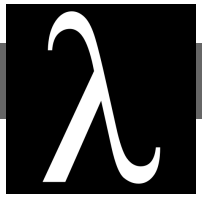


Out: Wednesday, 15 January 2025 at 18:00 ET
Due: Tuesday, 21 January 2025 at 23:59 ET

Total Points: 50

Contents

1 Preamble **3**
1.1 Getting the Assignment . . . 3
1.2 Collaboration . . . 4
1.3 Submission . . . 4
1.3.1 Code Solutions . . . 4
1.3.2 Code Submission Handin Script . . . 5
1.3.3 Code Deductions . . . 5
1.3.4 Written Solutions . . . 5
1.4 Late Policy . . . 6
1.5 Flavor Text . . . 6

2 LaTeX **7**
2.1 Basic Text and Math Mode . . . 7
Task 2.1. (1 point) . . . 8
2.2 15–150 Specific Commands . . . 8
Task 2.2. (1 point) . . . 9
Task 2.3. (1 point) . . . 9
2.3 Aesthetic Formatting . . . 9
Task 2.4. (1 point) . . . 10

3 Course Resources and Policy **11**
Task 3.1. (1 point) . . . 11
Task 3.2. (1 point) . . . 11
Task 3.3. (1 point) . . . 11
Task 3.4. (1 point) . . . 11
Task 3.5. (1 point) . . . 11
Task 3.6. (1 point) . . . 12
Task 3.7. (1 point) . . . 12
Task 3.8. (1 point) . . . 12
Task 3.9. (1 point) . . . 12
Task 3.10. (1 point) . . . 12
Task 3.11. (1 point) . . . 12

4 Types **13**
Task 4.1. (1 point) . . . 13
Task 4.2. (1 point) . . . 13
Task 4.3. (1 point) . . . 13
Task 4.4. (1 point) . . . 13
Task 4.5. (1 point) . . . 13

5 Evaluation **14**
Task 5.1. (3 points) . . . 14
Task 5.2. (3 points) . . . 15

Task 5.3. (2 points) . . . 15

6 Error Messages **16**
6.1 Reading Error Messages . . . 16
6.2 Interpreting Error Messages . . . 16
6.2.1 Evaluating an SML file . . . 16
6.2.2 The End of an Error . . . 17
Task 6.1. (1 point) . . . 17
Task 6.2. (1 point) . . . 17
Task 6.3. (1 point) . . . 17
Task 6.4. (1 point) . . . 17
Task 6.5. (1 point) . . . 17
Task 6.6. (1 point) . . . 17
Task 6.7. (1 point) . . . 18
Task 6.8. (1 point) . . . 18
Task 6.9. (1 point) . . . 18
Task 6.10. (1 point) . . . 18
Task 6.11. (1 point) . . . 18

7 Scope **19**
Task 7.1. (3 points) . . . 19
Task 7.2. (4 points) . . . 19
Task 7.3. (4 points) . . . 19

1 Preamble

Welcome to 15-150! This assignment introduces the course infrastructure and the SML runtime system, then asks some simple questions related to the first few lectures and the first lab.

Constraint: Make sure you have completed the the 15-150 Setup Reference. You will need it in order to complete this homework.

Please read the entire preamble section below. It contains useful instructions, which you are required to understand and follow.

1.1 Getting the Assignment

To download your homework assignment, follow the procedure below:

1. Navigate to the "Assignments" tab of the 15-150 Canvas page.
2. Within the "Homework" folder, you'll see an assignment called "Basics HW," with two files: `basics.pdf` and `basics.tar`.
3. Copy `basics.tar` into your AFS `15150` directory.
4. Navigate into your AFS `15150` directory.
5. Untar the `basics.tar` by typing `tar -xvf basics.tar`

Warning: Do not untar the file locally. This can lead to unexpected behavior.

After following this, you should be left with a directory `basics` with the following structure:

```
basics
├── code
│   ├── errors
│   │   ├── error1.sml
│   │   ├── error2.sml
│   │   ├── error3.sml
│   │   ├── error4.sml
│   │   ├── error5.sml
│   │   ├── smlnj
│   │   └── sources.mlb
│   └── sources.mlb
├── collab.txt
├── lib
│   ├── 150basis
│   │   └── (omitted)
│   ├── intro150basis
│   │   └── (omitted)
│   ├── libmap
│   └── pathconfig
├── Makefile
├── millet.toml
├── README.md
└── written
    └── written.tex
```

The files are used for the following purposes.

