Ian Campbell CS391 Finexam Report 6/25/2025

Journey Through a Compiler

Abstract

This report documents the design and implementation of a small compiler for a simply-typed λ-calculus–inspired language, written in Python. The compiler comprises two main passes: a type-checking phase that enforces static typing and a code-generation phase that translates typed abstract syntax trees (AST) into an instruction set before emitting executable Python code to the terminal. This report will describe the core data structures, logic, and design choices; and we illustrate the end to end compilation of three non trivial examples; Factorial, isPrime, and Fibinocci's Sequence. Unfortunately I was not able to get around to the 8 Queens puzzle, due to me wanting to fully understand everything I was writing and running out of time to implement further; however, this compiler compilers everything perfectly with what it is compatible with.

---

1. Language and AST Overview

The language's building blocks are all represented by distinct AST nodes, each encoding both syntax and intent. Integer literals like TMint(n) stand for the constant value n, while boolean literals TMbtf(b) represent the truth value b. A variable node TMvar(x) refers to the value bound to name x in the current scope. Anonymous functions are written as TMlam(x, τ, e), meaning "λ x:τ.e," and can be applied to arguments via TMapp(f, a), which denotes f a. To support recursion, TMfix(f, x, τ_1, τ_2, e) declares a function f of type $\tau_1 \rightarrow \tau_2$ whose body 'e' may call f on x. Primitive operations such as addition, multiplication, comparisons, ect., are encoded by TMopr(op, [e_1,…,$e_{\square}$]), where op is a string such as "+" or "<=". Conditional execution uses TMif0(t, e_1, e_2); if the test t evaluates to zero or false, the result is e_1; otherwise it's e_2. Compound data appears as pairs via TMtup(e_1,e_2), and the first or second projection is extracted with TMfst(e) and TMsnd(e), respectively. Finally, local definitions use TMlet(x, b, e), which binds the result of b to x when evaluating the body e.

These constructs are represented by Python classes inheriting from a common term base class which contains a field called ctag, this field contains the type of constructor the structure is. Each node carries the fields needed to complete compilation correctly. For example in term_opr, arg1 represents the operation taking place, arg2 holds the list for the arguments that the operation is taking place on. Term_op's ctag would be "TMopr" which mirrors the tag for the AST. These fields are the backbone to the constructs that give them meaning and purpose.

---

2. Static Types and Contexts

In this implementation, we implement a simple static type system with three type constructors: base which are integers and bools, function types and tuple types. By limiting ourselves with a three type system, we get a type system that's both powerful enough to express most common programming patterns in which you can build lists, records, higher‑order functions, etc., from these primitives, while it remains simple enough that the rules for checking types stay easy to understand. This minimal design makes the implementation straightforward, the proofs of soundness much easier, and it gives you a clean foundation you can later extend with more advanced features without getting lost in complexity. Hwoever, this is also an area I would need to extend logic to be able to compile the 8 Queens Puzzle. Due to my compiler's inability to compile any recursive or container types such as lists or arrays, it is not possible to compile partial solutions from building and traversing lists, let alone compiling the board for the problem in the first place. Therefore, in order to extend my project to be able to compile the 8 Queens Puzzle, there would need to be an addittion of recursive and container types.

2.1 Type Equality

Structural type equality is determined by the recursive function styp_equal, implemented through Python classes inheriting from a common styp base class. This function evaluates whether two given types are identical not just in their memory references but in their fundamental structural composition. It first compares the ctag field, which uniquely indicates the type constructor (STbas, STfun, or STtup), ensuring both types belong to the same category. After confirming matching constructor tags, styp_equal recursively inspects their corresponding internal fields. For instance, when comparing base types (STbas), it verifies that their internal names, such as "int" or "bool", match precisely. Function types (STfun) undergo deeper inspection by recursively checking both their domain and codomain types for structural equality; thus, the type

STfun(int,bool) structurally equals itself, but differs from STfun(int,int) due to mismatched codomain fields. Similarly, tuple types (STtup) necessitate checking both the first and second component types in their given order, distinguishing STtup(int,bool) from STtup(bool,int). These meticulous recursive checks on ctag and constituent fields are fundamental in accurately establishing structural equality, enabling the type system to reliably and precisely differentiate between genuinely equivalent and merely similar types.

