Lessons from Writing a Compiler

The prototypical compilers textbook is:

1. 600 pages on parsing theory.
2. Three pages of type-checking a first-order type system like C.
3. Zero pages on storing and checking the correctness of declarations (the “symbol table”).
4. Zero pages on the compilation model, and efficiently implementing separate compilation.
5. 450 pages on optimization and code generation.

The standard academic literature is most useful for the extreme frontend (parsing) and the extreme backend (SSA, instruction selection and code generation), but the middle-end is ignored. This is fine if you want to learn how to build, e.g., the next LLVM: a fat backend with a very thin frontend.

But what if you’re building a compiler on top of LLVM, such that it’s *all* frontend and middle-end? Semantic analysis, type checking, and checking the rules of declarations are the most important parts of modern compilers because this is where all the important diagnostics (other than syntax errors) are made.

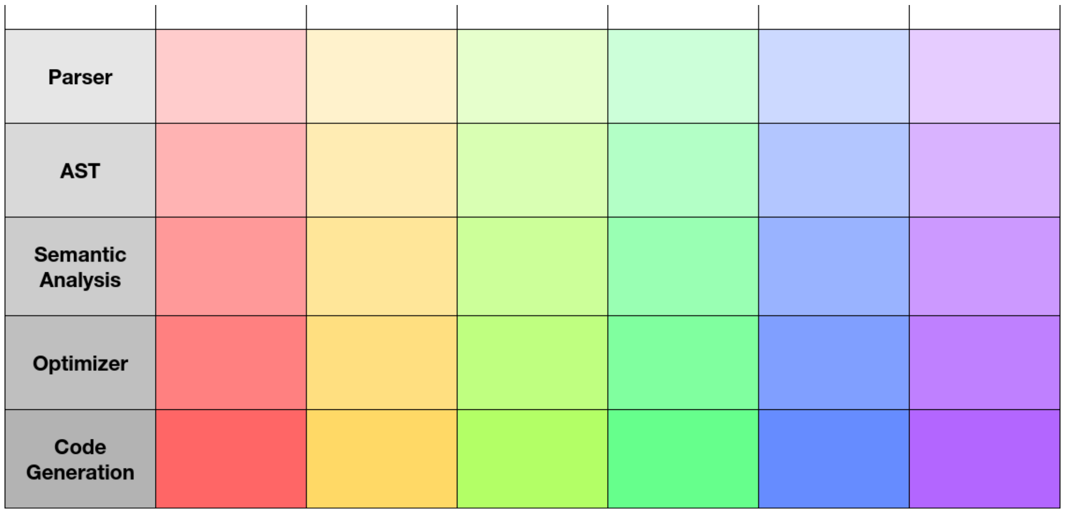
This article contains some of the lessons I learned writing the compiler for Austral, a new systems programming language with linear types that I've been working on for a while. The first few sections are high-level, the rest most specific to using OCaml to write a compiler.

Contents

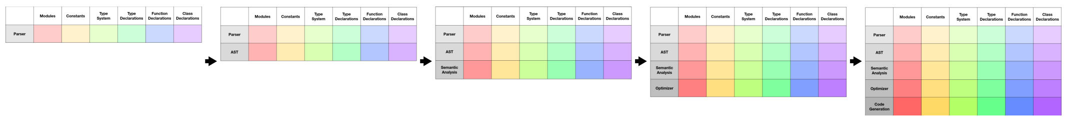
Implementation Strategies

Imagine a compiler as a 2D grid. The rows are the stages of compilation: from parsing, through the middleend, to the backend. The columns are the different facets of the programming language being implemented. For a smaller language like C, the different facets are: global variable declarations, enum declarations, type declarations, function declarations, and nothing else. More featureful languages add extra facets: interface and class definitions in C++, `trait` and `impl` definition in Rust, etc.

	Modules	Constants	Type System	Type Declarations	Function Declarations	Class Declarations



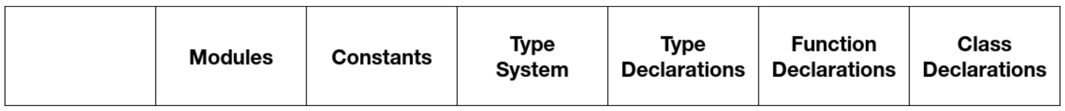
There are, broadly, two ways to implement a compiler. In the **waterfall** strategy, each stage is implemented one after the other: the complete parser, then the complete semantic analysis pass, then the complete type checking pass, and on to the backend stages.