- The `code` directory contains sub-directories for each problem that contains coding tasks. (For example, this homework has only the `errors` problem.)
- A `written` directory containing a `written.tex` file. This is a LaTeX template customized for the homework, pre-loaded with task headings.

  We highly recommend you use LaTeX (and this template in particular) when formatting your written submissions!
- An empty `collab.txt` file, to be used as specified in Section 1.2.
- A `Makefile`, which will help you to submit your code. More on that in Section 1.3.
- A `README` and a `lib` directory, which specify and provide any dependencies. These can be disregarded.

All future assignments will be distributed through Canvas in the same way.

1.2 Collaboration

We have a homework collaboration policy.

In keeping with this policy, you should use the `collab.txt` file in the handout to document your collaboration (as specified in the course policy).

If you appear to have collaborated with anyone not listed in your `collab.txt` file, this could be considered an **academic integrity violation**, and may result in disciplinary action.

1.3 Submission

Each homework will consist of code and written submissions.

Certain tasks require you to code in SML, while others require written explanations. Thus, you will submit your homework in two parts on Gradescope:

- A code submission, called `handin.zip`, into the "Basics (code)" assignment
- A written submission, called `writeup.pdf` [1], into the "Basics (written)" assignment

1.3.1 Code Solutions

To submit your code assignment, follow the procedure below:

1. Navigate to the `basics` directory for this assignment.
2. Run `make` in your terminal.
3. Copy `handin.zip` to your local computer.
4. Submit `handin.zip` to the "Basics (code)" assignment on Gradescope.
5. Gradescope will run a "handin script" (further explained below). Review the results and make sure the handin script awarded you full points.

Please get help from the course staff *immediately* if this does not work for you.

[1] This PDF can be called whatever you like

We recommend you **submit early** for time to fix any compilation errors or style violations.

1.3.2 Code Submission Handin Script

When you upload your code to Gradescope, the handin script has two basic purposes:

1. Ensuring your code compiles (e.g. file names are correct and present, has the correct types)
2. Checking for style violations

It is important to note that **the handin script is not a grading script**. Instead, it is "optimistic", meaning that it will report full points as long as your files exist and your code typechecks.

Your grade upon submission may not reflect your true final grade. If your solution is incorrect but has the correct type, it will ***naively*** be given full points.

Please thoroughly test (or prove!) the correctness of your code prior to the deadline. It is expected that you are confident in your code, as the script only provides feedback regarding style.

The handin script will also enforce 15-150 style guidelines[2]. The feedback can be found beneath each problem in the Gradescope "Results" tab.

In the case that your code has poor style, you will receive a deduction of 5% for a given problem. The feedback should explain any style violations you have received. Before the deadline, you may submit as many times as you like to fix your style violations.

After the deadline, style deductions are *permanent*.

1.3.3 Code Deductions

In the future, you may see some certain constraints in tasks.

Constraint: They look like this and will tell you something about the task. This may include not using a certain strategy, meeting a time complexity bound, and more.

If you do not meet these constraints, we reserve the right to give you zero credit for the problem. Note that violating these constraints will not be caught by the handin script.

1.3.4 Written Solutions

Note that we only accept written submissions in typed PDF form. We reserve the right to not grade your submission if you submit handwritten work or other formats besides PDF (e.g. MS Word files, images, etc.).

To submit your written assignment, follow the procedure below:

1. Upload your written assignment PDF to the "Basics (written)" Gradescope assignment.
2. Please put each problem on its own page, and use Gradescope to indicate which page the problem occurs on.

   If your pages are not attached to the correct tasks, we reserve the right to not grade it.

[2] 15-150 Style Guidelines can be found on our course website here

Note that marking pages does not affect your submission time. Thus, if you submitted close to the deadline, you can still mark your pages without pushing your submission time back.

Please contact course staff if you have any questions. If you attempt to contact us close to the deadline, please be aware that we may not be able to respond before the deadline.

1.4 Late Policy

As described in the course policy, you have a total of three late days this semester. Please familiarize yourself with the other specifics of this policy.

