Type systems

COMP 105 Assignment

Due Friday, March 15, 2019 **at 5:00 PM**

Contents

Overview

Even if you spend your whole career in the scripting ghetto, at some point you'll be expected to use a language with a *type system*. (Even languages that have never had type systems, like PHP and JavaScript, are getting them now.) You're already aware of type systems in C and C++, and you'll likely encounter them in other languages such as Java, C#, Swift, Go, Rust, and so on. This assignment will help you learn about type systems and polymorphism. You will understand what a type checker does, and you will be able to translate formal type-system rules into code for a type checker.[1] You will add typed primitives to an interpreter. And you will write a couple of explicitly typed, polymorphic functions. When you complete the assignment, you'll be pretty clear about how polymorphism works in languages like Java and Scala, and you might understand why many Go programmers are angry that they don't have it.

Setup

If you have not done so already, clone the book code:

```
git clone homework.cs.tufts.edu:/comp/105/build-prove-compare
```

You will modify two interpreters: `build-prove-compare/bare/tuscheme/tuscheme.sml` and `build-prove-compare/bare/timpcore/timpcore.sml`. You will compile your work with, e.g.,

```
mosmlc -o timpcore -toplevel -I /comp/105/lib timpcore.sml
mosmlc -o tuscheme -toplevel -I /comp/105/lib tuscheme.sml
```

Dire warnings

Your modified `timpcore.sml` and `tuscheme.sml` must compile using `mosmlc` without errors or warnings.

As in the ML homework, you must not use the functions `null`, `hd`, and `tl`. Use pattern matching.

You must not use the ML forms `#1` and `#2`. (They are manifestations of bad design, trying to masquerade as functions.) If you can't use pattern matching, use the `fst` and `snd` functions that are included with the interpreter's source code.

Your `typed-funs.scm` must load into `tuscheme` without warnings or errors, as in

```
tuscheme -q < typed-funs.scm
```

All the homework problems

Reading comprehension (10 percent)

These problems will help guide you through the reading. We recommend that you complete them before starting the other problems below. You can download the questions[2].

[1] You will not necessarily develop a deep understanding for how the rules work—that is, why *these* particular rules are good ones. That question is related to the so-called "type-soundness theorem," which is a *proof* that the type system predicts what happens when we run the code. Unfortunately, proofs of type soundness are relatively deep, and such proofs are beyond the scope of COMP 105.

[2] ./cqs.typesys.txt

1. Read section 6.3.3, which describes how Typed Impcore is extended with arrays. Examine code chunk 417, which shows the cases that have to be added to the type checker.

   For each case, name the type-system rule that applies. Each answer should be a rule name like Apply or GlobalAssign.

   - The rule for case `| ty (AAT (a, i)) = ...` is:
   - The rule for case `| ty (APUT (a, i, e)) = ...` is:
   - The rule for case `| ty (AMAKE (len, init)) = ...` is:
   - The rule for case `| ty (ASIZE a) = ...` is:

   Now pick one of the rules and explain, in informal English, what the rule is supposed to do.

   You are ready for exercise 2 in the pair problems.

2. Read section 6.6.3 on quantified types in Typed μScheme. In addition to the prose, read the transcripts in the first couple pages of that section: each value that has a quantified type is instantiated later in the transcript, giving you more examples to relate back to the prose.

   (a) Assume variable `syms` holds a list of symbols (it has type `(list sym)`). What expression do you write to compute its length? Pick exactly one of the options below.

   1. `(length syms)`
   2. `((o length sym) syms)`
   3. `((@ length sym) syms)`
   4. `((length sym) syms)`

   (b) You are given a function `positive?` of type `(int -> bool)`. Using the predefined function `o`, which has type `(forall ('a 'b 'c) (('b -> 'c) ('a -> 'b) -> ('a -> 'c)))`, what code do you write to compose `positive?` with `not`?

   (c) In testing, we sometimes use a three-argument function `third` that ignores its first two arguments and returns its third argument. Such a function has type

   ```
   (forall ('a 'b 'c) ('a 'b 'c -> 'c))
   ```

   There is only one sensible function that has this type. Using a `val` definition, define function `third` in Typed μScheme. You will need to use both `type-lambda` and `lambda`.

   You are ready for exercise TD.

