Lint Roller: An OCaml Linter Written in OCaml

Nenya Edjah, Theodore Liu, Richard Wang

May 3, 2017

1 Defining "linter"

Wikipedia defines a lint (we'll call it a linter) as "any tool that flags suspicious usage in software written in any computer language." By writing an OCaml linter, we want to detect opportunities to apply best practices where they have not already been applied. This is somewhat analogous to what the TFs do when they grade our problem sets on the basis of style and design. Ideally, a linter would do a lot of this work.

Consider, for example, the valid OCaml expression `1+2;;`. It will compile, and it will evaluate correctly to 3, but if it were placed in a large codebase, someone reviewing the code might have difficulty reading it. In this case, a better expression would be `1 + 2 ;;`, with spaces around the `+` operator and preceding the double semicolons, `;;`.

How about the expression `1 +  2 ;;`? It still compiles and evaluates fine, but it is far more difficult to read. Of course, this is an extreme case, but it might better convey the gist of what we want to do with a software linter. In essence, a software linter will recommend these best practices to the user before the code is compiled.

1.1 Opening remarks

We are going to establish that we want to concern ourselves more with the recommendation of good style and semantics as opposed to capturing serious syntax errors, which is a job usually left to the compiler. Of course, in the process of trying to improve style we might end up catching a few of these syntax errors. In any case, evaluating code before compile-time allows us to skirt a couple of nitty gritty details, which we will talk about later.

2 Initial steps

2.1 Getting set up

Our code can be found at this Github link. You will need to install `menhir` and `re2` before compiling by doing `opam install menhir re2`. You can compile our project by simply executing `make`.

We first pulled the Github repository for the standard project from this Github link.

2.2 Understanding the standard project

We understood that the implementation behind the standard project involved a lexer (ocamllex) and a parser (ocamlyacc). The code that was to be compiled and executed allowed a user to type in a (valid) OCaml expression which was then converted into a lexer buffer and tokenized. The tokens were then parsed to expressions.

2.2.1 Understanding the lexer, parser, and expr

In the process of tokenizing the buffer, the lexer took regular expressions that matched sequences of digits, characters, and symbols, and utilized a hash table to return the proper tokens for predefined or reserved words and symbols.

The parser then took the stream of tokens from the lexer and matched them according to a grammar in BNF which was written in `miniml_parse.mly`. The parser then returned an object of type expr, the specifics of which were defined in `expr.ml`.

3 Modifying the lexer, parser, and expr

We managed to find a comprehensive OCaml grammar online. We used this for a while in determining what grammar rules to form and what tokens to form.

3.1 The lexer

3.1.1 Supporting floats

We noticed that MiniML did not support floating point numbers. It was simple enough to write a regular expression in order to capture such numbers:

```
['0'-'9'] ['0'-'9' '_']* ('.' ['0'-'9' '_']*)? (['e' 'E'] ['+' '-'] ['0'-'9']
['0'-'9' '_']*)?
```

These are sent to a `FLOAT` token along with the corresponding value. Additionally, in `expr.ml` we created a float type in the abstract syntax of data which took a float.

3.1.2 Supporting character literals

Again, just a simple regular expression. This matches any regular character (wildcard) or escaped character, that are escaped using single quote literals. These are sent to a `CHAR` token.

3.1.3 Supporting string literals

Still just a simple regular expression. This matches any number of regular characters or escaped characters using a Kleene star, that are escaped using double quote literals. These are sent to a `STRING` token.

3.1.4 Resolving precedence of two-symbols over symbols

Our matching for symbols was initially too greedy and matched some important sets of two symbols (what we refer to as two-symbols) such as "`;;`", "`::`", "`->`". In order to resolve this issue we added another set of regular expressions that captures these two-symbols.

3.1.5 Supporting other operators

MiniML is incredibly restrictive in the number of operators that it accepts. We initially wanted to expand the number of supported operators by assigning a new token to each one of them, but we realized that it was smarter to support different operators based on what kind of operations they were performing, or what types they were performing operations on. At first, we supported more than a couple dozen `COMPAREBINOP` (comparison binary operators), `BOOLBINOP` (binary operators on booleans), `INTBINOP` (binary operators on integers), etc... along with the corresponding unary operators for certain types.