---

3. Type-Checking Pass

The type-checking pass is anchored by the Python function term_tpck00, which initiates the recursive checking process by calling term_tpck01 with an empty type context (CXnil()). This helper function, term_tpck01, examines each AST node by inspecting its ctag and recursively ensuring type correctness based on its structure. The following are compatible ctags in the type checker;

Integer literals (TMint) immediately yield the base type STbas("int"), while boolean literals (TMbtf) produce the base type STbas("bool"). Variables (TMvar) are resolved by a lookup in the current type context; if the variable is unbound, the type checker signals an error.

Lambda abstractions (TMlam(x, τ, e)) are checked by first extending the current context with the binding x:τ, then recursively type-checking the body e. The resulting type for the lambda node is STfun(τ, τ_e), where τ_e is the inferred type of the lambda's body.

Applications (TMapp(e_1,e_2)) undergo a two-step verification: the first expression (e_1) must have a function type STfun(τ_arg, τ_res), while the second expression (e_2) must match the argument type τ_arg. If this condition holds, the application node itself has type τ_res.

Primitive operations (TMopr(op, args)) enforce strict arity and operand-type checks according to the operation's semantics. For example, arithmetic operators such as "+" demand exactly two integer operands and return an integer type, while comparison operators require integers and return booleans. If the operands do not conform exactly, a clear type error arises.

Conditional expressions (TMif0) mandate that the test condition is of boolean type. Both the “then” and “else” branches are recursively checked, and they must have identical types; the conditional’s type is that shared type. Any deviation in these requirements results in an error.

Recursive definitions (TMfix(f,x,τ_x,τ_r,body)) are validated by extending the context with both f as a function type STfun(τ_x,τ_r) and x as type τ_x, then recursively type-checking the function’s body. The resulting type must precisely match the declared return type (τ_r). Upon successful verification, the overall type for the fixpoint node is confirmed as STfun(τ_x,τ_r).

Tuples (TMtup(e_1,e_2)) undergo recursive type-checking of their two subexpressions, forming the resulting pair type STtup($type_1$,$type_2$). Projections (TMfst(e) and TMsnd(e)) require their subexpression e to possess a tuple type (STtup(τ_1,τ_2)), subsequently yielding the respective component type (τ_1 or τ_2).

Finally, local bindings (TMlet(x,bound,body)) validate by first type-checking the bound expression (bound), extending the context with the new binding x:τ_b, then recursively type-checking the body. The type inferred for the entire let-expression is that of the body.

Throughout this type-checking phase, assertions ensure correctness at every step. Any mismatch in types, incorrect operator arities, or unbound variable usage triggers a concise AssertionError or TypeError, terminating compilation and immediately pinpointing the source of the error.

---

4. Compilation to Intermediate Representation

The language’s AST is translated into a concise intermediate representation (IR) using three-address instructions, defined by Python classes inheriting from a common tins base class. Each subclass represents a specific IR instruction form: tins_mov places literal values into registers, tins_opr applies primitive operations (such as arithmetic or comparisons) to operands, tins_app handles function application, tins_fun encodes function definitions, and tins_if0 manages conditional execution.

IR computations (tcmp) bundle two key elements together: a sequential list of tins instructions representing the exact computation steps, and a designated destination register (treg) that stores the final computation result. Registers themselves (treg) are virtual placeholders uniquely identified by prefixes (tmp, arg, or fun) along with numerical suffixes.

4.1 Registers and Environments

The compilation environment (cenv) maintains a mapping of source-level variables to these registers. Fresh register identifiers are consistently generated by helper functions—ttmp_new(), targ_new(), and tfun_new()—to ensure uniqueness and avoid conflicts.

4.2 Compilation Rules

The entry point for compilation is term_comp00(tm), which begins translating an AST by invoking term_comp01(tm, CENVnil()). Literal nodes (TMint and TMbtf) compile directly to move instructions (tins_mov). Variables (TMvar) resolve to existing registers found in the environment. Primitive operations (TMopr) compile their operands recursively, concatenate the resulting instructions, and conclude by emitting a tins_opr instruction.

Function applications (TMapp) compile both the function and argument expressions separately, and emit a single application instruction (tins_app). Lambda abstractions (TMlam) generate fresh registers for the function itself and its parameter, extend the compilation environment, compile the body, and emit a function definition instruction (tins_fun). Conditionals (TMif0) compile the condition, "then" branch, and "else" branch separately, then emit a single conditional instruction (tins_if0).

