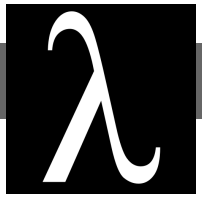

Contents

Welcome to 15-150!

My TAs are:

Their Andrew IDs are:

My lab section letter is:

---

1 Basics of SML

1.1 Expressions

Expressions are the basic unit of an SML program.

Example(s):

1.2 Types

A program *typechecks* if it has no type errors; type errors come from expressions that are *not well-typed*. If we have:

```
e : t
```

We say that expression `e` has type `t`.

Example(s):

There are also *function types*. A function type can be recognized by an arrow in it, such as:

```
e : t1 -> t2
```

An expression with this type has an *argument type* of `t1` and a *return type* of `t2`.

Example(s):

1.3 `val` Declarations

We can bind values to variable names (also called *identifiers*) using the syntax:

```
val <var> : <type> = <expr>
```

In order for this to typecheck, the expression `<expr>` must have type `<type>`.

Example(s):

1.4 `fun` Declarations

Functions in SML can be defined (or *declared*) using the `fun` keyword.

Example(s):

1.5 Scope

Declarations have an attribute called *scope*, which is everywhere it can be used. When a new declaration is made, that declaration only has access to any variable name bindings created *before* it. A declaration can't be made from bindings created *after* it; it's not in the scope of any such bindings.

We use environment binding notation to help us keep track of bindings. Note that this is not actual SML syntax. Annotate the following with environment binding notation:

```
val x : int = 10
val y : int =       x * x
val z : int =       x + y
```

1.6 Shadowing

Binding to variable names in SML is different from assignment in other programming languages. If you bind to the same variable twice, the variable still binds the first value between the first and the second binding; we say that the second binding *shadows* the first binding instead of replacing it everywhere, because prior to the second binding the first binding still exists!

Example(s):

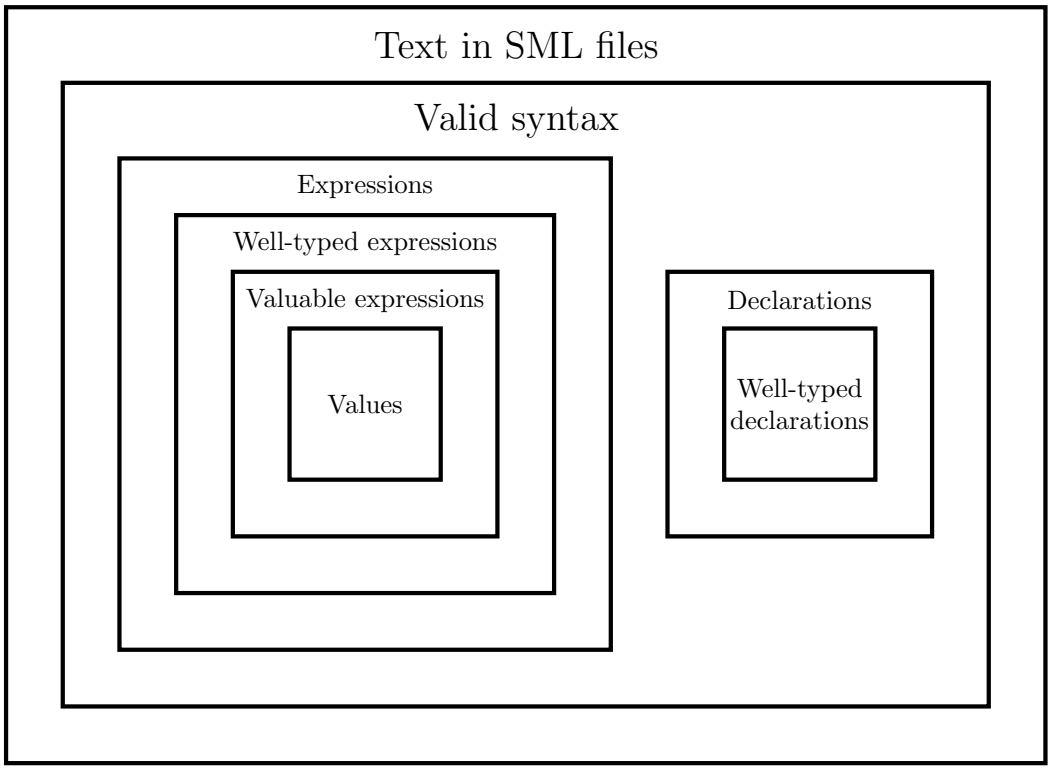

Figure 1: A simple hierarchy of text in SML.