1.5 Flavor Text

Occasionally you will see some text in gray boxes like these on your homeworks. This is what we call "flavor text" in this class. We include these to sprinkle in fun little stories about your homeworks; they are purely for the sake of entertainment. Reading these paragraphs are not necessary for completing the homework. To really drive home this point, here's an extremely irrelevant paragraph:

"According to all known laws of aviation, there is no way a bee should be able to fly. Its wings are too small to get its fat little body off the ground. The bee, of course, flies anyway because bees don't care what humans think is impossible. Yellow, black. Yellow, black. Yellow, black. Yellow, black. Ooh, black and yellow! Let's shake it up a little." [a]

[a] The opening lines of the Bee Movie.

2 LaTeX

Since some of you may not be familiar with using LaTeX, this section of the homework will guide you through the bare necessities that you should need for 15–150. We will also go over some special LaTeX commands pre-defined in our homework templates that will be pretty useful throughout the class. Note that if you decide to write your own documents in LaTeX outside of this class, you may need to look up more on how you need to format your LaTeX files.

For additional help on 15–150-relevant LaTeX, check out our 15150 LaTeX helpsheet (located on the website).

2.1 Basic Text and Math Mode

Let's start with the most basic thing that you can do in LaTeX: writing plain text on a page.

To do this, in the LaTeX handout for the homework assignment, type your text verbatim below the task that you are trying to complete. The text will automatically wrap around to the next line on the compiled PDF. If you want to start a new paragraph you should leave a blank line, or you can just add `\\` to the end of the line.

Unfortunately, even when writing plain text in LaTeX, there are already some caveats. Some characters have a special use in LaTeX syntax, like & and $, and require a backslash before them to appear correctly in plain text. If you get stuck on trying to type a certain character or you are getting compilation errors with your LaTeX, don't be afraid to do a quick web search to find out what the issue may be!

Now, for a concrete example: if Task 2 on a homework is asking for your favorite color and food, then you would find where it says `\task{2}` in the written.tex file and write your response below. This would look something like

written.tex

```
\task{2}
My favorite color is blue.

My favorite food is pizza.
```

and in the output `written.pdf`, you should get a page that contains

written.pdf

Task 2.
My favorite color is blue.

My favorite food is pizza.

LaTeX is also very useful for writing mathematical expressions. Whenever you are trying to write a mathematical expression, simply surround it with `\(` and `\)` (the environment inside the `\(...\)` is what we call math mode). You may also see examples online which surround math in dollar signs, e.g., `$2^x$`. This is the TeX primitive, which you can also use.

For example, if you write

written.tex

```
$f(x) = x^2 + x + 150$
```

then in the compiled pdf, you will get

written.pdf

$f(x) = x^2 + x + 150$

You can also write in "display" math mode by writing a mathematical expression inside `\[...\]` which puts the equation on a separate line and centers it.

There are some math symbols like $<$ and $>$ that only appear properly if they are in math mode or some other special environments such as `\code{...}` and `\begin{codeblock}...\end{codeblock}` (we will talk about these commands in the next section).

Task 2.1. (1 point)

Lillian is trying to write up her homework and ends up with the following in her LaTeX file:

```
I really love $150. It is my favorite class!$
```

This is probably not what Lillian was intending to write from looking at the compiled text. Fix this line of LaTeX to make it look normal and display the output in your submission. You do not have to include your code in your response; we only want to see the compiled output. (There should only be 1 main fix.)

2.2 15–150 Specific Commands

There are a handful of commands defined in the `written.tex` template distributed with each homework. Some of the most useful are

- `\code{...}` lets you write SML code with syntax highlighting inline.
- `\begin{codeblock}...\end{codeblock}` lets you write blocks of SML code with syntax highlighting (useful for writing out entire function declarations).
- `\codefile{path\to\file.sml}` imports and inserts an entire SML file, with syntax highlighting and line numbers.
  - `\codefile[linerange=2-5]{path\to\file.sml}` specifies a line range—in this case, lines 2 to 5.
- `\stepsTo` is the stepping symbol used when saying that a piece of code steps to another piece of code. This symbol looks like $\Longrightarrow$.
- `\eeq` is the symbol used for extensional equivalence. This symbol looks like $\cong$.
- `\neeq` is the symbol used for saying two things are not extensionally equivalent. This symbol looks like $\ncong$.

Important: the last three commands should always be used within math mode. See the commented "Sample usage" section at the top of `written.tex` for some examples on how to effectively use these commands.

