Pocaml: Poor Man's Ocaml

Leo Qiao, Peter Choi, Yiming Fang, Yunlan Li
Email: {flq2101, jc4883, yf2484, yl4387} @columbia.edu

January 6, 2022

1 Introduction

Pocaml is the "poor man's OCaml". It incorporates the main features of OCaml, such as parametric polymorphism, partial function application, lambda functions, pattern matching, and much of the same syntactic sugar, while also including a standard library for common use cases. One interesting aspect of Pocaml is that everything valid in this language can also be compiled by an OCaml compiler, in the same way C code can be compiled by a C++ compiler.

The input language is a functional language that follows the OCaml syntax. After scanning and parsing, the Abstract Syntax Tree (AST) gets lowered to a more concise form of Intermediate Representation (IR). Before generating LLVM code for the IR, our language goes through a pass of lambda lifting. Finally, in code generation, we link C libraries to generate builtin functions, and use OCaml's LLVM bindings to compile our IR to binary executable.

2 Language Tutorial

There are 2 ways to use Pocaml: locally and in a Docker container.

To setup Pocaml locally, make sure to have OCaml (¿4.08) and opam installed. Then, run `opam install . --deps-only` in the root directory of the Pocaml repository to install all dependencies, such as llvm and llvm ocaml bindings. To run a pocaml (.pml) file, run this command: `./pocaml file.pml`.

To use Pocaml with Docker, make sure to have Docker installed. Then, to run a command in our Pocaml Docker image, run the following: `./runDocker command`. For example, to run a pocaml file, run this command: `./runDocker ./pocaml file.pml`.

Let us create a new file called `playground.pml` to explore some of pocaml's features. Within this file, add the following code:

```
let value = 3 + 4 * -2
  let _ = print_int value
```

Run the program file and we see the output `-5` as expected. Here, we see two definitions at the top level. Definitions are made using the following syntax: `let VARIABLE (...PARAMS) = EXPRESSION` The first definition uses in-fix arithmetic operators and the second uses some simple IO. Other provided IO functions include `print_bool`, `print_char`, `print_string`, and `print_endline`. The reason for having separate print functions for each data type is that Pocaml is strongly typed. We also see the use of the wildcard for top-level definitions, as `print_int` returns some value that we do not care about (in particular, the unit type singleton value).

Let us now replace the code with the following:

```
let _ =
  let value = 3 + 4 * -2 in
    print_int value
```

Here, we see the use of the `let VARIABLE = EXPRESSION in EXPRESSION` expression, which cannot be used at the top-level, as it is an expression. This is why a wildcard top-level definition is used to wrap the program. Now, let us update the code like so:

```
let _ =
  let identity = fun a -> a in
    let value = 3 + 4 * -2 in
      print_int (identity value)
```

Here, we define a polymorphic identity lambda function which simply returns its argument. Lambda functions are defined with the `fun (...PARAM) -> BODY` syntax and are valid expressions. Before we continue building toward our final program, let us take a detour to explore parametric polymorphism:

```
let _ =
  let identity = fun a -> a in
    let value = 3 + 4 * -2 in
      let _ = print_int (identity value) in
        let value2 = "hello, world" in
          print_string (identity value2)
```

As shown here, the identity function is able to take multiple types of input during runtime. Now, replace the code with the following:

```
let _ =
  let fn_creator = fun is_flip x -> if is_flip then -1 * x else x in
```

```
    let do_flip = fn_creator true in
      let value = 3 + 4 * -2 in
        print_int (do_flip value)
```

The above demonstrates partial application of a function(`fn_creator`). A lambda expression which has `x` arguments can be applied `x - n` times to produce a function that takes in the rest of the `n` arguments. We also use a conditional expression, whose syntax is of the form `if EXPRESSION then EXPRESSION else EXPRESSION`.

Now, change the code to the following:

```
let _ =
  let fn_creator = fun is_flip x -> if is_flip then -1 * x else x in
    let do_flip = fn_creator true in
      let modify = fun do_flip -> fun x ->
        let res = do_flip x in
          if res < 0 then (-1 * res) else res
      in
      let abs = modify do_flip in
        let value = 3 + 4 * -2 in
          print_int (abs value)
```

The `modify` function is a higher-order function that modifies the `do_flip` function to return the absolute value of the argument instead. Running the program should now output `5`.

Now, change the code a final time:

```
let rec map f = function
  | head :: tail ->
      let r = f head in
        r :: map f tail
  | [] -> []

let _ =
  let fn_creator = fun is_flip x -> if is_flip then -1 * x else x in
    let do_flip = fn_creator true in
      let modify = function do_flip -> function x ->
        let res = do_flip x in
          if res < 0 then (-1 * res) else res
      in
      let modified_fns = map modify [do_flip] in
        let abs = list_hd modified_fns in
          let value = 3 + 4 * -2 in
            print_int (abs value)
```

We have now added a `map` function, which makes use of functional recursion and list cons. `map` takes in a function `f`, and returns a function which pattern matches against the input list. If the list is not empty, `f` is applied to the first element and added to the rest of the mapped over list. If it is empty on the other hand, it returns an empty list. `map` is then used to call `modify` on each function of a list that only contains `do_flip`. The modified function, `abs`, is finally accessed using `list_hd`.

3 Language Reference Manual

3.1 Lexical Aspects

3.1.1 Blanks

Characters including space, tab, carriage return (`\r`), line feed (`\n`), and form feed are considered blanks in Pocaml. They serve to separate the program into tokens.

3.1.2 Comments

Comments begin with the 2-character sequence `(*` and end with the 2-character sequence `*)`. Comments do not occur within a string or character literals. In nested comments, all opening `(*` should be closed with a corresponding *) like so:

```
(* this is a comment *)
(* this is a
multi-line
comment *)
(* this is a (* nested *) comment *)
(* this is not (* a valid comment *)
```

3.1.3 Identifier

Identifiers are either an underscore or a sequence of letters, digits and underscores that starts with a lowercase letter. Letters contain the lowercase and uppercase alphabets from ASCII.

lowercase-ident	::=	(_ | (a...z){*id-literal*})
id-literal	::=	(*letter* | *digit* | _)
letter	::=	A...Z | a...z
digit	::=	0...9

3.1.4 Integer literals

An integer literal is a sequence of one or more digits, optionally preceded by a minus sign. Integer literals are in decimal.

integer-literal `::= [-](`*digit*`){`*digit*`}`

3.1.5 Boolean literals

Boolean literals are: true, false.

3.1.6 Character literals

Characters include the regular set of characters and the escape sequence, which serve to delimit characters. A character literal is surrounded by single quotes.

$$\begin{array}{lll} \textit{char-literal} & ::= & \text{`} \ (\ \textit{regular-char} \mid \textit{escape-sequence}\ )\ \text{'} \\ \textit{escape-sequence} & ::= & \backslash\ (\ \text{'} \mid n \mid t\ ) \end{array}$$

3.1.7 String literals

Strings are a series of any character that is either an escaped double quote or a non-double quote. A string literal is surrounded by double quotes. For example:

```
"hello, world"
"The cat said, \"Hello, world!\""
```

3.1.8 List literals

Lists can be represented in two ways. The first is through semi-colon separated expressions within brackets, such as `[1; 2; 3]`. The second is through cons, such as `1 :: [2; 3]`.

3.2 Expressions

3.2.1 atom

An *atom* represents the most reduced form of an expression. An atom can be a literal (such as true), a variable. Also, `( EXPRESSION )` is considered an atom. Here are some examples of atoms:

```
(3)
true
id
(1 + 2)
```

3.2.2 Let-in

A *let-in* expression is used like so: `let VARIABLE = EXPRESSION in EXPRESSION`, such as `let x = 3 in x`. The variable x is scoped only within the expression x.

3.2.3 Functions

Functions can be defined in three different ways. The first is regular lambda functions, which is created using the keyword `fun`, followed by parameters, an `->`, then an expression whose value will be returned on invocation. Lambda functions are expressions and thus can be passed as arguments to other functions. For example:

```
let a = fun x -> x + 3
```

The second way functions can be defined is through the `function` keyword, followed by any number of match arms, which consists of a pipe, pattern, arrow, and expression.

```
let matcher = function
    | hd :: tl -> "at least one item"
    | [] -> "x is empty"
```

This is equivalent to

```
    let matcher lst = match lst with
      | hd :: tl -> "at least one item"
      | [] -> "x is empty"
```

Here, `hd` is the first element of the non-empty list, and `tail` is the rest of the list, which may be empty. Another useful tool is the wildcard variable, which can be used to catch all other cases like so:

```
let matcher = function
    | 3 -> "x is three"
    | _ -> "x is not three"
```

The third way to define a function is using a definition. This is expanded on in section 3.3.

3.2.4 Function Calls

A function application is a prefix expression `id arg1 arg2 ...` with one or more blank-separated expression parameters. Functions applications are curried. The values of the parameters are strictly evaluated from left to right and bound to the function's formal parameters using conventional static scoping rules.

Partial function applications are supported and a function that takes in the remaining arguments is returned. For example, the follow is valid:

```
let _ =
  let fn = fun x y -> x + y in
    let partial = fn 3 in
      let res = partial 6
(* res is 9 *)
```

3.2.5 Pattern Matching

In addition to the pattern matching used in functions, pattern matching can be used as standalone expressions as well. This is done the syntax `match EXPR with MATCH_ARMS`. `MATCH_ARMS` is the same as that described in the previous section. An example of pattern matching used as an expression is as follows:

```
let y =
  let x = 3 in match x with
    | 3 -> "is three"
    | _ -> "Not three"
(* y is "is three" *)
```

3.2.6 Operator expressions

The operators are `+`, `-`, `*`, `/`, `=`, $<$, $>$, $<=$, $>=$, `&&`, `||`, `::`, `;`, `not`.

An expression can be formed using the syntax `EXPR BINARY_OPERATOR EXPR` or `UNARY_OPERATOR EXPR`. All operators are binary operators except `not`. The type of the expression following `not` should be a boolean. For example, `not true` is `false`. Parentheses group expressions and can be used to assign precedence. The binary operators &&, || do the logical AND and OR operations on two boolean values.

3.2.7 Conditionals

The branching expression `if EXPR_1 then EXPR_2 else EXPR_3` evaluates to `EXPR_2` if `EXPR_1` evaluates to `true`. Otherwise, it evaluates to `EXPR_3`.

3.3 Definitions

A Pocaml program is a sequence of definitions. Thus, expressions cannot exist at the top-level of the program.

3.3.1 Functions

The syntax for a function declaration is as follows:

```
let id args = EXPR
let rec id args = EXPR
```

where

$$\begin{array}{lll} \textit{param} & ::= & \text{a...z (a...z | A...Z)} \\ \textit{args} & ::= & \textit{param} \mid \textit{param}\ \ \textit{args} \end{array}$$

The above (both forms) declare a function named `id` that takes in one or more parameters; `EXPR` is the body of the function. The scope of the function arguments is `EXPR`. All functions defined using the syntax above are recursive.

To demonstrate how a function that has multiple arguments is curried, here are two functions identical in effect. The second utilizes a lambda expression:

```
let fun1 a b = a + b
let fun2 a = fun b -> a + b
```

3.3.2 Variable Definitions

A declaration in the form `let ID = EXPR`. It produces a name to value binding that can be accessed globally within the same file. In addition, it is used only at the top level. Here is an example:

```
let x = if true then 3 else 2
```

3.4 Standard Library

$print(s : string)$
Print the string to the standard output.

$list\_map : (`a \to `b) \to `a\ list \to `b\ list$
Apply a function to each element of a list to return a new list with the original type.

$list\_iter : (`a \to unit) \to `a\ list \to unit$
Call a function with each element of a list.

$list\_append : `a\ list \to `a\ list \to `a\ list$
Return a new array containing the concatenation of two arrays

$list\_fold\_left : (`a \to `b \to `a) \to `a \to `b\ list \to `a$
fold_left f lst init applies function f on the current accumulator (initially *init*) and each element in *lst*, going from left to right. It returns the current accumulator after going through the whole list.

$list\_fold\_right : (`a \to `b \to `a) \to `a \to `b\ list \to `a$
fold_right f lst init applies function f on the current accumulator (initially *init*) and each element in *lst*, going from right to left. It returns the current accumulator after going through the whole list.

Other available stdlib functions are: *list_length*, *list_filter*, *list_rev*, *list_hd*, *list_tl*, *print_endline*, *print_int*, *print_bool*, *print_char*, *print_list*, *print_int_list*, *print_bool_list*, and *print_char_list*

3.5 Example

This example demonstrates how to implement Euclid's algorithm for finding the Greatest Common Denominator (GCD), printing the result after finding the answer. This code snippet showcases some of Pocaml's features, such as `let` expressions, recursive function call, type specification, and control flow statements.

```
let rec mod a b =
  if b > a then a
  else mod (a - b) b

let rec gcd a b =
    if b = 0 then a
    else gcd b (mod a b)

let print_gcd a b =
```

```
    print_endline (string_of_int (gcd a b))

let _ = print_gcd 15 5
```

4 Project Plan

4.1 Process

4.1.1 Planning

We first broke down the project into different compiler stages that are necessary for the language features that we wanted to implement. In the beginning, we determined on a high level what effect each compiler stage would achieve and its input and output. We then divided the semester into blocks of weeks where in each block we would focus on a specific compiler stage and complete its milestones. Within each block, we break down the milestone further into subtasks, based on suggestions from our project advisor John and our weekly group discussions. As our compiler gets bigger, bugs inevitably appear and we may find previous design decisions inflexible. We leveraged GitHub issues as they arise and assign the issues as tasks to each member. We meet regularly on Monday but also set aside an additional meeting time on Wednesday as needed.

4.1.2 Specification

From the outset, our team wanted to implement a compiler for a functional programming language. Functional programming languages have many cool features but each presents a big challenge on its own and could easily take a whole semester's time. With the aim of fitting the project into one semester while also implementing some cool features, we reached out to John before the submission of the Language Reference Manual to gauge the feasibility of implementing the features that we were interested in implementing. Eventually, under John's guidance, we made the following decisions:

1. make our language a subset of OCaml
2. select single-pattern polymorphic let-bindings, lambdas, pattern matching as the main features to incorporate in our language and implement either type inference or user-defined ADTs as time permits

Decision 1 delegates the language design aspect to OCaml to allow us to focus on the implementation of the language and to allow time for star-worthy features that would be hard to implement before we have a working compiler.

4.1.3 Development

Our development largely follows the stages of our compiler. In the beginning, we focused on making a MVP for the parser and the lexer as soon as possible. We started with the parser to define an AST for our language and then moved on to implement the parser. Once our lexer and parser were compiled by OCaml without errors, we split ourselves in two teams:

- (Peter, Yunlan): set up the testing framework using **ppx_expect** for OCaml code, test lexer and parser by writing a pretty printer for the AST and test cases, expand the lexer and parser to incorporate all language features in our LRM
- (Leo, Yiming): work towards a working end-to-end compiler by deciding on the necessary compiler stages, specifying the jobs of and creating a .ml file for each stage

After we had a fully working lexer and parser and well-defined compiler stages, we combined again to start implementing each stage: from first lowering (desugar) our original expressive AST into a much smaller abstract syntax tree, to tackling partial function application, lambda lifting, and code generation. After that, we had a working end-to-end compiler and started the implementation of pattern matching in code generation and monomorphization. An anecdote is that after we spent an afternoon on monomorphization and figured out its implementation, we realized that it was actually already achieved due to our design choice of a uniform run-time value representation (section 5.3) across all data types.

4.1.4 Testing

Whenever we complete a compiler stage, we would test the compilation process up to that stage. Assuming the correctness of the previous stages, this allowed us to perform unit test and integration test simultaneously.

For all stages before code generation, we would design each test case based on a specific task of the compiler stage and pretty print the modified syntax tree to determine if the desired changes were made.

To test our entire compiler, we would design test cases each centered on a specific aspect of our language, such as pattern matching, polymorphic functions, let-in bindings, etc.

For the C code that get linked to our compiled LLVM code, we unit tested, in C using simple assert statements, each built-in function and primitives to achieve partial function application.

More details about testing are shown in section 6.

4.2 Programming Style Guide

We set up dune[1] to use ocamlformat[2] to automatically format Ocaml source code when we run `dune build @fmt` at the root of our repository. The command will find recursively all `*.ml` and `*.mli` files in our project and show us the difference between the original file and the formatted file. We can then apply all the formatting change simply with `dune promote`.

4.3 Project Timeline

The major milestones of our project and its completion date are listed in the Table 1:

Date	Milestone
Oct 8	Project proposal submitted
Nov 9	Lexer, Parser completed
Nov 17	IR defined, lower AST to IR completed
Nov 18	Pretty Print completed
Nov 20	Run-time Value Representation completed
Nov 23	Lambda Lifting completed
Dec 3	C Built-ins completed
Dec 7	Code Generation completed
Dec 19	Automated test suites completed
Dec 22	Project Report and Presentation completed

Table 1: Project Timeline

4.4 Roles and Responsibilities

All the team members touched both the front-end and the back-end of the compiler. It's hard to attribute the completion of a specific compiler stage to a single person as in most cases, it was a collaborative effort of the whole team. For example, the whole team worked together in the same room in an afternoon to complete the initial version of the lexer and parser. Later, as we incorporate different features, such as pattern matching, into the AST or testing the parser, relevant parts of the lexer and parser were modified by different team members. Similar to the lexer and parser, the C library for allocating data on heap and supporting partial function application were also coded by the team together in person.

Table 2 roughly summarizes each team member's major involvement in compiler stages other than the lexer, parser, and C library mentioned above.

[1] https://github.com/ocaml/dune/tree/main
[2] https://github.com/ocaml-ppx/ocamlformat

Team Member	Responsibility
Leo Qiao	IR, Code Generation
Peter Choi	C built-ins, Automated Testing Suite
Yiming Fang	IR, Code Generation
Yunlan Li	Lambda Lifting, Automated Testing Suite

Table 2: Role and Responsibility of each Member

4.5 Software Development Environment

Tools:

- **GitHub-hosted Git Repository**: git for version control and GitHub issues and pull requests for project management
- **dune 2.9**: a build system for OCaml; allows us to easily build our OCaml library, run tests in OCaml and auto-formatting OCaml files
- **opam 2.0**: for managing our OCaml dependencies
- **docker**: for providing a standardized environment for building the pocaml compiler and testing
- **make** and **gcc**: for compiling our language built-ins and helper functions written in C

Languages:

- **OCaml 4.13.1**: for implementing our entire compiler front-end and performing code generation using the OCaml bindings for llvm.
- **llvm 11.0**: for generating intermediate representation
- **C**: for implementing language built-ins and helper functions for allocating variables, partial function applications, etc.

4.6 Project Log

Our git log shows a history of 192 commits[3] on branch main starting from October 21 to December 23. A complete history could be accessed on our public GitHub repository for Pocaml[4].

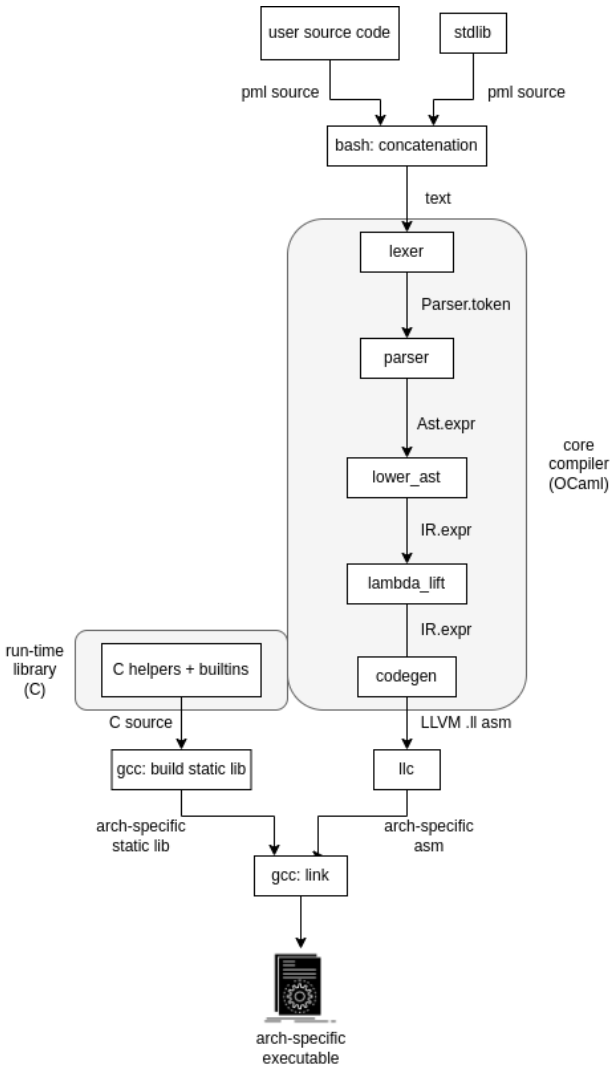

Figure 1: The compiler pipeline

5 Architectural Design

5.1 Compiler Pipeline

The current version of Pocaml compiler is a shellscript that glues various components together. The major components are: core compiler, standard library (stdlib), C run-time library and built-ins. These components are discussed in more details in the following sections.