3. Read about type equivalence starting on page 445 and page 446.

   You are given ML values `tau1` and `tau2`, which represent the respective Typed μScheme types `(forall ('a) 'a)` and `(forall ('b) 'b)`. Semantically, these types are equivalent. For each of the two ML expressions below, say whether the expression produces `true` or produces `false`. Write each answer immediately below the expression.

   (a) `tau1 = tau2`

   (b) `eqType (tau1, tau2)`

   You will soon be ready for exercise 23, but you first need to complete the next two comprehension questions.

4. Read section 6.6.5 on typing rules for expressions in Typed μScheme. For each of the expressions below, say if it is well typed, and if so, **what its type is**. If the expression is not well typed, say what typing rule fails and why.

```
; (a)
(if #t 1 #f)

; (b)
(let ([x 1]
      [y 2])
   (+ x y))

; (c)
(lambda ([x : int]) x)

; (d)
(lambda ([x : 'a]) x)

; (e)
(type-lambda ['a] (lambda ([x : 'a]) x))
```

You are almost ready for exercise 23.

5. Read Lesson 5 ("Program design with typing rules") of *Seven Lessons in Program Design*[3]. In particular, read the explanation of how the If rule is rewritten to add type-equality judgments $\tau \equiv \tau 3$ and $\tau 1 \equiv$ bool. Now look at the list of typing rules for expressions in Figure 6.9 on page 471 in *Programming Languages: Build, Prove, and Compare*. Identify one other rule that needs to be rewritten in the same fashion as If, for the same reason.

   The rule you found is named →

   You are now ready for exercise 23.

6. Exercise A below calls for you to add a primitive array type to Typed μScheme. Read it. Then read "Primitive type constructors of Typed uScheme" in section 6.6.9, which starts on page 455.

   (a) When you add a primitive type constructor for arrays, what chunk of the source code do you intend to add it to? (Give the page number, and if applicable, the letter. For example, page 451 has chunks 451a and 451b, and the letter is the simplest way to distinguish the two.)

   In section M.4, which starts on page 1263, read "Primitives of Typed μScheme."

   (b) Which set of primitive functions most resembles the functions you will need to add for arrays?

   (c) When you add primitives functions that operate on arrays, what chunk of the source code do you intend to add it to? (Just like we asked above, give the page number, and if applicable, the letter.)

   You are ready for Exercise A.

[3] ../design/lessons.pdf

Problems to do by yourself (27 percent)

On your own, please work exercise 8 on page 466 of *Build, Prove, and Compare* and problem **TD** described below.

8. *Adding lists to Typed Impcore*. Do exercise 8 on page 466 of *Build, Prove, and Compare*. The exercise requires you to **design new syntax** and to **write type rules** for lists.

Your typing rules must be *deterministic*. This means that in any given typing environment, any given expression has at most one type, and the type must be computable by a function that is given the abstract syntax and the typing environment as inputs.

Related reading:

- Study the new abstract syntax for arrays in section 6.3.2, which starts on page 414. Be sure you understand that you are seeing new syntactic forms, *not* functions.
- Each new form in code chunk 415a comes with a typing rule, which can be found in section 6.3.3, which starts on page 417. As long as you keep in mind the differences between lists and arrays, this section will help you imagine the sorts of rules you will need to write for lists.
- For another example of new forms and corresponding rules, study the sum-introduction forms left and right in section 6.4 near page 420.
- Finally, for help classifying rules, see the sidebar on "Formation, introduction, and elimination" on page 418.

*Hint: This exercise is more difficult than it first appears. I encourage you to scrutinize the lecture notes for similar cases, and to remember that you have to be able to type check **every** expression* at compile time. I recommend that you **do Pair Exercise 2 first**. It will give you more of a feel for monomorphic type systems.

Here are some things to watch out for:

- It's easy to conflate syntax, types, and values. In this respect, doing theory is significantly harder than doing implementation, because there's no friendly ML compiler to tell you that you have a type clash among `exp`, `tyex`, and `value`.
- It's especially easy to get confused about `cons`. You need to create a **new syntax** for `cons`. This syntax needs to be *different* from the `PAIR` constructor that is what `cons` *evaluates* to.
- Here's a good mental test case: it should be possible to write a recursive `reverse` function, and if `ns` is a list of integers, then `(car (reverse ns))` is an expression that should have type `int`. Even if list `ns` is empty.
- The empty list presents a challenge. Typed Impcore is monomorphic, which implies that any given piece of syntax has at most one type. But you want to allow for empty lists of *different* types. The easy way out is to design your syntax so that you have many different expressions, of different types, that all evaluate to empty lists. The most common mistake is to design a syntax that requires nondeterminism to compute the type of any term involving the empty list. But the problem requires a *deterministic* type system, because deterministic type systems are much easier to implement.