Recursive functions (TMfix) follow similar logic to lambda abstractions, but extend the environment with a binding to the function itself to allow recursion within the body. Tuples (TMtup) compile each component and emit a tuple instruction via tins_opr. Projections (TMfst, TMsnd) compile their tuple operand and emit projection operations. Finally, let-bindings (TMlet) compile the bound expression, extend the environment, compile the body expression under this new context, and combine their instruction lists.

After processing, term_comp00 returns the complete instructions alongside the final destination register packaged into a single tcmp, fully representing the computation's logic and outcome.

5. Python Code Emission

The final pass, embodied by the function tcmp_pyemit, takes a list of instructions from compiling IR (tcmp) and spits out human readable and executable Python code. It begins by inspecting the first instruction: if it's a function definition (TINSfun(fun_reg, body_cmp, arg_reg)), tcmp_pyemit emits a def funXXX(argYYY): header—where XXX and YYY come from the register names and have a helper function name(arg0) where it returns arg0.pffx + arg0.sffx, for example "tmp101"—then recursively emits each of the body's tins instructions with proper indentation before ending the function with return result_reg.

For top level computations or instruction sets that don't start with a funcion, it simply walks through the instruction set in order, translating each into the equivalent Python statement: A move instruction (TINSmov(dst, literal)) becomes an assignment for example dst = 42 or dst = True. An operator instruction (TINSopr(dst, op, [a, b])) is emitted as dst = a + b (or whatever string op is). A function application (TINSapp(dst, f, a)) turns into dst = f(a). A conditional (TINSif0(dst, test, then_cmp, else_cmp)) is rendered as an if test: block, in which the "then" sub instructions are emitted first, followed by dst = then_value, then an else: block for the "else" instructions and dst = else_value.

After all instructions are emitted, tcmp_pyemit appends a final print(dst) to display the program's result. Throughout this process, helper utilities like tins_emit manage indentation levels and name(r) consistently converts virtual registers (e.g. tmp101) into valid Python identifiers, ensuring the emitted code is both correct and human‑friendly.

6. Examples

In order to validate our compiler and emitter end to end, I chose three classic recursive examples—factorial, isPrime, and the Fibonacci sequence—each of which exercises increasingly complex combinations of recursion, arithmetic operations, and branching. Compiling factorial forces us to generate a fixpoint (TMfix) node, allocate fresh registers for the function label and its argument, and emit a correct TINSfun definition that the Python emitter can turn into a def funXXX(argYYY): … return construct. The isPrime example goes further by introducing an inner helper function for trial division, modulo operations (%), and nested conditionals; this pushes us to handle multiple TINSif0 blocks in sequence, maintain proper test and else ordering, and manage two layers of function definitions without name collisions. Finally, Fibonacci amplifies these challenges with mutually recursive calls to the same fixpoint, requiring that our compilation environment correctly re-bind the function register within its own body and that the emitter indent and order the Python code so that recursive calls resolve at runtime. Across all three, I discovered that ensuring fresh, collision free register naming (via ttmp_new, tfun_new, etc.), flattening nested instruction sets into properly indented Python, and preserving the precise semantics of zero tests versus booleans in if/else were the key hurdles to getting clean, executable output.

6.1 Factorial

For testing my compiler, I inserted the AST definition into the compiler, then used that output for the input of the emitter. below is the AST Definition, The type check, and the final output from the emitte

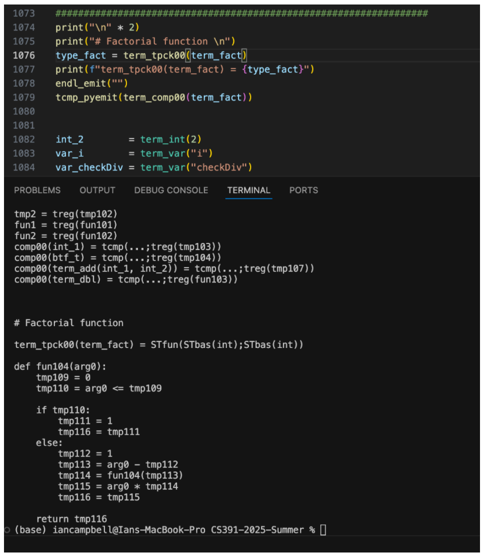
AST Definition:

```
term_fact = term_fix(
    "f","n", styp_int, styp_int,
    term_if0(
      term_lte(var_n, int_0),
      int_1,
      term_mul(var_n,
term_app(var_f, term_sub(var_n,
int_1)))
    )
)
```

Type-check: term_tpck00(term_fact) yields STfun(int,int).