The overall flow from a user-provided Pocaml source code to an architecture-specific executable is illustrated in figure 1 and described here. When given Pocaml source code by the user, the compiler prepends the user code with `stdlib`, which is simply a Pocaml source code file.

[3] A single commit on main may contain a series of commits since we squash merge feature branch into main.
[4] https://github.com/Pocaml/Pocaml

The resulting source code file is given to the core compiler written in OCaml. The core compiler is structured as distinct passes that takes in a representation of the user program and outputs a transformed representation that maintains its semantics. The passes are sequential in the following order: lexer, parser, lower_ast, lambda_lift, codegen. The output of each pass is fed as input to the next pass, and the output of the last pass (codegen) is a string that is written to standard output (stdout). The lexer pass takes in the program as a string and outputs a list of tokens. The parser pass outputs an abstract syntax tree specified in `Ast.expr`. The lower_ast pass outputs a smaller abstract syntax tree specified in `IR.expr`. The lambda_lift pass performs lambda lifting on the `IR.expr` to enable the language feature of lambda functions[5]. The codegen pass uses the LLVM OCaml bindings to output LLVM assembly to `stdout`.

There is also a C library that provides the Pocaml built-in functions as well as run-time data structures and helpers to enable features like pattern matching, partial application, parametric polymorphism and operator overloading.

At the end of the pipeline, the generated LLVM assembly and the C library are compiled and linked together using `gcc` and `llc` to create the architecture-specific executable.

5.2 Compile-time Language Representations

In the core compiler, there are two abstract syntax trees: AST and IR. The parser generates the AST, and the conversion from AST to IR is carried out in the lower_ast pass. The major differences between the two are the representation of type information and the variations of kinds of expressions.

First, having two expression representations provides convenience when dealing with type information. In parser, type annotations are part of the AST, so AST naturally have type information as a node in the AST tree. Yet, for later passes, it is more convenient to have type information attached to the other expression value nodes, so the lower_ast pass collapses the type annotations nodes onto a single type, which is then attached to each expression node in IR. However, note that this type information is not used by any of the later passes, because static type checker/inference was not a deliverable goal for this semester. (Currently, as a compromise, Pocaml implements a run-time type system to ensure type-safety; we plan on implementing a static type system in the near future)

Second, the IR is a smaller, desugared representation of AST. For example, expressions like if-then-else and pattern-matched function are lowered to lambda and pattern matching; operators are lowered to functions. Most notably, multi-parameter lambdas are lowered to nested single-parameter lambdas. Overall, the point is that the IR is smaller, so is easier to work with in later passes.

[5] https://en.wikipedia.org/wiki/Lambda_lifting

5.3 Run-time Value Representation

The run-time value representation refers to how we represent the Pocaml data types in memory when the executable process is running.

The details of value representation can be found in `builtins/builtins.h`. The mapping from Pocaml type to C type is the following. The `int` type is `int32_t`. The `char` type is `char`. The `unit` type is `int8_t`. The `bool` type is `bool`. The `string` type is `char *`. The list type is a pointer to the `_pml_list` type.

The most interesting one is the representation of lambdas. The lambda type is a pointer to a chunk of memory referred to as a `closure`. The closure enables partial application in Pocaml. It stores all the arguments that have been supplied so far and a reference to the original function that will be called when the required number of argument has been supplied. The memory layout is: function pointer (pointer to the run-time function that carries out the evaluation of the lambda), `int32_t` (number of arguments required), `int32_t` (number of arguments supplied so far), a dynamic array of currently supplied `_pml_val` arguments (the exact definition of `_pml_val` will be discussed next). The C helpers called `_make_closure` and `_apply_closure` facilitates the lambda creation and application in Pocaml. It is worth noting that, when the required number of arguments have been collected, the function is called with the input of a dynamic array that holds all the `_pml_val`. This means that all underlying Pocaml function have the signature `_pml_val f(_pml_val *)`, and arguments are retrieved by indexing into this dynamic array.

With these underlying data types defined, the first version of the `_pml_val` value representation was simply a pointer that points to the underlying data. This had a great property of being uniform across all data types. Despite being less performant (due to the need of dereferencing), this approach ended up being a huge win for us, as the uniform representation saved a substantial amount of development time and effort. At the same time, it enabled the feature of parametric polymorphism by itself without a monomorphization pass in the compiler pipeline.

If Pocaml had a static type system, the pointer-to-data representation would have sufficed. However, due to time constraints, the static type system was not possible. Without type information, operators like equality could not be overloaded across different data types. As a compromise, we opted for a run-time dynamic type system to quickly add type-safety and operator overloading to Pocaml. The resulting second and current version of `_pml_val` is a pointer to the `_pml_val_internal` struct. It is a struct that contained an enum that specifies the type of the value and a union of all Pocaml data types. This allowed us to do run-time type-checking and overload operators.

5.4 Run-time Language Representation and Initialization

The programming model of a Pocaml program is a sequence of top-level let-declarations that are evaluated sequentially. In code generation, this needs to be mapped to the LLVM programming model, which is execution of an entry `main` function.

To achieve this, Pocaml top-level variables are codegen'ed as global LLVM variables of type `_pml_val` that are initialized to `NULL`. A corresponding `init` function would then be codegen'ed, which stores the result of expression evaluation to the global variable. Inside the main function, the `init` functions

are called in the same order as the Pocaml top-level declarations.

5.5 Built-ins and Standard Library

The values defined in built-ins and stdlib are all variables that can be accessed inside the user program. Built-ins are defined in C with access to the internals, and stdlib is simply Pocaml code that are prepended to user program. Note that the operators are also defined as built-ins.

The need for such a distinction is that stdlib is much easier to view and update than built-ins. The guideline is that if the value can be defined in stdlib, then it should be. Built-ins are reserved for values or functions that really require manipulating the internal data representations.

6 Test Plan

6.1 Testing Phases

6.1.1 Unit Testing

For each individual stage of the compiler, we have a designated set of test suites to test it. This unit testing is used primarily during the process of software development, and its purpose is to ensure that the earlier stages are reliable enough for us to move onto developing later stages.

For different stages, we use different testing tools that fit the purpose most closely.

- Lexer, Parser, and AST

  For these stages, we wrote a pretty printing function to display the AST, which allowed us to judge whether the result meets our expectations.

  Without type inference, the output of pretty printing the syntax tree of a simple declaration could get very cumbersome with the type annotation for each expression using paranthesis. To solve this problem, we leveraged OCaml's inline-test library ppx_expect, which allowed us to promote the result of a test case by committing it to the test file. This saved us the effort to actually manually write out the result of pretty printing, which probably would become an impossible task as the test cases get complicated.

  To ensure the correctness of the test cases, we would as a team go through the promoted result and verify its correctness.

  All of our tests for this stage are included in the test/ast directory.

- IR, Lambda Lifting

  These two stages are tested using a separate, but similar methodology as in the AST. We have a separate function for pretty printing the IR, which have the core structure of the Pocaml language. We also utilized the ppx_expect library for easy updates of expected output instead of manually writing out the output ourselves. We used this testing stage to check that the lowering of AST and lambda lifting are both done correctly.

All of our tests for this stage are included in the test/ir directory.

- Builtin C Library

  For the builtin library implemented in the C language, we used C's assert statement to make sure that they are written correctly. Furthermore, we also implemented a debug mode which the user can specify when compiling Pocaml code, allowing the user to view more detailed debug information printed out by the builtin functions.

  All of our tests for this stage are included in the test/builtins directory.

6.1.2 Integration Testing

The integration testing is the most important type of testing for our compiler, because the success of test cases in the integration testing phrase assures us that individual stages are communicating to each other as expected. To this end, we designed the integration test cases most careful and comprehensively, and tried our best to make sure that these test cases cover the entirety of our language.

For end-to-end or integration testing, we couldn't use ppx_expect for these tests as the test cases are no longer interpreted by the OCaml compiler anymore. Instead, we used shell scripts to compare the actual result of the test program and the expected result.

All of our tests for this stage are included in the test/pml directory.

The coverage of the integration tests is summarized as follows:

- Variable and function declarations

  The declaration of variables and functions are tested to make sure that the scoping, value assignment, and partial function application all work properly.

- Control flow

  The if-else-then blocks are tested with different data and the we checked the output to see that the branching is correct.

- Pattern matching

  Since we have a functional language that supports pattern matching, we implemented tests that see the matched arms are executed as expected.

- Operators

  We have arithmetic tests for checking the output from the computations using binary or unitary operators.

- Types

  We support int, boolean, char in our language, and for most test cases for features of the language, we have a copy of the test for each of the primitive types we support.

- Standard library

  Our standard library contains printing functions and list functions such as iter, fold left, fold right, and filter written in Pocaml language, we tested that these functions can be called by tests and compute to the correct results.

6.1.3 System Testing

We have meaningful Pocaml code written to perform non-trivial tasks, such as GCD and DFS. These tests are consider to be special integration tests and are included in the test/pml folder.

6.2 Automation

We have automated shell script that compiles and runs all available integration test suites, and compare the results with the expected results contained in the .out files. The script outputs a .diff file to record the difference if any test case is different from the expected output. After running the test script, there will be a log file containing the relevant information during the testing process.

6.3 Test Suites

There are two type of test suites.

The first type is all test cases that are supposed to pass without any errors and generate some desired output. These cases are begin with test_* and have corresponding .out reference files containing expected output.

The second type is all test cases that are supposed to fail and generate some desired error messages, including both compile time and runtime errors. These cases begin with fail_* and have corresponding .err reference files containing expected error messages.

Please consult the test/pml folder for all tests suites we have. As of now, we have over 50 test cases implemented. The basic coverage of these test suites are described in 6.1.2.

6.4 Pocaml to LLVM Example

In this following example, we demonstrate a single representative test case which is compiled to LLVM. We chose this example because it demonstrates a few interesting features, including function/variable declaration, function application, if-else-then statement.

```
let f1 x = 10 * x
let f2 x = 100 * x
let g = 50
let result = if g > 50 then f2 g else f1 g
```

```
let _ = print_int result
```

The above example is compiled to the LLVM code shown below, after omitting the code generated for builtin functions and standard library functions.

```
define void @_init_f1() {
entry:
  %U145 = call i8* @_make_closure(i8* (i8**)* @U116, i32 1)
  store i8* %U145, i8** @f1, align 8
  ret void
}

define void @_init_f2() {
entry:
  %U146 = call i8* @_make_closure(i8* (i8**)* @U120, i32 1)
  store i8* %U146, i8** @f2, align 8
  ret void
}

define void @_init_g() {
entry:
  %U147 = call i8* @_make_int(i32 50)
  store i8* %U147, i8** @g, align 8
  ret void
}

define void @_init_result() {
entry:
  %_greater_than = load i8*, i8** @_greater_than, align 8
  %g = load i8*, i8** @g, align 8
  %U148 = call i8* @_apply_closure(i8* %_greater_than, i8* %g)
  %U149 = call i8* @_make_int(i32 50)
  %U150 = call i8* @_apply_closure(i8* %U148, i8* %U149)
  %match_val = alloca i8*, align 8
  br label %match2

end_match:                                        ; preds = %arm3, %arm, %no_match
  %match_val6 = load i8*, i8** %match_val, align 8
  store i8* %match_val6, i8** @result, align 8
  ret void

no_match:                                         ; preds = %match
  call void @_pml_error_nonexhaustive_pattern_matching()
  br label %end_match

match:                                            ; preds = %match2
```

```
  %match_result = call i1 @_match_pat_lit_bool(i8* %U150, i1 false)
  br i1 %match_result, label %arm, label %no_match

arm:                                              ; preds = %match
  %f1 = load i8*, i8** @f1, align 8
  %g1 = load i8*, i8** @g, align 8
  %U151 = call i8* @_apply_closure(i8* %f1, i8* %g1)
  store i8* %U151, i8** %match_val, align 8
  br label %end_match

match2:                                           ; preds = %entry
  %match_result4 = call i1 @_match_pat_lit_bool(i8* %U150, i1 true)
  br i1 %match_result4, label %arm3, label %match

arm3:                                             ; preds = %match2
  %f2 = load i8*, i8** @f2, align 8
  %g5 = load i8*, i8** @g, align 8
  %U152 = call i8* @_apply_closure(i8* %f2, i8* %g5)
  store i8* %U152, i8** %match_val, align 8
  br label %end_match
}

define void @_init_U14() {
entry:
  %print_int = load i8*, i8** @print_int, align 8
  %result = load i8*, i8** @result, align 8
  %U153 = call i8* @_apply_closure(i8* %print_int, i8* %result)
  store i8* %U153, i8** @U14, align 8
  ret void
}
```

6.5 Testing Roles

Each team member tested the part that he just finished before merging it to the main branch.

Yunlan and Peter setted up the ppx_expect framework for unit testing and wrote the pretty print function for the AST. Yiming added integration test suites and wrote the pretty print function for IR. Yiming and Yunlan worked on the automated shell script for running the test suites. Leo fixed some of the test suites and debugged the compiler for the errors that integration tests exposed.

7 Lessons Learned

- Leo Qiao:

This project was one of the most interesting and rewarding ones I have done so far. I have learned a great deal from it.

For writing code, fully understanding the inputs and outputs of what needs to be done is crucial. A compiler is a series of transformations, and it really helps to first "be the compiler" yourself and do these transformations by hand using some example inputs. This manual process gives a lot of insights into how the transformation can be automated using code. The resulting outputs can also be used as test cases. In addition, the process of turning these insights into actual code is quite easy (or even intuitive) in functional programming. Things like pure functions, algebraic data types and pattern matching corresponds very well with so many transformations in compiler development.

For architecture design, having abstractions early on was very helpful. For projects like this that start from scratch, many internal designs and representations will change over time. It would be very cumbersome and error-prone to have to change things throughout the entire codebase for a small design change. This abstraction is for not only components of the compiler, but also other things like data representation in the run-time. For example, our uniform run-time representation for different data types allowed us to update it in a limited number of places when we needed run-time type information and also gave us the pleasant surprise of having the property of parametric polymorphism.

For logistics and teamwork, it was great that our team members were in constant communication among each other. We all treated this project as an opportunity to learn from and with each other. I would say that long peer-programming sessions or hackathons were much more productive than the short weekly meetings. In addition, by integrating GitHub into Slack, we were able to get immediate notifications about code changes. When we see that our teammates are pushing code onto GitHub, it gives us the reminder and motivation to work on the project as well.

- Peter Choi:

  Having worked on this project, I am convinced that a fast yet so called "smooth-brain" worker, or one who uses brute force and iteration more than thoughtful deliberation, is likely to be more productive in the long run than a slow and "wrinkly brained" worker. Especially with such smart and increasingly knowledgeable teammates, I was able to save a lot of time by asking them questions and learning from them rather than spending hours trying to find a solution by myself. Of course, this is not to say that thinking ahead and actually designing the thing isn't important, but I found that often times it was much faster to code something, have someone take a look to make sure it was sane, then continue forward rather than attempting to get it right on the first try.

- Yiming Fang

  The most important lesson that I learned from the project is a new perspective rather than any specific skill or technique. I learned to regard programming languages from a critical lens. Before taking PLT, when I write programs in languages like popular C or Java, I considered the languages as rigid rules that I simply need to follow to get my things working. Now I consider language as interesting design problems, implemented by humans just like us. I would now consider the design choices they made, the trade-offs they took, why is their language is good/bad, and how I would approach it if I were the programmer implementing

it. This critical perspective gives me a lot of motivation to dig into the details programming languages and computer programs in general, which I find very enlightening.

Advice: I think time-management is key to the success of this project. Another key is good communication between team members. For our group, I think we got both keys by meeting frequently to discuss and solve problems, and also did a few hackathon-style workshops to get bulks of work done. These meetings and workshops kept us on top of the schedule and alleviated a lot of pressures.

- Yunlan Li:

  I learned that clean code is much more than putting ideas down into code in a easy-to-understand, aesthetic and well-documented way. More importantly, it also lies in how easily the algorithm behind it or in other words the design of the system could be explained to and understood by others. Sure, be happy when you figure out a solution, but also take a moment to ask yourself whether the solution could be easily communicated and better, if there's a more intuitive and straightforward way.

  And functional programming is cool! All those concepts in JavaScript, such as higher ordered functions, lambdas and lexical scope, all made sense to me after learning OCaml and implementing a compiler for a functional programming language.

  Advice: Have your lexer and parser completed as early as possible. Always a good idea to have a MVP first. In the case of compilers, have a working end-to-end compiler that does the absolutely most stupidest thing first, then go back and add in more features.

8 Code Listing

8.1 bin

```
open Pocaml

type action = Ast | IR | Lambda | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist =
    [
      ("-a", Arg.Unit (set_action Ast), "Print the AST");
      ("-i", Arg.Unit (set_action IR), "Print the IR");
      (*("-s", Arg.Unit (set_action Sast), "Print the SAST");*)
      ( "-lambda",
        Arg.Unit (set_action Lambda),
        "Print the IR after lambda-lifting" );
      ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
      ( "-c",
        Arg.Unit (set_action Compile),
        "Check and print the generated LLVM IR (default)" );
    ]
  in
  let usage_msg = "usage: ./main.native [-a|-i|-l|-c] [file.pml]" in
  let channel = ref stdin in
```

```
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let program = Parser.program Lexer.token lexbuf in
  match !action with
  | Ast -> print_string (Print.string_of_program program)
  | IR ->
      print_string
        (Print_ir.string_of_program (Lower_ast.lower_program program))
  | Lambda ->
      program |> Lower_ast.lower_program
              |> Lambda_lift.lambda_lift
              |> Print_ir.string_of_program
              |> print_string
  | _ -> (
      let m =
        program |> Lower_ast.lower_program
                |> Lambda_lift.lambda_lift
                |> Codegen.codegen
      in
      match !action with
      | Ast -> ()
      | IR -> ()
      | Lambda -> ()
      | LLVM_IR -> print_string (Llvm.string_of_llmodule m)
      | Compile ->
          Llvm_analysis.assert_valid_module m;
          print_string (Llvm.string_of_llmodule m))
```

Listing 1: main.ml

8.2 lib

```
type unary_op = Not

type binary_op =
  | PlusOp
  | MinusOp
  | TimesOp
  | DivideOp
  | LtOp
  | LeOp
  | GtOp
  | GeOp
  | EqOp
  | NeOp
  | OrOp
  | AndOp
  | ConsOp
  | SeqOp

(* type variable name *)
type tvar_id = string

(* type constructor name *)
type tcon_id = string

(* data constructor name *)
```

```
type dcon_id = string

(* variable name *)
type var_id = string

(* type annotation *)
type typ =
  | TVar of tvar_id
  | TCon of tcon_id
  | TApp of typ * typ
  | TArrow of typ * typ
  | TNone

type program = Program of definition list

and definition =
  | Def of var_id * param list * typ * expr
  | DefRecFn of var_id * param list * typ * expr

and expr =
  | Lit of literal
  | Var of var_id
  | UnaryOp of unary_op * expr
  | BinaryOp of expr * binary_op * expr
  | Conditional of expr * expr * expr
  | Letin of var_id * expr * expr
  | Lambda of param list * expr
  | Function of (pat * expr) list
  | Apply of expr * expr
  | Match of expr * (pat * expr) list
  | Annotation of expr * typ

and param = ParamAnn of var_id * typ

and literal =
  | LitInt of int
  | LitString of string
  | LitChar of char
  | LitList of expr list
  | LitBool of bool
  | LitUnit

and pat = PatId of var_id | PatLit of literal | PatCons of pat * pat

(* List Cons
   type list_literal =
     | ListCons of expr * list_literal
*)

let extract_program = function Program defs -> defs
```

Listing 2: ast.ml

```
let builtin_names =
  [
    "_not";
    "_add";
    "_minus";
    "_times";
```

```
    "_divide";
    "_less_than";
    "_less_equal";
    "_greater_than";
    "_greater_equal";
    "_equal";
    "_not_equal";
    "_or";
    "_and";
    "_cons";
    "_seq";
    "string_of_int";
    "string_of_bool";
    "string_of_char";
    "print_string";
    "error"
  ]
```

