15-150 Fall 2013
Homework 01

Out: Wednesday, 28 August 2013
Due: Tuesday, 3 September 2013 at 23:59 EST

1 Introduction

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

1.1 Getting The Assignment

The starter files for the homework assignment have been distributed through our `git` repository. To learn how to use it, read the documentation at

```
http://www.cs.cmu.edu/~15150/resources/git.pdf
```

If you still need help, ask a TA promptly and get started on the non-code questions.

In the first lab, you set up a clone of this repository in your AFS space. To get the files for this homework, log in to one of the UNIX servers via SSH or sit down at a cluster machine, change into your clone of the repository, and run

```
git pull
```

This should add a directory for Homework 1 to your copy of the repository, containing a copy of this PDF and some starter code in subdirectories. If this does not work for you, contact course staff immediately.

1.2 Submission

Submissions will be handled through Autolab, at

```
https://autolab.cs.cmu.edu
```

In preparation for submission, your `hw/01` directory should contain a file named exactly `hw01.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/01` directory (that contains a `code` folder and a file `hw01.pdf`). This should produce a file `hw01.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw01.tar` file via the "Handin your work" link.

The Autolab handin script does 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 PDF is valid; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number (usually either 0.0 or 1.0) corresponding to the "check" section of your latest handin on the "Handin History" page. If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

Your `hw01.sml` file must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

If you cannot access the Autolab site, notify the course staff immediately.

1.3 Due Date

This assignment is due on Tuesday, 3 September 2013 at 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

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, 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.

Task 2.1 (1%). You should have received an e-mail message with instructions on signing up for Piazza. Activate your account. There is announcement there that tells you a 'magic number'. What is the number?

Solution 2.1 3062

Task 2.2 (4%). Read 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.

1. Lars and Gillian write out a solution to Problem 3 on a whiteboard in the Gates-Hillman Center. Then they erase the whiteboard and run to the computer cluster. Sitting at opposite sides of the room, each student types up the solution.

   Solution 2.2 This is not permitted. Lars and Gillian did not allow sufficient time to elapse between discussing the problems and writing up their solutions.

2. Ben and Anshu are discussing Problem 1 over Skype. Meanwhile, Anshu is writing up his solution to that problem.

   Solution 2.2 Anshu's actions are not permitted. Students must wait at least 4 hours between when they discuss problems and when they work on the assignment.

3. Klaas is working late on a tricky question and just can't figure it out. To get a hint, he looks at the staff solution to the problem from when his friend Gail took it last semester.

   Solution 2.2 Klaas is clearly in violation of the policy. Students are never allowed to look at previous semester's solutions.

4. Bill, Ian, and Oguz eat lunch (at noon) while talking about their homework, and by the end of lunch, they have covered their napkins with notes and solutions. They throw out all of the napkins and go to class from 1pm-6pm. Then each individually writes up his solution.

   Solution 2.2 This is fine; the notes were discarded and more than 4 hours passed before anyone wrote up his or her solutions to the problem set.

5. Todd is working on a problem alone on a whiteboard in Gates. He accidentally forgets to erase his solution and goes home to write it up. Later, Sandra walks by, reads it, waits 4 hours, and then writes up her solution. Is Todd in violation of the policy? Is Sandra?

   Solution 2.2 Both are in violation of the policy.

   According to the University Policy on Cheating and Plagarism, you are responsible for preventing other students from accessing your work. This includes erasing whiteboards and setting appropriate file permissions. Thus, Todd is in violation of the policy.

   The only exception to the non-collaborative nature of homeworks is the whiteboard policy, which applies to discussions—which implies that both parties must be involved. Thus, reading someone else's solution when they are not present is in violtion of the policy, so Sandra is in trouble.

3 Types

In this section we will explore the step-by-step reasoning of type checking to better understand when an SML expression is well-typed and, if so, what its type is.