  You might consider whether similar difficulties arise with other kinds of data structures or whether lists are somehow unique.

You might consider what happens in C when you try to do something clever with a pointer of type `void *`, and to think of how C can address this issue using expression syntax only (that is, without resorting to a definition form.)

- You might want to see what happens to an ML program when you try to type-check operations on empty and nonempty lists. For this exercise to be helpful, you have to understand the *phase distinction* between a type error and a run-time error. For example, `hd 3` results in a type error, but `hd []` is well-typed—and results in a run-time error.

TD. *Polymorphic functions in Typed uScheme*. To hold your solution, create a file `typed-funs.scm`. Implement, in Typed μScheme, fully typed versions of these two functions:

- Function `drop` from the Scheme homework, problem B[4], which drops a given *number* of elements from the front of a list
- Function `takewhile`, from exercise 10 on page 212, which takes frontal elements that satisfy a given *predicate*

The problem has four parts:

(a) Write, in a `check-type`, the polymorphic type you expect `drop` to have.

(b) Write a definition of `drop`.

(c) Write, in a `check-type`, the polymorphic type you expect `takewhile` to have.

(d) Write a definition of `takewhile`.

If you are not able to write implementations of `drop` and `takewhile` with the proper types, you may get partial credit by commenting out the `check-type` forms in parts (a) and (c).

Related reading: Read section 6.6.3 on quantified types. Look especially at the definitions of `list2`, `list3`, `length`, and `revapp`. If you are not yet confident, go to section M.5 in Appendix M and study the definitions of `append`, `filter`, and `map`. Appendix M is included in the abridged edition; it starts on page 1257, which you'll find right after Chapter 12.

Problems you can work on with a partner (63 percent)

Please complete exercise 2 on page 463, exercise 23 on page 472, and Exercises **T** and **A** described below. You may work by yourself or with a partner. Most students prefer to work with a partner.

2. *Type-checking arrays in Typed Impcore*. Do exercise 2 on page 463 of *Build, Prove, and Compare*. My solution to this problem is 21 lines of ML.

Related reading:

- Study Lesson 5 ("Program design with typing rules") of *Seven Lessons in Program Design*[5]. Understand the model for implementing each form of judgment as an ML function. Understand the step-by-step procedure for implementing each rule.
- In *Programming Languages: Build, Prove, and Compare*, understand Table 6.1 on page 406. Identify what functions are available to you to call and what functions you will have to add code to.

[4] ./scheme.html#take
[5] ../design/lessons.pdf

- Study section 6.2.1, which starts on page 406. Understand the structure of function `typeof`, which takes three explicit typing environments, and internal function `ty`, which has access to those environments even though it takes only one parameter. Study the cases for `SET`, `IFX`, `EQ`, and `PRINT`. Develop an idea how typing rules and code are related.
- See how the `ARRAYTY` value constructor is defined in chunk 398g. An ML value constructed with `ARRAYTY` represents an array type in Typed Impcore. When you need to recognize an array type, you will pattern match using `ARRAYTY`. When you need to construct an array type, you will apply `ARRAYTY` to another ML value of type `ty`.
- Understand the typing rules in section 6.3.3, which starts on page 417.
- For a broader view of how Typed Impcore is extended with arrays, study section 6.3, which starts on page 414.

23. *Type checking Typed uScheme*. Do exercise 23 on page 472 of *Build, Prove, and Compare*: write a type checker for Typed uScheme. You will submit a modified interpreter and a file containing regression tests. Don't worry about the quality of your error messages, but do remember that your ML code must compile *without errors or warnings*.

Follow the step-by-step instructions listed below under the heading "How to build a type checker," which tells you how to build both the type checker and the regression tests.

Avoid this common mistake: Every possible syntactic form (which is every form except the literal-expression form) should be regression tested with at least one `check-type-error` test. The most common mistake is to forget these tests.