Compiled and emitted:

```
def fun104(arg0):
    tmp109 = 0
    tmp110 = arg0 <= tmp109

    if tmp110:
        tmp111 = 1
        tmp116 = tmp111
    else:
        tmp112 = 1
        tmp113 = arg0 - tmp112
        tmp114 = fun104(tmp113)
        tmp115 = arg0 * tmp114
        tmp116 = tmp115

    return tmp116
```



```
##################################################################
print("\n" * 2)
print("# Factorial function \n")
type_fact = term_tpck00(term_fact)
print(f"term_tpck00(term_fact) = {type_fact}")
endl_emit("")
tcmp_pyemit(term_comp00(term_fact))


int_2        = term_int(2)
var_i        = term_var("i")
var_checkDiv = term_var("checkDiv")
```

```
tmp2 = treg(tmp102)
fun1 = treg(fun101)
fun2 = treg(fun102)
comp00(int_1) = tcmp(...;treg(tmp103))
comp00(btf_t) = tcmp(...;treg(tmp104))
comp00(term_add(int_1, int_2)) = tcmp(...;treg(tmp107))
comp00(term_dbl) = tcmp(...;treg(fun103))



# Factorial function

term_tpck00(term_fact) = STfun(STbas(int);STbas(int))

def fun104(arg0):
    tmp109 = 0
    tmp110 = arg0 <= tmp109

    if tmp110:
        tmp111 = 1
        tmp116 = tmp111
    else:
        tmp112 = 1
        tmp113 = arg0 - tmp112
        tmp114 = fun104(tmp113)
        tmp115 = arg0 * tmp114
        tmp116 = tmp115

    return tmp116
(base) iancampbell@Ians-MacBook-Pro CS391-2025-Summer % 
```

This Python function correctly computes factorial as shown from the photo of my terminal. This was the easiest of functions due to its logic consistenting of an if statement and a single use of recursion.

6.2 IsPrime Test

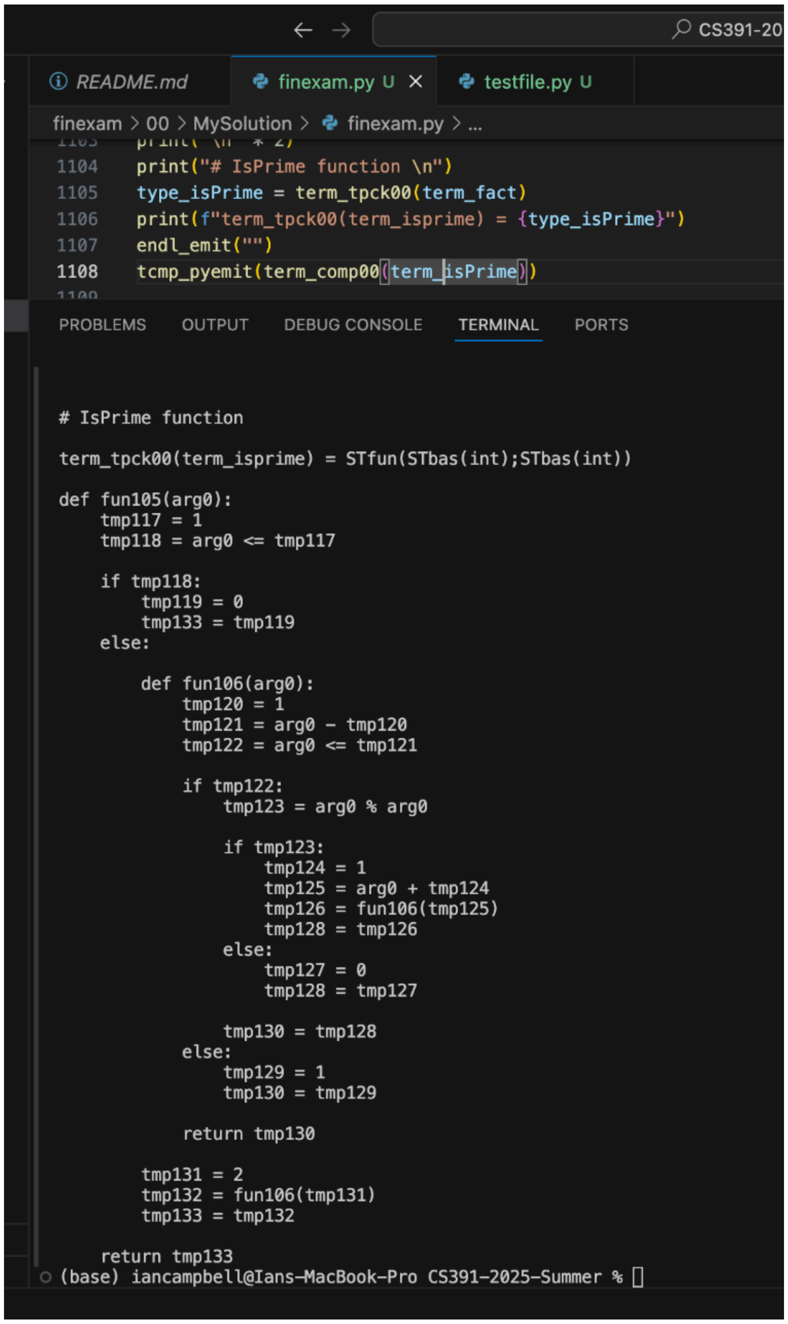
A more involved recursive definition nests two fix constructs to implement trial division up to √n. After type-checking to STfun(int,int), the emitted Python mirrors the nested recursion and yields 0 for prime and 1 for composite (or vice versa, per the chosen convention). This function stumped me for sometime due to a singular typo and redefinition of an already existing variable, "indent". In writing this my indent variable at the time was set to the definition of i in my code now, but I could not figure out why my code was emitting wrong in terms of indedentation. I realized after some time of scanning that I accidentally made my indent variable only a collection of spaces rather than an equation measure spaces by depth. As shown below, my implementation still successfully emits this function as well.