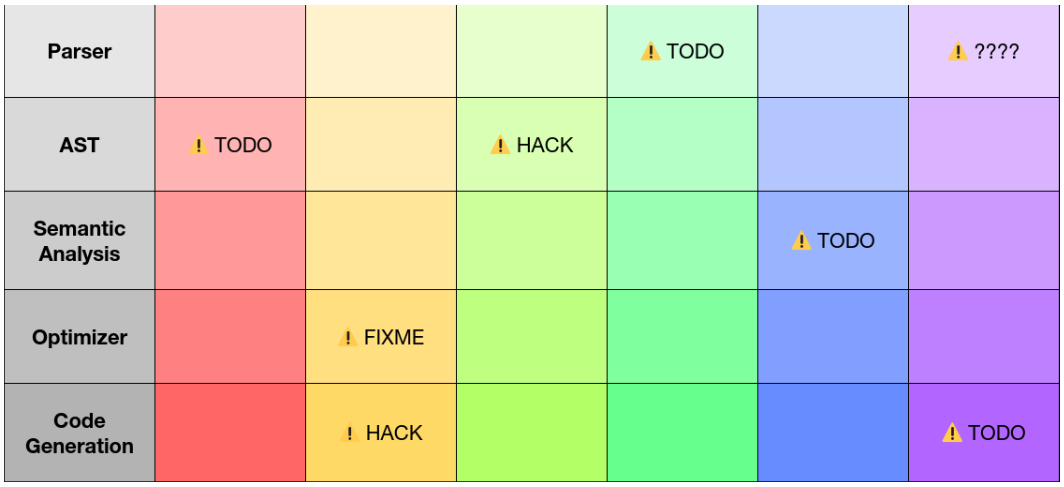


Tests can be be written after each stage is finished, and this might help catch bugs, since a shallower pipeline means fewer places to look for the source of the bug. It is difficult to identify where bugs happen in a compiler with many tens of intermediate representations, when you only have end-to-end tests.

The downside is getting to “Hello, world!” takes a very long time. When writing a compiler for an existing language, this matters less. But when building a new language, you want something you can play with at an early stage, so you can find what sticks out in the language design, what needs polishing, and what the language pragmatics are.

However, the reality of development pressures and expedience means the end result looks more like:

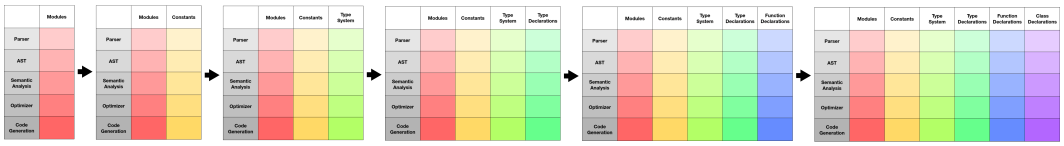




Parser				⚠ TODO		⚠ ????
AST	⚠ TODO		⚠ HACK			
Semantic Analysis					⚠ TODO	
Optimizer		⚠ FIXME				
Code Generation		⚠ HACK				⚠ TODO

With error diagnostics and semantic analysis checks being what is typically left out.

The alternative is the **verticals** strategy, where, at the beginning of development, you implement *all* stages of compilation, but they do nothing. The parser parses empty files, the ASTs types are empty or have only perfunctory placeholder cases, the backend emits nothing. And then you implement each language facet, one at a time.



The advantage of this is that from the start you have a compiler you can invoke and test. Having end-to-end tests from the start is a huge boon both in terms of ensuring the compiler implements the language definition *and* identifying regressions.

While in the waterfall strategy you can focus on ensuring each stage of compilation is correct, with the verticals strategy you can focus on ensuring each facet of the language being implemented is correct.

The problem is you're not implementing *the* language, but rather a succession of concentric subsets of the language, until you get to feature-completeness.

Austral’s compiler was written with a very unprincipled, waterfall-ish strategy: I went stage by stage, but omitting some harder-to-implement features, diagnostics, and checks, and then went back and filled those in. I have a sense that the strategy of proceeding by verticals is better, but I can’t endorse either strategy because I haven’t compared them.

Just Use OCaml

I’ve a whole graveyard of half-finished compiler projects and toy interpreters in my drive, in everything from C, C++, Common Lisp, Standard ML, Java, Haskell and OCaml.