My type checker is about 120 lines of ML. It is very similar to the type checker for Typed Impcore that appears in the book. The code could have been a little shorter, but I put some effort into error messages.

Related reading:

- Revisit Lesson 5 ("Program design with typing rules") of *Seven Lessons in Program Design*[6]. Understand the model for implementing each form of judgment as an ML function. Understand the step-by-step procedure for implementing each rule.
- In *Programming Languages: Build, Prove, and Compare*, understand Table 6.3 on page 441. Identify what functions are available to you to call and what functions you will write. Functions `asType`, `eqType`, and `instantiate` are written for you, and they are frequently overlooked.
- Study section 6.6.5, which starts on page 437—it gives the typing rules for expressions and definitions. You will implement each of these rules.
- Section 6.6.6 contains a long song and dance about type equivalence, starting on page 445. You do not need to understand any of the song and dance—you will get the important aspects later in the term—but you *do* need to understand functions `eqType` and `eqTypes` well enough to know how to use them.
- In section 6.6.7, which starts on page 450, there is even more song and dance, about substitution and instantiation. To implement your type checker successfully, you need to know only how to use functions `freetyvarsGamma` and `instantiate`.

[6] ../design/lessons.pdf

- In section 6.6.9, which starts on page 453, you need to know how to use function `asType`, which is defined on page 455.
- Study the instructions "How to build a type checker" below.

T. *Unit tests for type checkers*. Create a file `type-tests.scm`, and in that file, write three unit tests for Typed μScheme type checkers. Each test must use either `check-type` or `check-type-error`. If you wish, your file may include `val` bindings or `val-rec` bindings of names used in the tests. *Your file must load and pass all tests using the reference implementation of Typed μScheme:*

```
tuscheme -q < type-tests.scm
```

If you submit more than three tests, we will use only the *first* three.

Here is a complete example `type-tests.scm` file, with five tests:

```
(check-type cons (forall ('a) ('a (list 'a) -> (list 'a))))
(check-type (@ car int) ((list int) -> int))
(check-type
   (type-lambda ['a] (lambda ([x : 'a]) x))
   (forall ('a) ('a -> 'a)))
(check-type-error (+ 1 #t))                        ; extra example
(check-type-error (lambda ([x : int]) (cons x x))) ; another extra example
```

You may, if you wish, submit any of these example tests, provided you attribute them properly to me. But your tests will be evaluated on how well they find bugs in the type checkers everyone writes—so new tests are more likely to earn high grades.

Related reading: To be able to write `check-type` and `check-type-error` tests, you need to know the concrete syntax for *unit-test* and *type-exp*, which is shown in Figure 6.2 on page 424. Notice that the `check-type-error` form can accept any true definition, not just an expression.

A. *Polymorphic array primitives for Typed μScheme*. The only way to add arrays to Typed Impcore is to modify the type checker. In Type μScheme, we can do better: A great advantage of a polymorphic type system is that a language can be extended without touching its abstract syntax, its values, its type checker, or its evaluator. You will add arrays to Typed μScheme *without* changing any of these parts.

Extend Typed μScheme with an `array` type constructor and the polymorphic values `Array.make`, `Array.at`, `Array.put`, and `Array.size`. The contracts of the functions are as follows:

- `Array.make` takes as argument a nonnegative integer n and a value v, and it creates an array of n slots, each initialized with v.
- `Array.size` takes an array and returns the number of slots in the array.
- `Array.at` is the array-indexing operation; it takes an array and an index, and it returns the value stored at the given index.
- `Array.put` is the array-update operation; it takes an array, an index, and a value, and it stores the value at the given index.

If n is negative in `Array.make`, it causes a checked run-time error, as does indexing out of bounds (including negative indices). These errors are *run-time* errors, not type errors.

The exercise has three parts:

(a) What is the kind of the type constructor `array`? Add it to the initial Δ for Typed µScheme.

(b) What are the types of `Array.make`, `Array.at`, `Array.put`, and `Array.size`? (These types are polymorphic.)

(c) Edit the interpreter's source code to add the array primitives the initial Γ and ρ of Typed µScheme.

All you are doing is adding new primitives. There is no change to the type checker or to any existing code!

