Lab1 & Lab2 Report

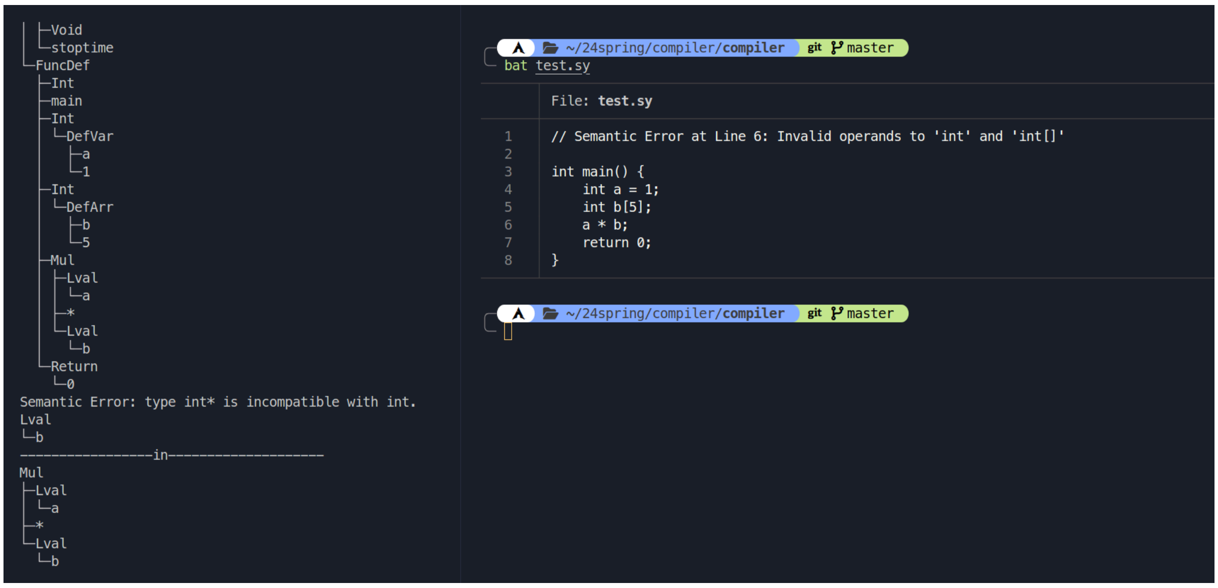
高晨昊 3210106354

Functionality

The compiler is implemented in OCaml from scratch with ocamllex and menhir.

It now can build the AST of a given SysY program(lab1), and then typecheck it(lab2). It has passed all the tests.

In addition, the compiler can print the ast in tree format, and detect different semantic errors with context information(the sub-ast where the error happens). Here is a glimpse.



```
│ ├─Void
│ └─stoptime
└─FuncDef
  ├─Int
  ├─main
  ├─Int
  │ └─DefVar
  │   ├─a
  │   └─1
  ├─Int
  │ └─DefArr
  │   ├─b
  │   └─5
  ├─Mul
  │ ├─Lval
  │ │ └─a
  │ ├─*
  │ └─Lval
  │   └─b
  └─Return
    └─0
Semantic Error: type int* is incompatible with int.
Lval
└─b
-----------------in--------------------
Mul
├─Lval
│ └─a
├─*
└─Lval
  └─b
```

```
~/24spring/compiler/compiler  git master
bat test.sy
       File: test.sy
   1   // Semantic Error at Line 6: Invalid operands to 'int' and 'int[]'
   2
   3   int main() {
   4       int a = 1;
   5       int b[5];
   6       a * b;
   7       return 0;
   8   }

~/24spring/compiler/compiler  git master
```

Figure 1:

Type Definition (`ast.ml`)

To define the type of an ast node, I came up with two methods. First, we can construct a type system that directly corresponds to the grammar. So the result will be something like:

```ocaml
type decl = btype * var_def list
and comp_unit = either_decl_funcdef list
and var_def = DefVar of id * exp | DefArr of id * int_const list
and func_def = func_type * id * func_f_params * block
and func_type = Void | Int
and func_f_params = func_f_param list
```

The benefit of this method is that the straitforward correspondance makes it easy to walk through the tree. If we get a type of `func_def`, we know immediately that it's a 4-tuple with `func_type`, `id`, `func_f_params`, and `block`. And we can call corresponding functions to handle them.

This is quite helpful especially when the analyzer can get more information from the structrue and give us more intuition about how to write the code, like whether a pattern matching is exhuastive or how to bind values. But the fact that every type is different from each other make it hard to dispatch functions for each type. We must write typechecker for each type and call them everywhere we get an instance of that type.