Task 2.2. (1 point)

Write the content inside the box in LaTeX making use of `\begin{codeblock}...\end{codeblock}` and `\code{...}`. The answer in your submission should look ***exactly*** like what in in the box below.

```
fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact (n - 1)
```

We can see by our type annotations that this function has type `int -> int`!

Task 2.3. (1 point)

Lillian is having another problem! She has the following lines of LaTeX in her homework

```
\task{2.3}
fact : int -> int
```

but when she compiles the LaTeX, it looks a bit strange.

One non-syntax-highlighting issue should really jump out at you...
Fix her issue so that her code ***also*** gives the text proper SML syntax highlighting, and display the output in your submission. You do not have to include your code in your response; we only want to see the compiled output.

2.3 Aesthetic Formatting

Beginning next week, we will dive into the wondrous world of proofs! The most readable way for you to LaTeX your proofs in this class is the `\begin{align*}...\end{align*}` environment. This environment will be very important when stepping through code in your proofs for this class. Here are a few main features about what you will write where "..." is:

- Everything is by default in math mode, but you can still use `\code{...}` like normal in here.
- You will need to use `\\` at the end of your lines to force new lines.
- Use `&` to indicate an alignment point to which each line will be horizontally shifted to line up vertically.
- Use `\tag{...}` to provide justification for the preceding content on a given line. Inside of this you'll be in text mode.

See the commented "Sample usage" section at the top of `written.tex` for an example on how to effectively use these commands.

Task 2.4. (1 point)

Write the content inside the box in LaTeX using `\begin{align*}...\end{align*}`. The answer in your submission should look ***mostly*** like what is in the box below to earn full points. We won't deduct for minor discrepancies. We are just looking for good alignment and proper use of some of the 15–150 template commands from before.

$$
\begin{aligned}
\texttt{fact 2} &\Longrightarrow \texttt{2 * fact 1} && \text{(by clause 2 of } \texttt{fact}\text{)} \\
&\Longrightarrow \texttt{2 * 1 * fact 0} && \text{(by clause 2 of } \texttt{fact}\text{)} \\
&\Longrightarrow \texttt{2 * 1 * 1} && \text{(by clause 1 of } \texttt{fact}\text{)} \\
&\cong \texttt{150 - 148}
\end{aligned}
$$

3 Course Resources and Policy

Please make sure you have access to the various course resources. We will post important information often. You can find more information about these resources in the Resources page of the course's website.

We are using a web-based discussion software called Piazza for the class. You are encouraged to post questions and answers, but please do not post anything that gives away answers or violates the academic integrity policy. If you think that your question might give away answers, you can make it a ***private*** question, visible only to the course staff. **You are responsible for checking Piazza periodically, as we sometimes use it to announce tips and updates to homeworks.**

Task 3.1. (1 point)

You should have received an e-mail message with instructions on signing up for Piazza. Activate your account. There is a pinned post titled "Basics HW Released" with an image. Briefly describe this image.

Task 3.2. (1 point)

Suppose you are registered for the 9:30 AM lab section, but you are really tired that morning. You decide to go to the 2:00 PM lab section instead later that day. Is this allowed?

Task 3.3. (1 point)

What is the terminal command that can be used to copy the file `handin.zip`, located on AFS in `sample@unix.andrew.cmu.edu:private/15150/basics`, to the Desktop on your local machine? This command should be executed from your local terminal.[3]

Task 3.4. (1 point)

Where can you find the LaTeX template for this week's written homework? (You should specify further than just "on Canvas")

Task 3.5. (1 point)

Imagine you have declared a function that meets the following specification:

```
pow2 : int -> int
```
REQUIRES: `n` ≥ 0

ENSURES: `pow2 n` $\cong 2^n$

Provide a sample test case for `pow2` using valid syntax.

Read through the collaboration policy on the course website. For each of the following situations, decide and explain whether or not the students' actions are permitted by the policy.

[3] This isn't the only way to copy files from AFS to your local machine. Visual Studio Code has a way that does not require a terminal command.

Task 3.6. (1 point)

Ting and Aileen are eating lunch together. Ting mentions that she had figured out how to solve a specific problem. Aileen, who hadn't previously thought about the problem, talks with Ting about a possible approach. They then each log off and write up the solution separately.

Task 3.7. (1 point)

Kiera and Alison are friends taking 15–150 together. During lecture, Kiera is confused by one of the examples that is covered. She asks Alison about it after class, so Alison explains it to her.