For part (c), you will need to write implementations in ML. To get you part of the way there, here are some helpful functions:

```
(* val arrayMake : int * value -> value *)
fun arrayMake (n, v) = ARRAY (Array.tabulate (n, (fn _ => v)))

(* val arraySize : value array -> value *)
fun arraySize a = NUM (Array.length a)

(* val arrayAt : value array * int -> value *)
fun arrayAt (a, i) =
 Array.sub (a, i) handle Subscript => raise RuntimeError "array subscript out of bounds"

(* val arrayPut : value array * int * value -> unit *)
fun arrayPut (a, i, v) =
 Array.update (a, i, v) handle Subscript => raise RuntimeError "array subscript out of bounds"
```

You will use these ML functions to implement the primitive Typed µScheme functions. Each primitive function has its own type in Typed µScheme, but to implement each Typed µScheme primitive, you must supply an ML function of type `value list -> value`.

To match an ML value into an array, you'll use the `ARRAY` value constructor, which is a secret: it's in the code but not the book. Consider code like this:

```
(fn ARRAY a => ... code using a ...
  | _ => raise BugInTypeChecking "should have had an array here")
```

If you emulate the implementations of existing primitives like `cdr` or the pattern matches in `arithOp`, which is used to implement + and *, and `cdr`, you'll be fine. Look in Appendix M or at the higher-order functions used to implement primitives on page 373.

Parts (a) and (b) ask you to write a kind and a type. The answers will appear in your code, but so we can find them, please also put the answers in your README file. Even if the code isn't perfect, you'll get partial credit for a good kind and good types.

Hint: You will modify two parts of the code that build the initial basis. Both parts are shown under "Building the initial basis" on page 457. Your definitions of `Array.make`, `Array.size`, `Array.at`, and `Array.put` can go next to primitives `null?`, `cons`, `car`, and `cdr`.

My solution to this problem, which takes advantages of the `binaryOp` and `unaryOp` functions already in the interpreter, is 12 lines of ML.

Related reading: Read "Primitive type constructors of Typed µScheme" in section 6.6.9, which starts on page 455 of *Build, Prove, and Compare*. Look at "Primitives of Typed µScheme" in section M.4

Appendix M on page 1257. Focus on polymorphic primitives, such as `null?`, `cons`, `car`, and `cdr` in code chunk 1264a. To help with the implementation of `Array.put` look at the implementation of the `print` primitive, which also returns `unit`.

What to submit and how to submit it

Even if you decide to work without a partner, you'll submit individual work and pair work separately.

Submitting individual work

Using script `submit105-typesys-solo`, please submit

- A `README` file containing
  - A list of the problems you completed
  - The names of the people with whom you collaborated
- File `cqs.typesys.txt`[7] containing your answers to the reading-comprehension questions
- A PDF file `lists.pdf` containing your work on exercise 8.
- A file `typed-funs.scm` containing your code for problem **TD** above

Submit early and often!

Submitting pair work

For your joint work with your partner, *one* of you should use script `submit105-typesys-pair` to submit these files:

- A `README` file containing
  - Your answers to parts (a) and (b) of Exercise A (the kind and the types will also be in your `tuscheme.sml`, but we want to see them in a readable notation)
  - A high-level description of the design and implemenation of your solutions
  - Your name, your partner's name (if you have a partner, and the names of the people with whom you collaborated
- File `timpcore.sml`, containing the Typed Impcore interpreter extended with your type-checking code for arrays (exercise 2)
- File `tuscheme.sml`. containing the Typed μScheme interpreter extended with your type checker (exercise 23) and with array primitives (Exercise A)
- File `regression.scm`, containing the regression tests for your type checker for Typed μScheme, in which each group of tests is identified by a comment saying which step of the testing process the tests belong to
- File `type-tests.scm`, containing up to three tests to be used to test your classmates' type checkers (exercise **T**)

How your work will be evaluated

We will evaluate the functional correctness of your *code* by extensive testing.

[7] ./cqs.typesys.txt

We will evaluate your *regression tests* by looking at *coverage*: a syntactic form is well covered if there is a `check-type` test for the form and if there is also at least one `check-type-error` test for *every* way the form can go wrong.

We will evaluate your *unit-test cases* by using them to look for bugs in other people's code. The more bugs your tests find, the better they are.

We will evaluate the *structure and organization* of your Typed µScheme code using the same criteria as used in previous homework assignments. We will evaluate the structure and organization of your ML code using similar criteria for naming and documentation. For indentation and layout, we'll look for conformance to the Style Guide for Standard ML Programmers,[8] within the constraints imposed by the code from the book.