This results in ugly and hard-to-maintain code. So I turned to the second method, which is to use a single type to represent all the ast nodes:

```
type ast =
  | Decl of ast list (* DefVar | DefArr list *)
  | CompUnit of ast list (* FuncDef | Decl list *)
  | DefVar of string * ast (* Exp*)
  | DefArr of string * int list
  | FuncDef of
      value_type
      * string
      * ast list
      * ast (* IntParam | ArrParam list * Block *)
```

This definition is more ambiguous, although more concise. The main reason is that we know that `DefVar` will only be comprised of `string * exp`, but we can only refer to ast when definining it. Also, when we are pattern matching the second part of DefVar, it will not immediately be a `exp`, but an `ast`. We must further pattern match it to get the `exp`, although we know it's impossible to be anything else from the `inversion`, which is to say, the only way to construct a `DefVar` in parser is to combine a string with a exp, so the second part of `DefVar` must be an `exp`.

To get around the warning of non-exhaustive pattern matching, we can use `[@@warning "-8"]` to suppress it and recover it through `[@@warning "+8"]`.

If-then-else Problem

An classic problem when implementing a compiler is how to handle if-then-else ambiguity. We want it to be closet matching. And this can be done by rewriting the grammar or suppressing a precedence rule. To make it more readable, I choose the latter.

Fisrt declare the precedence

```
%nonassoc THEN
%nonassoc ELSE
```

where THEN is a phantom token that is just for precedence, and ELSE is the real token.

Then we can write the grammar like this:

```
| IF; LPARE; e = exp; RPARE; s1 =stmt; ELSE; s2 =stmt {IfElse (e, s1, s2)}
| IF; LPARE; e = exp; RPARE; s =stmt; {IfThen (e, s)}  %prec THEN
```

after reading stmt, the parser will choose the following else branch if there is one, since ELSE has higher precedence than THEN.

Type Checking

Here is the definition of value types:

```
type value_type =
  | IntType
  | VoidType
  | ArrayType of int list
  | FuncType of value_type * value_type list
[@@deriving equal]
```

The definition is straitforward, the `ArrayType` is something like `int(*)[10][10]`, and will be just `int*` if the list is empty. `FuncType` has one return type and a list of parameter types.

The table is updated in a functional way. Actually, we use `Map.Poly.t` to store the table and here are the type definitions:

```
type table= (string, value_type) Map.Poly.t
type ctx = table list
type ctxes = ctx * ctx (* func_ctx, var_ctx*)
```

each table is a map from id to it's type. And we have a separate table for functions and variables.

To type check an ast, we can take three different actions(my own terminology):
- `Check/TypeCheck`: check the all the content of the tree, return `unit`
- `Get`: get the type of the tree, it must be something typeable, return `value_type`
- `Update`: update the context, return `ctxes`

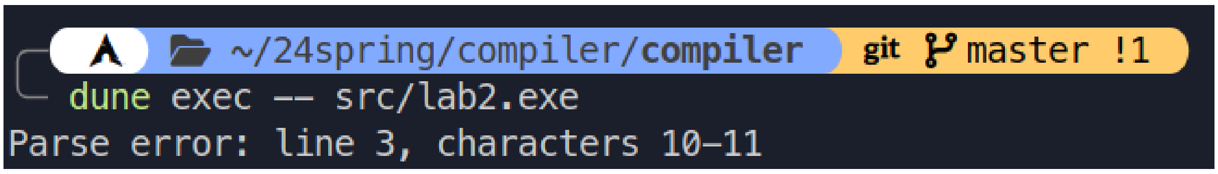
For example, given a program, we will check it, since we dont want get anything back from the program. Given a list of stmt, we will update through it, since the context will be changed as we go though it. Given a exp, we will get the type of it, since we want to know the type of the expression to conduct the type checking.

The whole procedure will be:
- typecheck program
- update through comp_unit
  - decl and func_def will update the context
  - for func_def, we will also go into the block and update through it, where we will carry a return_type to check it
  - get_exp is called when we need to confirm some constraints is met.

During the updating, the ctxes will be passed around by `List.fold_left`. If we meet a new scope, we will push a new table into ctxes and pass it into the block. There is no need to pop the table since the ctxes outside the block will not be affected by the passing.