OCaml sits at a comfortable point in language space for building compilers. It has enough tutorials and practical documentation to be productive. It has all the functional programming features you want: algebraic data types and pattern matching with exhaustiveness checking. But it’s not too strict, and you write imperative code if you want.

And OCaml is not terribly big-brained, so you don’t have to be put off. You can ignore much of the language and write spartan, workmanlike OCaml that solves the problem and doesn’t leave potential contributors scratching their head.

Compilers are unusual in that they rarely require large ecosystems, unlike most modern applications. A good parser generator is all you need, so using more obscure languages like OCaml does not involve paying as much of a cost, as, say, writing a web app in Haskell vs. Python. You won’t find yourself having to reinvent the entire ecosystem from scratch because few dependencies you need don’t exist.

Just Use Menhir

You need a parser. This is the most mind-numbing part of the compiler. Writing a parser by hand is tricky and not worth the alleged performance increases[1]. Parsing performance likely won’t be the bottleneck.

Parser combinators are very popular. It’s cute that you can parameterize grammars by making them into functions. But the problem with parser combinators is they implement parsing expression grammars. PEGs are popular too, but they’re too rigid.

In a PEG, the disjunction or choice operator (typically denoted by the pipe character `|`) is *ordered* choice. If you have a context-free grammar like `foo := a | b | c`, you can expect the parser generator to be smart about which case to take given input.

In PEGs – and, therefore, in parser combinators – choices are tried in the order in which they are listed. This leads to huge headaches.

Things like left recursion lead to difficult to debug infinite loops, you will have to contort your grammar to fit the rules of PEG. The end result does *not* look like the cute parser combinator one-liners that draw everyone in.

Good old fashioned parser generators like Menhir or ANTLR don’t have this problem. Choice is unordered, like in the EBNF notation you find in language specifications. Left recursion is implemented “intelligently”, usually without requiring you to transform your grammar.

Menhir is a mature parser generator. It is integrated into OCaml's build system: dune. The documentation is alright, but there's enough practical tutorials to get you started.

What Tests are Useful

There's, roughly, two kinds of tests:

1. Unit tests are written in the same language the compiler is written in. In my case, OCaml.

2. End-to-end tests. Here you have a test runner that treats the compiler as a black box: a binary file that is invoked with command line arguments. Tests can either check that compilation *fails* and produces an expected set of error messages, or that compilation *succeeds* and the generated program, when executed, produces expected output.

You should write end-to-end tests first. Yes, they are coarse-grained, and they may not help you figure out exactly *where* in the compiler a bug resides. But the benefit is that they treat the compiler as a black box, which lets you iterate the compiler's internals without causing tests to break. You can change the AST structure, intermediate representation, passes, environment representation, backend, everything, as long as the CLI interface remains the same. This is useful for early-stage development where things are changing rapidly.

Investing time in writing a good end-to-end test runner was a huge boon for the development of the Austral compiler. I wish I had done it earlier.

Having lots of very specific unit tests for every aspect of a pass is useful mainly when the compiler has reached a stage of maturity where its internals are fixed. If you are still iterating on the internals, those tests are a liability: they will fail to compile (due to the changes you made to the compiler's code) more often than they will catch bugs.

Caveat: Languages differ by how convenient they make writing tests, and what is convenient is what gets written. I find myself writing a lot more tests in Python, for example, than in OCaml. Partly because Python makes fewer guarantees at compile time, and partly because Python's dynamism makes things like test autodiscovery easier.

Compilation Model

The simplest compiler is a whole-program compiler: it processes an ordered set of files, one after the other, and spits out generated code for the whole thing. The module dependency DAG can be topologically sorted to a linear order.

For an MVP language, this is fine, because very little code is written in the language. The problem is scalability: whole-program compilation for large codebases is intrinsically slow.

Being able to typecheck, compile, and run code at a high frequency makes development less frustrating. Having to wait ten seconds to build hundreds of thousands of lines of code, *ab initio*, for every change that you make is frustrating. Performance requires separate compilation.

Separate compilation requires careful thinking. It can't be an afterthought of language design: you have to design the language's module system and compilation model with separate compilation in mind. Having this figured out and sketched out from the start is a good idea.

Code Organization

It's generally good to separate *intermediate representations* from *passes*. The former are types, the latter are a set of functions. This helps keep modules short and to the point. Besides, there's not always a one-to-one mapping from IRs to passes, you will be running multiple different analysis passes on the same representation.

The exception to this is where it would break encapsulation. Sometimes you want types to be opaque, so that the module that defines them can enforce invariants about their contents.

Bootstrapping Issues

(This section only applies to new programming languages and not e.g. Fortran)

Compilers are often written in the language they compile. This shows the language is mature enough to handle the non-trivial task of building a compiler, and it also makes contributions easier, because contributors only need to know one language, rather than both the host and implementation languages.

So, you've decided to build a new programming language, but you can't write the compiler in a non-existent language. The basic idea here is:

1. Write the **bootstrapping compiler**, in a language like OCaml.

2. Write the **self-hosting compiler**, in your new language, and use the bootstrapping compiler to build it.

When do you move on from 1 to 2? Writing the simplest, dumbest possible MVP compiler is a good idea: it lets you iterate with the language early and see what makes sense and what doesn't. Immediately jumping off to build a production compiler at this stage is a bad idea: building a compiler is hard, and if your only tool is a rickety MVP compiler held together by spit and bailing wire, writing the self-hosting compiler will take longer and be more frustrating than it should be.

The bootstrapping compiler doesn't have a set expiration date. You can iteratively improve it until there's enough interest in, and users of, the language to justify the time investment in writing a second, production-quality compiler.

If iterative improvement is too slow, you can take the Fred Brooks route: throw the first one away. Build a brain-dead simple bootstrapping compiler in OCaml, then rewrite it from scratch into a production-quality compiler—but still in OCaml. And then only migrate to the self-hosted compiler when the language is mature enough that this is worth doing.

The Environment

The **environment** (also called the ***symbol table***, especially in compilers for C-like languages) is where user declarations, like type definitions and function definitions, are stored. This is the central data structure in a compiler.

Type checking, which is easily the largest part of modern language compilers, requires pervasive access to the environment.

For example, to typecheck a function call, you need the function's signature, which comes from the environment. Constructor calls require the definition of the type being constructed, which comes from the environment. The environment is also the natural place to store extra information that is not strictly needed for compilation but lets you provide richer diagnostics, such as the cross-reference graph. An API documentation generator can draw essentially all the information it needs from the environment.

Despite the centrality of the environment, compiler literature typically glosses over it, for two reasons:

1. First, most textbook compilers are for relatively simple, C-like languages. Since C has no explicit modules and a global namespace, and only four kinds of declarations (global variables, enums, types, and functions), the environment can be very simple: it's literally just a hash table from declaration names to definitions.

2. Compiler textbooks typically focus on the executable part of source text: the bodies of functions. This is where most optimizations, control flow analysis, SSA, code generation etc. happen.

The language determines how sophisticated the environment has to be. C and C++ can have relatively simple environments. Ada, which has a more complicated module system, requires a correspondingly more complicated environment. So some of this advice is not portable across languages, and an environment data structure that works for one may be lacking for another.

Austral is a relatively simple language. Code is organized in modules, which have names, and are non-hierarchical[2]. Modules contain declarations, of which there are six kinds: constants, records, unions, functions, type classes, and type class instances. All declarations *except* type class instances can be identified by a name.

In the Austral compiler, the environment is an opaque type that's only accessible through an API. Internally, it is a set of tables, where each table stores a different kind of language object. There's a table of modules, whose rows looks like this:

```
type mod_rec = ModRec of {
      id: mod_id;
      name: module_name;
      interface_file: file_id option;
      interface_docstring: docstring;
      body_file: file_id;
      body_docstring: docstring;
      kind: module_kind;
      (** Whether or not the module is unsafe. *)
      imported_instances: DeclIdSet.t;
      (** The set of imported typeclass instances. *)
      imports_from: ModIdSet.t;
      (** The set of modules this module imports from. *)
    }
```

In addition to this there's a table of declarations, a table of typeclass instance methods, and a table of monomorphs (to implement generics).

The environment has a simple CRUD API, and each function basically performs integrity checks like ensuring that you don't have two declarations with the same name in the same module.

Rows are indexed by a typed ID, but can be retrieved by name where applicable or, where necessary, by a predicate over rows.

Rows in one table can refer to rows in another using the typed identifiers. The CRUD functions can enforce relational integrity, ensuring, for example, that if you add a type class instance to the environment, the type class it links to exists already.