Techniques and advice

How to run internal `Unit` tests

Many students prefer to include internal `Unit` tests for their type checkers. If you include such tests, here's how you should run them:

1. Compile your interpreter using something like the following:

```
mosmlc -o a.out -toplevel -I /comp/105/lib timpcore.sml
mosmlc -o a.out -toplevel -I /comp/105/lib tuscheme.sml
```

2. Run the unit tests *from the Unix command line* using

```
./a.out -q < /dev/null
```

How to print information when debugging

Within each interpreter, you can use ML functions `print`, `println`, `eprint`, and `eprintln`.[9] Each of these functions expects a single string, and they write to standard output and standard error, respectively. To produce strings, you can use internal functions `expString`, `typeString`, `defString`, and `intString`.

The most common use case is with predefined function `app`. Here's an example:

```
let ...
    val _ =
      app eprint [ "In IF, true branch ", expString e2, " has type "
                 , typeString tau2, " and false branch ", expString e3
                 , " has type ", typeString tau3, "\n"
                 ]
    ...
in  ...
end
```

[8] ../handouts/mlstyle.pdf

[9] Function `print` is predefined; the others are implemented in the interpreter itself.

General advice about type-related code

Here's some generic advice for writing any of the type-checking code, but especially the array primitives you will add to Typed μScheme:

1. Compile early. You could use this command:

   ```
   mosmlc -I /comp/105/lib -o tuscheme tuscheme.sml
   ```

2. Compile insanely often.

3. Compile from within your editor, and use an editor that can jump straight to the location of the first error. With Vim, use `:make`, and with Emacs, use `M-x compile`.

4. Come up with examples in Typed μScheme.

5. Figure out how those examples are *represented* in ML.

6. Keep in mind the distinction between the term language (values of array type, values of function type, values of list type) and the type language (array types, function types, list types).

7. If you're talking about a thing in the term language, you should be able to give its type.

8. If you're talking about a thing in the type language, you should be able to give its kind.

How to build a type checker

Building a type checker is the first COMP 105 exercise of significant scope. You must approach it systematically. *Do not copy and paste the Typed Impcore code into Typed μScheme.* Copying and pasting would be a grave strategic error. You will be *much* better off adding a brand new type checker to the `tuscheme.sml` interpreter, one step at a time.

Writing the whole type checker before running any of it will make you miserable. Make the types in the interpreter work for you, start small, and implement one rule at a time. For each rule, use the techniques explained in Lesson 5 ("Program design with typing rules") of *Seven Lessons in Program Design*[10]. To know what rules to implement in what order, follow these steps:

1. The initial basis contains code for predefined functions that you will not be able to typecheck until your work is complete. Your first step should therefore be to disable those functions. I suggest that you find the line in the source code that corresponds to the binding of value `fundefs` on page 457 of the book:

   ```
   val fundefs   =
   (* predefined {\tuscheme} functions, as strings (generated by a script) *)
   ```

   Replace the line `val fundefs =` with these two lines:

   ```
   val predefined_included = false
   val fundefs = if not predefined_included then [] else
   ```

   Verify that your modified interpreter compiles with `mosmlc`.

2. Start function `typeof`. I recommend defining an internal function `ty`, just as in the type checker for Typed Impcore. Create the first draft of `ty` by writing a clausal definition that has one case for

[10] ../design/lessons.pdf

each syntactic form of Typed μScheme. On the right-hand side of each clause, raise the `LeftAsExercise` exception.

Verify that your modified interpreter compiles with `mosmlc`.

3. Write a function `literal` that computes the type of a literal value. Start with just numbers, Booleans, and symbols—you can add types for list literals later.

   Verify that your modified interpreter compiles with `mosmlc`.

4. Write the case for `typeof/ty` that handles `LITERAL` expressions—it should call `literal`.

5. Create a test file `regression.scm` containing a comment and three unit tests:

   ```
   ;; step 5

   (check-type 3 int)
   (check-type #t bool)
   (check-type 'hello sym)
   ```

   Verify that your modified interpreter compiles with `mosmlc`.