Listing 3: builtins.ml

```
module L = Llvm
open Ir
open Fresh
module StringMap = Map.Make (String)

exception CodegenError of string

let error s = raise (CodegenError s)

let not_implemented () = error "Not implemented"

type env = Env of L.llvalue StringMap.t * env | EnvNone

let print_lltype_of_llvalue llval =
  print_endline (L.string_of_lltype (L.type_of llval))

(* translate : I.program -> Llvm.module *)
let codegen (Program definitions) =
  let context = L.global_context () in
  let the_module = L.create_module context "pocaml" in

  (* Get types from the context *)
  let void_t = L.void_type context in
  let pml_char_t = L.i8_type context in
  let pml_bool_t = L.i1_type context in
  let pml_unit_t = L.i8_type context in
  let pml_int_t = L.i32_type context in
  let pml_string_t = L.pointer_type (L.i8_type context) in
  let pml_val_t = L.pointer_type (L.i8_type context) in
  let pml_func_t = L.function_type pml_val_t [| L.pointer_type pml_val_t |] in
  let pml_init_t = L.function_type void_t [||] in

  (* Define main function and get the builder *)
  let main_func =
    let ftype = L.function_type pml_int_t [||] in
    L.define_function "main" ftype the_module
  in
  let main_builder =
    let entry_block = L.entry_block main_func in
```

```
  L.builder_at_end context entry_block
in

(* Declare helpers *)
let get_arg_f =
  let ftype =
    L.function_type pml_val_t [| L.pointer_type pml_val_t; pml_int_t |]
  in
  L.declare_function "_get_arg" ftype the_module
in

let pml_error_nonexhaustive_pattern_matching =
  let ftype = L.function_type void_t [||] in
  L.declare_function "_pml_error_nonexhaustive_pattern_matching" ftype
    the_module
in

let match_pat_cons_f =
  let ftype = L.function_type pml_bool_t [| pml_val_t |] in
  L.declare_function "_match_pat_cons" ftype the_module
in

let match_pat_cons_end_f =
  let ftype = L.function_type pml_bool_t [| pml_val_t |] in
  L.declare_function "_match_pat_cons_end" ftype the_module
in

let match_pat_lit_int_f =
  let ftype = L.function_type pml_bool_t [| pml_val_t; pml_int_t |] in
  L.declare_function "_match_pat_lit_int" ftype the_module
in

let match_pat_lit_char_f =
  let ftype = L.function_type pml_bool_t [| pml_val_t; pml_char_t |] in
  L.declare_function "_match_pat_lit_char" ftype the_module
in

let match_pat_lit_bool_f =
  let ftype = L.function_type pml_bool_t [| pml_val_t; pml_bool_t |] in
  L.declare_function "_match_pat_lit_bool" ftype the_module
in

let match_pat_lit_string_f =
  let ftype = L.function_type pml_bool_t [| pml_val_t; pml_string_t |] in
  L.declare_function "_match_pat_lit_string" ftype the_module
in

let match_pat_lit_unit_f =
  let ftype = L.function_type pml_bool_t [| pml_val_t; pml_unit_t |] in
  L.declare_function "_match_pat_lit_unit" ftype the_module
in


let match_pat_lit_list_end_f =
  let ftype = L.function_type pml_bool_t [| pml_val_t |] in
  L.declare_function "_match_pat_lit_list_end" ftype the_module
in

let list_get_head_f =
```

```
    let ftype = L.function_type pml_val_t [| pml_val_t |] in
    L.declare_function "_list_get_head" ftype the_module
  in

  let list_get_tail_f =
    let ftype = L.function_type pml_val_t [| pml_val_t |] in
    L.declare_function "_list_get_tail" ftype the_module
  in

  let make_int_f =
    let ftype = L.function_type pml_val_t [| pml_int_t |] in
    L.declare_function "_make_int" ftype the_module
  in

  let make_bool_f =
    let ftype = L.function_type pml_val_t [| pml_bool_t |] in
    L.declare_function "_make_bool" ftype the_module
  in

  let make_char_f =
    let ftype = L.function_type pml_val_t [| pml_char_t |] in
    L.declare_function "_make_char" ftype the_module
  in

  let make_string_f =
    let ftype = L.function_type pml_val_t [| pml_string_t |] in
    L.declare_function "_make_string" ftype the_module
  in

  let make_unit_f =
    let ftype = L.function_type pml_val_t [||] in
    L.declare_function "_make_unit" ftype the_module
  in

  let make_closure_f =
    let ftype =
      L.function_type pml_val_t [| L.pointer_type pml_func_t; pml_int_t |]
    in
    L.declare_function "_make_closure" ftype the_module
  in

  let apply_closure_f =
    let ftype = L.function_type pml_val_t [| pml_val_t; pml_val_t |] in
    L.declare_function "_apply_closure" ftype the_module
  in

  (* Declare the builtins *)
  let pml_empty_list =
    L.declare_global pml_val_t "_pml_empty_list" the_module
  in

  let builtins : L.llvalue StringMap.t =
    let builtin m n =
      let f = L.declare_global pml_val_t n the_module in
      StringMap.add n f m
    in
    List.fold_left builtin StringMap.empty Builtins.builtin_names
  in

```

```
158  let builtins_env : env = Env (builtins, EnvNone) in
159
160  (* Declare the builtin-init function *)
161  let builtins_init : L.llvalue =
162    L.declare_function "_init__builtins" pml_init_t the_module
163  in
164
165  (* Build call in main for the buildin-init function *)
166  let _ = L.build_call builtins_init [||] "" main_builder in
167
168  (* Define top-level values *)
169  let toplevels : L.llvalue StringMap.t =
170    let toplevel m (Def (n, _)) =
171      let v = L.define_global n (L.const_null pml_val_t) the_module in
172      StringMap.add n v m
173    in
174    List.fold_left toplevel StringMap.empty definitions
175  in
176  let toplevels_env : env = Env (toplevels, builtins_env) in
177
178  (* This `lookup` is actually more complicated than just looking up the llvalue.
179     Since LLVM globals are actually pointers to the values, we have to first load
180     the globals into a local variable, and then use it accordingly. However,
181     the current hacky approach actually loads it everytime it is being looked up,
182     so there could be multiple locals variables that point to the same global
     variable.
183     It works, but we should probably think of something better. *)
184  let lookup n builder env =
185    let rec lookup' n f = function
186      | Env (m, parent_env) -> (
187          try StringMap.find n m with Not_found -> lookup' n f parent_env)
188      | EnvNone -> f ()
189    in
190    let lookup_toplevel () =
191      let unfound () = error ("codegen: unbound variable " ^ n) in
192      let global = lookup' n unfound toplevels_env in
193      L.build_load global n builder
194    in
195    lookup' n lookup_toplevel env
196  in
197
198  let add_var_to_scope k v env =
199    let m = StringMap.singleton k v in
200    Env (m, env)
201  in
202
203  (* Function that builds the expression evaluation *)
204  let rec build_expr f builder env = function
205    | Lambda (t, vid, e) ->
206        error "codegen: should not have non-top-level lambdas expr"
207    | Lit (_, lit) -> build_expr_lit builder lit
208    | Var (t, vid) -> (lookup vid builder env, builder)
209    | Letin (t, vid, e1, e2) ->
210        let llval1, builder = build_expr f builder env e1 in
211        let env = add_var_to_scope vid llval1 env in
212        build_expr f builder env e2
213    | Apply (t, e1, e2) ->
214        let llval1, builder = build_expr f builder env e1 in
215        let llval2, builder = build_expr f builder env e2 in
```

```
216         let llval =
217           L.build_call apply_closure_f [| llval1; llval2 |] (fresh_name ())
218             builder
219         in
220         (llval, builder)
221     | Match (t, e, arms) ->
222         (* evaluate e first *)
223         let e', builder = build_expr f builder env e in
224         (* allocate a result variable (build_alloca): pointer to pml_val_t *)
225         let match_val_ptr = L.build_alloca pml_val_t "match_val" builder in
226         (* make a end_match block that load result var into an llvalue and return
    it with new builder *)
227         let end_match_block = L.append_block context "end_match" f in
228         (* recursively make match blocks with append_block *)
229         let rec build_match_arms = function
230           | [] ->
231               (* no match: non-exhaustive pattern matching *)
232               let block = L.append_block context "no_match" f in
233               let builder = L.builder_at_end context block in
234               let _ =
235                 L.build_call pml_error_nonexhaustive_pattern_matching [||] ""
236                   builder
237               in
238               let _ = L.build_br end_match_block builder in
239               block
240           | (pat, arm_e) :: arms ->
241               let next_match_block = build_match_arms arms in
242               let match_block = L.append_block context "match" f in
243               let match_builder = L.builder_at_end context match_block in
244               let arm_block = L.append_block context "arm" f in
245               let arm_builder = L.builder_at_end context arm_block in
246               (* build match *)
247               let match_result =
248                 match pat with
249                 | PatLit (_, lit) -> (
250                     match lit with
251                     | LitInt n ->
252                         let llval = L.const_int pml_int_t n in
253                         L.build_call match_pat_lit_int_f [| e'; llval |]
254                           "match_result" match_builder
255                     | LitChar c ->
256                         let llval = L.const_int pml_char_t (Char.code c) in
257                         L.build_call match_pat_lit_char_f [| e'; llval |]
258                           "match_result" match_builder
259                     | LitString s ->
260                         let llval = L.build_global_stringptr s "" builder in
261                         L.build_call match_pat_lit_string_f [| e'; llval |]
262                           "match_result" match_builder
263                     | LitBool b ->
264                         let llval = L.const_int pml_bool_t (Bool.to_int b) in
265                         L.build_call match_pat_lit_bool_f [| e'; llval |]
266                           "match_result" match_builder
267                     | LitUnit ->
268                         let llval = L.const_int pml_unit_t 69 in
269                         L.build_call match_pat_lit_unit_f [| e'; llval |]
270                           "match_result" match_builder
271                     | LitListEnd ->
272                         L.build_call match_pat_lit_list_end_f [| e' |]
273                           "match_result" match_builder)
```

```
                | PatDefault _ -> L.const_int pml_bool_t (Bool.to_int true)
                | PatCons _ ->
                    L.build_call match_pat_cons_f [| e' |] "match_result"
                      match_builder
                | PatConsEnd _ ->
                    L.build_call match_pat_cons_end_f [| e' |] "match_result"
                      match_builder
              in
              (* conditional jump to arm_block *)
              let _ =
                L.build_cond_br match_result arm_block next_match_block
                  match_builder
              in
              (* build arm *)
              let env =
                match pat with
                | PatLit _ -> env
                | PatDefault (_, vid) -> add_var_to_scope vid e' env
                | PatCons (_, vid1, vid2) ->
                    let e1 =
                      L.build_call list_get_head_f [| e' |] "" arm_builder
                    in
                    let e2 =
                      L.build_call list_get_tail_f [| e' |] "" arm_builder
                    in
                    let env1 = add_var_to_scope vid1 e1 env in
                    let env2 = add_var_to_scope vid2 e2 env1 in
                    env2
                | PatConsEnd (_, vid1) ->
                    let e1 =
                      L.build_call list_get_head_f [| e' |] "" arm_builder
                    in
                    let env1 = add_var_to_scope vid1 e1 env in
                    env1
              in
              let arm_val, arm_builder = build_expr f arm_builder env arm_e in
              let _ = L.build_store arm_val match_val_ptr arm_builder in
              let _ = L.build_br end_match_block arm_builder in
              match_block
        in
        (* jump to the first match block *)
        let first_match_block = build_match_arms arms in
        let _ = L.build_br first_match_block builder in
        let end_match_builder = L.builder_at_end context end_match_block in
        let match_val =
          L.build_load match_val_ptr "match_val" end_match_builder
        in
        (match_val, end_match_builder)
  (* build arm *)
  and build_expr_lit builder = function
    | LitInt n ->
        let llval =
          L.build_call make_int_f
            [| L.const_int pml_int_t n |]
            (fresh_name ()) builder
        in
        (llval, builder)
    | LitChar c ->
        let llval =
```

```
          L.build_call make_char_f
            [| L.const_int pml_char_t (Char.code c) |]
            (fresh_name ()) builder
        in
        (llval, builder)
    | LitString s ->
        let llval = L.build_global_stringptr s (fresh_string_name ()) builder in
        let llval =
          L.build_call make_string_f [| llval |] (fresh_name ()) builder
        in
        (llval, builder)
    | LitBool b ->
        let llval =
          L.build_call make_bool_f
            [| L.const_int pml_bool_t (Bool.to_int b) |]
            (fresh_name ()) builder
        in
        (llval, builder)
    | LitUnit ->
        let llval = L.build_call make_unit_f [||] (fresh_name ()) builder in
        (llval, builder)
    | LitListEnd ->
        let llval = L.build_load pml_empty_list "pml_empty_list" builder in
        (llval, builder)
  in

  (* Define top-level lambda's; keep a dictionary (key: top-level name, value:
    lambda's function definition)*)
  let topfuncs : L.llvalue StringMap.t =
    let topfunc m (Def (n, e)) =
      match e with
      | Lambda (t, _, _) ->
          (* get chaining lambda arguments *)
          let (typed_params, body) : (int * typ * string) list * expr =
            let rec collapsed_lambda' i e =
              match e with
              | Lambda (t, p, body) ->
                  let collapsed_t_p, collapsed_body =
                    collapsed_lambda' (i + 1) body
                  in
                  ((i, t, p) :: collapsed_t_p, collapsed_body)
              | _ -> ([], e)
            in
            collapsed_lambda' 0 e
          in
          (* define the anonymous function *)
          let f = L.define_function (fresh_name ()) pml_func_t the_module in
          let entry_block = L.entry_block f in
          let builder = L.builder_at_end context entry_block in
          let params_ptr = Array.get (L.params f) 0 in
          (* get the parameters into the environment *)
          let params_env =
            let get_param m (i, _, n) =
              let v =
                L.build_call get_arg_f
                  [| params_ptr; L.const_int pml_int_t i |]
                  n builder
              in
              StringMap.add n v m
```

```
            in
            let params_env' =
              List.fold_left get_param StringMap.empty typed_params
            in
            Env (params_env', EnvNone)
          in
          (* build the expression evaluation *)
          let evaluated_expr, builder = build_expr f builder params_env body in
          (* add the return statement *)
          let _ = L.build_ret evaluated_expr builder in
          (* map the top-level lambda name to the anonymous function *)
          StringMap.add n f m
      | _ -> m
    in
    List.fold_left topfunc StringMap.empty definitions
  in

  (* Define top-level init functions; keep a dictionary (key: top-level name,
    value: init's function definition)*)
  let topinits : L.llvalue StringMap.t =
    let topinit m (Def (n, e)) =
      let f = L.define_function ("_init_" ^ n) pml_init_t the_module in
      let entry_block = L.entry_block f in
      let builder = L.builder_at_end context entry_block in
      match e with
      | Lambda _ ->
          let argn =
            let rec argn' = function
              | Lambda (_, _, body) -> 1 + argn' body
              | _ -> 0
            in
            argn' e
          in
          let global = StringMap.find n toplevels in
          let topfunc = StringMap.find n topfuncs in
          let llval =
            L.build_call make_closure_f
              [| topfunc; L.const_int pml_int_t argn |]
              (fresh_name ()) builder
          in
          (* store the llval in the top-level *)
          let _ = L.build_store llval global builder in
          (* return void *)
          let _ = L.build_ret_void builder in
          StringMap.add n f m
      | _ ->
          let global = StringMap.find n toplevels in
          (* build the expression evaluation *)
          let evaluated_expr, builder = build_expr f builder EnvNone e in
          (* store the result in the top-level *)
          let _ = L.build_store evaluated_expr global builder in
          (* return void *)
          let _ = L.build_ret_void builder in
          StringMap.add n f m
    in
    List.fold_left topinit StringMap.empty definitions
  in

  (* Build calls in main for the top-evel init functions *)
```

```
  let () =
    let build_call_init f = ignore (L.build_call f [||] "" main_builder) in
    List.iter
      (fun (Def (n, _)) -> build_call_init (StringMap.find n topinits))
      definitions
  in

  (* Terminate main with a return *)
  let _ = L.build_ret (L.const_int pml_int_t 0) main_builder in

  (* Return the module *)
  the_module
```

Listing 4: codegen.ml

```
let helper (prefix : string) =
  let n = ref 0 in
  fun () -> n := !n + 1; prefix ^ string_of_int !n

let fresh_name = helper "U"

let fresh_lambda_name = helper "L"

let fresh_string_name = helper "S"
```

Listing 5: fresh.ml

```
(* type variable name *)
type tvar_id = string

(* variable name *)
type var_id = string

(* type annotation *)
type typ =
  | TUnit
  | TInt
  | TBool
  | TChar
  | TString
  | TList of typ
  | TVar of tvar_id
  | TArrow of typ * typ
  | TNone

type program = Program of definition list

and definition = Def of var_id * expr

and expr =
  | Lit of typ * literal
  | Var of typ * var_id
  | Letin of typ * var_id * expr * expr
  | Lambda of typ * var_id * expr
  | Apply of typ * expr * expr
  | Match of typ * expr * (pat * expr) list

and literal =
  | LitInt of int
```

```
  | LitChar of char
  | LitString of string
  | LitBool of bool
  | LitUnit
  | LitListEnd

and pat =
  | PatDefault of typ * var_id
  | PatLit of typ * literal
  | PatCons of typ * var_id * var_id
  | PatConsEnd of typ * var_id

let typ_of_expr = function
  | Lit (typ, _) -> typ
  | Var (typ, _) -> typ
  | Letin (typ, _, _, _) -> typ
  | Lambda (typ, _, _) -> typ
  | Apply (typ, _, _) -> typ
  | Match (typ, _, _) -> typ
```

Listing 6: ir.ml

```
open Ir
open Print_ir
open Fresh

exception LambdaLiftError of string

let error s = raise (LambdaLiftError s)

(* helper to extract list of definitions from a program *)
let extract_defs = function Program defs -> defs

(* creates an application of var to a lambda *)
let apply_actual (e : expr) (var : expr) =
  let typ_after_app =
    match typ_of_expr e with
    | TArrow (_, t1) -> t1
    | ty ->
        error
          ("[ Lambda_lift.apply_actual ] apply " ^ string_of_expr var
         ^ " to expr (" ^ string_of_expr e
         ^ ") : expected type TArrow, but has " ^ string_of_typ ty)
  in
  Apply (typ_after_app, e, var)

let add_formal_helper (formal : string) (formal_t : typ) = function
  | (Lambda (_, _, _)) as e ->
      Lambda (TArrow(formal_t, typ_of_expr e), formal, e)
  | e -> Lambda (TArrow (formal_t, typ_of_expr e), formal, e)

(* adds a formal parameter to a lambda to create a new lambda *)
let add_formal (lambda : expr) (var : expr) =
  let formal_t, formal_name =
    match var with
    | Var (ty, n) -> (ty, n)
    | _ ->
        let actual_typ = typ_of_expr var in
        error
```

```
          ("[ Lambda_lift.add_formal ] expected a Var, but "
         ^ string_of_expr var ^ "is of type " ^ string_of_typ actual_typ)
  in
  add_formal_helper formal_name formal_t lambda

(*
   function lift:
   bool -> Ir.program -> Ir.expr list -> Ir.expr -> (Ir.expr * Ir.program)

   switch (bool) determines if a lambda expression should be lifted
    - avoid lifing nested lambdas: we turn off switch when we recursivly
      lift lambda in expr inside a Lambda
    - avoid lifting top-level lambdas: turn off switch when we lift expr inside
      a top-level declaration
*)
let rec lift switch (p : program) (ctx : expr list) = function
  | Lambda (ty, id, e) ->
      let rec get_id_typ = function
        | TArrow(a,b) -> get_id_typ a
        | ty -> ty
      in
      let id_typ = get_id_typ ty in
      let e', p' = lift false p (Var(id_typ, id) :: ctx) e in
      let l = Lambda (ty, id, e') in
      if switch then
        (* add closure as parameters to lambda *)
        let lambda_with_closure = List.fold_left add_formal l ctx in
        let lambda_name = fresh_lambda_name () in
        (* create a global definition *)
        let lifted_lambda_def = Def (lambda_name, lambda_with_closure) in
        let lifted_lambda_var =
          Var (typ_of_expr lambda_with_closure, lambda_name)
        in
        (* substitute original lambda with named function applied with params in
    closure *)
        let new_expr = List.fold_left apply_actual lifted_lambda_var (List.rev ctx
    ) in
        (* update global definition *)
        let defs = match p' with Program ds -> ds in
        (new_expr, Program (lifted_lambda_def :: defs))
      else
        (l, p')
  | Letin (ty, id, e1, e2) ->
      let e1', p1 = lift true p ctx e1 in
      let e2', p2 = lift true p1 (Var (typ_of_expr e1, id) :: ctx) e2 in
      (Letin (ty, id, e1', e2'), p2)
  | Apply (ty, e1, e2) ->
      let e1', p1 = lift true p ctx e1 in
      let e2', p2 = lift true p1 ctx e2 in
      (Apply (ty, e1', e2'), p2)
  | Match (ty, e1, cases) ->
    (*
      takes in
        - an accumalator (program, context, list of match arms seen)
        - next match arm to lift lambdas from
      returns the updated accumulator
    *)
      let lift_lambda_in_case (p, ctx, lst) (pat, e) =
        (* no expr in pat -> don't have to worry about lifting lambdas in pat *)
```

```
        let update_ctx ctx = function
          | PatDefault (ty, id) -> List.rev (Var (ty, id) :: ctx)
          | PatLit (_, _) -> ctx
          | PatCons (ty, id1, id2) ->
              List.rev (Var (ty, id2) :: Var (ty, id1) :: ctx)
          | PatConsEnd (ty, id) -> List.rev (Var (ty, id) :: ctx)
        in
        let ctx' = update_ctx ctx pat in
        let e', p' = lift true p ctx' e in
        (p', ctx, (pat, e') :: lst)
      in
      let e1', p' = lift true p ctx e1 in
      let p'', _, cases' =
        List.fold_left lift_lambda_in_case (p', ctx, []) cases
      in
      (Match (ty, e1', List.rev cases'), p'')
  | e -> (e, p)

(*
   function lambda_lift: Ir.program -> Ir.program
*)
let lambda_lift (p : program) =
  let defs = extract_defs p in
  let lift_def (p : program) (d : definition) =
    let id, e = match d with Def (id, e) -> (id, e) in
    let e', p' = lift false p [] e in
    let defs = extract_defs p' in
    let new_def = Def (id, e') in
    (* defs are in reverse order *)
    Program (new_def :: defs)
  in
  let lifted_program = List.fold_left lift_def (Program []) defs in
  Program (List.rev (extract_defs lifted_program))
```

