches, new_errs =
    Common.partition_either match_to_json res.RP.matches
  in
  let errs = new_errs @ res.RP.errors in
  let count_errors = List.length errs in
  let count_ok = List.length files - count_errors in
  let time_field =
    match res.RP.rule_profiling with
    | None -> []
    | Some x -> json_time_of_profiling_data x
  in
  [
    ("matches", J.Array matches);
    ("errors", J.Array (errs |> List.map R2c.error_to_json));
    ( "stats",
      J.Object
        [ ("okfiles", J.Int count_ok); ("errorfiles", J.Int count_errors) ] );
  ]
  @ time_field
  [@@profiling]

let json_of_profile_info profile_start =
  let now = Unix.gettimeofday () in
  (* total time, but excluding J.string_of_json time that comes after *)
  (* partial copy paste of Common.adjust_profile_entry *)
  Hashtbl.add !Common._profile_table "TOTAL" (ref (now -. profile_start), ref 1);
```

```
    (* partial copy paste of Common.profile_diagnostic *)
    let xs =
      Hashtbl.fold (fun k v acc -> (k, v) :: acc) !Common._profile_table []
      |> List.sort (fun (_k1, (t1, _n1)) (_k2, (t2, _n2)) -> compare t2 t1)
    in
    xs
    |> List.map (fun (k, (t, cnt)) ->
           (k, J.Object [ ("time", J.Float !t); ("count", J.Int !cnt) ]))
    |> fun xs -> J.Object xs

  let json_of_exn e =
    (* if (ouptut_as_json) then *)
    match e with
    | Parse_mini_rule.InvalidRuleException (pattern_id, msg) ->
        J.Object
          [
            ("pattern_id", J.String pattern_id);
            ("error", J.String "invalid rule");
            ("message", J.String msg);
          ]
    | Parse_mini_rule.InvalidLanguageException (pattern_id, language) ->
        J.Object
          [
            ("pattern_id", J.String pattern_id);
            ("error", J.String "invalid language");
            ("language", J.String language);
          ]
    | Parse_mini_rule.InvalidPatternException (pattern_id, pattern, lang, message)
      ->
        J.Object
          [
            ("pattern_id", J.String pattern_id);
            ("error", J.String "invalid pattern");
            ("pattern", J.String pattern);
            ("language", J.String lang);
            ("message", J.String message);
          ]
    | Parse_mini_rule.UnparsableYamlException msg ->
        J.Object
          [ ("error", J.String "unparsable yaml"); ("message", J.String msg) ]
    | Parse_mini_rule.InvalidYamlException msg ->
        J.Object [ ("error", J.String "invalid yaml"); ("message", J.String msg) ]
    | exn ->
        J.Object
          [
            ("error", J.String "unknown exception");
            ("message", J.String (Common.exn_to_s exn));
          ]

  (*****************************************************************************)
  (* Error *)
  (*****************************************************************************)
  ⟨function JSON_report.error 454b⟩

  ⟨function JSON_report.match_to_error 454c⟩
```

semgrep/reporting/Matching_report.mli

⟨semgrep/reporting/Matching_report.mli 456⟩≡

⟨*type* Matching_report.match_format 111c⟩

⟨*signature* Matching_report.print_match 112a⟩

⟨*signature* Matching_report.join_with_space_if_needed 112c⟩

semgrep/reporting/Matching_report.ml

⟨semgrep/reporting/Matching_report.ml 457a⟩≡
```
(* Yoann Padioleau
 *
 * Copyright (C) 2013 Facebook
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * version 2.1 as published by the Free Software Foundation, with the
 * special exception on linking described in file license.txt.
 *
 * This library 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.
 *)
open Common
module PI = Parse_info

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

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

⟨type Matching_report.match_format 111c⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Matching_report.join_with_space_if_needed 112d⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨function Matching_report.print_match 112b⟩
```

F.20 semgrep/tainting/

semgrep/tainting/Tainting_generic.mli

⟨semgrep/tainting/Tainting_generic.mli 457b⟩≡
```
⟨signature Tainting_generic.check 193d⟩
```

semgrep/tainting/Tainting_generic.ml

⟨semgrep/tainting/Tainting_generic.ml 458⟩≡
```
  ⟨pad/r2c copyright 11⟩
  module AST = AST_generic
  module V = Visitor_AST
  module R = Tainting_rule
  module R2 = Mini_rule
  module Flag = Flag_semgrep
  module PM = Pattern_match

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Simple wrapper around the tainting dataflow-based analysis in pfff.
   *
   * Here we pass matcher functions that uses semgrep patterns to
   * describe the source/sink/sanitizers.
   *)
  let _logger = Logging.get_logger [ __MODULE__ ]

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)

  module F2 = IL

  module DataflowY = Dataflow.Make (struct
    type node = F2.node

    type edge = F2.edge

    type flow = (node, edge) Ograph_extended.ograph_mutable

    let short_string_of_node n = Display_IL.short_string_of_node_kind n.F2.n
  end)

  let match_pat_eorig pat =
    match pat with
    | [] -> fun _ -> false
    | xs ->
        let xs =
          xs
          |> List.map (function
               | AST.E e -> e
               | _ ->
                   failwith "Only Expr patterns are supported in tainting rules")
        in
        let pat = Common2.foldl1 (fun x acc -> AST.DisjExpr (x, acc)) xs in
        fun eorig ->
          (* the rule is just used by match_e_e for profiling stats *)
          let rule =
            {
              R2.id = "<tainting>";
              pattern = AST.E pat;
              pattern_string = "<tainting> pat";
              message = "";
              severity = R2.Error;
              languages = [];
            }
```

```
        in

        let env =
          Matching_generic.empty_environment None Config_semgrep.default_config
        in
        let matches_with_env = Semgrep_generic.match_e_e rule pat eorig env in
        matches_with_env <> []

  let match_pat_exp pat exp =
    let eorig = exp.IL.eorig in
    match_pat_eorig pat eorig

  ⟨function Tainting_generic.match_pat_instr 196a⟩

  ⟨function Tainting_generic.config_of_rule 196b⟩

  (*****************************************************************************)
  (* Main entry point *)
  (*****************************************************************************)

  ⟨function Tainting_generic.check2 197c⟩
```

F.21 semgrep/typing/

semgrep/typing/Typechecking_generic.mli

⟨semgrep/typing/Typechecking_generic.mli 459a⟩≡

```
  ⟨signature Typechecking_generic.compatible_type 107c⟩
```

semgrep/typing/Typechecking_generic.ml

⟨semgrep/typing/Typechecking_generic.ml 459b⟩≡

```
  ⟨pad/r2c copyright 11⟩
  open AST_generic

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Poor's man typechecker on literals (for now).
   *
   * todo:
   *  - local type inference on AST generic? good coverage?
   *  - we could allow metavars on the type itself, as in
   *    foo($X: $T) ... $T x; ...
   *    which would require to transform the code in the generic_vs_generic
   *    style as typechecking could also bind metavariables in the process
   *)

  (*****************************************************************************)
  (* Entry point *)
  (*****************************************************************************)

  ⟨function Typechecking_generic.compatible_type 107d⟩
```

F.22 semgrep/tests/

semgrep/tests/Test.ml

⟨*constant* Test.verbose 460a⟩≡ (460h)
```
let verbose = ref false
```

⟨*constant* Test.dump_ast 460b⟩≡ (460h)

⟨*function* Test.any_gen_of_string 460c⟩≡ (460h)
```
let any_gen_of_string str =
  Common.save_excursion Flag_parsing.sgrep_mode true (fun () ->
  let any = Parse_python.any_of_string str in
  Python_to_generic.any any
  )
```

⟨*function* Test.parse_generic 460d⟩≡ (460h)

⟨*function* Test.ast_generic_of_file 460e⟩≡ (460h)

⟨*function* Test.dump_ast_generic 460f⟩≡ (460h)

⟨*constant* Test.options 460g⟩≡ (460h)
```
let options = [
  "-verbose", Arg.Set verbose,
  " verbose mode";
  ]
```

⟨semgrep/tests/Test.ml 460h⟩≡
```
open Common
open OUnit
module E = Error_code
module P = Pattern_match
module R = Mini_rule
module T = Tainting_rule

(*****************************************************************************)
(* Purpose *)
(*****************************************************************************)
(* Unit tests runner (and a few dumpers) *)

(*****************************************************************************)
(* Flags *)
(*****************************************************************************)
```
⟨*constant* Test.verbose 460a⟩

⟨*constant* Test.dump_ast 460b⟩

⟨*constant* Test.tests_path 215a⟩
⟨*constant* Test.data_path 215b⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*function* Test.ast_fuzzy_of_string 215c⟩

⟨*function* Test.any_gen_of_string 460c⟩

⟨*function* Test.parse_generic 460d⟩

```
let parsing_tests_for_lang files lang =
  files |> List.map (fun file ->
    (Filename.basename file) >:: (fun () ->
      let {Parse_target. errors = errs; _ } =
         Parse_target.parse_and_resolve_name_use_pfff_or_treesitter lang file in
      if errs <> []
      then failwith (String.concat ";" (List.map Error_code.string_of_error errs));
    )
  )

(*
   For each input file with the language's extension, locate a pattern file
   with the '.sgrep' extension.

   If foo/bar.sgrep is not found, POLYGLOT/bar.sgrep is used instead.
*)
⟨function Test.regression_tests_for_lang 215d⟩

let tainting_test lang rules_file file =
  let rules =
    try
      Parse_tainting_rules.parse rules_file
    with exn ->
      failwith (spf "fail to parse tainting rules %s (exn = %s)"
                rules_file
                (Common.exn_to_s exn))
  in
  let ast =
      try
        let { Parse_target. ast; errors; _ } =
          Parse_target.parse_and_resolve_name_use_pfff_or_treesitter lang file
        in
        if errors <> []
        then pr2 (spf "WARNING: fail to fully parse %s" file);
        ast
      with exn ->
        failwith (spf "fail to parse %s (exn = %s)" file
                  (Common.exn_to_s exn))
  in
  let rules =
    rules |> List.filter (fun r -> List.mem lang r.T.languages) in
  let matches = Tainting_generic.check rules file ast in
  let actual =
    matches |> List.map (fun m ->
      { E.typ = SemgrepMatchFound(m.P.rule_id.id,m.P.rule_id.message);
        loc   = fst m.range_loc;
        sev   = Error; }
      )
  in
  let expected = Error_code.expected_error_lines_of_files [file] in
  Error_code.compare_actual_to_expected actual expected

let tainting_tests_for_lang files lang =
  files |> List.map (fun file ->
   (Filename.basename file) >:: (fun () ->
    let rules_file =
      let (d,b,_e) = Common2.dbe_of_filename file in
      let candidate1 = Common2.filename_of_dbe (d,b,"yaml") in
      if Sys.file_exists candidate1
```

```
      then candidate1
      else failwith (spf "could not find tainting rules file for %s" file)
    in
    tainting_test lang rules_file file
  ))

(*****************************************************************************)
(* Tests *)
(*****************************************************************************)

(* This differs from pfff/tests/<lang>/parsing because here we also use
 * tree-sitter to parse; certain files do not parse with pfff but parses here
 *)
let lang_parsing_tests =
  "lang parsing testing" >::: [
   (* languages with only a tree-sitter parser *)
    "C#" >::: (
      let dir = Filename.concat (Filename.concat tests_path "csharp") "parsing" in
      let files = Common2.glob (spf "%s/*.cs" dir) in
      let lang = Lang.Csharp in
      parsing_tests_for_lang files lang
    );
    "Lua" >::: (
      let dir = Filename.concat (Filename.concat tests_path "lua") "parsing" in
      let files = Common2.glob (spf "%s/*.lua" dir) in
      let lang = Lang.Lua in
      parsing_tests_for_lang files lang
    );
    "Rust" >::: (
      let dir = Filename.concat (Filename.concat tests_path "rust") "parsing" in
      let files = Common2.glob (spf "%s/*.rs" dir) in
      let lang = Lang.Rust in
      parsing_tests_for_lang files lang
    );
    "Kotlin" >::: (
      let dir = Filename.concat (Filename.concat tests_path "kotlin") "parsing" in
      let files = Common2.glob (spf "%s/*.kt" dir) in
      let lang = Lang.Kotlin in
      parsing_tests_for_lang files lang
    );
    (* here we have both a Pfff and tree-sitter parser *)
    "Java" >::: (
      let dir= Filename.concat (Filename.concat tests_path "java") "parsing" in
      let files = Common2.glob (spf "%s/*.java" dir) in
      let lang = Lang.Java in
      parsing_tests_for_lang files lang
    );
    "Go" >::: (
      let dir= Filename.concat (Filename.concat tests_path "go") "parsing" in
      let files = Common2.glob (spf "%s/*.go" dir) in
      let lang = Lang.Go in
      parsing_tests_for_lang files lang
    );
    "Ruby" >::: (
      let dir = Filename.concat tests_path "ruby/parsing" in
      let files = Common2.glob (spf "%s/*.rb" dir) in
      let lang = Lang.Ruby in
      parsing_tests_for_lang files lang
    );
    "Javascript" >::: (
```