   Verify that your interpreter correctly typechecks the literals used in the tests above. Run

   ```
   ./tuscheme -q < regression.scm
   ```

   You **must** remember the `./` in `./tuscheme`, or otherwise you will be testing my code, not your own code.

   If you are working on a departmental server, you can try the command

   ```
   regression-test-tuscheme
   ```

   As you build your type checker, you will continually add "regression" tests to file `regression.scm`. They are called "regression" tests because they are designed to prevent *regressions*—a regression is a bug introduced into previously working code.

6. Write the case for `typeof` that handles IF-expressions, which I plan to show in class. Add regression tests for a few IF-expressions that have different types. Also add tests for some IF-expressions that are ill-typed.

   - Add the comment `;; step 6` to your `regression.scm` file.
   - Add some `check-type` unit tests for `if` to your `regression.scm` file.
   - Add some `check-type-error` unit tests for `if` to your `regression.scm` file.
   - *Verify that your interpreter compiles and passes all its unit tests.* If something goes wrong with a unit test, make sure the unit test is OK—test it by running `/comp/105/bin/tuscheme -q < regression.scm`.

7. Implement the VAR rule. Add regression tests that check the types of some primitive functions. Be sure to include at least one `check-type-error` test.

   Verify that your interpreter compiles and passes all its regression tests.

8. Now turn your attention to function `elabdef`, which is right next to `typeof`. It takes a true definition, a kind environment, and a typing environment, and it returns a new typing environment and a string.

   - The new typing environment contains a binding for whatever name is defined.
   - The string shows the *type* of whatever name is defined, which you get by applying function `typeString` to the type.

   Write four clauses for `elabdef`, each to raise `LeftAsExercise`. There should be one clause each for `VAL`, `VALREC`, `EXP`, and `DEFINE`.

   Verify that your interpreter compiles and passes all its regression tests.

9. Continuing work with `elabdef`, implement the VAL rule for definitions. Then the EXP rule.

   Add a "step 9" comment and a couple of `val` bindings to your regression-test file, along with `check-type` *and* `check-type-error` tests that use those bindings.

   Verify that your interpreter compiles and passes all its regression tests.

10. Return to `typeof`. Implement the rule for function application. Add regression tests that apply functions. Include both `check-type` and `check-type-error` tests. You should be able to apply some primitive arithmetic and comparison functions.

    Verify that your interpreter compiles and passes all its regression tests.

11. Implement LET binding. The Scheme version is slightly more general than what I plan to cover in class. Be careful with your contexts. Add both `check-type` and `check-type-error` tests.

    Verify that your interpreter compiles and passes all its regression tests.

12. Once you've got LET working, LAMBDA should be quite similar.

    Add suitable regression tests, including both `check-type` and `check-type-error` tests, and verify that your interpreter compiles and passes its regression tests.

13. Knock off SET, WHILE, and BEGIN.

    Add suitable regression tests, including both `check-type` and `check-type-error` tests, and verify that your interpreter compiles and passes its regression tests.

14. There are a couple of different ways to handle LETSTAR. As usual, the simplest way is to treat it as syntactic sugar for nested LETs. Implement type checking for `LETSTAR`.

    Add suitable regression tests, including both `check-type` and `check-type-error` tests, and verify that your interpreter compiles and passes its regression tests.

15. Implement the LETREC rule. Don't overlook the side condition: every right-hand side must be a LAMBDA expression.

    Add suitable regression tests, including both `check-type` and `check-type-error` tests, and verify that your interpreter compiles and passes its regression tests.

16. Go back to `elabdef`, and knock off the definition forms VALREC and DEFINE. (Remember that DEFINE is syntactic sugar for VALREC.) As in LETREC, the right-hand side of VALREC must be a LAMBDA expression.

Add `val-rec` and `define` definitions to your regression-test file, and add regression tests for the names you define. Include both `check-type` and `check-type-error` tests.

Verify that your interpreter compiles and passes all its regression tests.

Your `elabdef` is now complete.

17. Return to `typeof` and implement TYAPPLY and TYLAMBDA. Save these cases for after the last class lecture on the topic. (Those are the *only* parts that have to wait until the last lecture; you can have your entire type checker, except for those two constructs, finished before the last class.)

    Add suitable regression tests, including both `check-type` and `check-type-error` tests, and verify that your interpreter compiles and passes its regression tests.