The environment, then, is just a record of tables storing different specific language constructs. The idea comes from reading the source code of Coalton, a typed, ML-inspired Lisp built on top of Common Lisp by Robert Smith, whom you should follow on Twitter.

In an earlier version of the Austral compiler, the environment matched the structure of language constructs: it was a map from module names to module objects, and module objects were basically a map from declaration names to declaration objects. But making updates to deeply nested structure is inconvenient. So the concrete lesson here is: *keep the environment flat*.

If I had time to refactor the environment, I think I'd structure it like this. First, define an interface along the lines of this:

```
module type ENVIRONMENT = sig
    type id
    type row
    type row_input
    type table

    val empty : table

    (* Create *)
    val store : table -> row_input -> id

    (* Retrieve *)
    val get : table -> id -> row option
    val filter : table -> (row -> bool) -> row list

    (* Update *)
    val replace : table -> id -> row_input -> table

    (* Delete *)
    val delete : table -> id -> table
end
```

(`row_input` is basically `row` but without the ID. The only way to create an ID should be by adding a row to a table. So the table has to keep an internal row ID counter.)

This is the most basic interface, relying exclusively on IDs for identifying rows. Most—but not necessarily all—language constructs are identifiable by name, so it is reasonable to create a more specific interface:

```
module type NAME_ENVIRONMENT = sig
    include ENVIRONMENT

    type name

    val get_by_name : table -> name -> row option
end
```

And then I'd implement the different environments like so (omitting some details for simplicity):

```
module ModuleEnv : NAME_ENVIRONMENT = sig
    type id = mod_id

    type name = module_name

    type row = { id: mod_id; name: module_name; docstring: string; }

    type row_input = { name: module_name; docstring: string; }

    type table = { next_row_id: int; rows: row list }

    (* ... functions implemented here ... *)
end

type decl =
  | ConstDecl of { name: identifier; type: ty; value: expr }
  | RecordDecl of { name: identifier; typarams: typarams; fields: field list }
  | ...

module DeclEnv : NAME_ENVIRONMENT = sig
    type id = decl_id

    type name = identifier

    type row = { id: decl_id; name: identifier; decl: decl; }
    type row_input = { name: identifier; decl: decl; }

    type table = { next_row_id: int; rows: row list }

    (* ... *)
end
```

Finally wrap up all the environments into a single `env` type:

```
type env = Env of {
    module_env: ModuleEnv.table;
    decl_env: DeclEnv.table;
    ...
}
```

The `env` API consists of thin wrappers around the functions in the underlying table modules, plus whatever integrity checks you have to enforce. For a functional implementation, a function to insert a declaration would:

1. Take the declarations environment out of the `env`.
2. Mutate it using `DeclEnv.store`.
3. Replace it back into an `env` instance.
4. Return the new `env`.

Identifying Declarations

You need a way to identify declarations, so that AST nodes can point to them. Most declarations can be identified by their name. If you have:

```
namespace foo {
    void bar();
}
```

Then the function `bar` can be identified by the string `foo::bar`. In most languages this is all it takes. But some languages – Haskell, Rust, Austral – have declarations that don't have identifying names. Namely, type class/trait instances:

```
class Foo a where
  foo :: a -> String

instance Foo Int where
  foo _ = "Int"

instance Foo Float where
  foo _ = "Float"
```

Instances are identified not by name but by the `(typeclass name, type)` pair.

A uniform, general, type-safe way to refer to declarations is to simply use typed, opaque identifiers, like so:

```
type module_id
type decl_id
```

Or, if you want to be more specific:

```
type module_id

type decl_id =
  | DeclIdConst of const_id
  | DeclIdType of type_id
  | DeclIdFunc of func_id
  | DeclIdClass of class_id
  | DeclIdInstance of instance_id

type const_id
type type_id
type func_id
type class_id
type instance_id
```

Typed identifiers have fewer degrees of freedom than strings. In addition, you can design your API such that declaration identifiers cannot be constructed manually, rather, they can only be created by storing declarations in the environment:

```
val store_constant_decl : env -> (name * type * value) -> (env * const_id)
```

Errors

Initially, I wanted to have something along the lines of this:

```
open Identifier
open Type

type err =
  | NoSuchName of identifier
  (** No identifier with name found in the environment. *)
  | TypeMismatch of { expected: ty; got: ty; }
  (** Type error: expected a type other than what we got. *)
  |
  ...
```

An error being just a giant variant of different error cases. The problem with this is dependencies: the `Error` module has to import `identifier` from the `Identifier` module, `ty` (the type to represent Austral types) from the `Type` module. Which means those modules can't raise errors, because importing the `Error` module would be a circular dependency. This is a problem as new types of errors are added.