We eventually discovered that there are, in actuality, an infinite number of possible operators (not all of them are predefined, but all of them can be assigned to act as operators). In order to solve this problem, we wrote regular expressions that captured such sequences of symbols and returned an appropriate `INFIX` or `PREFIX` token depending on whether said operator was an infix operator or a prefix operator. For those which we could not generally capture with regular expressions, such as the infix operators `land` and `lxor`, we put entries for in the hash table.

3.1.6 Supporting tuples

Much like how we initially captured lists, we create `OPEN`, `COMMA`, `CLOSE` tokens corresponding to the appropriate tuple opening constructions, commas as delimiters, and tuple closing constructions. A fun (or terrible) quirk of OCaml is that `(, )` directly correspond to and are interchangeable with `begin, end`. This means that `begin 1, 2, 3 end` is interpreted as a tuple containing the integers 1, 2, and 3, and `begin end` is interpreted as unit. For this reason, both parentheses and begin/end are sent to their respective `OPEN` and `CLOSE` tokens.

3.2 The parser

3.2.1 Switching the parser

Somewhere down the line, we decided to switch the parser generator from `ocamlyacc` to `menhir`. The reason for this is that `menhir` gave us some powerful tools to do matching on optional tokens,

matching on delimited lists of tokens, and also let us assign the expressions returned sequences of tokens to variables.

3.2.2 Supporting simple lists

Initially, we realized that a list must be comprised of an opening `[`, some number of delimiters `;`, and a closing `]`, which we sent to the tokens `LISTOPEN`, `DELIMITER`, and `LISTCLOSE` respectively. Lists are represented in the abstract syntax as `Cons` expressions, where the end of the list is represented by a terminating `Nil`. For example, the list `[1;2;3]` will be parsed into the expression `Cons(1, Cons(2, Cons(3, Nil)))`.

3.2.3 Supporting all well-formed lists

The former list construction doesn't support nested lists due to the need to assign a precedence to how `LISTOPEN` and `LISTCLOSE` are parsed. In order to resolve this problem, we created a new subset of the grammar called `listexp` that handles this issue. Somewhere along the way, we renamed the `LISTOPEN` and `LISTCLOSE` tokens to `OPENBRACKET` and `CLOSEBRACKET`.

```
...
listexp:
  | exp SEMICOLON listexp { $1 :: $3 }
  | exp                  { [$1] }
  |                      { [] }
...
exp:
...
| OPENBRACKET listexp CLOSEBRACKET  { List.fold_right
                                        (fun x y -> Cons (x, y)) $2 Nil }
...
```

Another issue that we had was the supporting the existence of a trailing semicolon at the end of a list. This was covered in the listexp rules.

3.2.4 Simplifying expressions

With regards to operators, just like how in the lexer we only ultimately cared about whether or not we were working with infix or prefix operators, we only care about the same in the parser. In a similar simplification, we ultimately sent literals (int, float, string, char, unit, boolean) to a Const expression that described its type. We do not care about what literal values those tokens corresponded to, just their type.

3.2.5 Supporting let, let ... in, let rec, let rec ... in

We were able to condense the grammar rules in the parser for all sorts of let statements into just one rule, written below:

```
| LET x=boption(REC) y=nonempty_list(ID)
      EQUALS z=exp a=option(preceded(IN, exp))
      { let h, t =
          match y with
          | h :: t -> h, t
          | [] -> failwith "Cannot get here because nonempty_list" in
        let l = List.fold_right (fun x y -> Fun (x, y)) t z in
        match x, a with
        | false, None -> Let (h, l)
        | false, Some b -> LetIn (h, l, b)
        | true, None -> LetRec (h, l)
        | true, Some b -> LetRecIn (h, l, b) }
```

We were able to condense all the variants of let statements into one rule because the different types of let statements really derive from the simplest `let x = y` sort of statement. This rule matches against an optional `rec` keyword, represented by the menhir statement `boption(REC)`. Similarly, the statement `option(preceded(IN,exp))` will match against the let ... in constructions, only if there is a trailing expression. We only care about the existence of the rec keyword and the let ... in construction in both cases.