Error Handlling

Currently, the compiler will report two errors: Syntax Error and Semantic Error. Syntax Error will be report when we failed to get the ast from the parser, the exception will be catched and an error message about the the position of the failure will be printed.



```
~/24spring/compiler/compiler  git  master !1
dune exec -- src/lab2.exe
Parse error: line 3, characters 10-11
```

Figure 2:

For Semantic Error, it will first be raised with a string to illustrate the error. However, as the exception goes up, it will be cautht by the callers where we can access the information about the sub-ast where the error happens and even its parent sub-ast, which is helpful. So the handler will add this information to the Exception and raise it again. Finally the top level of type checker will catch the exception and print all the informatioin inside.

The catch and raise mechanism looks like this:

```
let try_check f tr =
  try f () with
  | SemanticErrorWithCurTree (msg, tree) ->
      raise (SemanticErrorWithCurTreeParentTree (msg, tree, tr))
  | SemanticError msg -> raise (SemanticErrorWithCurTree (msg, tr))
```

If the exception has no tree or only one tree with it, we will add another one.

The handler can be installed by:

```
and check_stmt_exn ctxes stmt return_type =
  let f () = check_stmt ctxes stmt return_type in
  try_check f (ast_to_tree stmt)
```

Tricky Part

Some part of the typechecker is tricky. Especially the part about array type. We must check it carefully. For example, when indexing over an array, there are four case:

- no index at all, the return type is original array type
- too much index, raise an error
- exact index, the return type is int
- some index, the return type is array type with the rest of the dims

Other parts are trivial.

Run, Compile and Test

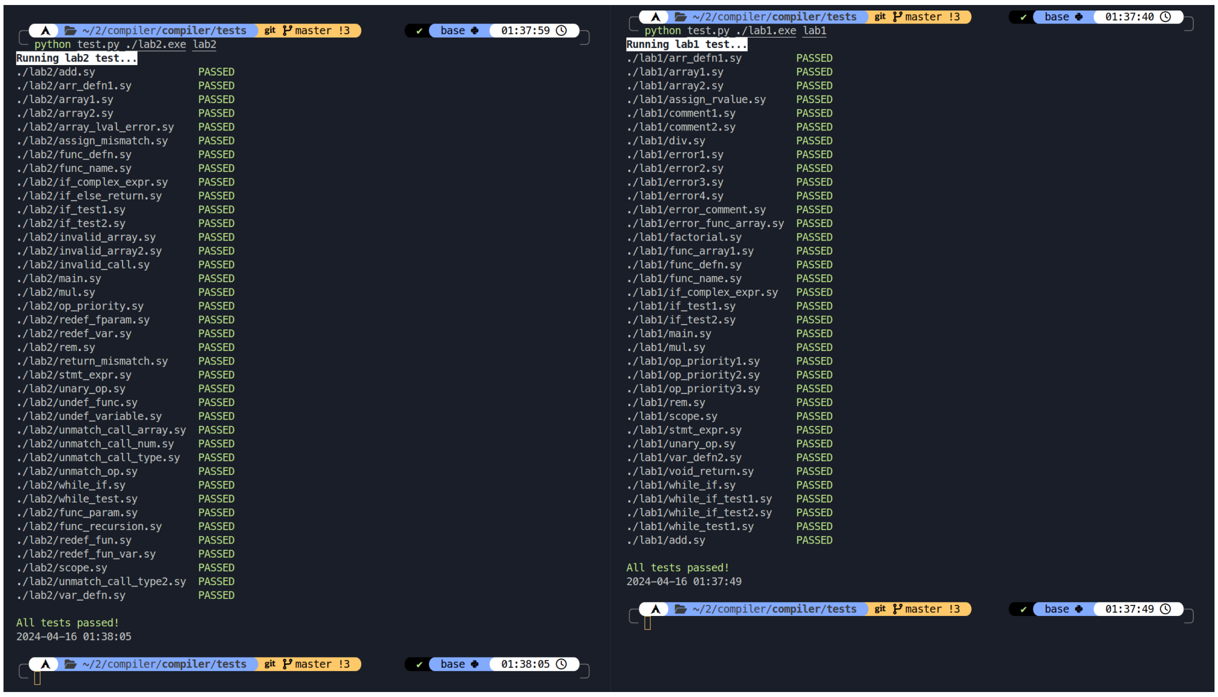
To compile the program. If no immediate OCaml environment is available, docker is recommended. Just use `ocaml/opam:latest` image. debian:latest works fine too.