An application expression `e1 e2` has type `t2` if `e1` has type `t1 -> t2` and `e2` has type `t1`. In an arrow type like `t1 -> t2`, `t1` is the *argument type* and `t2` is the *result type*. Therefore, this application is well-typed if the function expression `e1` has an arrow type, and the argument expression `e2` has the correct argument type. The application `e1 e2`, then, has the corresponding result type.

Using the notation from class for type bindings, we write `e: t` to mean that `e` has type `t`. We can summarize the above *typing rule* as follows:

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

For example, suppose `intToString` has type `int -> string`. Consider the application expression `intToString 6`. We already said that `intToString` has type `int -> string`, an arrow type with argument type `int` and result type `string`. Clearly `6` has type `int`. Since this is the correct argument type, the application `intToString 6` has the corresponding result type (`string`).

We can summarize this informal discussion as follows:

i `intToString : int -> string`

ii `6 : int`

iii `(intToString 6) : string` by (APP)

Task 3.1 (3%). The infix operator `^` has type `string * string -> string`. An expression of the form `e1 ^ e2` has type `string` if `e1` has type `string` and `e2` has type `string`.

Determine the type of the expression:

```
(intToString 5) ^ (intToString 1)
```

Describe your reasoning in the same manner as above, first informally using English(!), then summarized 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.

> **Solution 3.1** By the same reasoning as in the example, `intToString 5` and `intToString 1` each have type `string`. Therefore, the whole expression has type `string` since the two subexpressions of `^` each have type `string`.
>
> i `intToString : int -> string`
>
> ii `5 : int`
>
> iii `(intToString 5) : string` by (APP)

iv `1 : int`

v `(intToString 1) : string` by (APP)

vi `^ : string * string -> string`

vii `(intToString 5) ^ (intToString 1) : string` by (APP)

Task 3.2 (2%). Explain why the expression `intToString "2"` is not well-typed.

Solution 3.2 Note that the expression `"2"` has type `string`. We assume that the expression `intToString` has the type `int -> string`. Therefore, the argument in this application does not have the type indicated by the type of the function resulting in a type error.

4 Evaluation

A well-typed expression can be evaluated. If its evaluation terminates, the result is a *value*. If the expression is already a value (such as an integer numeral, or a function), it is not evaluated further. In an expression like `e1 ^ e2`, the infix concatenation operator `^` evaluates its two subexpressions (`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 `(intToString 7) ^ "1"`. Assume that the application `(intToString 7)` evaluates to the value `"7"`. The expression `"1"` is already a value. So the expression `(intToString 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 (also sometimes stated "evaluates to") `e'`. We can summarize the relevant facts about evaluation in this example as:

i `(intToString 7)` $\Longrightarrow$ `"7"`

ii `"1"` $\Longrightarrow$ `"1"`

iii `(intToString 7)^"1"` $\Longrightarrow$ `"7"^"1"`

iv `"7"^"1"` $\Longrightarrow$ `"71"`

Now we ask you to perform a similar analysis on another example. Assume that the expression `fact 5` evaluates to `120`, and that `intToString` function has the usual behavior, e.g. `intToString 42` evaluates to `"42"`.

Task 4.1 (3%). Determine the value that results from the following expression:

```
intToString (fact 5)
```

Explain your reasoning informally in the same manner as above.

Solution 4.1 The expression evaluates to the value `"120"`. The subexpression `fact 5` evaluates to `120` by the assumption. Therefore the original expression evaluates to the expression `intToString 120`, which then evaluates to the value `"120"`.

Task 4.2 (3%). Now use the $\Longrightarrow$ notation from class, as above, to express the key evaluation facts in your analysis.

Solution 4.2

`fact 5` $\Longrightarrow$ `120`
`intToString (fact 5)` $\Longrightarrow$ `intToString 120`
`intToString 120` $\Longrightarrow$ `"120"`

5 Interpreting Error Messages

Download the file `hw01.sml` from the `git` repository as described in Section 1.1.

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

```
use "hw01.sml";
```

at the SML REPL prompt. Unfortunately, the file has some errors that must be corrected. The next five tasks will guide you through the process of correcting these errors.