One solution would be to split each module into two: a module that has the types, and another that has the functions, e.g. `Type` and `TypeSystem`. Then `Error` can just import the types, and the modules that have actual code in them (and can raise errors) can import the

`Error` module. This breaks encapsulation: sometimes, you want opaque types, hidden behind an interface. Yes, even in functional programming.

I have a much better solution at present:

```
(** An Austral compiler error. The module name, span, and source context rarely
    have to be passed in explicitly: they are added where the error is throw in
    the context of {!adorn_error_with_span}. *)
type austral_error = AustralError of {
      module_name: module_name option;
      (** The name of the module where the error occurred, if available. *)
      kind: error_kind;
      (** The error kind. *)
      text: err_text list;
      (** The error text. *)
      span: span option;
      (** The source span where the error happened, if available. *)
      source_ctx: source_ctx option;
      (** The source code where the error happened, if available. *)
    }

(** Represents a category of errors. *)
type error_kind =
  | ParseError
  (** A parse error. *)
  | TypeError
  (** A type error. *)
  ...
  | InternalError
  (** An internal error in the compiler. *)

(** An individual error text element. *)
and err_text =
  | Text of string
  (** Human-readable prose. *)
  | Code of string
  (** Austral text, like the name of an identifier or a type. *)
  | Break
  (** A paragraph break. *)
```

An `austral_error` is a fairly generic record, it stores:

1. The name of the module where the error is thrown.
2. The kind (essentially a typed title).

3. The source span where the error happened (file, line/column interval).
4. The source context, which is the source code at the given span plus a few lines before and after.
5. Finally, the error message using the `err_text` type. This is essentially a very minimalistic markup language so we can render text both to the console and as HTML.

This is the error as a value. You can wrap it in an exception:

```
(** The exception that carries Austral errors. *)
exception Austral_error of austral_error
```

Errors are constructed like this:

```
AustralError {
    module_name = None;
    span = None;
    source_ctx = None;
    kind = TypeError;
    text = [
        Text "Expected a value of type";
        Code (type_string a);
        Text ", but got a value of type";
        Code (type_string b);
    ];
}
```

And then thrown by being wrapped in an exception.

The module name, span, and source context are optional because they don't have to be passed in where the error is thrown. Rather, code that's higher up the call tree can catch an error, put that contextual information in, and rethrow the error. This means you don't have to pass the contextual information all the way down the call tree, you can just add it at the point where it is convenient:

```
(** Run the callback, and if it throws an error that doesn't have a span, put
    the given span in the error and rethrow it. *)
let adorn_error_with_span (new_span: span) (f: unit -> 'a): 'a =
  try
    f ()
  with Austral_error error ->
    let (AustralError { module_name; kind; text; span; source_ctx }) = error in
    (* Does the error we caught already have a span? *)
    match span with
    | Some _ ->
       (* The error already has a span, do nothing. *)
       raise (Austral_error error)
    | None ->
       (* Create a new AustralError object with the new span, and rethrow it *)
       let new_err = AustralError { module_name; kind; text; span = Some new_span; sourc
       raise (Austral_error new_err)
```

Then, having an ID is proof that a declaration exists.

Remarks

Writing a correct, production-quality compiler with useful diagnostics is difficult and time consuming work. However, it can be some of the most rewarding work in software engineering, because compilers communicate with the messy real world through a very narrow interface at the beginning: the parser. Beyond that, you have a great deal of freedom to experiment with new ways of organizing software, you can be as original and avant-garde as you want.

Footnotes

1. If you have a working compiler with an automatically-generated parser, and profiling says you're spending a lot of time parsing, then it might be worth investigating whether you

can write a faster parser by hand. As always: measure. ↩

---

2. As in languages like Java, Common Lisp, and Haskell, modules in Austral are “conceptually” but not “actually” hierarchical. That is, you can have a module with a name like `App.Core.Storage.Sql`, and the dots serve to create a conceptual hierarchy to the module name. But the module namespace is a flat map from module names to modules. ↩