Listing 7: lambda_lift.ml

```
{
  open Parser

  module L = Lexing

  exception Error

  type token = Parser.token

  let semi_stack = ref [SEQ]

  let push_semi t = semi_stack := t :: !semi_stack

  let pop_semi () = semi_stack := List.tl !semi_stack

  let get_semi () = List.hd !semi_stack
}

let blanks = [' ' '\t' '\r' '\n']+
let digit = ['0'-'9']
let uppercase = ['A'-'Z']
let lowercase = ['a'-'z']
let letter = (uppercase | lowercase)
```

```
let escaped_char = ("\\n" | "\\t" | "\'")
let char_literal = ([^ '\''] | escaped_char )
let string_literal = ([^ '"'] | "\\\"")+
let id_literal = (letter | digit | '_')
(* TODO: fix var_id regex: support 'a *)
let capitalized_ident = uppercase id_literal*
let lowercase_ident = ('_' | (lowercase) id_literal*)
let integer_literal = ['-']? digit (digit | '_')*
let regular_char = [^ '\'' '\\']
let variable_id = lowercase (lowercase | uppercase)*

rule token = parse
  blanks { token lexbuf }
| "(*" { comment [lexbuf.L.lex_start_p] lexbuf }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| "not" { NOT }
| ";"  { get_semi () }
| "list" { LIST }
| "["    { push_semi LSEP; LEFT_BRAC }
| "]"    { pop_semi (); RIGHT_BRAC }
| "("    { push_semi SEQ; LEFT_PAREN }
| ")"    { pop_semi (); RIGHT_PAREN }
| "if" { IF }
| "then" { THEN }
| "else" { ELSE }
| "let" { LET }
| "in"  { IN }
| "fun" { FUN }
| "function" { FUNCTION }
| "rec"   { REC }
| "match" { MATCH }
| "with"  { WITH }
| "|"     { PIPE }
| "->"    { ARROW }
| ":"     { COLON }
| "::"    { CONS }
| "="     { EQ }
| "<"     { LT }
| "<="    { LE }
| ">"     { GT }
| ">="    { GE }
| "&&"    { AND }
| "||"    { OR }
| "true"  { LITBOOL(true)  }
| "false" { LITBOOL(false) }
| integer_literal+ as lit { LITINT(int_of_string lit) }
| '"' (string_literal as lit) '"'  { LITSTRING(lit) }
| '\''(char_literal as lit)'\'' { LITCHAR(lit) }
| lowercase_ident as var { VARIABLE(var) }
| eof { EOF }

and comment level = parse
  "*)" { match level with
        | [_] -> token lexbuf
        | _::level' -> comment level' lexbuf
        | [] -> failwith "bug in comment scanner"
```

```
        }
| "(*" { comment (lexbuf.L.lex_start_p :: level) lexbuf }
| '\n' { L.new_line lexbuf;
         comment level lexbuf
        }
| _    { comment level lexbuf }
| eof  { raise Error }
```

Listing 8: lexer.mll

```
module A = Ast
module I = Ir
module P = Print
open Fresh

exception LowerAstError of string

let error s = raise (LowerAstError s)

let not_implemented () = error "Not implemented"

let rec char_list_of_string = function
  | "" -> []
  | s ->
      String.get s 0
      :: char_list_of_string (String.sub s 1 (String.length s - 1))

let fresh_if_wild = function "_" -> fresh_name () | s -> s

let rec annotate t1 t2 =
  match (t1, t2) with
  | t1, I.TNone -> t1
  | I.TNone, t2 -> t2
  | t1, I.TVar _ -> t1
  | I.TVar _, t2 -> t2
  | I.TBool, I.TBool -> I.TBool
  | I.TChar, I.TChar -> I.TChar
  | I.TInt, I.TInt -> I.TInt
  | I.TString, I.TString -> I.TString
  | I.TUnit, I.TUnit -> I.TUnit
  | I.TArrow (t11, t12), I.TArrow (t21, t22) ->
      I.TArrow (annotate t11 t21, annotate t12 t22)
  | I.TList t1', I.TList t2' -> I.TList (annotate t1' t2')
  | _ -> error "Can't collapse type annotation"

let no_annotation = annotate I.TNone

(* A.program -> I.program *)
let rec lower_program = function
  | A.Program defs -> I.Program (List.map lower_def defs)

and lower_def = function
  | A.Def (avar, aparams, atyp, abody) ->
      I.Def (fresh_if_wild avar, lower_lambda no_annotation aparams atyp abody)
  | A.DefRecFn (avar, aparams, atyp, abody) ->
      I.Def (fresh_if_wild avar, lower_lambda no_annotation aparams atyp abody)

and lower_expr ann = function
  | A.Lit lit -> lower_lit ann lit
```

```
50    | A.Var var_id -> I.Var (ann I.TNone, var_id)
51    | A.UnaryOp (aop, e) -> lower_unary_op ann aop e
52    | A.BinaryOp (e1, aop, e2) -> lower_binary_op ann aop e1 e2
53    | A.Conditional (cond, e1, e2) -> lower_conditional ann cond e1 e2
54    | A.Letin (avar_id, e1, e2) -> lower_letin ann avar_id e1 e2
55    | A.Lambda (ps, e) -> lower_lambda ann ps A.TNone e
56    | A.Function arms -> lower_function ann arms
57    | A.Apply (e1, e2) -> lower_apply ann e1 e2
58    | A.Match (e, arms) -> lower_match ann e arms
59    | A.Annotation (e, t) -> lower_expr (annotate (ann (lower_typ t))) e
60
61  and lower_function ann arms =
62    let n = fresh_name () in
63    let match' = lower_match no_annotation (A.Var n) arms in
64    I.Lambda
65      ( ann I.TNone,
66        n,
67        match')
68
69  and lower_unary_op ann aop e =
70    I.Apply
71      ( ann I.TNone,
72        I.Var (I.TNone, P.string_of_unop aop),
73        lower_expr no_annotation e )
74
75  and lower_binary_op ann aop e1 e2 =
76    I.Apply
77      ( ann I.TNone,
78        I.Apply
79          ( I.TNone,
80            I.Var (I.TNone, P.string_of_binop aop),
81            lower_expr no_annotation e1 ),
82        lower_expr no_annotation e2 )
83
84  and lower_conditional ann cond e1 e2 =
85    I.Match
86      ( ann I.TNone,
87        lower_expr no_annotation cond,
88        [
89          (I.PatLit (I.TNone, I.LitBool true), lower_expr no_annotation e1);
90          (I.PatLit (I.TNone, I.LitBool false), lower_expr no_annotation e2);
91        ] )
92
93  and lower_letin ann var_id e1 e2 =
94    I.Letin
95      ( ann I.TNone,
96        fresh_if_wild var_id,
97        lower_expr no_annotation e1,
98        lower_expr no_annotation e2 )
99
100 and lower_apply ann e1 e2 =
101   I.Apply (ann I.TNone, lower_expr no_annotation e1, lower_expr no_annotation e2)
102
103 and lower_lambda ann aparams aoutput_typ abody =
104   let rec get_lambda_ityp = function
105     | [] -> lower_typ aoutput_typ
106     | A.ParamAnn (_, atyp) :: ps -> I.TArrow (lower_typ atyp, get_lambda_ityp ps)
107   in
108   let lambda_ityp = get_lambda_ityp aparams in
```

```
109   match aparams with
110   | [] -> lower_expr (annotate (ann lambda_ityp)) abody
111   | ParamAnn (avar_id, _) :: aps ->
112       I.Lambda
113         ( lambda_ityp,
114           fresh_if_wild avar_id,
115           lower_lambda no_annotation aps aoutput_typ abody )
116
117 and lower_match ann e arms =
118   let typ' = ann I.TNone in
119   let e' = lower_expr no_annotation e in
120   let lower_arm = function
121     | arm_pat, arm_e -> (lower_pat arm_pat, lower_expr no_annotation arm_e)
122   in
123   let arms' = List.map lower_arm arms in
124   I.Match (typ', e', arms')
125
126 and lower_pat =
127   let lower_literal = function
128     | A.LitInt i -> I.LitInt i
129     | A.LitChar c -> I.LitChar c
130     | A.LitBool b -> I.LitBool b
131     | A.LitList [] -> I.LitListEnd
132     | A.LitList _ -> error "Can't lower pattern matching on non-emtpy list"
133     | A.LitString s -> I.LitString s
134     | A.LitUnit -> I.LitUnit
135   in
136   function
137   | A.PatId avar_id -> I.PatDefault (I.TNone, fresh_if_wild avar_id)
138   | A.PatLit lit -> I.PatLit (I.TNone, lower_literal lit)
139   | A.PatCons (pat1, pat2) -> (
140       match (pat1, pat2) with
141       | A.PatId avar_id1, A.PatId avar_id2 ->
142           I.PatCons (I.TNone, fresh_if_wild avar_id1, fresh_if_wild avar_id2)
143       | A.PatId avar_id1, A.PatLit (A.LitList []) ->
144           I.PatConsEnd (I.TNone, fresh_if_wild avar_id1)
145       | _ -> error "Can't lower recursive patterns yet")
146
147 and lower_lit ann alit =
148   match alit with
149   | A.LitInt i -> I.Lit (ann I.TNone, I.LitInt i)
150   | A.LitChar c -> I.Lit (ann I.TNone, I.LitChar c)
151   | A.LitBool b -> I.Lit (ann I.TNone, I.LitBool b)
152   | A.LitList [] -> I.Lit (ann I.TNone, I.LitListEnd)
153   | A.LitList (e :: es) ->
154       let outerCons x =
155         I.Apply
156           ( ann I.TNone,
157             I.Apply
158               ( I.TNone,
159                 I.Var (I.TNone, Print.string_of_binop ConsOp),
160                 lower_expr no_annotation e ),
161             x )
162       in
163       let innerCons =
164         List.fold_right
165           (fun e l ->
166             I.Apply
167               ( I.TNone,
```

```
                I.Apply
                  (I.TNone, I.Var (I.TNone, Print.string_of_binop ConsOp), e),
                l ))
          (List.map (lower_expr no_annotation) es)
          (I.Lit (I.TNone, I.LitListEnd))
      in
      outerCons innerCons
  | A.LitString s -> I.Lit (ann I.TNone, I.LitString s)
  | A.LitUnit -> I.Lit (ann I.TNone, I.LitUnit)

and lower_typ = function
  | A.TVar tvar_id -> I.TVar tvar_id
  | A.TCon "int" -> I.TInt
  | A.TCon "bool" -> I.TBool
  | A.TCon "char" -> I.TChar
  | A.TCon "()" -> I.TUnit
  | A.TCon "string" -> I.TString
  | A.TApp (A.TCon "list", t) -> I.TList (lower_typ t)
  | A.TArrow (t1, t2) -> I.TArrow (lower_typ t1, lower_typ t2)
  | A.TNone -> I.TNone
  | A.TCon _ -> error "Can't lower any TCon besides built-in types"
  | A.TApp _ -> error "Can't lower any TApp besides list"
```

Listing 9: lower_ast.ml

```
%{
    open Ast

    let unescape s =
      Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun u -> u)

    let unescape_char s =
      let unescaped_s = unescape s in
      String.get unescaped_s 0
%}

%token EOF
%token <int> LITINT
%token <string> LITSTRING
%token <string> LITCHAR
%token <bool> LITBOOL
%token <string> VARIABLE
%token LIST LEFT_BRAC RIGHT_BRAC
%token NOT
%token PLUS MINUS TIMES DIVIDE LT LE GT GE EQ NE OR AND CONS
%token IF THEN ELSE
%token LET IN
%token FUN REC FUNCTION
%token MATCH WITH PIPE
%token ARROW
%token LEFT_PAREN RIGHT_PAREN
%token COLON
%token SEQ LSEP

%nonassoc LET IN FUN MATCH WITH ARROW FUNCTION
%left PIPE
%left SEQ
%right LSEP
%nonassoc IF THEN ELSE
```

```
%right OR AND
%left LT LE GT GE EQ NE
%right CONS
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc NOT

%start program
%type <Ast.program> program

%%

program:
  defs EOF { Program ($1) }

defs:
    /* nothing */    { [] }
  | def defs         { $1 :: $2 }

def:
    LET LEFT_PAREN VARIABLE COLON typ RIGHT_PAREN EQ expr    { Def($3, [], $5, $8) }
  | LET VARIABLE params_opt COLON typ EQ expr                { Def($2, $3, $5, $7) }
  | LET VARIABLE params_opt EQ expr                          { Def($2, $3, TNone, $5) }
  | LET REC VARIABLE params_opt COLON typ EQ expr            { DefRecFn($3, $4, $6, $8) }
  | LET REC VARIABLE params_opt EQ expr                      { DefRecFn($3, $4, TNone, $6)}

params_opt:
    /* no param */  { [] }
  | params          { $1 }

params:
    LEFT_PAREN VARIABLE COLON typ RIGHT_PAREN           { [ ParamAnn($2, $4) ] }
  | LEFT_PAREN VARIABLE COLON typ RIGHT_PAREN params    { ParamAnn($2, $4) :: $6 }
  | VARIABLE                                            { [ ParamAnn($1, TNone) ] }
  | VARIABLE params                                     { ParamAnn($1, TNone) :: $2 }

typ:
    VARIABLE LIST { TApp(TCon("list"), TVar($1)) }
  | VARIABLE      { TVar($1) }
  | LEFT_PAREN RIGHT_PAREN { TCon("()") }
  | typ ARROW typ { TArrow($1, $3) }

literal:
  | LEFT_BRAC list_literal RIGHT_BRAC   { LitList($2) }
  | LITINT                              { LitInt($1) }
  | LITBOOL                             { LitBool($1) }
  | LITSTRING                           { LitString(unescape($1)) }
  | LITCHAR                             { LitChar(unescape_char($1)) }
  | LEFT_PAREN RIGHT_PAREN              { LitUnit }

list_literal:
  | /* nothing */           { [] }
```

```
  | expr                    { [$1] }
  | expr LSEP list_literal  { $1 :: $3 }

apply:
    apply atom { Apply($1, $2) }
  | atom { $1 }

atom:
    literal { Lit($1) }
  | var { $1 }
  | LEFT_PAREN expr COLON typ RIGHT_PAREN { Annotation($2, $4) }
  | LEFT_PAREN expr RIGHT_PAREN { $2 }

expr:
    NOT expr         { UnaryOp(Not, $2) }
  | expr PLUS   expr { BinaryOp($1, PlusOp, $3) }
  | expr MINUS  expr { BinaryOp($1, MinusOp, $3) }
  | expr TIMES  expr { BinaryOp($1, TimesOp, $3) }
  | expr DIVIDE expr { BinaryOp($1, DivideOp, $3) }
  | expr LT expr { BinaryOp($1, LtOp, $3) }
  | expr LE expr { BinaryOp($1, LeOp, $3) }
  | expr GT expr { BinaryOp($1, GtOp, $3) }
  | expr GE expr { BinaryOp($1, GeOp, $3) }
  | expr EQ expr { BinaryOp($1, EqOp, $3) }
  | expr NE expr { BinaryOp($1, NeOp, $3) }
  | expr OR expr { BinaryOp($1, OrOp, $3) }
  | expr AND expr { BinaryOp($1, AndOp, $3) }
  | expr CONS expr { BinaryOp($1, ConsOp, $3) }
  | expr SEQ expr { BinaryOp($1, SeqOp, $3) }
  | IF expr THEN expr ELSE expr { Conditional($2, $4, $6) }
  | LET VARIABLE EQ expr IN expr { Letin($2, $4, $6) }
  | MATCH expr WITH match_arms { Match($2, $4) }
  | FUN params ARROW expr { Lambda($2, $4) }
  | FUNCTION match_arms { Function($2) }
  | apply { $1 }

var:
  VARIABLE         { Var($1) }

match_arms:
  | match_arms_                    { $1 }
  | PIPE match_arms_               { $2 }

match_arms_:
  | pat ARROW expr                 { [($1, $3)] }
  | pat ARROW expr PIPE match_arms_ { ($1, $3) :: $5 }

pat:
  | pat_                          { $1 }
  | LEFT_PAREN pat_ RIGHT_PAREN    { $2 }

pat_:
    VARIABLE  { PatId($1) }
  | literal   { PatLit($1) }
  | pat CONS pat  { PatCons($1, $3) }
```

Listing 10: parser.mly

```
open Ir
```

```
2
3
4  exception PrintIrError of string
5  let error s = raise (PrintIrError s)
6
7  let rec string_of_typ = function
8    | TUnit -> "unit"
9    | TInt -> "int"
10   | TBool -> "bool"
11   | TChar -> "char"
12   | TString -> "string"
13   | TList typ -> string_of_typ typ
14   | TVar tvar_id -> tvar_id
15   | TArrow (t1, t2) -> string_of_typ t1 ^ " -> " ^ string_of_typ t2
16   | TNone -> "None"
17
18
19 let annotate typ id =
20   let type_str = string_of_typ typ in match type_str with
21   | "" -> id
22   | typ_str -> "( " ^ id ^ " : " ^ typ_str ^ " )"
23
24
25 let rec string_of_expr = function
26   | Lit (typ, literal) -> annotate typ (string_of_lit literal)
27   | Var (typ, id) -> annotate typ id
28   | Letin (typ, id, e1, e2) ->
29       "( let " ^ annotate typ id ^ " = " ^ string_of_expr e1 ^ " in " ^
     string_of_expr e2
30       ^ " )"
31   | Lambda (typ, var_id, e) ->
32       annotate typ ("( fun " ^ var_id ^ " -> " ^ string_of_expr e ^ " )")
33   | Apply (typ, e1, e2) -> annotate typ ("( " ^ string_of_expr e1 ^ " " ^
     string_of_expr e2 ^ " )")
34   | Match (typ, e, lst) ->
35       annotate typ ("(\n match " ^ string_of_expr e ^ " with\n" ^
     string_of_match_arms lst ^ "\n)")
36
37 and string_of_lit = function
38   | LitInt int -> string_of_int int
39   | LitBool bool -> string_of_bool bool
40   | LitChar char -> "\'" ^ String.make 1 char ^ "\'"
41   | LitString string -> "\"" ^ string ^ "\""
42   | LitUnit -> "()"
43   | LitListEnd -> "[]"
44
45 and string_of_pattern = function
46   | PatDefault (typ, id) -> annotate typ id
47   | PatLit (typ, lit) -> annotate typ (string_of_lit lit)
48   | PatCons (typ, id1, id2) ->
49       annotate typ ("( " ^ id1 ^ " :: " ^ id2 ^ " )")
50   | PatConsEnd (typ, id) -> annotate typ id
51
52 and string_of_match_arm (pat, e) =
53   "|  " ^ string_of_pattern pat ^ " -> " ^ string_of_expr e
54
55 and string_of_match_arms arms =
56   String.concat "\n" (List.map string_of_match_arm arms)
57
```

```
let string_of_def = function
  | Def (var_id, expr) ->
     (* let type_str = string_of_typ typ in
      let formatted_type_str = if type_str = "" then "" else " : " ^ type_str in
      let params_str = string_of_params params_opt in
      let formatted_params_str =
        if params_str = "" then "" else " " ^ params_str
      in*)
      ("let " ^ var_id ^ " = " ^ string_of_expr expr)

let string_of_program = function
  | Program defs -> String.concat "\n" (List.map string_of_def defs)

let print_prog = function
  | str ->
      let lexbuf = Lexing.from_string str in
      let prog = Parser.program Lexer.token lexbuf in
      print_endline (string_of_program(Lower_ast.lower_program(prog)))
```