For cases where we only have one "argument" in the let statement, for example `let x = x + 1`, then our fold statement will not match on the empty tail, and we will have a let statement as expected.

However, for cases where we have many arguments, for example `let f x = x + 1`, then what our fold statement does is treat this many-argument let statement as a series of nested anonymous functions. In our example, `let f x = x + 1` can be rewritten as `let f = fun x -> x + 1`.

3.2.6 Supporting match ... with

We decided to represent match statements in a system analogous to how we implemented lists. If we consider a match statement, it is really comprised of matching some variable to a list of conditions or restraints. A simpler way to understand this is that match statements can also be written as a long line of match conditions delimited by pipes; it is mainly for purposes of readability that we conventionally write match statements separated by new lines. Therefore, an MCons will contain a tuple that describes the expression to be matched against, the resulting code if the match is successful, and any following MCons relevant to that match statement (or a MNil if there are none left). Of course, the reason why we separate Cons, Nil, and MCons, MNil, is because lists and match statements are not the same thing and therefore shouldn't be represented as such in our abstract syntax tree.

```
matchexp:
  | PIPE exp DOT exp matchexp         { ($2, $4) :: $5 }
  | exp DOT exp matchexp              { ($1, $3) :: $4 }
  | PIPE exp DOT exp                  { [($2, $4)] }
  | exp DOT exp                       { [($1, $3)] }
...
```

```
exp:
...
| MATCH exp WITH matchexp           { let l = List.fold_right
                                              (fun (x1, x2) y ->
                                                 MCons (x1, x2, y))
                                              $4 MNil in
                                        Match ($2, l) }
...
```

3.2.7 Supporting tuples, continued

Because tuples don't have the same loose rules as lists have (for example, you may not end a tuple in a comma whereas in a list you are able to end in a semicolon), we were able to simply use the menhir statement `x=separated_nonempty_list(COMMA, exp)` to construct a Tuple of x.

4 Style checking

These checks are related to pure style with no regard to the actual content or purpose of the code. They are naive checks that either go character by character or simply use regular expressions to find errors.

4.1 Overlength lines

This style check was very straightforward to implement. After reading in the file to a string, we simply split the string on newlines to get a list of strings. After that, a simple iteration through the list to check the lengths allowed us to find the lines that were over the 80 character limit.

4.2 Trailing whitespace

Most code editors trim trailing whitespace at the end of lines on save, but this is nevertheless a good check to have in a linter. Using a regular expression, we captured all whitespace before newlines in the file. If anything was captured, we could raise an error and print it out.

4.3 Space around operators

Spacing around operators makes code much easier to decipher and interpret. To detect this style, we again used a regular expression to match against common operators and then checked the characters adjacent to the operator to determine if they were spaces. If we detected no spaces, we raised a warning and recommended the proper spacing around the operators.

4.4 Matching parentheses and other delimiters

Matching parentheses and other delimiters is not only a style thing but also extremely important in ensuring code that compiles. Before starting the check for matching delimiters, we first used a regular expression to remove all the delimiters inside string and character literals since these obviously should not affect the overall pairing of delimiters. Additionally, we used another regular expression to remove all commented code because these comments also have no affect on the overall pairing of delimiters. After this preprocessing was completed, we could begin with checking for matching delimiters.

Our attack was to use a stack. We went through the code character by character and added the character to the stack if it was an opening delimiter. If it was a closing delimiter then we checked the top of the stack for the matching opening delimiter. If they did match then we popped the top from the stack. If they did not match then we raised a warning in the terminal and also popped the delimiter from the stack. If we ever came across a closing delimiter when the stack was empty, this clearly signified an unmatched delimiter, so we raised a warning. In the event that we finished parsing all the code and there were still opening delimiters on the stack, then we again knew that there were unmatched delimiters and we raised warnings accordingly.