Task 5.1 (2%). What error message do you see when you evaluate the unmodified `hw01.sml` file? What caused this error? How can it be fixed?[1]

Correct this one error in the `hw01.sml` file and evaluate it again using the same command.

Solution 5.1 The syntax error is:

```
hw01.sml:10.5 Error: syntax error: inserting  VAL
```

This syntax error is caused by a missing vertical bar in the second clause of the declaration of the function `f`. The error can be fixed by inserting the bar.

Task 5.2 (2%). With that error corrected, what is the first of the remaining errors? What caused this error? How do you fix it?
Again, correct just this one error in the `hw01.sml` file and evaluate it.

Solution 5.2 The first of the remaining error messages is:

```
hw01.sml:13.30-13.40 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
  in expression:
    2 * pi
```

This error is caused because the function `*` has type `real * real -> real`, but its argument of `2` is an `int`. This can be fixed by changing `2` to `2.0`.

Task 5.3 (2%). What is the first of the remaining error messages after these two bugs have been corrected? What does this error message mean? How do you fix this error?

Fix this error and evaluate the file again. There's now a new error.

Solution 5.3 The first of the new errors is:

[1] *Hint:* Compare the syntax of the case statement in `f` and the one in `fact`. What is different?

```
hw01.sml:16.5-16.13 Error: can't find function arguments in clause
```

This error is caused by the missing argument in the declaration of the function `semicirc`. It is fixed by adding the argument declaration `(r : real)` after `area`.

Task 5.4 (2%). What error message does the evaluation of the resulting file produce? How do you fix this error?[2]

After you fix this error there should be two type error messages remaining.

Solution 5.4 The remaining error message is:

```
./hw1.sml:16.34-16.37 Error: unbound variable or constructor: pie
```

This error is caused by the use of the unbound variable `pie` in the function `semicirc`. It can be fixed by changing the spelling of that variable to `pi`.

Task 5.5 (2%). What are these error messages? What do these error messages mean? How do you fix this error?[3]

When you correct this final error and evaluate the file there should be no more error messages.

Solution 5.5 The remaining type error messages are:

```
hw01.sml:19.29-19.39 Error: operator and operand don't agree [tycon mismatch]
  operator domain: real * real
  operand:         real * int
  in expression:
    pi * r
hw01.sml:19.5-19.39 Error: right-hand-side of clause doesn't agree with function re
  expression:  int
  result type:  real
  in declaration:
    area = (fn d : int => <exp> * <exp> * d: real)
```

The first error means that we are trying to give `*` a `real` and an `int` argument, instead of two `real`s. The seccond argument is saying that although `area` is declared to evaluate to a `real`, it actually evaluates to an `int`. This can be fixed by changing the type of `area`'s argument `r` from `int` to `real`.

[2] *Hint:* There are two simple ways to fix this error. Please choose the one that keeps the rest of the functions unchanged.

[3] *Hint:* Both error messages are caused by the same error.

6 Specs and Functions

Consider the following function:

```
(* eval : int list -> int *)
fun eval ([] : int list) : int = 0
  | eval (x::xs) = x + 10 * eval(xs)
```

A specification for this function has typical form

```
(* eval : int list -> int
 * REQUIRES:  . . .
 * ENSURES: . . .
 *)
```

The function *satisfies* this spec if for all values `n` of type `int list` that satisfy the assumption from the requires-condition, `eval n` evaluates as described in the ensures-condition.

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. A digit is an integer value in the range `0 - 9`.

Task 6.1 (2%).

```
(* eval: int list -> int
 * REQUIRES: true
 * ENSURES: eval(n) evaluates to a non-negative integer
 *)
```

Solution 6.1 The given function does not match the specification. If given a list containing a negative integer, the function may evaluate to a non-negative integer. For example:
`eval [~1]` $\Longrightarrow^*$ `~1`

Task 6.2 (2%).

```
(* eval: int list -> int
 * REQUIRES: n is a list of digits
 * ENSURES: eval(n) evaluates to a non-negative integer
 *)
```

Solution 6.2 The function satisfies the specifications.

Task 6.3 (2%).

```
(* eval: int list -> int
 * REQUIRES:  n is a list of positive digits
 * ENSURES: eval(n) evaluates to a non-negative integer
 *)
```

Solution 6.3 The function satisfies the specifications.

Task 6.4 (2%).

```
(* eval: int list -> int
 * REQUIRES:  n is a list of digits
 * ENSURES: eval(n) evaluates to a integer
 *)
```

Solution 6.4 The function satisfies the specifications.

Task 6.5 (2%). Which *one* of these specifications gives the *most* information about the applicative behavior of the function `eval`? Say why, briefly.

Solution 6.5 The second spec is the most informative. The `REQUIRES` statement captures the whole domain of the function (the third spec leaves out lists containing `0`), and the `ENSURES` clause conveys the fact that for `n is a list of digits`, `eval n` always evaluates to an integer that is non-negative, information which the fourth spec omits.

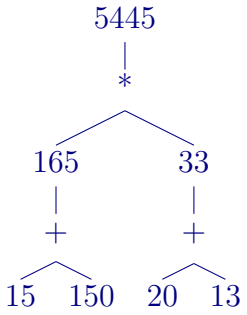
7 Parallel Computing

In lab we showed how to draw a computation tree for an expression, whose structure reflects the order in which its sub-expressions can be evaluated. Each non-leaf node is be labelled with an operator, and its children are sub-trees representing the sub-expressions to be combined by that operator. Leaf nodes are labelled with values, such as integer numerals.

Task 7.1 (2%). Draw the computation tree for the following expression:

```
(15 + 150) * (20 + 13)
```

Solution 7.1



We define the *work* of a computation tree to be the total number of non-leaf nodes (i.e. the number of nodes labelled with operations). The *span* of a computation tree is the number of edges along the longest path from the root to a leaf.

Task 7.2 (2%). What are the work and span for the above computation tree?

Solution 7.2 The work for the tree is 3 and the span is 2.

Suppose we have an expression whose computation tree has work W and span S. No matter how many processors are usable for parallel evaluation, the number of steps required to evaluate the expression must be at least S, because to evaluate (the expression represented by) a node we must first evaluate its children (its immediate sub-expressions), because the value at the node depends on the values of these sub-expressions; this is a *data dependency*. Also note that if each of P processors performs one evaluation step in parallel during each *time cycle*, it would require at least W/P time cycles to perform all of the W operations required to fully evaluate the expression. These observations give the intuition behind *Brent's Theorem*:

Theorem 1 (Brent's Theorem)**.** *If an expression, e, evaluates to a value with work, W, and span, S, then evaluating e on a P-processor machine requires at least* $\max(W/P, S)$ *steps.*

Task 7.3 (2%). Use Brent's Theorem to find a lower bound on the number of steps required to evaluate the computation tree for `(15 + 150) * (20 + 13)` on a machine with $P = 2$ processors.

Solution 7.3 By Brent's Theorem, we have $\max(W/P, S) = \max(3/2, 2) = 2$

Task 7.4 (2%). Describe a possible assignment of the nodes in this computation tree to two processors that achieves this lower bound. In particular, for each time step, say what node each processor is evaluating. If a processor is idle during a time step say so.

Solution 7.4 During time step one, processor one can evaluate one of the two additions and processor two can evaluate the other addition. During time step two, processor one can evaluate the multipliation and processor two can be idle.

Consider the task of baking and frosting n cupcakes in a bakery with multiple ovens and frosting machines. Assume that each cupcake takes the same time (t minutes) to bake and the same time (t minutes) to frost.

Task 7.5 (3%). What is the work for this task? Justify your answer briefly.

Solution 7.5 The work is 2nt, since each cupcake must be baked once and frosted once.

Task 7.6 (3%). If you had an arbitrary number of ovens and frosting machines, how quickly could you finish the task? What is the span of this task? Justify your answer briefly.

Solution 7.6 With an arbitrary number of ovens and frosting machines, it would take 2t minutes to complete the task. Therefore the span of the task is 2, since that is the length of the longest task dependency.