2 Getting Started

Setup time - check out this doc! Once you're done, you should be able to open the REPL using `smlnj` and run

```
- "Hello, " ^ "world!";
val it = "Hello, world!" : string
```

Make sure this works! Hit Ctrl-D to exit the REPL once you're done.

3 Expressions

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

Task 1.

Type

```
2 + 2;
```

into the REPL and press Enter. What is the output?

The output should have been

```
val it = 4 : int
```

This indicates the type and the value of evaluating the expression, as well as the name that this value was bound to. `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 - 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 as "`4` has type `int`".

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

Note that semicolons are *not* required [1] when writing SML code into a file. Furthermore, any code you submit for homework should not use semicolons, or you may lose style points.

Task 2.

Type

```
2 + 2
```

into the REPL and press Enter. What is the 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] With the rare exception of `use` statements.

3.1 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 (PEMDAS!!). You can insert parentheses into expressions to force a particular order of evaluation.

Task 3.

Type

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

into the REPL. What would you expect the result to be? What is the actual result?

Now, type

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

into the REPL. Is the result the same? Why or why not?

4 Types

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

Task 4.

Type

```
"foo";
```

into the REPL. 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. Infix means you insert the function between its arguments, so this is used on strings similar to how `+` is used on integers.

Task 5.

Type

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

into the REPL. What is the result?

We can write a program that does not typecheck (has an expression that's not well-typed) to see what SML does in that situation. For example, concatenation only works given two strings.

Task 6.

What happens when you type

```
3 ^ 7;
```

into the REPL?

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!

5 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 7.

Type

```
2 + 2;
```

into the REPL. Then, type

```
it * 5;
```

into the REPL. 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 `20`, the result of the most recent expression evaluation (the value of `it * 5` 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. The SML syntax for value declaration allows you to *bind* an identifier (or variable) to a value, which lets you refer to it by a more mnemonic name.

A value declaration uses the keyword `val` and has the syntax:

```
val <varname> = <expr>
```

This declaration tries to evaluate the expression `<expr>` to a *value* and then *binds* that value to the *variable* (or identifier) named `<varname>`.[2]

If we want to explicitly state the desired type of the variable, we can give it a *type annotation* as follows:

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

We prefer type annotations as done above, but we can also write:

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

Or even:

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

[2]Sometimes we call this *creating a `val` binding*. Bindings can be combined to form declarations. A binding is an atomic declaration. Each use of `val` is one binding, and a single declaration can consist of many. Two declarations can be combined by simply writing them one after the other, which we could regard as just one declaration.

Declarations have an attribute called *scope*. A declaration's scope is wherever it can be used.[3] If we can declare some "thing" using a declaration, then we say that "thing" is *within the scope* of that declaration.

We could make declarations in the SML/NJ REPL, but we can also make them as part of a `let`-expression using the syntax:

```
let val <varname> = <expr1> in <expr2> end
```

The value of this `let`-expression is obtained by evaluating `<expr1>`, binding its result to the variable named `<varname>`, and using this declaration while evaluating `<expr2>`. This declaration is not available for use outside of `<expr2>`, and we say that the *scope* of that declaration is this expression.

The SML declaration syntax is much more general than we have indicated here, but this introduces the key notions of scope and binding.

Task 8.

Type

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

into the REPL. What is the result? How does it differ from just typing the following?

```
2 + 2;
```

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

Task 9.

Type

```
x;
```

into the REPL. What is the result?

Task 10.

Now type

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

into the REPL. What is the result?

Task 11.

How about

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

What happens? Why?

[3] The concept of scope will be used throughout the semester; if you have any questions, please ask your TAs!

Task 12.

After that, type

```
z * z;
```

into the REPL. What happens? Why?

Variables in SML refer to values, but are not *re-assignable* like variables in imperative programming languages.

Each time a variable is declared, SML creates a new *binding* and binds that variable to a value. This binding is available, unchanged, throughout the *scope* of the declaration that introduced it.

If the name was used before, the new binding *shadows* the previous one: the old binding is still around, but new uses of the variable refer to the most recent one.

Task 13.

Type

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

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

We can introduce multiple identifiers in a single declaration using tuples. When we write a tuple, its elements are comma separated:

```
val (x : int, y : int) = (3,4)
```

This declaration binds `x` to `3` and `y` to `4`.

A tuple of values is itself a value, so we can also do this:

```
val z : int * int = (3,4)
```

This binds `z` to the tuple `(3, 4)`. Note that tuples have what's called a *product type*, indicated by the `*` in it. A tuple will have `*` in its type to separate the types of its elements. For 3-tuples, there would be two `*`'s in its type.