4.5 Matching quotes

Matching quotations is also very important for having compilable code. Before starting the check for matching quotations, we performed similar preprocessing to the matching delimiters check. We removed all comments with a regular expression. We also replaced all escaped quotations with a "`#`" to avoid complications when determining whether a quote was escaped properly within other quotes or not.

We also used a stack to check for unmatched quotations. However, we quickly realized that the stack would have, at maximum, one element. This is because matching quotations is greedy, and it is impossible to distinguish an opening quote from a closing quote. Thus, we proceeded in the same manner. If we came across a single or double quote, we checked the stack for a matching single or double quote. If the stack contained a matching quote, we "closed" the quotations by popping the element off the stack. If the stack did not contain a matching quote, we could safely ignore the quote since the opposite quote inside a quote will properly compile. If the stack was empty, we could add the quote to the stack to "open" a quote.

Another thing we wanted to check is that character literals were the correct length. Inside of a character literal, there can only be one character, so we also kept an additional variable that measured the length of a quote so that we coud properly assess whether character literals were the right length. There are some minor subtleties with this check since the "backslash" character is used to escape characters and should not be counted towards the length of the character literal.

4.6 Indentation

In our initial proposal, we assumed that indentation would be simple to check, but this turned out to be wildly incorrect. Indentation is highly variable and fluid; we intuitively know what "looks

right" but it turns out to be difficult to capture this intuition in code.

Our initial strategy was to look at the first and last words or characters in a line to determine what the indentation level of the next line should be. This works at the most basic level, but once you start moving into strange things like match statements or longer if-else statements, this heuristic no longer works.

In the end, we were unable to get indentation properly working, so we decided to leave it out of the finished product. You can find the (somewhat ugly) code we used for this in `indent.ml`.

5 Abstract syntax tree traversals

5.1 Singular match statements

We traverse the tree and match against Match statements. If we find that they only contain one pattern, this is extremely inefficient code. These can usually be replaced by let-in statements unless the match is incomplete. If we ever encounter these singular matches, we simply "recommend" that the user switch to a let statement. We cannot ensure that this will be a full match, though.

5.2 Opportunities for partial application

Again, this is simply a matter of correctly traversing the AST that we generated through our parsing. However, we ran out of time on the project and did not reach this style checking, so this is currently left unimplemented. However, this is an easy

6 Type inferencing

We found, used, and modified an excellent resource about type inferencing. Type inferencing is powerful because it allows us to recommend fixes to the user based on type errors before compile-time. For example, if the types in lists are mismatched, that could be reported as a possible error. If the types surrounding an infix operator are mismatched, that could also be reported as a possible error. With these motivations in mind, we set out to implement type inferencing in OCaml.

We were able to infer the types for some predefined infix operators, as well as support polymorphism.

6.1 Different approaches

We sought to understand the Hindley-Milner type system, a classic type system for the lambda calculus. In this system, we have expressions that are monotyped and polytyped, and we want to infer, based on the context in which expressions appear in and how they are used, what the types of the expressions are. OCaml uses a complicated algorithm based on the union-find data structure which is both poorly-documented and more of an optimization to a simpler type inferencing algorithm. We settled on performing unification of types as our method of type inferencing.

The idea in performing unification of types is that we first annotate the abstract syntax tree (derived from the parser) with the types of the variables. At this stage, we do not know what the types of the variables are, so we assign a new polymorphic type (or, if the variable has already been assigned one of these types, we use the same time). After annotating the abstract syntax tree with these types, we collect whatever substitutions need to be made, before unifying all of these substitutions. After unifying all of these substitutions, we will have inferred the most general type of the expression.

6.2 Expanding the expressions we can infer

We added inferencing support for let statements, let ... in statements, constants, lists, tuples, and infix operators.

6.2.1 Inference on constants

When we annotate constants here, we really see that it is unnecessary to find what the value is in type inferencing. Instead of going to a variable representing a polymorphic type, all constants or literals are sent to the same variable structure, except representing their respective types. This is powerful because when we unify our type environment, we do not need to create another type variable that represents a constant.

6.2.2 Inference on let (in) statements

Consider the expression `Let (x, e)`. In the process of annotating, we have the annotated version of `e` reflects that the type of the let statement goes from the type of `x` to a new polymorphic type. However, this isn't of especial importance (we don't need to add it to our list of substitutions). The reason for this is that the let statement construction doesn't inherently enforce any types among the expressions that comprise it. A similar process occurs for expressions of the form `LetIn (x, e1, e2)`.

6.2.3 Inference on lists

Naturally, lists are comprised of other expressions joined together by the Cons construction. We recursively annotate the elements inside of the list. The important thing to realize here is that were we to enforce a type on the list (list elements must have a single type), we would do it outside of the annotation step. In other words, we would rather infer the individual types of elements inside the list, and then traverse through the list to see if there is a type mismatch. This method is much simpler.

6.2.4 Inference on infix operators

We predefine a list of predefined infix operators and store them in a hash table. When we annotate the infix expression, if the infix operator is predefined in our hash table, then we add the appropriate

substitution. Otherwise, we assign a polymorphic polytype to the expression. The pitfall in this scheme is that we are unable to detect reassignments in any infix operator we predefine. For example, it is entirely possible in OCaml to reassign the `( + )` operator, although it's certainly not recommended. We do not have a good way to detect updated definitions to these operators yet.

7 Looking forward

We feel proud about what we have accomplished with the linter, but there are definite areas for improvement.

7.1 Increasing acceptable syntax

Even though we covered a large portion of OCaml's syntax, there still are a lot of subtleties remaining in the language as well as whole features that we still not have covered, including modules and objects. In future versions of this linter, it would be nice to have these features of OCaml covered so that users can check their style when using these features as well.

7.2 Improving style checks

Our attempts to check for mismatched delimiters and quotations are good but somewhat naive in their attempts. First, we replace all quotes and comments with a singular character, so if there are large comments in a file, the line and column numbers indicating the location of an error can be wildly inaccurate. Additionally, the checks are still not very context sensitive since they only examine one character at a time. This can be improved by increasing the size of our sliding window or by storing more information about the file as we go through it.

We also left indentation completely unfinished. A different outlook or approach to indentation might be the answer we need to correctly and robustly detect indentation errors. Our current approach is too narrow and naive to be useful or correct.

7.3 Improving AST traversals

We shed a great deal of blood, sweat, and tears to correctly parse the OCaml syntax into an abstract syntax tree. Yet, in the time given, all we were able to do with this AST is detect singular match statements. As mentioned before, there is ample opportunity to use the AST to detect chances for partial application, but there are undoubtedly more uses for the AST. A step we were aiming to complete was writing a complete traverser for the annotated AST. Remember that our AST is an expr, and our annotated expression results in an annotated AST. For this to be effective, though, we would need to expand the number of constructs that we can annotate and perform unification on. The logic behind what kinds of substitutions need to be integrated in the type environment is already challenging, so this will be a difficult step.

7.4 Integrating type inference

Again, we did hard work to make type inferencing work on our subset of the OCaml language, but we were not able to use this type inferencing to make recommendations about the code. In the future, we *would* like to be able to make recommendations about improper types, and we can definitely leverage our type inferencing to do this.

In the event that we extend our supported syntax, we would also want to extend our type inferences ability to infer on a large portion of the OCaml syntax.

8 Conclusion

This was an incredible project and experience. We learned a lot about a wide variety of things, including lexing, parsing, type inference, abstract syntax trees, regular expressions, and more. We also learned about a lot of the subtleties of the OCaml language and the difficulty in accommodating all such subtleties. We also gained some valuable life lessons in time management, version control, setting goals, and working as a team. We are proud of what we accomplished in the time given, but we want to continue to iterate on this project in the future.

For now, we will leave the style and design score up to the TFs.