18. Complete your `literal` function by making sure it handles list literals formed with `PAIR` or `NIL`. The book has *three* rules for list literals; follow them rigorously, and your code will work with no problems.

    Add suitable regression tests, including both `check-type` and `check-type-error` tests, and verify that your interpreter compiles and passes its regression tests.

    Your `typeof` function is now complete.

    Your entire type checker is now complete.

19. Return to the code you modified in step 1. Bind

    ```
    val predefined_included = true
    ```

    Verify that your interpreter compiles and that it can typecheck the predefined functions of Typed μScheme.

Avoid common mistakes

In exercise 8, it's a common mistake to try to create a type system that prevents programmers from applying `car` or `cdr` to the empty list. Don't do this! Such a type system is too complicated for COMP 105. As in ML, taking `car` or `cdr` of the empty list should be a well-typed term that causes an error at run time.

In exercise 8, it's common to write a nondeterministic type system by accident. The rules, typing context, and syntax have to work together to determine the type of every expression. But you're free to choose whatever rules, context, and syntax you want.

In exercise 8, it's inexplicably common to forget to write a typing rule for the construct that tests to see if a list is empty.

There are already interpreters on your PATH with the same name as the interpreters you are working on. So remember to get the version from your current working directory, as in

```
ledit ./timpcore
```

Just plain `timpcore` will get the system version.

In exercise 23, it's a common mistake to write only `check-type` tests, forgetting the `check-type-error` tests. To earn full credit for your regression tests, you must use `check-type-error`.

ML equality is broken! The = sign gives equality of *representation*, which may or may not be what you want. For example, in Typed uScheme, you must use the `eqType` function to see if two types are equal. If you use built-in equality, **you will get wrong answers**.

It's a common mistake to call `ListPair.foldr` and `ListPair.foldl` when what you really meant was `ListPair.foldrEq` or `ListPair.foldlEq`.

It's not a common mistake, but it can be devastating: when you're writing the type of a polymorphic primitive function, write the type variable with an ASCII quote mark, as in `'a`, not with a Unicode right quote mark, as in `’a`. Thanks, Apple!

It's not a common mistake, but don't define any new exceptions. And don't raise any exceptions besides `TypeError`. (If you don't finish, you might also raise `LeftAsExercise`.)

What is and is not hard or time-consuming

In exercise 8 on page 466, I am asking you to create new type rules on your own. Many students find this exercise easy, but many find it very difficult. The "difficult" people have my sympathy; you haven't had much practice *creating* new rules of your own, and as you may remember from class, creating is the highest level of cognitive task on Bloom's hierarchy.

Problem **TD**, writing `drop` and `takewhile` in Typed µScheme, requires that you really understand *instantiation* of polymorphic values. Once you get that, the problem is not difficult, but the type checker is persnickety. A little of this kind of programming goes a long way.

Exercise 2, type-checking arrays in Typed Impcore, has a lot of related reading—you'll fill in any ideas or details that you missed in class. But aside from the amount of reading, this exercise is probably the easiest exercise on the homework. You need to be able to duplicate the kind of reasoning and programming that we will do in class for the language of expressions with LET and VAR.

Exercise 23, the full type checker for Typed µScheme, presents two kinds of difficulty:

- You have to understand the connection between typing judgments, typing rules, and code. Be sure that you follow Lesson 5 ("Program design with typing rules") of *Seven Lessons in Program Design*[11] and the examples that are there.
- You have to understand a moderately sophisticated ML program (the interpreter) and then build a relatively big and independent extension of it.

For the first item, we'll add to Lesson 5 with some talk in class about the concepts and the connection between type theory and type checking. For the second item, it's not so difficult **provided** you remember what you've learned about building big software: don't write it all in one go. Instead, start with a tiny language and grow it very slowly, testing at each step—just as instructed in the guide above. As in yoga, the slow way is the fastest.

Exercise A, adding arrays to Typed µScheme, requires you to understand how primitive type constructors and values are added to the initial basis. And it requires you to write *ML code* that manipulates *µScheme representations*. The task is not inherently difficult, but there are two challenges:

- Because the task is not inherently difficult, it won't get any air time in class. You'll rely on the book.
- Understanding how *ML* code relates to a *µScheme* primitive is not trivial.

[11] ../design/lessons.pdf

To address these challenges, your best bets are to study the way the existing primitives are implemented and to emulate the code that you see.