Lab 01

15-150 Fall 2024

Due: Thu 29th Aug, 2024 at 5:00pm

Interview

This assignment will be evaluated in an interview with the course staff on the date above. The points in the tasks are meaningless and submission of solutions is not required.

The instructions for the interview are in the interview guide.

1 Expressions

From here, we can type in expressions for SML/NJ to evaluate. For example, if we want to add $2+2$, we write `2 + 2;`

Task 1.1 (1 points) Enter the text

```
2 + 2;
```

at the SML prompt and press Enter. What is SML's output?

The output line, `val it = 4 : int`, indicates the type and the result of evaluating the expression. "`it`" is used as a default name for the value if a name is not provided by you, `4` is the value or result, and `int` is the type of the expression — in this case, meaning an integer. SML uses types to ensure at compile time that programs cannot go wrong in certain ways; the phrase `4 : int` can be read "4 has type `int`".

Notice that the expression was terminated with a semicolon; if we do not do this, the SML interpreter does not know to evaluate the expression and expects more input.

Task 1.2 (1 points) Enter the text

```
2 + 2
```

(no semicolon) into at the SML prompt and press Enter. What is SML's output? After doing that, enter a semicolon. What happens now?

As you can see, it is possible to put the semicolon on the next line and still get the same result.

1.1 Parentheses

In an arithmetic class long ago, you probably learned some standard rules of operator precedence — for example, multiply before you add, but anything grouped in parentheses gets evaluated first. SML follows the exact same rules of precedence. You can, of course, insert parentheses into expressions to force a particular order of evaluation.

Task 1.3 (1 points) Enter the text

```
1 + 2 * 3 + 4;
```

at the prompt. What would you expect the result to be? What is the actual result?

Now, enter

```
(1 + 2) * (3 + 4);
```

at the prompt. Is the result the same? Why?

2 Types

There are more types than just `int` in SML. For example, there is a type `string` for strings of text.

Task 2.1 (1 points) Enter the text

```
"foo";
```

at the SML prompt. What is the result?

Instead of seeing a number as the output, you see a string here. It is possible to concatenate two strings, using the infix `^` operator. This can be used just like `+` is used on integers.

Task 2.2 (1 points) Enter the text

```
"foo" ^ " bar";
```

at the SML prompt. What is the result?

We can write a program that is not well–typed to see what SML does in that situation.

Task 2.3 (1 points) What happens when you enter the expression

```
3 ^ 7;
```

at the SML prompt?

This is an example of one of SML's error messages — you should start to familiarize yourselves with them, as you will be seeing them quite a lot this semester, at least until you get used to types!

3 Variables

Above, we mentioned that the results of computations are bound to the variable `it` by default. This means that once we have done one computation, we can refer to its result in the next computation:

Task 3.1 (1 points) Enter

```
2 + 2;
```

at the SML prompt. Then, enter

```
it * 2;
```

at the SML prompt. What is the result?

As you see, before the second evaluation the value bound to `it` was 4 (the value of `2+2`), and now `it` is bound to 8, the result of the most recent expression evaluation (the value of `it * 2` with `it` bound to `4`).

Of course, you shouldn't get into the habit of using `it` like this! The SML runtime system only uses it (i.e., `it`) as a convenient default and a way to help you debug code. Usually you will want to choose a more mnemonic name by which to refer to a value, and the SML syntax for *declarations* allows you to do this.

A simple form of declaration has the syntax

```
val <varname> = <exp>
```

This declaration binds the value of `<exp>` to `<varname>`. If we want to mention the desired type of this variable explicitly we can write

```
val <varname> : <type> = <exp>
```

The SML declaration syntax is much more general than we have indicated here, but this gives us enough to work with for now and also introduces the key notions of scope and binding.

Task 3.2 (1 points) Enter the declaration

```
val x : int = 2 + 2;
```

at the SML prompt. What is the result? How does it differ from just typing `2 + 2;`?

As you can see, that declaration binds the value of `2 + 2` to the variable `x`. We can now use the variable.

Task 3.3 (1 points) Enter

```
x;
```

at the SML prompt; what is the result?

Task 3.4 (1 points) Now enter

```
val y : int = x * x;
```

at the SML prompt. What is the result?

Task 3.5 (1 points) How about

```
val y : int list = x * x;
```

What happens? Why?

Task 3.6 (1 points) After that, enter

```
z * z;
```

at the SML prompt. What happens? Why?

Variables in SML refer to values, but are not *assignable* like variables in imperative programming languages. Each time a variable is declared, SML creates a fresh variable and binds it to a value. This binding is available, unchanged, throughout the *scope* of the declaration that introduced it. If the name was already taken, the new definition *shadows* the previous definition: the old definition is still around, but uses of the variable refer to the new definition. We'll talk about this more in lecture and subsequently.

Task 3.7 (1 points) Type the following at the SML prompt:

```
val x : int = 3;
val x : int = 10;
val x : string = "hello, world";
```

What are the value and type of `x` after each line?

4 Functions

Now that we have written some basic SML expressions, we can take a look at something a little more interesting: loading a program from files. We have provided the file `summorial.sml` for you in the handout.

Go to the directory which contains the starter code for this lab. At the SML prompt, type `use "summorial.sml";`. The output from SML should look like

```
- use "summorial.sml";
[opening summorial.sml]
...
val it = () : unit
-
```

Now that you have done this, you have access to everything that was defined in `summorial.sml`, as if you had copied and pasted the contents of the file into SML/NJ.

4.1 Applying functions

Type the line

```
fun intToString (x: int): string = Int.toString x
```

at the prompt (you can also copy and paste it from the file `summorial.sml`). This function can be invoked by writing `intToString(37)`. However, the parentheses around the argument are actually unnecessary. It doesn't matter whether we write `intToString 37` or `(((intToString) (37)))` — both are evaluated exactly the same.

> **Task 4.1** (1 points) Enter
>
> `(intToString 37) ^ " " ^ (intToString 42);`
>
> at the SML prompt. What is the result?

4.2 Defining functions

We will generally require you to use a simple standard format for commenting function definitions: the SML code for a function definition should be preceded by comments that name the function's input/output, a "requires" (pre-)condition, and an "ensures" (post-)condition that describes how the function behaves when applied to arguments that satisfy the pre-condition. We may waive this requirement when the function is part of the SML basis, or has been specified earlier.

For example:

```
(* factorial n ==> r
 * REQUIRES: n >= 0
 * ENSURES:  r is the product of all integers between 1 and n
 *)
fun factorial (0: int): int = 1
  |  factorial n = n * factorial (n-1);
```

We read this specification as saying that:

> *For all values* `n : int` *such that* $n \geq 0$, `factorial(n)` *evaluates to the product of the integers in from 1 to* n.

> **Coding Task 4.2** (4 points) Complete the following template for an SML function `summorial` that calculates the sum of all integers from 0 to n.

```
(* summorial n ==> r
 * REQUIRES: n >= 0
 * ENSURES:  r is the sum of all integers between 0 and n
 *)
fun summorial (0: int): int = raise Fail "Implement me"
  | summorial n = raise Fail "Implement me"
```

Task 4.3 (3 points) Complete the specification for the following function:

```
(* summorial' (n, a) ==> r
 * REQUIRES: n >= 0
 * ENSURES:  FILL ME IN
 *)
fun summorial' (0: int, a: int): int = a
  | summorial' (n, a) = summorial' ((n-1), n+a)
```