Task 3.8. (1 point)

Elijah is working late on a tricky question and just can't figure it out. To get a hint, he messages a friend who is also taking the course and goes to bed. The next morning, he reads the friend's hints and works out the solution from there.

Task 3.9. (1 point)

Amy is stuck on a homework problem and is not sure what to do. She begins to read a functional programming textbook from the library for conceptual review. While reading, Amy stumbles upon an example that solves the problem she is stuck on. She decides to find the textbook solution for it and use it in her own code. Afterwards, she cites the textbook in her `collab.txt`.

Task 3.10. (1 point)

Yun and Rong are living in the same house and are both taking 15–150. Rong is working on a problem alone on a whiteboard in her living room. She accidentally forgets to erase her solution and writes it up alone later. Later, Yun, who had forgotten the assignment until the last day, walks by and sees the solution. She reads it, erases it, then writes up her solution.

Task 3.11. (1 point)

Jacky is working on a homework problem. He asks the question to ChatGPT, but notices some errors in its output. Jacky fixes the errors manually before submitting.

4 Types

In order to properly compile, an SML program must only contain well-typed expressions. We can document the types of the expressions that we use in our programs using type annotations, as in `15150 : int`. However, SML performs automatic type checking using various typing rules, regardless of whether these annotations are included.

One such typing rule concerns application expressions. In a function type like `t1 -> t2` (for some types `t1` and `t2`), `t1` is the ***argument type*** and `t2` is the ***result type***. Therefore, an application `e2 e1` is well-typed if the expression `e2` has a function type `t1 -> t2`, and the argument expression `e1` has the correct argument type `t1`. The application then has the corresponding result type `t2`. We can write this typing rule for function application as follows:

> If `e2 : t1 -> t2` and `e1 : t1`, then `(e2 e1) : t2`.

For the following expressions, if they are well-typed, state its type (explanation ***is not*** necessary). If they are not well-typed, put "not well-typed" (NWT) and ***briefly explain*** why this is the case.

Because reasoning about types is an important skill for the course, make sure you have a full understanding of the following tasks and avoid using the SML/NJ REPL. If we have reason to suspect that you are using the SML/NJ REPL, you will not get credit for the question.

Task 4.1. (1 point)

```
5 / 3 + 1
```

Task 4.2. (1 point)

```
(fn x => x + 1)
```

Task 4.3. (1 point)

```
(2 + 4, 8.0)
```

Task 4.4. (1 point)

```
"15" ^ "150"
```

Task 4.5. (1 point)

```
(fn n => if 3 < 0 then false else n * n)
```

5 Evaluation

Now that we have talked about well-typed expressions, we can talk about evaluation. Since type-checking happens before evaluation, only well-typed expressions can be evaluated.

Three things can happen when an expression is trying to get evaluated, it could: raise an exception, loop forever, or ***reduce to a value***. If the expression is already a value, such as an integer numeral or a function (functions are values), it is not evaluated further. Expressions that can reduce to a value (or are already values) are called ***valuable expressions***.

We also want to consider the order in which expressions are evaluated during function application. Consider a well-typed expression like `f e1` where `f` is a function value and `e1` is an expression. First, the argument `e1` is evaluated. If it can be reduced to a value, the value is passed into `f`. Otherwise, the whole expression ends up raising an exception or looping forever.

Let's take a look at an example. In an expression like `e1 ^ e2`, the infix concatenation operator `^` evaluates its two arguments, `e1` and `e2`, from left to right, then returns the string obtained by concatenating the two strings that result from these evaluations.

Using the notation from class, we write $e \Longrightarrow e'$ when `e` reduces to `e'` in a finite number of steps (when an expression "reduces to" a value we may also say "evaluates to"). We can visualize the evaluation of the expression `(Int.toString 7) ^ "1"` as such:

```
   (Int.toString 7) ^ "1"
⟹ "7" ^ "1"
⟹ "71"
```

Now we ask you to perform a similar analysis on another example. Assume that the expression `fact 4` evaluates to `24`, and that the `Int.toString` function has the usual behavior, e.g. `Int.toString 150` evaluates to `"150"`.

Task 5.1. (3 points)

Determine the value that results from the following expression:

```
"7" ^ Int.toString (fact 4)
```

Use the $\Longrightarrow$ notation from class, as above, to express the key evaluation facts in your analysis.

Task 5.2. (3 points)

Caroline has been working on a code trace with the following functions:

```
fun oopsie (x : int, y : int) : int = x + 5
fun whoops (x : int) : int = 15 div x
```

Here is her work:

```
    oopsie(whoops 3, whoops 0)
⟹ (whoops 3) + 5                    (def. of oopsie)
⟹ (15 div 3) + 5                    (def. of whoops)
⟹ 5 + 5                                      (math)
⟹ 10                                         (math)
```

Based on her trace, she believes that the expression is valuable and reduces to `10`. However, when she tested out the original expression into the REPL, the exception `Div` was raised. (Test this out to see it yourself!)

Identify which step Caroline did incorrectly and briefly explain why it was incorrect. Your justification should be more than just restating what the REPL says.

Note: An explanation obtaining full points does not simply explain how to step through the expression yourself. Rather, it points out Caroline's flaw in reasoning and why it's incorrect.

Task 5.3. (2 points)

Annoyed that she has been outsmarted by you, Caroline decides to trace through an expression without any numbers. Instead, she decides to let `x : int` represent any arbitrary integer and tries stepping through some code again. Note that this means that we are assuming that x has the type `int` and is valuable.

Here is her work:

```
    (x div x) * x
⟹ 1 * x                                      (math)
⟹ x                                          (math)
```

Caroline insists that she is correct because this is just basic math. However, you being the expert know that this trace is once again incorrect. Explain why. Your justification should be more than just restating what the REPL says.

Hint: Are these steps accurate for every possible value of `x`?

6 Error Messages

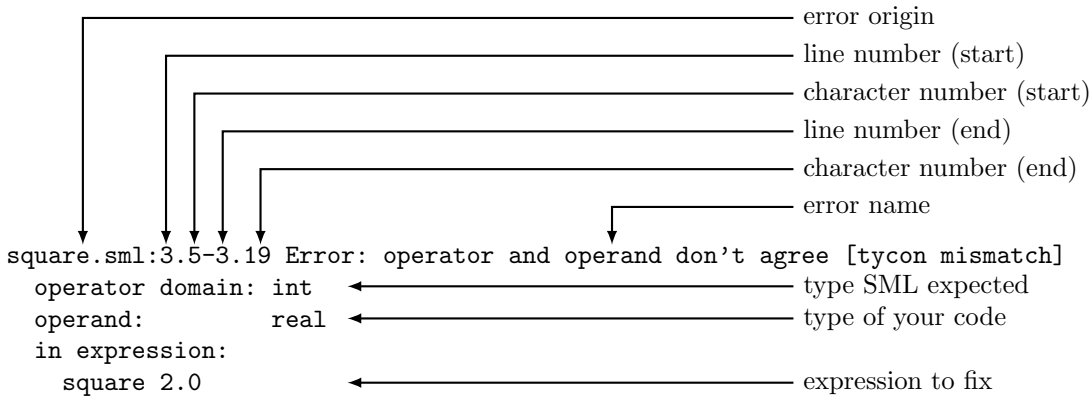
6.1 Reading Error Messages

Suppose we had the following code in a file named `square.sml`:

```
fun square (n : int) : int = n * n

val result = square 2.0
```

If we try to compile it by typing `./smlnj square.sml` into our terminal, SML/NJ gives us an error message! Here's how to read it:



To fix it, we could change the `2.0` to `2`.

6.2 Interpreting Error Messages

For this task, we will be using the files in the `code/errors/` directory.

6.2.1 Evaluating an SML file

Before we begin, `cd` into the `code/errors/` directory.

If you run the command

```
./smlnj sources.mlb
```

all of the files in `code/errors/` will be evaluated. The files will be evaluated in the order that they appear in the following tasks.

You can also run each code file individually. You can either open the SML/NJ REPL[4] and evaluate the desired file *within* the SML/NJ REPL via the command

```
use "filename.sml";
```

or you can specify the file when opening the REPL via

```
./smlnj filename.sml
```

[4] If the `./smlnj` command is not found, consult the 15150 Setup Reference for instructions on setting up your `PATH`.

where `filename.sml` is the name of the SML file you want to run.

6.2.2 The End of an Error

Unfortunately, these files have some errors that must be corrected. The next five tasks will guide you through the process of correcting these errors. **Make sure to submit written responses to Gradescope in order to receive full credit. You may not use any functions from the Real library**

Task 6.1. (1 point)

What error message do you see when you evaluate the unmodified `code/errors/error1.sml` file? What caused this error? How can it be fixed?
Note: You should not need to rewrite the function body for this error![5]

Task 6.2. (1 point)

Correct this *one* error in the file `code/errors/error1.sml` and re-evaluate the file, using the same command as before.

With the first error corrected, the file `code/errors/error1.sml` should now compile without errors. However, you will encounter a new set of errors in the file `code/errors/error2.sml`.

Task 6.3. (1 point)

What is the first error in this set? What caused this error? How do you fix it?[6]

Task 6.4. (1 point)

Correct this error in the file `code/errors/error2.sml` and evaluate the file again.

The file `code/errors/error2.sml` should now compile cleanly, but `code/errors/error3.sml` should have some errors upon evaluation.

Task 6.5. (1 point)

What are the two error messages? They should both reference the same line of SML code. What do these error messages mean? How do you fix them?

Task 6.6. (1 point)

Both errors should disappear with one correction. Fix the errors, and evaluate the file `code/errors/error3.sml` again.

[5] If you are confused, look at how the `fact` function in `code/errors/error2.sml` is written
[6] *Think about types.*

The errors in `code/errors/error3.sml` have been fixed, but there is now yet another set of errors in `code/errors/error4.sml`!

Task 6.7. (1 point)

What is the first error message you see now? What does this error message mean? How do you fix this error?

Task 6.8. (1 point)

Once again, fix the error, then re-evaluate the `code/errors/error4.sml` file.

There should be one more error message, now in `code/errors/error5.sml`.

Task 6.9. (1 point)

What is the error? What caused it? How do you fix this error?

Task 6.10. (1 point)

Correct the error in `code/errors/error5.sml` and evaluate the file once more.

Now, if you run the command `./smlnj sources.mlb`, there should be no more error messages from any of the files in `code/errors/`!

Task 6.11. (1 point)

Submit your code to Gradescope and verify that it compiles cleanly on Gradescope.

Look at the Gradescope output; you should notice that the grader found a style error in the submitted code. Fix the style violation and submit again to Gradescope, checking to confirm that no style points were lost.
Note: No written answer is needed for this task.

7 Scope

Recall from lab that we say that a declaration is *within the scope* of a binding if we can use that binding to make that declaration. Consider the following example:

```
val x : int = 3
val y : int = x + 1
val x : int = 10
val z : int = x + 1
```

We say that `y` is in the scope of the first binding of `x`, but not in the scope of the second binding of `x` (because it was created after `y` was bound), so `y` binds to `4`. We can say that `z` is in the scope of the second binding of `x`; `z` is ***NOT*** in the scope of the first binding of `x` because the second binding ***shadows*** the first, so `z` binds to `11`. For any identifier bound multiple times, new declarations are only in the scope of that identifier's *most recent* binding.

The built-in function

```
real : int -> real
```

returns the `real` value corresponding to a given `int` input; for example, `real 1` evaluates to `1.0`. Conversely, the built-in function

```
trunc : real -> int
```

returns the integral part (intuitively, the digits before the decimal point) of its input; for example, `trunc 3.9` evaluates to `3`. Feel free to try these functions out in the SML/NJ REPL.

Once you understand these functions, you should solve the questions in this section ***without*** first trying them out in the SML/NJ REPL. Reasoning about types is an important skill, and it's helpful to make sure you have a full understanding without relying on the SML/NJ REPL.

Task 7.1. (3 points)

Consider the following code fragment:

```
fun squareit (a : real) : real = a * a
fun squareit (b : real) : int = trunc b * trunc b
fun bopit    (c : real) : real = squareit (c + 1.0)
```

Does this typecheck? Briefly explain why or why not.

Task 7.2. (4 points)

Consider the following code fragment:

```
1 val x : int = 3
2 fun foo (w : int, x : int) : int = 2 + x
3 val y : int = x
4 val x : int = 4
5 val z : int = foo (y, x)
```

What value does `w` get bound to when `foo` gets called on line 5? Why?

Task 7.3. (4 points)

Consider the following code fragment:

```
1 val y : int = 2
2 fun bar (x : int) : int = x + y
3 val y : int = 10
4 val z : int = bar y
```

What value does `z` get bound to on line 4? Why?

This assignment has a total of 50 points.