AST Definition:

```
int_2        = term_int(2)
var_i        = term_var("i")
var_checkDiv = term_var("checkDiv")

term_isPrime = \
  term_fix("isPrime", "n", styp_int, styp_int, \
    term_if0(term_lte(var_n, int_1), \
      int_0, \
      term_app( \
        term_fix("checkDiv", "i", styp_int, styp_int, \
          term_if0(term_lte(var_i, term_sub(var_n, int_1)), \
            term_if0(term_mod(var_n, var_i), \
              term_app(var_checkDiv, term_add(var_i, int_1)), \
              int_0), \
            int_1) \
        ), \
        int_2 \
     ) \
    ) \
  )
```
Type-check: term_tpck00(term_isprime) = STfun(STbas(int);STbas(int))

Compiled and Emitted:

```
def fun105(arg0):
    tmp117 = 1
    tmp118 = arg0 <= tmp117

    if tmp118:
        tmp119 = 0
        tmp133 = tmp119
    else:

        def fun106(arg0):
            tmp120 = 1
            tmp121 = arg0 - tmp120
            tmp122 = arg0 <= tmp121

            if tmp122:
                tmp123 = arg0 % arg0

                if tmp123:
                    tmp124 = 1
                    tmp125 = arg0 + tmp124
                    tmp126 = fun106(tmp125)
                    tmp128 = tmp126
                else:
                    tmp127 = 0
             tmp128 = tmp127

                tmp130 = tmp128
            else:
                tmp129 = 1
                tmp130 = tmp129

            return tmp130

        tmp131 = 2
        tmp132 = fun106(tmp131)
        tmp133 = tmp132

    return tmp133
(base) iancampbell@Ians-MacBook-Pro cs391- %
```



```
print("\n" * 2)
print("# IsPrime function \n")
type_isPrime = term_tpck00(term_fact)
print(f"term_tpck00(term_isprime) = {type_isPrime}")
endl_emit("")
tcmp_pyemit(term_comp00(term_isPrime))
```

```
# IsPrime function

term_tpck00(term_isprime) = STfun(STbas(int);STbas(int))

def fun105(arg0):
    tmp117 = 1
    tmp118 = arg0 <= tmp117

    if tmp118:
        tmp119 = 0
        tmp133 = tmp119
    else:

        def fun106(arg0):
            tmp120 = 1
            tmp121 = arg0 - tmp120
            tmp122 = arg0 <= tmp121

            if tmp122:
                tmp123 = arg0 % arg0

                if tmp123:
                    tmp124 = 1
                    tmp125 = arg0 + tmp124
                    tmp126 = fun106(tmp125)
                    tmp128 = tmp126
                else:
                    tmp127 = 0
                    tmp128 = tmp127

                tmp130 = tmp128
            else:
                tmp129 = 1
                tmp130 = tmp129

            return tmp130

        tmp131 = 2
        tmp132 = fun106(tmp131)
        tmp133 = tmp132

    return tmp133
(base) iancampbell@Ians-MacBook-Pro CS391-2025-Summer %
```

6.3 Fibonacci Sequence

After solving my indentation issue, the Fibonacci Sequence, in turn also compiled and emitted perfectly. I have had no issues with running either of them in my test file, which I have put in my solutions folder. This shows that my compiler can perfectly compile and emit AST that has compatible logic.