```
      let dir = Filename.concat tests_path "js/parsing" in
      let files = Common2.glob (spf "%s/*.js" dir) in
      let lang = Lang.Javascript in
      parsing_tests_for_lang files lang
    );
  ]
```

⟨*constant* `Test.lang_regression_tests` 216⟩

```
let full_rule_regression_tests =
  "full rule" >:: (fun () ->
      let path = Filename.concat tests_path "OTHER/rules" in
      Test_engine.test_rules ~ounit_context:true [path]
  )

let lang_tainting_tests =
  let taint_tests_path = Filename.concat tests_path "tainting_rules" in
  "lang tainting rules testing" >::: [
    "tainting Python" >::: (
      let dir = Filename.concat taint_tests_path "python" in
      let files = Common2.glob (spf "%s/*.py" dir) in
      let lang = Lang.Python in
      tainting_tests_for_lang files lang
    );
    "tainting Typescript" >::: (
      let dir = Filename.concat taint_tests_path "ts" in
      let files = Common2.glob (spf "%s/*.ts" dir) in
      let lang = Lang.Typescript in
      tainting_tests_for_lang files lang
    );
  ]
```

⟨*constant* `Test.lint_regression_tests` 218a⟩

```
let eval_regression_tests =
  "eval regression resting" >:: (fun () ->
      let dir = Filename.concat tests_path "OTHER/eval" in
      let files = Common2.glob (spf "%s/*.json" dir) in
      files |> List.iter (fun file ->
        let (env, code) = Eval_generic.parse_json file in
        let res = Eval_generic.eval env code in
        OUnit.assert_equal ~msg:(spf "%s should evaluate to true" file)
          (Eval_generic.Bool true) res
      )
  )

(*****************************************************************************)
(* Main action *)
(*****************************************************************************)
```

⟨*function* `Test.test` 219c⟩

```
(*****************************************************************************)
(* Extra actions *)
(*****************************************************************************)
```
⟨*function* `Test.ast_generic_of_file` 460e⟩
⟨*function* `Test.dump_ast_generic` 460f⟩

```
(*****************************************************************************)
(* The options *)
```

```
(*****************************************************************************)
```

⟨*constant* `Test.options` 460g⟩

```
(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)
```

⟨*constant* `Test.usage` 218b⟩

⟨*function* `Test.main` 219a⟩

⟨*toplevel* `Test._1` 219b⟩

Glossary

CST = Concrete Syntax Tree
AST = Abstract Syntax Tree
IL = Intermediate Language
CFG = Control Flow Graph
SAST = Static Application Security Testing

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[App04] Andrew Appel. *Modern Compiler in ML*. Cambridge University Press, 2004. cited page(s) 11

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. For an introduction see http://en.wikipedia.org/wiki/Literate_Program. cited page(s) 10, 11

[LDF+16] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 4.04): Documentation and user's manual*. November 2016. Available at http://caml.inria.fr/pub/docs/manual-ocaml/. cited page(s) 11

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 11

[Pad16] Yoann Padioleau. *Principia Softwarica, Fundamental Literate System Programs*. 2016. cited page(s) 10