Listing 11: print_ir.ml

```
open Ast

let rec string_of_typ = function
  | TVar tvar_id -> tvar_id
  | TCon tcon_id -> tcon_id
  | TApp (_, _) -> "TApp"
  | TArrow (t1, t2) -> string_of_typ t1 ^ " -> " ^ string_of_typ t2
  | TNone -> ""

let string_of_param = function
  | ParamAnn (var_id, typ) ->
      let typ_string = string_of_typ typ in
      if typ_string = "" then var_id
      else "( " ^ var_id ^ " : " ^ typ_string ^ " )"

let string_of_params = function
  | params -> String.concat " " (List.map string_of_param params)

let string_of_unop = function Not -> "_not"

let string_of_binop = function
  | PlusOp -> "_add"
  | MinusOp -> "_minus"
  | TimesOp -> "_times"
  | DivideOp -> "_divide"
  | LtOp -> "_less_than"
  | LeOp -> "_less_equal"
  | GtOp -> "_greater_than"
  | GeOp -> "_greater_equal"
  | EqOp -> "_equal"
  | NeOp -> "_not_equal"
  | OrOp -> "_or"
  | AndOp -> "_and"
  | ConsOp -> "_cons"
  | SeqOp -> "_seq"

let rec string_of_expr = function
  | Lit literal -> string_of_lit literal
```

```
39    | Var id -> id
40    | UnaryOp (op, e) -> string_of_unop op ^ " " ^ string_of_expr e
41    | BinaryOp (e1, binop, e2) ->
42        "( " ^ string_of_expr e1 ^ " " ^ string_of_binop binop ^ " "
43        ^ string_of_expr e2 ^ " )"
44    | Conditional (e1, e2, e3) ->
45        let pred = string_of_expr e1 in
46        let br1 = string_of_expr e2 in
47        let br2 = string_of_expr e3 in
48        "( if " ^ pred ^ " then " ^ br1 ^ " else " ^ br2 ^ " )"
49    | Letin (id, e1, e2) ->
50        "( let " ^ id ^ " = " ^ string_of_expr e1 ^ " in " ^ string_of_expr e2
51        ^ " )"
52    | Lambda (params, e) ->
53        "( fun " ^ string_of_params params ^ " = " ^ string_of_expr e ^ " )"
54    | Function arms ->
55        "(\n function " ^ string_of_match_arms arms ^ "\n)"
56    | Apply (e1, e2) -> "( " ^ string_of_expr e1 ^ " " ^ string_of_expr e2 ^ " )"
57    | Match (e, lst) ->
58        "(\n match " ^ string_of_expr e ^ " with\n" ^ string_of_match_arms lst
59        ^ "\n)"
60    | Annotation (expr, typ) ->
61        let expr_string = string_of_expr expr in
62        let typ_string = string_of_typ typ in
63        "( " ^ expr_string ^ " : " ^ typ_string ^ " )"
64
65  and string_of_lit = function
66    | LitInt int -> string_of_int int
67    | LitString str -> "\"" ^ str ^ "\""
68    | LitBool bool -> string_of_bool bool
69    | LitChar char -> "\'" ^ String.make 1 char ^ "\'"
70    | LitList list -> "[" ^ String.concat ";" (List.map string_of_expr list) ^ "]"
71    | LitUnit -> "()"
72
73  and string_of_pattern = function
74    | PatId id -> id
75    | PatLit lit -> string_of_lit lit
76    | PatCons (p1, p2) ->
77        "( " ^ string_of_pattern p1 ^ " :: " ^ string_of_pattern p2 ^ " )"
78
79  and string_of_match_arm (pat, e) =
80    "|  " ^ string_of_pattern pat ^ " -> " ^ string_of_expr e
81
82  and string_of_match_arms arms =
83    String.concat "\n" (List.map string_of_match_arm arms)
84
85  let string_of_def = function
86    | Def (var_id, params_opt, typ, expr) ->
87        let type_str = string_of_typ typ in
88        let formatted_type_str = if type_str = "" then "" else " : " ^ type_str in
89        let params_str = string_of_params params_opt in
90        let formatted_params_str =
91          if params_str = "" then "" else " " ^ params_str
92        in
93        "let " ^ var_id ^ formatted_params_str ^ formatted_type_str ^ " = "
94        ^ string_of_expr expr
95    | DefRecFn (var_id, params_opt, typ, expr) ->
96        let type_str = string_of_typ typ in
97        let formatted_type_str = if type_str = "" then "" else " : " ^ type_str in
```

```
      let params_str = string_of_params params_opt in
      let formatted_params_str =
        if params_str = "" then "" else " " ^ params_str
      in
      "let rec " ^ var_id ^ formatted_params_str ^ formatted_type_str ^ " = "
      ^ string_of_expr expr

let string_of_program = function
  | Program defs -> String.concat "\n" (List.map string_of_def defs)

let print_prog = function
  | str ->
      let lexbuf = Lexing.from_string str in
      let prog = Parser.program Lexer.token lexbuf in
      print_endline (string_of_program prog)
```

Listing 12: print.ml

8.3 builtins

```
let string_of_int : int -> string = fun _ -> ""
let string_of_char : char -> string = fun _ -> ""
let string_of_bool : bool -> string = fun _ -> ""
let string_of_ : int -> string = fun _ -> ""
let string_of_int : int -> string = fun _ -> ""
let print_string : string -> unit = fun _ -> ()
let error : string -> unit = fun _ -> ()
```

Listing 13: builtins.ml

```
#ifndef _POCAML_BUILTINS_H_
#define _POCAML_BUILTINS_H_

#include <stdint.h>
#include <stdbool.h>

/* pocaml primitives */

typedef enum {
PML_CHAR,
PML_BOOL,
PML_UNIT,
PML_INT,
PML_STRING,
PML_LIST,
PML_CLOSURE
} _pml_type;

typedef   char      _pml_char;
typedef   bool    _pml_bool;
typedef   int8_t    _pml_unit;
typedef   int32_t   _pml_int;
typedef   _pml_char *_pml_string;
typedef struct _pml_val_internal
{
  _pml_type type;
  union {
    _pml_char c;
```

```
    _pml_bool b;
    _pml_unit u;
    _pml_int i;
    _pml_string s;
    void *l;
    void *closure;
  };
} _pml_val_internal;
typedef   _pml_val_internal   *_pml_val;
typedef   _pml_val  _pml_func(_pml_val*);
typedef struct _pml_list {
  _pml_val data;
  struct _pml_list *next;
} _pml_list;

typedef void _pml_init(void);

/* type related helpers */
_pml_char _pml_get_char(_pml_val);
_pml_bool _pml_get_bool(_pml_val);
_pml_unit _pml_get_unit(_pml_val);
_pml_int _pml_get_int(_pml_val);
_pml_string _pml_get_string(_pml_val);
_pml_list *_pml_get_list(_pml_val);
void *_pml_get_closure(_pml_val);

/* closure */
typedef struct _pml_closure_md
{
  _pml_func *fp;
  _pml_int required;
  _pml_int supplied;
} _pml_closure_md;

#define _closure_fp(closure) (((_pml_closure_md *)(closure))->fp)
#define _set_closure_fp(closure, f_ptr) (((_pml_closure_md *)(closure))->fp = (
    f_ptr))
#define _closure_required(closure) (((_pml_closure_md *)(closure))->required)
#define _set_closure_required(closure, n) (((_pml_closure_md *)(closure))->
    required = (n))
#define _closure_supplied(closure) (((_pml_closure_md *)(closure))->supplied)
#define _set_closure_supplied(closure, n) (((_pml_closure_md *)(closure))->
    supplied = (n))
#define _start_of_args_in_closure(closure) ((_pml_val *)((_pml_closure_md *)(
    closure) + 1))
#define _closure_size(closure) (sizeof(_pml_closure_md) + sizeof(_pml_val) * (((
    _pml_closure_md *)(closure))->supplied))
#define _closure_size_with_args(n) (sizeof(_pml_func *) + 2 * sizeof(_pml_int) +
    sizeof(_pml_val) * ((n)-1))

_pml_val _make_closure(_pml_func *fp, _pml_int num_args);
_pml_val _apply_closure(_pml_val closure, _pml_val arg);

/* primitives */
_pml_val _make_int(_pml_int n);
_pml_val _make_bool(_pml_bool b);
_pml_val _make_char(_pml_char c);
_pml_val _make_string(_pml_char *s);
_pml_val _make_int(_pml_int n);
```

```
_pml_val _make_unit();
_pml_val _make_list(_pml_val data, _pml_val next_list);

extern _pml_val _pml_empty_list;

/* pml helpers */
_pml_val _get_arg(_pml_val *params, _pml_int i);
_pml_bool _match_pat_cons(_pml_val val);
_pml_bool _match_pat_cons_end(_pml_val val);
_pml_bool _match_pat_lit_int(_pml_val val, _pml_int pat);
_pml_bool _match_pat_lit_char(_pml_val val, _pml_char pat);
_pml_bool _match_pat_lit_bool(_pml_val val, _pml_bool pat);
_pml_bool _match_pat_lit_string(_pml_val val, _pml_string pat);
_pml_bool _match_pat_lit_unit(_pml_val val, _pml_unit pat);
_pml_bool _match_pat_lit_list_end(_pml_val val);
_pml_val _list_get_head(_pml_val val);
_pml_val _list_get_tail(_pml_val val);
void _pml_error(_pml_string s);
void _pml_error_nonexhaustive_pattern_matching(void);

/* builtins */
_pml_func _builtin__add;
extern _pml_val _add;
_pml_init _init__add;

_pml_func _builtin__minus;
extern _pml_val _minus;
_pml_init _init__minus;

_pml_func _builtin__times;
extern _pml_val _times;
_pml_init _init__times;

_pml_func _builtin__divide;
extern _pml_val _divide;
_pml_init _init__divide;

_pml_func _builtin__less_than;
extern _pml_val _less_than;
_pml_init _init__less_than;

_pml_func _builtin__less_equal;
extern _pml_val _less_equal;
_pml_init _init__less_equal;

_pml_func _builtin__greater_than;
extern _pml_val _greater_than;
_pml_init _init__greater_than;

_pml_func _builtin__greater_equal;
extern _pml_val _greater_equal;
_pml_init _init__greater_equal;

_pml_func _builtin__equal;
extern _pml_val _equal;
_pml_init _init__equal;

_pml_func _builtin__not_equal;
extern _pml_val _not_equal;
```

```
_pml_init _init__not_equal;

_pml_func _builtin__or;
extern _pml_val _or;
_pml_init _init__or;

_pml_func _builtin__and;
extern _pml_val _and;
_pml_init _init__and;

_pml_func _builtin__cons;
extern _pml_val _cons;
_pml_init _init__cons;

_pml_func _builtin__seq;
extern _pml_val _seq;
_pml_init _init__seq;

_pml_func _builtin_string_of_int;
extern _pml_val string_of_int;
_pml_init _init_string_of_int;

_pml_func _builtin_string_of_char;
extern _pml_val string_of_char;
_pml_init _init_string_of_char;

_pml_func _builtin_string_of_bool;
extern _pml_val string_of_bool;
_pml_init _init_string_of_bool;

_pml_func _builtin_print_string;
extern _pml_val print_string;
_pml_init _init_print_string;

_pml_func _builtin_error;
extern _pml_val error;
_pml_init _init_error;

_pml_init _init__builtins;

#endif
```

Listing 14: builtins.h

```
#include <stdlib.h>
#include <stdio.h>
#include "builtins.h"


_pml_val _add;

_pml_val _builtin__add(_pml_val *args)
{
  _pml_val left, right;

  left = (_pml_val) args[0];
  right = (_pml_val) args[1];

  _pml_int res = _pml_get_int(left) + _pml_get_int(right);
```

```
16
17    return _make_int(res);
18  }
19
20  void _init__add()
21  {
22    _add = _make_closure(_builtin__add, 2);
23  }
```

Listing 15: _add.c

```
1   #include <stdlib.h>
2   #include "builtins.h"
3
4   _pml_val _and;
5
6   _pml_val _builtin__and(_pml_val *args)
7   {
8     _pml_val left, right;
9
10    left = (_pml_val) args[0];
11    right = (_pml_val) args[1];
12
13    _pml_bool res = _pml_get_bool(left) && _pml_get_bool(right);
14
15    return _make_bool(res);
16  }
17
18  void _init__and()
19  {
20    _and = _make_closure(_builtin__and, 2);
21  }
```

Listing 16: _and.c

```
1   #include <assert.h>
2   #include <stdio.h>
3   #include <string.h>
4   #include <stdlib.h>
5   #include "builtins.h"
6
7
8   void *_dup_closure(void *closure)
9   {
10    _pml_int num_args, dup_size;
11    void *dup_closure;
12
13    num_args = _closure_required(closure);
14    dup_size = _closure_size(closure);
15
16      dup_closure = malloc(_closure_size_with_args(num_args));
17
18    memcpy(dup_closure, closure, dup_size);
19
20    return dup_closure;
21  }
22
23  void _add_arg_to_closure(void *closure, _pml_val arg)
24  {
```

```
  _pml_int supplied = _closure_supplied(closure);
  _start_of_args_in_closure(closure)[supplied] = arg;
}

_pml_val _apply_closure(_pml_val _closure, _pml_val arg)
{
  /*
   * if have the required # args -> apply func with args, return result
   * else -> duplicate a closure, add arg to the closure, update supplied
  */

#ifdef BUILTIN_DEBUG
  printf("[debug] _apply_closure\n");
#endif

  void *closure = _pml_get_closure(_closure);

  _pml_val ret;
  _pml_int required = _closure_required(closure);
  _pml_int supplied = _closure_supplied(closure);
  _pml_func *fn = _closure_fp(closure);

  if (required == supplied + 1) {
    /* get the arguments and store in an array */
    _pml_val *args = malloc(sizeof(_pml_val) * required);
    _pml_int args_size = sizeof(_pml_val) * supplied;

    memcpy(args, _start_of_args_in_closure(closure), args_size);
    args[required - 1] = arg;

    ret = fn(args);
    free(args);
  } else {
    void *dup_c = _dup_closure(closure);
    _add_arg_to_closure(dup_c, arg);
    _set_closure_supplied(dup_c, supplied + 1);
    ret = malloc(sizeof(_pml_val_internal));
    ret->type = PML_CLOSURE;
    ret->closure = dup_c;
  }

  return ret;
}
```

Listing 17: _apply_closure.c

```
#include <stdlib.h>
#include "builtins.h"

_pml_val _cons;

_pml_val _builtin__cons(_pml_val *args)
{
  _pml_val left, right;

  left = (_pml_val) args[0];
  right = (_pml_val) args[1];

  _pml_val res = _make_list(left, right);
```

```
14
15    return res;
16  }
17
18  void _init__cons()
19  {
20    _cons = _make_closure(_builtin__cons, 2);
21  }
```

Listing 18: _cons.c

```
1  #include <stdlib.h>
2  #include "builtins.h"
3
4
5  _pml_val _divide;
6
7  _pml_val _builtin__divide(_pml_val *args) {
8    _pml_val left, right;
9
10   left = (_pml_val) args[0];
11   right = (_pml_val) args[1];
12
13   _pml_int res = _pml_get_int(left) / _pml_get_int(right);
14
15   return _make_int(res);
16 }
17
18 void _init__divide() {
19     _divide = _make_closure(_builtin__divide, 2);
20 }
```

Listing 19: _divide.c

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include "builtins.h"
4
5  _pml_val _equal;
6
7  _pml_val _builtin__equal(_pml_val *args)
8  {
9    _pml_val left, right;
10
11   left = (_pml_val) args[0];
12   right = (_pml_val) args[1];
13
14   _pml_bool res;
15   switch (left->type) {
16     case PML_CHAR:
17       res = _pml_get_char(left) == _pml_get_char(right);
18       return _make_bool(res);
19     case PML_BOOL:
20       res = _pml_get_bool(left) == _pml_get_bool(right);
21       return _make_bool(res);
22     case PML_UNIT:
23       res = _pml_get_unit(left) == _pml_get_unit(right);
24       return _make_bool(res);
25     case PML_INT:
```

```
      res = _pml_get_int(left) == _pml_get_int(right);
      return _make_bool(res);
    case PML_STRING:
      res = 0 == strcmp(_pml_get_string(left), _pml_get_string(right));
      return _make_bool(res);
    default:
      _pml_error("This type does not support equality operator");
  }

  _pml_error("This type does not support equality operator");
  return _make_int(420);
}

void _init__equal()
{
  _equal = _make_closure(_builtin__equal, 2);
}
```

Listing 20: _equal.c

```
#include <stdio.h>
#include <stdlib.h>
#include "builtins.h"

void _pml_error(_pml_string s) {
    fprintf(stderr, "%s\n", s);
    exit(EXIT_FAILURE);
}

void _pml_error_nonexhaustive_pattern_matching() {
    _pml_error("Non-exhaustive pattern matching");
}
```

Listing 21: _error.c

```
#include "builtins.h"

_pml_val _get_arg(_pml_val *params, _pml_int i) {
    return params[i];
}
```

Listing 22: _get_arg.c

```
#include <stdlib.h>
#include <string.h>
#include "builtins.h"

_pml_val _greater_equal;

_pml_val _builtin__greater_equal(_pml_val *args)
{
  _pml_val left, right;

  left = (_pml_val) args[0];
  right = (_pml_val) args[1];

  _pml_bool res;
  switch (left->type) {
    case PML_CHAR:
      res = _pml_get_char(left) >= _pml_get_char(right);
```

```
      return _make_bool(res);
    case PML_BOOL:
      res = _pml_get_bool(left) >= _pml_get_bool(right);
      return _make_bool(res);
    case PML_UNIT:
      res = _pml_get_unit(left) >= _pml_get_unit(right);
      return _make_bool(res);
    case PML_INT:
      res = _pml_get_int(left) >= _pml_get_int(right);
      return _make_bool(res);
    case PML_STRING:
      res = 0 >= strcmp(_pml_get_string(left), _pml_get_string(right));
      return _make_bool(res);
    default:
      _pml_error("This type does not support equality operator");
      return NULL;
  }
}

void _init__greater_equal()
{
  _greater_equal = _make_closure(_builtin__greater_equal, 2);
}
```

Listing 23: _greater_equal.c

```
#include <stdlib.h>
#include <string.h>
#include "builtins.h"

_pml_val _greater_than;

_pml_val _builtin__greater_than(_pml_val *args)
{
  _pml_val left, right;

  left = (_pml_val) args[0];
  right = (_pml_val) args[1];

  _pml_bool res;
  switch (left->type) {
    case PML_CHAR:
      res = _pml_get_char(left) > _pml_get_char(right);
      return _make_bool(res);
    case PML_BOOL:
      res = _pml_get_bool(left) > _pml_get_bool(right);
      return _make_bool(res);
    case PML_UNIT:
      res = _pml_get_unit(left) > _pml_get_unit(right);
      return _make_bool(res);
    case PML_INT:
      res = _pml_get_int(left) > _pml_get_int(right);
      return _make_bool(res);
    case PML_STRING:
      res = 0 > strcmp(_pml_get_string(left), _pml_get_string(right));
      return _make_bool(res);
    default:
      _pml_error("This type does not support equality operator");
      return NULL;
```

```
  }
}

void _init__greater_than()
{
  _greater_than = _make_closure(_builtin__greater_than, 2);
}
```

Listing 24: _greater_than.c

```
#include "builtins.h"

void _init__builtins()
{
    _init__add();
    _init__minus();
    _init__times();
    _init__divide();
    _init__less_than();
    _init__less_equal();
    _init__greater_than();
    _init__greater_equal();
    _init__equal();
    _init__not_equal();
    _init__or();
    _init__and();
    _init__cons();
    _init__seq();
    _init_string_of_int();
    _init_string_of_char();
    _init_string_of_bool();
    _init_print_string();
    _init_error();
}
```

Listing 25: _init.c

```
#include <stdlib.h>
#include <string.h>
#include "builtins.h"


_pml_val _less_equal;

_pml_val _builtin__less_equal(_pml_val *args) {
  _pml_val left, right;

  left = (_pml_val) args[0];
  right = (_pml_val) args[1];

  _pml_bool res;
  switch (left->type) {
    case PML_CHAR:
      res = _pml_get_char(left) <= _pml_get_char(right);
      return _make_bool(res);
    case PML_BOOL:
      res = _pml_get_bool(left) <= _pml_get_bool(right);
      return _make_bool(res);
    case PML_UNIT:
```

```
      res = _pml_get_unit(left) <= _pml_get_unit(right);
      return _make_bool(res);
    case PML_INT:
      res = _pml_get_int(left) <= _pml_get_int(right);
      return _make_bool(res);
    case PML_STRING:
      res = 0 <= strcmp(_pml_get_string(left), _pml_get_string(right));
      return _make_bool(res);
    default:
      _pml_error("This type does not support equality operator");
      return NULL;
  }
}

void _init__less_equal() {
    _less_equal = _make_closure(_builtin__less_equal, 2);
}
```

Listing 26: _less_equal.c

```
#include <stdlib.h>
#include <string.h>
#include "builtins.h"


_pml_val _less_than;

_pml_val _builtin__less_than(_pml_val *args) {
  _pml_val left, right;

  left = (_pml_val) args[0];
  right = (_pml_val) args[1];

  _pml_bool res;
  switch (left->type) {
    case PML_CHAR:
      res = _pml_get_char(left) < _pml_get_char(right);
      return _make_bool(res);
    case PML_BOOL:
      res = _pml_get_bool(left) < _pml_get_bool(right);
      return _make_bool(res);
    case PML_UNIT:
      res = _pml_get_unit(left) < _pml_get_unit(right);
      return _make_bool(res);
    case PML_INT:
      res = _pml_get_int(left) < _pml_get_int(right);
      return _make_bool(res);
    case PML_STRING:
      res = 0 < strcmp(_pml_get_string(left), _pml_get_string(right));
      return _make_bool(res);
    default:
      _pml_error("This type does not support equality operator");
      return NULL;
  }
}

void _init__less_than() {
    _less_than = _make_closure(_builtin__less_than, 2);
```

```
}
```

Listing 27: _less_than.c

```
#include <stdlib.h>
#include "builtins.h"


_pml_val _make_closure(_pml_func *fp, _pml_int num_args) {
#ifdef BUILTIN_DEBUG
  printf("[debug] _make_closure\n");
#endif
    void *closure = malloc(_closure_size_with_args(num_args));
    _pml_val_internal *v = malloc(sizeof(_pml_val_internal));

    _set_closure_fp(closure, fp);
    _set_closure_required(closure, num_args);
    _set_closure_supplied(closure, 0);

    v->type = PML_CLOSURE;
    v->closure = closure;

    return v;
}
```

Listing 28: _make_closure.c

```
#include <string.h>
#include <stdlib.h>
#include "builtins.h"

_pml_val _make_int(_pml_int n) {
    _pml_val_internal *p = malloc(sizeof(_pml_val_internal));
    p->type = PML_INT;
    p->i = n;
    return (_pml_val) p;
}

_pml_val _make_bool(_pml_bool b) {
    _pml_val_internal *p = malloc(sizeof(_pml_val_internal));
    p->type = PML_BOOL;
    p->b = b;
    return (_pml_val) p;
}

_pml_val _make_char(_pml_char c) {
    _pml_val_internal *p = malloc(sizeof(_pml_val_internal));
    p->type = PML_CHAR;
    p->c = c;
    return (_pml_val) p;
}

_pml_val _make_string(_pml_char *s) {
    _pml_val_internal *v = malloc(sizeof(_pml_val_internal));
    size_t n = strlen((char *) s) + 1;
    _pml_char *p = malloc(sizeof(_pml_char) * n);
    strcpy(p, s);
    v->type = PML_STRING;
    v->s = p;
```

```
    return (_pml_val) v;
}

_pml_val_internal _unit_internal = {
    .type = PML_UNIT,
    .u = 69
};

_pml_val _unit = &_unit_internal;

_pml_val _make_unit() {
    return _unit;
}

_pml_val_internal _pml_empty_list_internal = {
    .type = PML_LIST,
    .l = NULL
};

_pml_val _pml_empty_list = &_pml_empty_list_internal;

_pml_val _make_list(_pml_val data, _pml_val next_list_val) {
    _pml_list *next_list = (_pml_list *)next_list_val->l;
    _pml_list *list = malloc(sizeof(_pml_list));
    list->data = data;
    list->next = next_list;
    _pml_val_internal *v = malloc(sizeof(_pml_val_internal));
    v->type = PML_LIST;
    v->l = list;
    return (_pml_val) v;
}
```

Listing 29: _make_primitives.c

```
#include <stdlib.h>
#include "builtins.h"


_pml_val _minus;

_pml_val _builtin__minus(_pml_val *args) {
  _pml_val left, right;

  left = (_pml_val) args[0];
  right = (_pml_val) args[1];

  _pml_int res = _pml_get_int(left) - _pml_get_int(right);

  return _make_int(res);
}

void _init__minus() {
    _minus = _make_closure(_builtin__minus, 2);
}
```

Listing 30: _minus.c

```
#include <stdlib.h>
#include <string.h>
```

```c
#include "builtins.h"

_pml_val _not_equal;

_pml_val _builtin__not_equal(_pml_val *args)
{
  _pml_val left, right;

  left = (_pml_val) args[0];
  right = (_pml_val) args[1];

  _pml_bool res;
  switch (left->type) {
    case PML_CHAR:
      res = _pml_get_char(left) != _pml_get_char(right);
      return _make_bool(res);
    case PML_BOOL:
      res = _pml_get_bool(left) != _pml_get_bool(right);
      return _make_bool(res);
    case PML_UNIT:
      res = _pml_get_unit(left) != _pml_get_unit(right);
      return _make_bool(res);
    case PML_INT:
      res = _pml_get_int(left) != _pml_get_int(right);
      return _make_bool(res);
    case PML_STRING:
      res = 0 != strcmp(_pml_get_string(left), _pml_get_string(right));
      return _make_bool(res);
    default:
      _pml_error("This type does not support equality operator");
      return NULL;
  }
}

void _init__not_equal()
{
  _not_equal = _make_closure(_builtin__not_equal, 2);
}
```

Listing 31: _not_equal.c

```c
#include <stdlib.h>
#include "builtins.h"

_pml_val _or;

_pml_val _builtin__or(_pml_val *args)
{
  _pml_val left, right;

  left = (_pml_val) args[0];
  right = (_pml_val) args[1];

  _pml_bool res = _pml_get_bool(left) || _pml_get_bool(right);

  return _make_bool(res);
}

void _init__or()
```

```
{
  _or = _make_closure(_builtin__or, 2);
}
```

Listing 32: _or.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "builtins.h"

_pml_bool _match_pat_cons(_pml_val val) {
    _pml_list *l = _pml_get_list(val);
    return l != NULL;
}

_pml_bool _match_pat_cons_end(_pml_val val) {
    _pml_list *l = _pml_get_list(val);
    if (l == NULL)
        return false;
    l = l->next;
    return l == NULL;
}

_pml_bool _match_pat_lit_int(_pml_val val, _pml_int pat) {
    _pml_int i = _pml_get_int(val);
    return i == pat;
}

_pml_bool _match_pat_lit_char(_pml_val val, _pml_char pat) {
    _pml_char c = _pml_get_char(val);
    return c == pat;
}

_pml_bool _match_pat_lit_bool(_pml_val val, _pml_bool pat) {
    _pml_bool b = _pml_get_bool(val);
    return b == pat;
}

_pml_bool _match_pat_lit_string(_pml_val val, _pml_string pat) {
    _pml_string s = _pml_get_string(val);
    return strcmp(s, pat) == 0;
}

_pml_bool _match_pat_lit_unit(_pml_val val, _pml_unit pat) {
    _pml_unit u = _pml_get_unit(val);
    return u == pat;
}

_pml_bool _match_pat_lit_list_end(_pml_val val) {
    _pml_list *l = _pml_get_list(val);
    return l == NULL;
}

_pml_val _list_get_head(_pml_val val) {
    _pml_list *l = _pml_get_list(val);

    if (l == NULL)
        _pml_error("Can't get head of list with length 0");
```

```
54
55      return l->data;
56  }
57
58  _pml_val _list_get_tail(_pml_val val) {
59      _pml_list *l = _pml_get_list(val);
60
61      if (l == NULL)
62          _pml_error("Can't get tail of list with length 0");
63
64      _pml_val_internal *v = malloc(sizeof(_pml_val_internal));
65      v->type = PML_LIST;
66      v->l = l->next;
67
68      return v;
69  }
```

Listing 33: _pattern_match.c

```
1  #include <stdlib.h>
2  #include "builtins.h"
3
4  _pml_val _seq;
5
6  _pml_val _builtin__seq(_pml_val *args)
7  {
8    return args[1];
9  }
10
11 void _init__seq()
12 {
13   _seq = _make_closure(_builtin__seq, 2);
14 }
```

Listing 34: _seq.c

```
1  #include <stdlib.h>
2  #include "builtins.h"
3
4
5  _pml_val _times;
6
7  _pml_val _builtin__times(_pml_val *args) {
8    _pml_val left, right;
9
10   left = (_pml_val) args[0];
11   right = (_pml_val) args[1];
12
13   _pml_int res = _pml_get_int(left) * _pml_get_int(right);
14
15   return _make_int(res);
16 }
17
18 void _init__times() {
19     _times = _make_closure(_builtin__times, 2);
20 }
```

Listing 35: _times.c

```c
#include <stdlib.h>
#include <stdio.h>
#include "builtins.h"


_pml_val error;

_pml_val _builtin_error(_pml_val *args)
{
  _pml_string s;

  s = (_pml_string) args[0];

  fprintf(stderr, "Error: %s", s);
  exit(1);

  return _make_unit();
}

void _init_error()
{
  error = _make_closure(_builtin_error, 1);
}
```

Listing 36: error.c

```c
#include "builtins.h"

_pml_char _pml_get_char(_pml_val val) {
    if (val->type != PML_CHAR)
        _pml_error("Run-time type error: value is not char type");
    return val->c;
}

_pml_bool _pml_get_bool(_pml_val val) {
    if (val->type != PML_BOOL)
        _pml_error("Run-time type error: value is not bool type");
    return val->b;
}

_pml_unit _pml_get_unit(_pml_val val) {
    if (val->type != PML_UNIT)
        _pml_error("Run-time type error: value is not unit type");
    return val->u;
}

_pml_int _pml_get_int(_pml_val val) {
    if (val->type != PML_INT)
        _pml_error("Run-time type error: value is not int type");
    return val->i;
}

_pml_string _pml_get_string(_pml_val val) {
    if (val->type != PML_STRING)
        _pml_error("Run-time type error: value is not string type");
    return val->s;
}

_pml_list *_pml_get_list(_pml_val val) {
```

```
    if (val->type != PML_LIST)
        _pml_error("Run-time type error: value is not list type");
    return (_pml_list *) val->l;
}

void *_pml_get_closure(_pml_val val) {
#ifdef BUILTIN_DEBUG
  printf("[debug] _pml_get_closure: got %d\n", val->type);
#endif

    if (val->type != PML_CLOSURE)
        _pml_error("Run-time type error: value is not function type");
    return (void *) val->closure;
}
```

Listing 37: get_types.c

```
#include <stdlib.h>
#include <stdio.h>
#include "builtins.h"


_pml_val print_string;

_pml_val _builtin_print_string(_pml_val *args)
{
  _pml_string s;

  s = _pml_get_string(args[0]);

  printf("%s", s);
  return _make_unit();
}

void _init_print_string()
{
  print_string = _make_closure(_builtin_print_string, 1);
}
```

Listing 38: print_string.c

```
#include <stdlib.h>
#include <stdio.h>
#include "builtins.h"

#define BOOL_NUM_CHAR    6
#define BOOL_STR_SIZE    (sizeof(_pml_char) * BOOL_NUM_CHAR)


_pml_val string_of_bool;

_pml_val _builtin_string_of_bool(_pml_val *args)
{
  _pml_bool bool_val;
  _pml_string bool_str = "false";
  _pml_char res[BOOL_STR_SIZE];

  bool_val = _pml_get_bool(args[0]);
  if (bool_val) {
```

```
    bool_str = "true";
  }
  sprintf(res, "%s", bool_str);

#ifdef BUILTIN_DEBUG
  printf("[debug] string_of_bool %d -> %s\n", bool_val, res);
#endif

  return _make_string(res);
}

void _init_string_of_bool()
{
  string_of_bool = _make_closure(_builtin_string_of_bool, 1);
}
```

Listing 39: string_of_bool.c

```
#include <stdlib.h>
#include <stdio.h>
#include "builtins.h"

#define CHAR_MAX_CHAR_NUM    2
#define CHAR_STR_MAX_SIZE    (sizeof(_pml_char) * CHAR_MAX_CHAR_NUM)

_pml_val string_of_char;

_pml_val _builtin_string_of_char(_pml_val *args)
{
  _pml_char char_val;
  _pml_char res[CHAR_STR_MAX_SIZE];

  char_val = _pml_get_char(args[0]);
  res[0] = char_val;
  res[1] = '\0';

#ifdef BUILTIN_DEBUG
  printf("[debug] string_of_char %c -> %s\n", char_val, res);
#endif

  return _make_string(res);
}

void _init_string_of_char()
{
  string_of_char = _make_closure(_builtin_string_of_char, 1);
}
```

Listing 40: string_of_char.c

```

#include <stdlib.h>
#include <stdio.h>
#include "builtins.h"


#define INT_MAX_CHAR_NUM    10
#define INT_STR_MAX_SIZE    (sizeof(_pml_char) * INT_MAX_CHAR_NUM)

```

```
10
11 _pml_val string_of_int;
12
13 _pml_val _builtin_string_of_int(_pml_val *args)
14 {
15   _pml_int int_val;
16   _pml_char res[INT_STR_MAX_SIZE];
17
18   int_val = _pml_get_int(args[0]);
19   sprintf(res, "%d", int_val);
20
21 #ifdef BUILTIN_DEBUG
22   printf("[debug] string_of_int %d -> %s\n", int_val, res);
23 #endif
24
25   return _make_string(res);
26 }
27
28 void _init_string_of_int()
29 {
30   string_of_int = _make_closure(_builtin_string_of_int, 1);
31 }
```

Listing 41: string_of_int.c

```
1  CC=gcc
2  CFLAGS=-I. $(PML_CFLAGS)
3  DEPS=buildins.h
4  OBJ= _init.o \
5   _apply_closure.o \
6   _get_arg.o \
7   get_types.o \
8   _make_closure.o \
9   _make_primitives.o \
10  _pattern_match.o \
11  _error.o \
12  _add.o \
13  _and.o \
14  _cons.o \
15  _divide.o \
16  _equal.o \
17  _greater_equal.o \
18  _greater_than.o \
19  _less_equal.o \
20  _less_than.o \
21  _minus.o \
22  _not_equal.o \
23  _or.o \
24  _seq.o \
25  _times.o \
26  string_of_int.o \
27  string_of_bool.o \
28  string_of_char.o \
29  print_string.o \
30  error.o
31
32 .PHONY: default
33 default: pml_builtins.a
34
```

```
%.o: %.c $(DEPS)
  $(CC) -c -o $@ $< $(CFLAGS)

pml_builtins.a: $(OBJ)
  ar rcs $@ $^

.PHONY: clean
clean:
  rm -f *.o pml_builtins.a
```

Listing 42: Makefile

8.4 stdlib

```
(*****************)
(*  List  *)
(*****************)

let rec list_length = function [] -> 0 | _ :: xs -> 1 + list_length xs

let rec list_map f = function [] -> [] | x :: xs -> f x :: list_map f xs

let rec list_iter f = function
  | [] -> ()
  | x :: xs ->
      f x;
      list_iter f xs

let rec list_append xs ys =
  match xs with [] -> ys | x :: xs -> x :: list_append xs ys

let rec list_filter pred = function
  | [] -> []
  | x :: xs -> if pred x
               then x :: (list_filter pred xs)
               else list_filter pred xs

let rec list_mem el = function
  | [] -> false
  | x :: xs ->
      if x = el then
          true
      else
          list_mem el xs

let rec list_fold_left f m = function
  | [] -> m
  | x :: xs -> list_fold_left f (f m x) xs

let list_rev xs =
  list_fold_left (fun l x -> x :: l) [] xs

let rec fold_right_helper f xl m = match xl with
    | [] -> m
    | x :: xs -> fold_right_helper f xs (f x m)

let list_fold_right f xlist m =
  let xr = list_rev xlist in fold_right_helper f xr m
```

```
45
46 let list_hd = function
47   | x :: xs -> x
48   | _ -> error "Trying to get head from empty list"
49
50 let list_tl = function
51   | x :: xs -> xs
52   | _ -> error "Unable to get tail from emtpy/one-element list"
53
54
55 (****************)
56 (*  I/O   *)
57 (****************)
58
59 let print_endline s =
60   print_string s;
61   print_string "\n"
62
63 let print_int n = print_string (string_of_int n)
64 let print_bool b = print_string (string_of_bool b)
65 let print_char c = print_string (string_of_char c)
66 let print_list print_el lst =
67   let _ = print_string "[ " in
68   let _ = list_iter (fun el -> let _ = print_el el in print_string " " ) lst in
69     print_endline "]"
70
71 let print_int_list l1 = print_list print_int l1
72 let print_bool_list l2 = print_list print_bool l2
73 let print_char_list l3 = print_list print_char l3
```

Listing 43: stdlib.pml

8.5 test

8.5.1 ast

```
1 open Pocaml.Print
2
3 (* open Pocaml.Parser *)
4 (* open Pocaml.Lexer *)
5
6 let%expect_test _ =
7   print_prog "let fn (a: int) : int = 3";
8   [%expect {| let fn ( a : int ) : int = 3 |}]
9
10 let%expect_test _ =
11   print_prog "let fn (a: int) = 3";
12   [%expect {| let fn ( a : int ) = 3 |}]
13
14 let%expect_test _ =
15   print_prog "let rec fn (a: int): int = 3";
16   [%expect {| let rec fn ( a : int ) : int = 3 |}]
17
18 let%expect_test _ =
19   print_prog "let rec fn (a: int) b (c: int): int = 3";
20   [%expect {| let rec fn ( a : int ) b ( c : int ) : int = 3 |}]
21
```

```
let%expect_test _ =
  print_prog "let a: int = 3";
  [%expect {| let a : int = 3 |}]

let%expect_test _ =
  print_prog "let a = \"some string\"";
  [%expect {| let a = "some string" |}]

let%expect_test _ =
  print_prog "let a = false";
  [%expect {| let a = false |}]

let%expect_test _ =
  print_prog "let a = 'c'";
  [%expect {| let a = 'c' |}]

let%expect_test _ =
  print_prog "let a = [1;2;3;]";
  [%expect {| let a = [1;2;3] |}]
```

Listing 44: def.ml

```
open Pocaml.Print

let%expect_test "annotated literal" =
  print_prog "let (a : int) = (3 : int)";
  [%expect {| let a : int = ( 3 : int ) |}]

let%expect_test "annotated variable" =
  print_prog "let _ = let a = 3 in (a : int)";
  [%expect {| let _ = ( let a = 3 in ( a : int ) ) |}]

let%expect_test "annotated expr with unary operator" =
  print_prog "let a = (not true : bool)";
  [%expect {| let a = ( _not true : bool ) |}]

let%expect_test "annotated expr with binary operator" =
  print_prog "let a = (3 + 5 : int)";
  [%expect {| let a = ( ( 3 _add 5 ) : int ) |}]

let%expect_test "annotated conditional expression" =
  print_prog "let a = (if true then 1 else 2 : int)";
  [%expect {| let a = ( ( if true then 1 else 2 ) : int ) |}]

let%expect_test "annotated let in expression" =
  print_prog "let a = (let b = 3 in b : int)";
  [%expect {| let a = ( ( let b = 3 in b ) : int ) |}]

let%expect_test "annotated lambda expression" =
  print_prog "let a = (fun (a: int) -> (a + 1 : int) : int -> int)";
  [%expect
    {| let a = ( ( fun ( a : int ) = ( ( a _add 1 ) : int ) ) : int -> int ) |}]

let%expect_test "annotated function application" =
  print_prog "let a = (print \"hello\" : ())";
  [%expect {| let a = ( ( print "hello" ) : () ) |}]

let%expect_test "annotated function application with multiple arguments" =
  print_prog "let a = (fn 0 f 1 : bool)";
```

```
  [%expect {| let a = ( ( ( ( fn 0 ) f ) 1 ) : bool ) |}]

(* TODO: add test case for annotated match expression *)
let%expect_test "annotated match expression" =
  print_prog "let a = ( match 3 with | _ -> 1 : int )";
  [%expect {|
    let a = ( (
     match 3 with
    |  _ -> 1
    ) : int ) |}]
```

Listing 45: expr.ml

8.5.2 ir

```
open Pocaml.Print_ir


let%expect_test _ =
  print_prog "let fn (a: int) : int = 3";
  [%expect {| let fn = ( ( fun a -> ( 3 : int ) ) : int -> int ) |}]

let%expect_test _ =
  print_prog "let fn (a: int) = 3";
  [%expect {| let fn = ( ( fun a -> ( 3 : None ) ) : int -> None ) |}]

let%expect_test _ =
  print_prog "let rec fn (a: int): int = 3";
  [%expect {| let fn = ( ( fun a -> ( 3 : int ) ) : int -> int ) |}]

let%expect_test _ =
  print_prog "let rec fn (a: int) b (c: int): int = 3";
  [%expect {| let fn = ( ( fun a -> ( ( fun b -> ( ( fun c -> ( 3 : int ) ) : int
    -> int ) ) : None -> int -> int ) ) : int -> None -> int -> int ) |}]

let%expect_test _ =
  print_prog "let a: int = 3";
  [%expect {| let a = ( 3 : int ) |}]

let%expect_test _ =
  print_prog "let a = \"some string\"";
  [%expect {| let a = ( "some string" : None ) |}]

let%expect_test _ =
  print_prog "let a = false";
  [%expect {| let a = ( false : None ) |}]

let%expect_test _ =
  print_prog "let a = 'c'";
  [%expect {| let a = ( 'c' : None ) |}]

let%expect_test _ =
  print_prog "let a = [1;2;3;]";
  [%expect {| let a = ( ( ( ( ( _cons : None ) ( 1 : None ) ) : None ) ( ( ( ( (
    _cons : None ) ( 2 : None ) ) : None ) ( ( ( ( ( _cons : None ) ( 3 : None ) )
    : None ) ( [] : None ) ) : None ) ) : None ) ) : None ) |}]

let%expect_test _ =
```

```
  print_prog "let a: () = ()";
  [%expect {| let a = ( () : unit ) |}]
```

Listing 46: def.ml

```
open Pocaml.Print_ir

let%expect_test "annotated literal" =
  print_prog "let (a : int) = (3 : int)";
  [%expect {| let a = ( 3 : int ) |}]

let%expect_test "annotated variable" =
  print_prog "let _ = let a = 3 in (a : int)";
  [%expect {| let U2 = ( let ( a : None ) = ( 3 : None ) in ( a : int ) ) |}]

let%expect_test "annotated expr with unary operator" =
  print_prog "let a = (not true : bool)";
  [%expect {| let a = ( ( ( _not : None ) ( true : None ) ) : bool ) |}]

let%expect_test "annotated expr with binary operator" =
  print_prog "let a = (3 + 5 : int)";
  [%expect {| let a = ( ( ( ( ( _add : None ) ( 3 : None ) ) : None ) ( 5 : None )
    ) : int ) |}]

let%expect_test "annotated conditional expression" =
  print_prog "let a = (if true then 1 else 2 : int)";
  [%expect {|
    let a = ( (
     match ( true : None ) with
    |  ( true : None ) -> ( 1 : None )
    |  ( false : None ) -> ( 2 : None )
    ) : int ) |}]

let%expect_test "annotated let in expression" =
  print_prog "let a = (let b = 3 in b : int)";
  [%expect {| let a = ( let ( b : int ) = ( 3 : None ) in ( b : None ) ) |}]

let%expect_test "annotated lambda expression" =
  print_prog "let a = (fun (a: int) -> (a + 1 : int) : int -> int)";
  [%expect
    {| let a = ( ( fun a -> ( ( ( ( ( _add : None ) ( a : None ) ) : None ) ( 1 :
   None ) ) : int ) ) : int -> None ) |}]

let%expect_test "annotated function application" =
  print_prog "let a = (print (\"hello\" : string) : ())";
  [%expect {| let a = ( ( ( print : None ) ( "hello" : string ) ) : unit ) |}]

let%expect_test "annotated function application with multiple arguments" =
  print_prog "let a = (fn 0 f 1 : bool)";
  [%expect {| let a = ( ( ( ( ( ( fn : None ) ( 0 : None ) ) : None ) ( f : None
    ) ) : None ) ( 1 : None ) ) : bool ) |}]

(* TODO: add test case for annotated match expression *)
let%expect_test "annotated match expression" =
  print_prog "let a = ( match 3 with | _ -> 1 : int )";
  [%expect {|
    let a = ( (
     match ( 3 : None ) with
    |  ( U3 : None ) -> ( 1 : None )
```

```
    ) : int ) |}]
```

Listing 47: expr.ml

```
open Pocaml
open Pocaml.Print_ir


let print_prog_ll = function
  | str ->
      let lexbuf = Lexing.from_string str in
      let prog = Parser.program Lexer.token lexbuf in
      Lower_ast.lower_program prog |> Lambda_lift.lambda_lift |> string_of_program
     |> print_endline

let%expect_test "lift lambdas in list" =
  print_prog_ll "let lambda_list : int -> int list = ([(fun (a:int) -> (a:int) :
    int -> int); (fun (b:int) -> (b:int) : int -> int);] : int -> int list)";
  [%expect {|
    let L1 = ( ( fun a -> ( a : int ) ) : int -> None )
    let L2 = ( ( fun b -> ( b : int ) ) : int -> None )
    let lambda_list = ( ( ( ( ( _cons : None ) ( L1 : int -> None ) ) : None ) ( (
     ( ( ( _cons : None ) ( L2 : int -> None ) ) : None ) ( [] : None ) ) : None )
    ) : int -> int ) |}]

let%expect_test "lift lambdas in letin expression" =
  print_prog_ll "let (a : int -> int) = let a = (3 : int) in (fun (b: int) -> ((b
    : int) + (a : int) : int) : int -> int)";
  [%expect {|
    let L3 = ( ( fun a -> ( ( fun b -> ( ( ( ( ( _add : None ) ( b : int ) ) :
    None ) ( a : int ) ) : int ) ) : int -> None ) ) : int -> int -> None )
    let a = ( let ( a : int -> int ) = ( 3 : int ) in ( ( ( L3 : int -> int ->
    None ) ( a : int ) ) : int -> None ) ) |}]

let%expect_test "lift lambdas in letin expression 2" =
  print_prog_ll " let f = let a = 3 in (fun b -> let c = 6 in (fun d -> b : int ->
     int) : int -> int -> int)";
  [%expect {|
    let L4 = ( ( fun a -> ( ( fun b -> ( ( fun c -> ( ( fun d -> ( b : None ) ) :
    None -> None ) ) : None -> None -> None ) ) : None -> None -> None -> None ) )
    : None -> None -> None -> None -> None )
    let L5 = ( ( fun a -> ( ( fun b -> ( let ( c : None ) = ( 6 : None ) in ( ( (
    ( ( ( ( L4 : None -> None -> None -> None -> None ) ( a : None ) ) : None ->
    None -> None -> None ) ( b : None ) ) : None -> None -> None ) ( c : None ) ) :
     None -> None ) ) ) : None -> None ) ) : None -> None -> None )
    let f = ( let ( a : None ) = ( 3 : None ) in ( ( ( L5 : None -> None -> None )
     ( a : None ) ) : None -> None ) ) |}]

let%expect_test "lift lambdas in match arms" =
  print_prog_ll "let a = match 3 with | 3 -> fun b -> b | 4 -> fun c -> c";
  [%expect {|
    let L6 = ( ( fun b -> ( b : None ) ) : None -> None )
    let L7 = ( ( fun c -> ( c : None ) ) : None -> None )
    let a = ( (
     match ( 3 : None ) with
    |  ( 3 : None ) -> ( L6 : None -> None )
    |  ( 4 : None ) -> ( L7 : None -> None )
    ) : None ) |}]

```

```
let%expect_test "lift lambdas in expr being patterned match on" =
  print_prog_ll "
    let a = (
      match (let a = 3 in
        (fun (b : int) -> (a : int) : int -> int)
        : int -> int)
      with | _ -> 0
      : int)
  ";
  [%expect {|
    let L8 = ( ( fun a -> ( ( fun b -> ( a : int ) ) : int -> None ) ) : None ->
    int -> None )
    let a = ( (
     match ( let ( a : int -> int ) = ( 3 : None ) in ( ( ( L8 : None -> int ->
    None ) ( a : None ) ) : int -> None ) ) with
    |  ( U1 : None ) -> ( 0 : None )
    ) : int ) |}]

let%expect_test "don't lift top level lambdas" =
  print_prog_ll "let f = fun a -> fun b -> a";
  [%expect {|
    let f = ( ( fun a -> ( ( fun b -> ( a : None ) ) : None -> None ) ) : None ->
    None ) |}]

let%expect_test "don't lift immediately nested lambdas" =
  print_prog_ll "let a = let a = 1 in fun b -> fun c -> a";
  [%expect {|
    let L9 = ( ( fun a -> ( ( fun b -> ( ( fun c -> ( a : None ) ) : None -> None
    ) ) : None -> None ) ) : None -> None -> None )
    let a = ( let ( a : None ) = ( 1 : None ) in ( ( ( L9 : None -> None -> None )
     ( a : None ) ) : None -> None ) ) |}]

let%expect_test "lift not immediately nested lambda" =
  print_prog_ll "let f = fun a -> let b = 1 in fun c -> a";
  [%expect {|
    let L10 = ( ( fun a -> ( ( fun b -> ( ( fun c -> ( a : None ) ) : None -> None
    ) ) : None -> None -> None ) ) : None -> None -> None -> None )
    let f = ( ( fun a -> ( let ( b : None ) = ( 1 : None ) in ( ( ( ( ( L10 : None
    -> None -> None -> None ) ( a : None ) ) : None -> None -> None ) ( b : None )
    ) : None -> None ) ) ) : None -> None ) |}]

let %expect_test "lift lambda in application" =
  print_prog_ll "let a = ((fun a -> a) 3) + 4";
  [%expect {|
    let L11 = ( ( fun a -> ( a : None ) ) : None -> None )
    let a = ( ( ( ( ( _add : None ) ( ( ( L11 : None -> None ) ( 3 : None ) ) :
    None ) ) : None ) ( 4 : None ) ) : None ) |}]
```

Listing 48: lambda_lift.ml

8.5.3 builtins

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "../../builtins/builtins.h"

void test__add()
```

```
{
  _pml_int *a = malloc(sizeof(_pml_int)),
           *b = malloc(sizeof(_pml_int));
  _pml_val builtin_add = _make_closure(_builtin__add, 2);
  _pml_val res;

  *a = 6, *b = 9;
  res = _apply_closure(builtin_add, a); /* partial function application */
  res = _apply_closure(res, b);

  assert(*(_pml_int *)res == *a + *b);
  free(a);
  free(b);
}

void test__minus()
{
  _pml_int *a = malloc(sizeof(_pml_int)),
           *b = malloc(sizeof(_pml_int));
  _pml_val builtin_minus = _make_closure(_builtin__minus, 2);
  _pml_val res;

  *a = 6, *b = 9;
  res = _apply_closure(builtin_minus, a); /* partial function application */
  res = _apply_closure(res, b);

  assert(*(_pml_int *)res == *a - *b);
  free(a);
  free(b);
}

void test__times()
{
  _pml_int *a = malloc(sizeof(_pml_int)),
           *b = malloc(sizeof(_pml_int));
  _pml_val builtin_times = _make_closure(_builtin__times, 2);
  _pml_val res;

  *a = 6, *b = 9;
  res = _apply_closure(builtin_times, a); /* partial function application */
  res = _apply_closure(res, b);

  assert(*(_pml_int *)res == *a * *b);
  free(a);
  free(b);
}

void test__divide()
{
  _pml_int *a = malloc(sizeof(_pml_int)),
           *b = malloc(sizeof(_pml_int));
  _pml_val builtin_divide = _make_closure(_builtin__divide, 2);
  _pml_val res;

  *a = 6, *b = 9;
  res = _apply_closure(builtin_divide, a); /* partial function application */
  res = _apply_closure(res, b);

  assert(*(_pml_int *)res == *a / *b);
```

```
  free(a);
  free(b);
}

void test__less_than()
{
  _pml_int *a = malloc(sizeof(_pml_int)),
           *b = malloc(sizeof(_pml_int));
  _pml_val builtin_less_than = _make_closure(_builtin__less_than, 2);
  _pml_val res;

  *a = 6, *b = 9;
  res = _apply_closure(builtin_less_than, a); /* partial function application */
  res = _apply_closure(res, b);

  assert(*(_pml_int *)res == *a < *b);
  free(a);
  free(b);
}

void test__less_equal()
{
  _pml_int *a = malloc(sizeof(_pml_int)),
           *b = malloc(sizeof(_pml_int));
  _pml_val builtin_less_equal = _make_closure(_builtin__less_equal, 2);
  _pml_val res;

  *a = 6, *b = 6;
  res = _apply_closure(builtin_less_equal, a); /* partial function application */
  res = _apply_closure(res, b);

  assert(*(_pml_int *)res == *a <= *b);
  free(a);
  free(b);
}

void test__greater_than()
{
  _pml_int *a = malloc(sizeof(_pml_int)),
           *b = malloc(sizeof(_pml_int));
  _pml_val builtin_greater_than = _make_closure(_builtin__greater_than, 2);
  _pml_val res;

  *a = 6, *b = 9;
  res = _apply_closure(builtin_greater_than, a); /* partial function application
    */
  res = _apply_closure(res, b);

  assert(*(_pml_int *)res == *a > *b);
  free(a);
  free(b);
}

void test__greater_equal()
{
  _pml_int *a = malloc(sizeof(_pml_int)),
           *b = malloc(sizeof(_pml_int));
  _pml_val builtin_greater_equal = _make_closure(_builtin__greater_equal, 2);
  _pml_val res;
```

```
124
125   *a = 6, *b = 6;
126   res = _apply_closure(builtin_greater_equal, a); /* partial function application
        */
127   res = _apply_closure(res, b);
128
129   assert(*(_pml_int *)res == *a >= *b);
130   free(a);
131   free(b);
132 }
133
134 void test__equal()
135 {
136   _pml_int *a = malloc(sizeof(_pml_int)),
137            *b = malloc(sizeof(_pml_int));
138   _pml_val builtin_equal = _make_closure(_builtin__equal, 2);
139   _pml_val res;
140
141   *a = 6, *b = 6;
142   res = _apply_closure(builtin_equal, a); /* partial function application */
143   res = _apply_closure(res, b);
144   assert(*(_pml_int *)res == (*a == *b));
145   free(a);
146   free(b);
147 }
148
149 void test__not_equal()
150 {
151   _pml_int *a = malloc(sizeof(_pml_int)),
152            *b = malloc(sizeof(_pml_int));
153   _pml_val builtin_not_equal = _make_closure(_builtin__not_equal, 2);
154   _pml_val res;
155
156   *a = 6, *b = 9;
157   res = _apply_closure(builtin_not_equal, a); /* partial function application */
158   res = _apply_closure(res, b);
159   assert(*(_pml_int *)res == (*a != *b));
160   free(a);
161   free(b);
162 }
163
164 void test__or()
165 {
166   _pml_int *a = malloc(sizeof(_pml_int)),
167            *b = malloc(sizeof(_pml_int));
168   _pml_val builtin_or = _make_closure(_builtin__or, 2);
169   _pml_val res;
170
171   *a = 0, *b = 9;
172   res = _apply_closure(builtin_or, a); /* partial function application */
173   res = _apply_closure(res, b);
174   assert(*(_pml_int *)res == (*a || *b));
175   free(a);
176   free(b);
177 }
178
179 void test__and()
180 {
181   _pml_int *a = malloc(sizeof(_pml_int)),
```

```
182             *b = malloc(sizeof(_pml_int));
183   _pml_val builtin_and = _make_closure(_builtin__and, 2);
184   _pml_val res;
185
186   *a = 0, *b = 9;
187   res = _apply_closure(builtin_and, a); /* partial function application */
188   res = _apply_closure(res, b);
189   assert(*(_pml_int *)res == (*a && *b));
190   free(a);
191   free(b);
192 }
193
194 void test__cons()
195 {
196   _pml_int *a = malloc(sizeof(_pml_int));
197   _pml_list_node *b = malloc(sizeof(_pml_list_node));
198   _pml_int *data = malloc(sizeof(_pml_int));
199   _pml_val builtin_cons = _make_closure(_builtin__cons, 2);
200   _pml_val res;
201
202   *a = 6;
203   *data = 3;
204   b->data = data;
205
206   res = _apply_closure(builtin_cons, a);
207   res = _apply_closure(res, b);
208
209   assert(*(_pml_int *)((_pml_list_node *)res)->data == *a);
210
211   free(a);
212   free(data);
213   free(b);
214 }
215
216 void test__seq()
217 {
218   _pml_int *a = malloc(sizeof(_pml_int)),
219            *b = malloc(sizeof(_pml_int));
220   _pml_val builtin_seq = _make_closure(_builtin__seq, 2);
221   _pml_val res;
222
223   *a = 0, *b = 9;
224   res = _apply_closure(builtin_seq, a); /* partial function application */
225   res = _apply_closure(res, b);
226   assert(*(_pml_int *)res == *b);
227   free(a);
228   free(b);
229 }
230
231 int main()
232 {
233   test__add();
234   test__minus();
235   test__times();
236   test__divide();
237   test__less_than();
238   test__less_equal();
239   test__greater_than();
240   test__greater_equal();
```

```
  test__equal();
  test__not_equal();
  test__or();
  test__and();
  test__cons();
  test__seq();
}
```

Listing 49: closure.c

8.5.4 demo

```
let rec map f = function
  | head :: tail ->
      let r = f head in
      r :: map f tail
  | [] -> []

let _ =
  let fn_creator = fun is_flip x -> if is_flip then -1 * x else x in
  let do_flip = fn_creator true in

  let modify = function
    | do_flip -> (
        function
        | x ->
            let res = do_flip x in
            if res < 0 then -1 * res else res)
  in

  let modified_fns = map modify [ do_flip ] in
  let abs = list_hd modified_fns in

  let value = 3 + (4 * -2) in
  print_int (abs value)
```

Listing 50: demo1.pml

```
let lambda =
  let a = 1 in
  let b = 2 in
  let str = "pocaml" in
    fun lst ->
      list_iter
        ( fun el ->
            match el with
              | 1 -> (fun x -> print_string str) 1
              | _ -> print_int (a + b)
        )
      lst

let _ = lambda [ 6; 1; 9 ]
```

Listing 51: demo_lambda_lifting.pml

```
let edges = [
    ['a'; 'b'; '1']; ['a'; 'c'; '2'];
    ['a'; 'd'; '3']; ['b'; 'e'; '4'];
    ['c'; 'f'; '5']; ['d'; 'e'; '6'];
```

```
    ['e'; 'f'; '7']; ['e'; 'g'; '8']
]

let nodes = [['a'];['b'];['c'];['d'];['e'];['f'];['g']]

let rec successors n = function
    | [] -> []
    | hd :: edges -> match hd with
        | s :: t ->
            if s = n then
                (list_hd t) :: (successors n edges)
            else
                successors n edges
        | _ -> error "edges formatted incorrectly"

let _ = print_string "Successors of a: ";
        print_list print_char (successors 'a' edges)
let _ = print_string "Successors of b: ";
        print_list print_char (successors 'b' edges)
let _ = print_endline " "


let rec dfs edges visited = function
    | [] -> list_rev visited
    | n :: nodes ->
        if list_mem n visited then
            dfs edges visited nodes
        else
            dfs edges (n::visited) (list_append (successors n edges) nodes)

let _ = print_string "DFS from a: ";
        print_char_list (dfs edges [] ['a'])
let _ = print_string "DFS from e: ";
        print_char_list (dfs edges [] ['e'])


let rec same_comp a b = function
    | hd_in :: tl_in ->
        if (list_mem a hd_in) && (list_mem b hd_in) then
            true
        else
            same_comp a b tl_in
    | [] -> false

let rec get_comp a = function
    | hd_in :: tl_in ->
        if (list_mem a hd_in) then
            hd_in
        else
            get_comp a tl_in
    | [] -> [a]

let rec kruskal comps res = function
| [] -> res
| hd :: tl ->
    let u = list_hd hd in
    let v = list_hd (list_tl hd) in

    if (same_comp u v comps) then
```

```
        kruskal comps res tl
    else
        let res = hd :: res in
        let comp_u = (get_comp u comps) in
        let comp_v = (get_comp v comps) in
        let comps = list_filter (fun a -> (list_mem u a) = false) comps in
        let comps = list_filter (fun a -> (list_mem v a) = false) comps in
        let comp_uv = list_append comp_u comp_v in
        let comps = comp_uv :: comps in
        kruskal comps res tl

let _ = print_endline "Minimum Spanning Tree: ";
print_list (print_list print_char) (kruskal nodes [] edges)
```

Listing 52: demo_graphs.pml

8.5.5 pml (integration tests)

```
let x = true && 1
let _ = print_bool x
```

Listing 53: fail_and_1.pml

```
let f x = 3 + x
let y = g 4
let _ = print_int y
```

Listing 54: fail_apply_1.pml

```
let f x = list_hd x
let g y = (f y) + 3
let _ = print_int (g [])
```

Listing 55: fail_apply_2.pml

```
let x = 5
    let y = 10 in
y + 5
let z = x + y / 3
let _ = print_int z
```

Listing 56: fail_binops_1.pml

```
let x = 3
(* this is a invalid comment )
let _ = print_int x
```

Listing 57: fail_comments_1.pml

```
let x = if 3 + 2 > 4 then true else "happy"
let _ = print_int x
```

Listing 58: fail_conditional_1.pml

```
let f1 x = 10 * x
let f2 x = 100 * x
let g = 50
let result = if g > 50 then f2 g else f1 g else 10
(* Extra Else *)
let _ = print_int result
```

Listing 59: fail_conditional_2.pml

```
let x = true
let y = x :: [2; 3; 4]
(* type mismatch *)
let _ = print_int_list y
```

Listing 60: fail_cons_1.pml

```
let x = 'hello'
let y = x :: ['e'; 'l'; 'l'; 'o']
let _ = print_char_list y
```

Listing 61: fail_cons_2.pml

```
let f = function
    1 -> "1"
  | true -> "2"
  | _ -> "_"

let _ = print_string "f 1 = "; print_endline (f 1)
let _ = print_string "f 2 = "; print_endline (f true)
let _ = print_string "f others = "; print_endline (f 3)
```

Listing 62: fail_function_1.pml

```
let f = function
    hd :: tl -> tl
  | [] -> "empty"

let _ = print_string "f [1; 2; 3] = "; print_int_list (f "hello")
let _ = print_string "f others = "; print_endline (f [])
```

Listing 63: fail_function_2.pml

```
let f = function
    true -> "true"
  | _ -> "false"

let _ = print_string "f true = "; print_endline (f true)
let _ = print_string "f others = "; print_endline (f 0)
```

Listing 64: fail_function_3.pml

```
let x =
    let y =
        let z =
            let f = 10 in
        f + 5 in
    z + 2 in
f + 1

let _ = print_int x
```

Listing 65: fail_letin_1.pml

```
let x =
    let y =
        let z =
            let f = true in
        f || false in
    z && true in
```

```
z || false

let _ = print_bool x
```

Listing 66: fail_letin_2.pml

```
let x = false || (false || (false || 9))
let _ = print_bool x
```

Listing 67: fail_or_1.pml

```
let x = ((3)
let _ = print_int x
```

Listing 68: fail_parse_2.pml

```
let f a = match a with
| 2 -> print_int a
| true -> print_int 3
| _ -> print_int 10

let _ = f 3
```

Listing 69: fail_pattern_matching_1.pml

```
let f a b =
    let x =
        match a with
        | 1 -> match b with
            | 2 -> x
            | _ -> b
        | 2 -> match f with
            | 1 -> a
            | _ -> b
        | _ -> 0
    in
x
(* scoping error *)

let _ = print_int (f 1 2)
```

Listing 70: fail_pattern_matching_2.pml

```
let f a = match a with
| hd :: tl -> list_length tl
| [] -> 0
| _ -> error "Input should be a list"

let _ = print_int (f 3)
```

Listing 71: fail_pattern_matching_3.pml

```
let f a = match a with
| hd :: tl -> match tl with
    | hd1 :: tl1 -> hd :: hd1 :: (list_rev tl1)
    | _ -> error "empty list"
| _ -> []

let _ = print_int_list (f [])
```

Listing 72: fail_pattern_matching_4.pml

```
let rec f = function
    | xh :: xs -> xh + f xs
    | _ -> 0
let y = f [1; 4; 5; 6; true]
let _ = print_int y
```

Listing 73: fail_rec_1.pml

```
let x = []
let z = list_hd x
let _ = print_int z
```

Listing 74: fail_std_head_1.pml

```
let x = []
let z = list_tl x
let _ = print_int_list z
```

Listing 75: fail_std_tail_1.pml

```
let x = [1; 2; 3]
let z = list_fold_right (fun a l -> a - l) x []
let _ = print_int_list z
```

Listing 76: fail_stdlib1.pml

```
let x = [1; 2; "3"]
let z = list_rev x
let _ = print_int_list z
```

Listing 77: fail_stdlib2.pml

```
let x = [1; 2; 3]
let z = list_filter (fun a -> a + 2) x
let _ = print_int_list z
```

Listing 78: fail_stdlib3.pml

```
let x = [1; 2; 3]
let z = list_map (fun a -> list_tl a) x
let _ = print_bool_list z
```

Listing 79: fail_stdlib4.pml

```
let x = [1; 2; 3]
let z = list_fold_left (fun a l -> a + l) [] x
let _ = print_int z
```

Listing 80: fail_stdlib5.pml

```
let test_cases = [
  "pocaml!";
  "\tpocaml";
  "x :: xs";
  "\"quoted\"";
  1
]

let _ = list_map print_endline test_cases
```

Listing 81: fail_string_literal.pml

```
let x = true && (true && (true && false))
let _ = print_bool x
```

Listing 82: test_and_1.pml

```
let f x = 3 + x
let y = f 4
let _ = print_int y
```

Listing 83: test_apply_1.pml

```
let f x = list_hd x
let g y = (f y) + 3
let _ = print_int (g [1; 2; 3])
```

Listing 84: test_apply_2.pml

```
let x = 5
let y = 10
let z = x + y / 3
let _ = print_int z
```

Listing 85: test_binops_1.pml

```
let x = 3
(* this is a comment *)
let _ = print_int x
```

Listing 86: test_comments_1.pml

```
let x = 3
(* this is a (* nested *) comment *)
let _ = print_int x
```

Listing 87: test_comments_2.pml

```
let x = if 3 + 2 > 4 then 4 else 6
let _ = print_int x
```

Listing 88: test_conditional_1.pml

```
let f1 x = 10 * x
let f2 x = 100 * x
let g = 50
let result = if g > 50 then f2 g else f1 g
let _ = print_int result
```

Listing 89: test_conditional_2.pml

```
let x = 1
let y = x :: [2; 3; 4]
let _ = print_int_list y
```

Listing 90: test_cons_1.pml

```
let x = 'h'
let y = x :: ['e'; 'l'; 'l'; 'o']
let _ = print_char_list y
```

Listing 91: test_cons_2.pml

```
let f = function
    1 -> "1"
  | 2 -> "2"
  | _ -> "_"

let _ = print_string "f 1 = "; print_endline (f 1)
let _ = print_string "f 2 = "; print_endline (f 2)
let _ = print_string "f others = "; print_endline (f 3)
```

Listing 92: test_function_1.pml

```
let f = function
    hd :: tl -> tl
  | [] -> "empty"

let _ = print_string "f [1; 2; 3] = "; print_int_list (f [1; 2; 3])
let _ = print_string "f others = "; print_endline (f [])
```

Listing 93: test_function_2.pml

```
let f = function
    true -> "true"
  | _ -> "false"

let _ = print_string "f true = "; print_endline (f true)
let _ = print_string "f others = "; print_endline (f false)
```

Listing 94: test_function_3.pml

```
let x =
    let y =
        let z =
            let f = 10 in
        f + 5 in
    z + 2 in
y + 1

let _ = print_int x
```

Listing 95: test_letin_1.pml

```
let x =
    let y =
        let z =
            let f = true in
        f || false in
    z && true in
y || false

let _ = print_bool x
```

Listing 96: test_letin_2.pml

```
let x = false || (false || (false || true))
let _ = print_bool x
```

Listing 97: test_or_1.pml

```
let f a = match a with
| 2 -> print_int a
| 1 -> print_int 3
| _ -> print_int 10

let _ = f 3
```

Listing 98: test_pattern_matching_1.pml

```
let f a b = match a with
| 1 -> match b with
    | 2 -> print_int a
    | _ -> print_int b
| 2 -> match b with
    | 1 -> print_int a
    | _ -> print_int b
| _ -> print_int 0

let _ = f 1 2
```

Listing 99: test_pattern_matching_2.pml

```
let f a = match a with
| hd :: tl -> list_length tl
| [] -> 0
| _ -> error "Input should be a list"

let _ = print_int (f [1; 1; 1; 1])
```

Listing 100: test_pattern_matching_3.pml

```
let f a = match a with
| hd :: tl -> match tl with
    | hd1 :: tl1 -> hd :: hd1 :: (list_rev tl1)
    | _ -> tl
| _ -> []

let _ = print_int_list (f [1; 2; 3; 4])
```

Listing 101: test_pattern_matching_4.pml

```
let rec f = function
    | xh :: xs -> xh + f xs
    | _ -> 0
let y = f [1; 4; 5; 6; 7]
let _ = print_int y
```

Listing 102: test_rec_1.pml

```
let x = [1]
let z = list_hd x
let _ = print_int z
```

Listing 103: test_std_head_1.pml

```
let x = [1; 3; 4; 5]
let z = list_tl x
let _ = print_int_list z
```

Listing 104: test_std_tail_1.pml

```
let x = [1; 2; 3]
let z = list_fold_right (fun a l -> a :: l) x []
let _ = print_int_list z
```

Listing 105: test_stdlib1.pml

```
let x = [1; 2; 3]
let z = list_rev x
let _ = print_int_list z
```

Listing 106: test_stdlib2.pml

```
let x = [1; 2; 3]
let z = list_filter (fun a -> (a > 1)) x
let _ = print_int_list z
```

Listing 107: test_stdlib3.pml

```
let x = [1; 2; 3]
let z = list_map (fun a -> (a > 1)) x
let _ = print_bool_list z
```

Listing 108: test_stdlib4.pml

```
let x = [1; 2; 3]
let z = list_fold_left (fun a l -> a + l) 0 x
let _ = print_int z
```

Listing 109: test_stdlib5.pml

```
let test_cases = [
  "pocaml!";
  "\tpocaml";
  "x :: xs";
  "\"quoted\""
]

let _ = list_map print_endline test_cases
```

Listing 110: test_string_literal.pml

8.6 scripts

```
FROM ubuntu:focal
ENV TZ=America/New_York
ARG DEBIAN_FRONTEND=noninteractive
RUN apt-get update -yq && \
    apt-get upgrade -yq && \
    apt-get install -yq --no-install-suggests --no-install-recommends \
        ocaml \
        menhir \
        llvm-11 \
        llvm-11-dev \
        m4 \
        git \
        aspcud \
        ca-certificates \
        python2.7 \
        pkg-config \
```

```
        cmake \
        opam && \
    ln -s /usr/bin/lli-11 /usr/bin/lli && \
    ln -s /usr/bin/llc-11 /usr/bin/llc

RUN opam init --auto-setup --yes --disable-sandboxing

WORKDIR '/pocaml'
COPY ./pocaml.opam .
RUN opam install . --deps-only --yes

WORKDIR '/home/pocaml'

ENTRYPOINT ["opam", "config", "exec", "--"]

CMD ["bash"]
```

Listing 111: Dockerfile

```
#!/usr/bin/env bash

# stop script when a command returns with non-zero
set -e

# important names
fpath=""
fname=""
basename=""
build_dir="_pml_build"
builtins_ar="pml_builtins.a"
stdlib_dir="stdlib"

# get tools
POCAML="dune exec -- bin/main.exe"
LLC=llc
CC=cc
OCAMLC=ocamlc

KEEP_BUILD_DIR=false
COMPILE_C_LIB_ONLY=false
CLEAN_PREV_BUILD=false
RUN_AFTER_COMPILATION=false
SKIP_BUILD_C_LIB=false
TYPECHECK=false
POCAMLC_FLAGS=""

Usage() {
  echo
  echo "usage: pocamlc [otpions] [-f <path_to_pml_file>]"
  echo
  echo "options:"
  echo "-a  Only compile Pocaml's C static library."
  echo "-b  Use existing ./_pml_build and its files for build"
  echo "-c  Clean C object files from previous build before rebuild."
  echo "-d  Compile Pocaml C builtins with debugging information."
  echo "-r  Run the executable after compilation."
  echo "-s  Skip building Pocaml's C static library."
  echo "-t  Typecheck with ocamlc."
  echo "-x  Clean and remove all build artifacts."
```

```
  echo "-h  Display pocamlc usage."
  echo

  exit 0;
}

BuildCLib() {
    [ "$SKIP_BUILD_C_LIB" = true ] && return

    cd builtins
    ([ "$CLEAN_PREV_BUILD" = true ] || [ "$COMPILE_C_LIB_ONLY" = true ]) && make
    clean
    make PML_CFLAGS="${POCAMLC_FLAGS}"
    [ -d "../${build_dir}" ] || mkdir ../${build_dir}
    cp ${builtins_ar} ../${build_dir}
    cd ..

    [ "$COMPILE_C_LIB_ONLY" = true ] && exit 0
    echo
}

[ $# -eq 0 ] && Usage

# parse options and argument
while getopts 'abcdrstxf:h' opt; do
  case $opt in
    a)  # only compile pocaml C library into $build_dir
  COMPILE_C_LIB_ONLY=true
  BuildCLib
  ;;
    b)  # keep &build_dir if already exists
  KEEP_BUILD_DIR=true
  ;;
    c)  # clean C object files
  CLEAN_PREV_BUILD=true
  ;;
    d) # compile with debugging info
  POCAMLC_FLAGS+=" -D BUILTIN_DEBUG"
  ;;
    r) # run compiled executable
  RUN_AFTER_COMPILATION=true
  ;;
    s)  # skip building pocaml C library
  SKIP_BUILD_C_LIB=true
  ;;
    t)  # type check before compilation
  TYPECHECK=true
  ;;
    f) # file
  fpath=$OPTARG
  fname=${fpath##*/}
  basename="${fname%.*}"
  ;;
    x) # clean build artifacts
  rm -rf $build_dir
  dune clean
  cd builtins
  make clean
  cd ..
```

```
  exit
  ;;
    h | *) # help
  Usage
  ;;
  esac
done

# create build_dir
[ "$KEEP_BUILD_DIR" = false ] &&
(
    rm -rf $build_dir
    mkdir $build_dir
)

# if '-f' specified, copy source file to build_dir
# else, get input from stdin
if [[ $fpath ]]; then
  cp ${fpath} ${build_dir}/${fname}.orig
else
  fname="a.pml"
  fpath=${build_dir}/${fname}
  basename="${fname%.*}"
  cat > $fpath.orig
fi

# prepend stdlib
fname_with_stdlib=${build_dir}/${fname}
echo > $fname_with_stdlib
for stdlib_file in ${stdlib_dir}/**; do
  { printf "(* pocamlc: %s *)\n" $stdlib_file;
    cat $stdlib_file;
    printf "\n" >> $fname_with_stdlib;
  } >> $fname_with_stdlib
done
{ printf "(* pocamlc: %s *)\n" $fname;
  cat ${build_dir}/${fname}.orig
} >> $fname_with_stdlib


# typecheck with ocamlc
if [ $TYPECHECK = true ]; then
  cp builtins/builtins.ml ${build_dir}/
  cd $build_dir
  {
    cat builtins.ml
    printf "\n"
    cat $fname
  } > ${basename}.ml
  $OCAMLC -i ${basename}.ml
  rm -f builtins.ml ${basename}.ml
  cd ..
fi

# pocaml -> llvm
$POCAML -c ${build_dir}/${fname}> ${build_dir}/${basename}.ll

# build builtins C static library
BuildCLib
```

```

# link the generated llvm with builtin
cd ${build_dir}
$LLC -relocation-model=pic ${basename}.ll > ${basename}.s
$CC -o ${basename}.exe ${basename}.s ${builtins_ar}

# if '-r' specified, execute
[ "$RUN_AFTER_COMPILATION" = true ] && ./${basename}.exe

# exit successfully
exit 0
```

Listing 112: pocamlc

```
#!/usr/bin/env bash

# stop script when a command returns with non-zero
set -e

POCAML="./pocamlc"
PFLAGS=""
COMPILE_ONLY=0
FROM_STDIN=0
CLEAN=0

c_lib="_pml_build/pml_builtins.a"

Usage()
{
  echo
  echo "usage:  pocaml [options] [<path_to_pml_file>...]"
  echo
  echo "-c  Only compile .pml files"
  echo "-h  Display this help"
  echo "-x  Clean and remove all build artifacts before compilation"
  echo

  exit 1
}

SetFlags()
{
    if [ $COMPILE_ONLY -eq 1 ]
    then
  PFLAGS="-c"
    else
  PFLAGS="$PFLAGS -r"

  if [ -f $c_lib ] && [ $CLEAN -eq 0 ]
  then
      # use existing C static lib
      PFLAGS="$PFLAGS -bs"
  else
      PFLAGS="$PFLAGS -c"
  fi
    fi

    if [ $FROM_STDIN -eq 0 ]
    then
```

```
  PFLAGS="$PFLAGS -f"
    fi

    echo "reached here flags: $PFLAGS"
}

[ $# -eq 0 ] && Usage

while getopts chx opt; do
    case $opt in
  c)
      COMPILE_ONLY=1
      ;;
  x)
      CLEAN=1
      ;;
  *)
      Usage
      ;;
    esac
done

shift $(($OPTIND - 1))

if [ $# -eq 0 ]
then
    FROM_STDIN=1
fi

SetFlags

for file in $@
do
    $POCAML $PFLAGS $file
done

if [ $FROM_STDIN -ne 0 ]
then
    $POCAML $PFLAGS
fi

exit 0
```

Listing 113: pocaml

```
#!/bin/bash

docker_tag=pocaml/pocaml
docker_work_dir=/home/pocaml

docker_cmd_override=$*

docker build -t $docker_tag . > /dev/null 2>&1

if [ $? -ne 0 ]; then
    echo "docker build failed"
else
    set -e
fi
```

```

docker run -it -v "$(pwd)":"$docker_work_dir" -w="$docker_work_dir" $docker_tag
    $docker_cmd_override
```

Listing 114: runDocker

```
POCAMLC="./pocamlc"

ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0
pflags="-c"

Usage() {
    echo "Usage: testall.sh [options] [.pml files]"
    echo "-k    Keep intermediate files"
    echo "-l  Run tests locally"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
  echo "FAILED"
  error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
  SignalError "$1 differs"
  echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
  SignalError "$1 failed on $*"
  return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
```

```
    eval $* && {
  SignalError "failed: $* did not report an error"
  return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.pml//'`
    reffile=`echo $1 | sed 's/.pml$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
    testdir="`echo $1 | sed 's/\/[^\/]*$//'`"
    build_dir="_pml_build"

    echo "$basename...\c"

    echo 1>&2
    echo "###### Testing $basename" 1>&2
    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.diff ${basename}.out" &&
    Run $POCAMLC '-bsf' ${testdir}/${basename}.pml 1>&2 &&
    Run "${build_dir}/${basename}.exe" > ${basename}.out &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
  echo "OK"
  echo "###### SUCCESS" 1>&2
    else
    echo "FAILED"
  echo "###### FAILED" 1>&2
  globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.pml//'`
    reffile=`echo $1 | sed 's/.pml$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
    testdir="`echo $1 | sed 's/\/[^\/]*$//'`"
    build_dir="_pml_build"

    echo "$basename...\c"

    echo 1>&2
    echo "###### Testing $basename" 1>&2
    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.diff ${basename}.err" &&
    RunFail $POCAMLC '-bsrf' ${testdir}/${basename}.pml "2>" ${basename}.err ">>"
```

```
    $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff
    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
    echo "OK"
  echo "###### SUCCESS" 1>&2
    else
  echo "###### FAILED" 1>&2
  globalerror=$error
    fi
}

CompileLib() {
    Run $POCAMLC -a 1>> $globallog 2>&1
}

while getopts khl c; do
    case $c in
  k) # Keep intermediate files
      keep=1
      ;;
  l) # Run tests locally
      pflags="-l $pflags"
      ;;
  h) # Help
      Usage
      ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="test/pml/*.pml"
fi

CompileLib

for file in $files
do
    case $file in
  *test_*.pml)
      Check $file 2>> $globallog
      ;;
      *fail_*.pml)
      CheckFail $file 2>> $globallog
      ;;
  *)
      echo "unknown file type $file"
      globalerror=1
      ;;
    esac
```

```
171 done
172
173 exit $globalerror
```

Listing 115: testall.sh