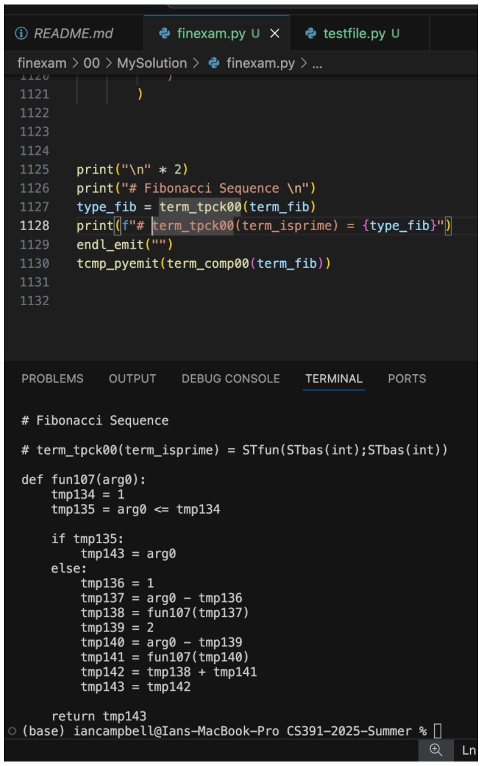
AST Definition:

```
term_fib = \
        term_fix("f","n", styp_int, styp_int,
              term_if0(
                term_lte(var_n, term_int(1)),
                var_n,
                term_add(
                  term_app(var_f, term_sub(var_n, term_int(1))),
                  term_app(var_f, term_sub(var_n, term_int(2)))
                )
            )
        )
```

Type Check: term_tpck00(term_fib) = STfun(STbas(int);STbas(int))

Compiled and Emitted:

```
def fun107(arg0):
    tmp134 = 1
    tmp135 = arg0 <= tmp134

    if tmp135:
        tmp143 = arg0
    else:
        tmp136 = 1
        tmp137 = arg0 - tmp136
        tmp138 = fun107(tmp137)
        tmp139 = 2
        tmp140 = arg0 - tmp139
        tmp141 = fun107(tmp140)
        tmp142 = tmp138 + tmp141
        tmp143 = tmp142

    return tmp143
```



```
            )



print("\n" * 2)
print("# Fibonacci Sequence \n")
type_fib = term_tpck00(term_fib)
print(f"# term_tpck00(term_isprime) = {type_fib}")
endl_emit("")
tcmp_pyemit(term_comp00(term_fib))
```

```
# Fibonacci Sequence

# term_tpck00(term_isprime) = STfun(STbas(int);STbas(int))

def fun107(arg0):
    tmp134 = 1
    tmp135 = arg0 <= tmp134

    if tmp135:
        tmp143 = arg0
    else:
        tmp136 = 1
        tmp137 = arg0 - tmp136
        tmp138 = fun107(tmp137)
        tmp139 = 2
        tmp140 = arg0 - tmp139
        tmp141 = fun107(tmp140)
        tmp142 = tmp138 + tmp141
        tmp143 = tmp142

    return tmp143
(base) iancampbell@Ians-MacBook-Pro CS391-2025-Summer %
```

Conclusion

In summary, this compiler project delivers a clear, end-to-end demonstration of how high-level language constructs are systematically transformed into executable code. We begin with an AST layer whose node definitions align one to one with the surface syntax, proceed through a static type checker that catches errors early, and then translate terms into a concise instruction set that clearly describes logic to make it easier to interpret. Finally, our Python emitter turns that IR back into readable, runnable Python—both serving as a validation of the compiler's correctness and as a lightweight interpreter in its own right. By keeping each phase focused yet interoperable, we not only reinforce foundational compiler-construction principles—like environment handling, register allocation, and recursive code generation—but also leave the door open for future extensions (such as sum types, pattern matching, or optimizations) without sacrificing clarity or pedagogical value. Overall, this modular pipeline strikes a balance between theoretical rigor and practical usability, making it an ideal teaching tool and a solid basis for further experimentation.