Here I reiterate the steps to compile the program:

> Tested on docker image debian:latest and ocaml/opam:latest, the latter is recommended where opam is immediately available.
>
> ```
> #debian:latest only
> apt install opam
> opam init
> eval $(opam env --switch=default)
>
> #install dependencies
> #make sure compiler.opam is in .
> cd <dict>
> opam install . --deps-only
>
> #compile
> dune build
>
> #Done, the executables are under _build/default/src/lab1.exe and _build/default/
> src/lab2.exe
> ```
>
> To run the executable, just type `./lab1.exe <input.sy>`, `./lab2.exe <input.sy>` when executables are in . or run `dune exec -- src/lab1.exe <input.sy>`. If `<input.sy>` is left blank, `test.sy` will be the default.
>
> Other Information
>
> ```
> opam --version
> 2.0.10
> ocaml --version
> The OCaml toplevel, version 5.1.1
> ```
>
> — README.md

The dependency includes `menhir` `ppx_jane` `printbox` `printbox-text`.

Here is the result of the test:



```
~/2/compiler/compiler/tests  git  master !3            ✓  base  01:37:59
python test.py ./lab2.exe lab2
Running lab2 test...
./lab2/add.sy                PASSED
./lab2/arr_defn1.sy          PASSED
./lab2/array1.sy             PASSED
./lab2/array2.sy             PASSED
./lab2/array_lval_error.sy   PASSED
./lab2/assign_mismatch.sy    PASSED
./lab2/func_defn.sy          PASSED
./lab2/func_name.sy          PASSED
./lab2/if_complex_expr.sy    PASSED
./lab2/if_else_return.sy     PASSED
./lab2/if_test1.sy           PASSED
./lab2/if_test2.sy           PASSED
./lab2/invalid_array.sy      PASSED
./lab2/invalid_array2.sy     PASSED
./lab2/invalid_call.sy       PASSED
./lab2/main.sy               PASSED
./lab2/mul.sy                PASSED
./lab2/op_priority.sy        PASSED
./lab2/redef_fparam.sy       PASSED
./lab2/redef_var.sy          PASSED
./lab2/rem.sy                PASSED
./lab2/return_mismatch.sy    PASSED
./lab2/stmt_expr.sy          PASSED
./lab2/unary_op.sy           PASSED
./lab2/undef_func.sy         PASSED
./lab2/undef_variable.sy     PASSED
./lab2/unmatch_call_array.sy PASSED
./lab2/unmatch_call_num.sy   PASSED
./lab2/unmatch_call_type.sy  PASSED
./lab2/unmatch_op.sy         PASSED
./lab2/while_if.sy           PASSED
./lab2/while_test.sy         PASSED
./lab2/func_param.sy         PASSED
./lab2/func_recursion.sy     PASSED
./lab2/redef_fun.sy          PASSED
./lab2/redef_fun_var.sy      PASSED
./lab2/scope.sy              PASSED
./lab2/unmatch_call_type2.sy PASSED
./lab2/var_defn.sy           PASSED

All tests passed!
2024-04-16 01:38:05
~/2/compiler/compiler/tests  git  master !3            ✓  base  01:38:05
```

```
~/2/compiler/compiler/tests  git  master !3            ✓  base  01:37:40
python test.py ./lab1.exe lab1
Running lab1 test...
./lab1/arr_defn1.sy        PASSED
./lab1/array1.sy           PASSED
./lab1/array2.sy           PASSED
./lab1/assign_rvalue.sy    PASSED
./lab1/comment1.sy         PASSED
./lab1/comment2.sy         PASSED
./lab1/div.sy              PASSED
./lab1/error1.sy           PASSED
./lab1/error2.sy           PASSED
./lab1/error3.sy           PASSED
./lab1/error4.sy           PASSED
./lab1/error_comment.sy    PASSED
./lab1/error_func_array.sy PASSED
./lab1/factorial.sy        PASSED
./lab1/func_array1.sy      PASSED
./lab1/func_defn.sy        PASSED
./lab1/func_name.sy        PASSED
./lab1/if_complex_expr.sy  PASSED
./lab1/if_test1.sy         PASSED
./lab1/if_test2.sy         PASSED
./lab1/main.sy             PASSED
./lab1/mul.sy              PASSED
./lab1/op_priority1.sy     PASSED
./lab1/op_priority2.sy     PASSED
./lab1/op_priority3.sy     PASSED
./lab1/rem.sy              PASSED
./lab1/scope.sy            PASSED
./lab1/stmt_expr.sy        PASSED
./lab1/unary_op.sy         PASSED
./lab1/var_defn2.sy        PASSED
./lab1/void_return.sy      PASSED
./lab1/while_if.sy         PASSED
./lab1/while_if_test1.sy   PASSED
./lab1/while_if_test2.sy   PASSED
./lab1/while_test1.sy      PASSED
./lab1/add.sy              PASSED

All tests passed!
2024-04-16 01:37:49
~/2/compiler/compiler/tests  git  master !3            ✓  base  01:37:49
```

Figure 3: