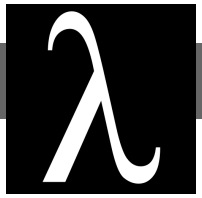


Out: Tuesday, 19 May 2020 at 22:00 ET
Due: Thursday, 21 May 2020 at 22:00 ET

Total Points: 60

Contents

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 first lab.

Constraint: Please read this entire section. It contains useful instructions, which you are required to understand and follow. Thanks!

1.1 Getting the Assignment

To download the homework files, SSH into your afs space (see provided resources for help doing that), and `cd` into the `private` directory. We recommend having a directory inside `private` devoted to 150, e.g. `private/15150`. Navigate into that directory and execute the following command.

```
150m20 homework basics
```

Notice the name of this assignment, `basics`, at the end. For future assignments, we will use the same command to download assignments, but replace the name after homework. The name of a given assignment is shown in the top-right corner of the first page of the assignment.

This should create a directory `basics` with the following structure:

```
basics
├── writeup.pdf
├── code
│   └── errors
│       └── errors.sml
├── collab.txt
├── Makefile
└── written
    └── written.tex
```

The files are used for the following purposes.

- Under the `code` directory, there are sub-directories for each problem containing coding tasks. (For example, this homework has only the `errors` problem.)
- 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 `written` directory with a `written.tex` file underneath it. 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! It will make your solutions easier to format and cleaner overall. More information can be found on the tools page.

1.2 Collaboration

We have a homework collaboration policy.

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

It is very important that you do this. 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

Code submissions will be handled through Gradescope.

1.3.1 Code Solutions

To submit your code asignment, navigate to the `basics` directory for this assignment. You can then run `make` in your terminal. This should produce a file `handin.zip`, which you should scp to your local machine. Then, please go to the Gradescope webpage (linked above), find the assignment **Basics (code)**. Submit your `handin.zip`; Gradescope should run a 'handin script' (described below). Please review the results of the handin script and make sure your work was submitted properly. Please get help from the course staff *immediately* if this does not work for you.

When you upload your code submission to Gradescope, the Gradescope handin script will do some basic checks on your submission: making sure that the file names are correct, making sure that no files are missing, making sure that your code compiles cleanly.

It is extremely important to note that ***the handin script is not a grading script***. Instead, the handin script is "optimistic"; in other words, it assumes that as long as your files exist and the declaration typechecks, you will receive full points. If your code has an obvious error (such as an expected declaration not existing, a file not typechecking, a syntax error, etc.), points will be immediately deducted. However, if your solution is incorrect but has the correct type, it will ***temporarily*** give full points. **The handin script only checks that your code compiles and has the correct type.** Please make sure you test (or prove!) the correctness of your code prior to the deadline. It is expected that you are confident in your code, given that the handin script does not provide nontrivial feedback.

The "style" column has a maximum of `0.0` points, because it is expected that you will have no style deductions. However, it is possible to get up to `-10.0` points, in the case that your code has poor style. Clicking on the point value under the "style" column will explain the style violations. You may submit as many times as you like, which will allow you to fix style issues before the deadline. Note that after the deadline, style deductions are *permanent*. Thus, we highly recommend submitting early and often.

The "written" column score will be populated with your written score from Gradescope after manual grading of written problems is completed.

1.3.2 Written Solutions

Please submit your written solutions to Gradescope **Basics (written)** assignment. Note that we only accept 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.). When you submit, you will be asked to indicate which page of your submission each problem occurs on. 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.

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, there are no late days this semester. Please email the instructors if you are unable to submit due to extenuating circumstances. Please familiarize yourself with the other specifics of this policy.

2 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 Tools page of the course's Web site.

We are using 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 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 there with an image. Briefly describe this image.

Task 2. (1 point)

What is the terminal command that can be used to download the first lab? From what directory should you execute this command?

Task 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.[1]

Task 4. (1 point)

Where can you find the LaTeX template for this week's written homework?

Task 5. (1 point)

Identify one of the code editors suggested in the Great Practical Ideas for 150 guide on the 15-150 website.

Task 6. (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.

[1] 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.

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

Task 7. (1 point)

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

Task 8. (1 point)

Harrison and James are friends taking 150 together. During lecture, James is confused by one of the examples that is covered. He asks Harrison about it after class, so Harrison explains it to him.

Task 9. (1 point)

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

Task 10. (1 point)

Soumil is stuck on a homework problem and is not sure what to do. He begins to read a functional programming textbook he found at the library for help. In the book, there is an example problem that solves the problem he is stuck on. He then reads its solution, and uses that example to construct his answer. Is this permitted by the policy? What if he mentions in `collab.txt` that he consulted the textbook?

Task 11. (1 point)

Tim and Kaz are living in the same house and are both taking 150. Tim is working on a problem alone on a whiteboard in their living room. He accidentally forgets to erase his solution and writes it up alone later. Later, Kaz, who had forgotten the assignment until the last day, walks by and sees the solution. He reads it, erases it, then writes up his solution.

3 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 (abbreviated APP) as follows:

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

For example, suppose `Int.toString` has type `int -> string`. Consider the application expression `Int.toString 7`. We already said that `Int.toString` has type `int -> string`, a function type with argument type `int` and result type `string`. Clearly `7` has type `int`. Since this is the correct argument type, the application `Int.toString 7` has the corresponding result type `string`.

We can formalize this discussion as follows:

1. `Int.toString : int -> string`
2. `7 : int`
3. `(Int.toString 7) : string` by [APP]

Task 12. (3 points)

Determine the type of the expression:

```
(Int.toString 115) ^ (Int.toString 35)
```

Describe your reasoning in the same manner as above, first informally using English, then summarize using the more formal notation. If part of your reasoning exactly corresponds to that found in the example feel free to cite the correspondence rather than copying everything.

Task 13. (2 points)

Explain why the expression `Int.toString 2.0` is not well-typed.

For each of the following expressions, state its type. You do not need to provide additional reasoning. If it is not well-typed, put "not well-typed" (NWT). 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.

Task 14. (1 point)

```
5 / 3 + 1
```

Task 15. (1 point)

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

Task 16. (1 point)

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

Task 17. (1 point)

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

Task 18. (1 point)

```
fun f n = if n < 0 then false else n * n
```

4 Evaluation

A well-typed expression can be evaluated. If its evaluation terminates without raising an exception, the result is 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. 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.

Here is an example: Consider the expression `(Int.toString 7) ^ "1"`. Assume that the application `Int.toString 7` evaluates to the value `"7"`. The expression `"1"` is already a value. So the expression `(Int.toString 7) ^ "1"` evaluates to `"71"`, the string built by concatenating `"7"` and `"1"`.

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 summarize the relevant facts about evaluation in this example as:

```
   (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 19. (2 points)

Determine the value that results from the following expression:

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

Explain your reasoning informally in the same manner as above.

Task 20. (3 points)

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

5 Error Messages

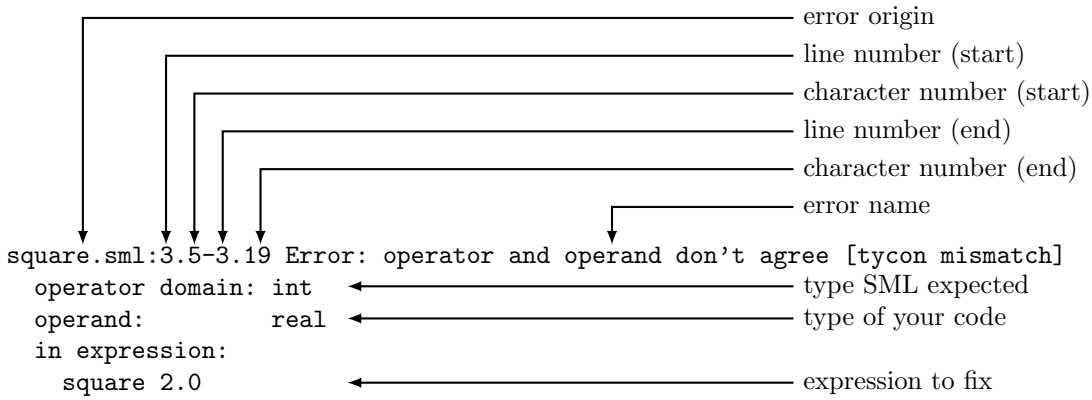
5.1 Reading Error Messages

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`.

5.2 Interpreting Error Messages

For this task, we will be using the `code/errors/errors.sml` file. Before we begin, `cd` into the `code/errors/` directory.

You can evaluate the SML declarations in this file using the command

```
use "errors.sml";
```

within the SML/NJ REPL. Alternatively, you can run the file directly via

```
smlnj errors.sml
```

If the `smlnj` command is not found, consult the Lab 1 handout for instructions on setting up your `PATH`.

Unfortunately, the file has 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.**

Task 21. (2 points)

What error message do you see when you evaluate the unmodified `code/errors/errors.sml` file? What caused this error? How can it be fixed?

Note: You don't need to rewrite the function body!

Correct this one error in the file `code/errors/errors.sml` and evaluate the file again using the same command as before.

Task 22. (2 points)

With the first error corrected, you will encounter a set of errors. What is the first error in this set? What caused this error? How do you fix it?[2] Note that these errors should reference different lines than the first error.

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

Task 23. (2 points)

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

Both errors should disappear with one correction. Fix the errors, and evaluate the file again.

Task 24. (2 points)

There is now yet another set of errors. What is the first error message you see now? What does this error message mean? How do you fix this error?

Once again, fix the error, then re-evaluate the file.

Task 25. (2 points)

There should be one more error message. What is it? What caused it? How do you fix this error?

Task 26. (5 points)

When you correct this final error and evaluate the file there should be no more error messages. Submit your code to Gradescope and verify that it compiles cleanly on Gradescope.

Task 27. (0 points)

Although this task is worth zero points, **you will receive negative points should you fail to complete it**! 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.

[2] *Think about types.*

6 Specs and Functions

Consider the following function:

```
(* decimal : int -> int list *)
fun decimal (n : int) : int list =
    if n < 10 then [n] else (n mod 10) :: decimal (n div 10)
```

A specification for this function has typical form

> `decimal : int -> int list`
>
> REQUIRES: . . .
>
> ENSURES: . . .

The function ***satisfies*** this spec if for all values `n` of type `int` that satisfy the proposition described in the **Requires**, the evaluation of `decimal n` satisfies the proposition described in the **Ensures**.

For each of the following specifications, say whether or not this function satisfies the specification. If not, give an example to illustrate what goes wrong.

Note: A (decimal) digit is an integer value in the range `0 - 9`.

Task 28. (2 points)

> `decimal : int -> int list`
>
> REQUIRES: $n > 0$
>
> ENSURES: `decimal n` evaluates to a non-empty list of digits

Task 29. (2 points)

> `decimal : int -> int list`
>
> REQUIRES: $n \geq 0$
>
> ENSURES: `decimal n` evaluates to a non-empty list of digits

Task 30. (2 points)

> `decimal : int -> int list`
>
> REQUIRES: true
>
> ENSURES: `decimal n` evaluates to a non-empty list of digits

Task 31. (2 points)

```
decimal : int -> int list
```

REQUIRES: $n \geq 0$

ENSURES: `decimal n` evaluates to a list of digits

Task 32. (2 points)

Which ***one*** of these specifications gives the ***most*** useful information about the behavior of the function `decimal`? Say why, briefly.

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 inthe 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 in your head, ***without*** first trying them out in the SML/NJ REPL. Remember, you won't have access to SML/NJ during an exam, so you should be able to reason about code by hand.

Task 33. (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 34. (6 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?

This assignment has a total of 60 points.