Declarations are useful in other ways! We can use them to write tests for functions. Let's say we wanted to test the built-in `+` and `^` operators. We could write:

```
val 3 : int = 1 + 2;
val 10 : int = 5 + 5;
val "hello, world" = "hello," ^ " world";
```

SML/NJ will not output anything because we didn't create any new bindings; `3` always binds `3`, `"hello, world"` always binds `"hello, world"`, etc.

Values *always* bind themselves, and we will use this fact to write some tests for other functions later.[4]

[4]The only *value* that a value can bind is *itself*.

6 Functions

6.1 Using Files

We have written some basic SML expressions, so now we can take a look at getting input from files. We have provided the file `code/practice/playground.sml` in the tarfile you downloaded.

First, move into the `code/practice/` directory:

```
> ls
code README.md
> cd code/practice/
> ls
playground.sml
```

Then, from the SML/NJ REPL, type `use "playground.sml";`. The output from SML should look like

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

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

If this didn't work, you might not be in the right directory. Exit the REPL by pressing Ctrl+D, then type `pwd` and `ls` to verify you're in the right directory and the file `playground.sml` exists in the working directory.

6.2 Applying Functions

In this file, notice that there are functions defined. For example, there is

```
fun fst (x : int, y : int) : int = x
fun snd (x : int, y : int) : int = y
fun diag (x : int) : int * int = (x,x)
```

Notice the return type of `diag` is `int * int` which means it returns a tuple of ints, for example `(1,1)`. The `diag` function can be invoked by writing `diag(37)`. However, the parentheses around the argument are actually unnecessary. It doesn't matter whether we write `diag 37` or `(((diag) (37)))` – both are evaluated exactly the same.

Good SML style uses only as many parentheses as is ***necessary*** to make your intent clear. For instance, even though

```
2 * 6 + 3 * 5
(2 * 6) + (3 * 5)
```

evaluate to the same thing, we prefer the second form, because it is clearer to see the order of operations.

However, since

```
diag 5
```

is parsed the same way as

```
diag(5)
```

while also using fewer characters, we prefer the first form.

Note that the following two expressions are very different, for reasons we will explore later in the course.

```
fst diag(5)
fst (diag 5)
```

This is yet another reason to prefer minimal use of parentheses during function application.

Task 14.

What is the type of `fst`, and what does it do?

Task 15.

What is the type of `snd`, and what does it do?

Task 16.

What is the type of `diag`, and what does it do?

6.3 Defining Functions

We will generally require you to use a standard format for commenting function definitions: the SML code for a function definition should be preceded by a comment that specifies the function's type, a "requires" clause, and an "ensures" clause. We call this ***a function's spec***. A REQUIRES clause is a logical guarantee about the input (i.e. assumptions you're making about the input), whereas an ENSURES clause tells you what you're guaranteeing about the output. These allow us to formally reason about the behavior of functions, which is a key part of 15-150.

We may waive this requirement when the function is part of the SML Basis Library, or has been specified earlier. When the function has no pre-conditions, we use the notation "`REQUIRES: true`" (i.e. all arguments of the correct type satisfy the pre-condition). You do not ever need to specify in your REQUIRES that the input be of the correct type, nor do you need to specify in the ENSURES what the output type is. This is the point of the type annotation.

For example,

```
(* incr : int -> int
* REQUIRES: true
* ENSURES: incr x ==> the next integer after x
*)
fun incr (x : int) : int = x + 1
```

It's a bit superfluous to write specs for functions that are this straightforward, but this is the general form we'll ask you to use when documenting your code, and it will be much more helpful later on when your code becomes very complex!

Notice that our ENSURES clause is just written in English, not any kind of executable code. As mentioned in lecture, we do not execute our REQUIRES and ENSURES clauses alongside the code (like you may have done in 15-122), but rather these are logical guarantees that help us, the people reading this code, reason about it more easily.

We also ask you to test your code for correctness. Here are some tests for `incr`:

```
val 2 = incr 1
val 5 = incr 4
val ~3 = incr ~4
```

Note: Numeric negation in SML uses ~, a unary operator that has type `int -> int` by default. It's a function.

What's happening here is we're telling SML to *bind* the value `2` to whatever `incr 1` evaluates to. If `incr 1` isn't `2`, then SML would complain that we're trying to bind `2` to something that's not `2`. However, if `incr 1` *is* `2` (like we want), then we're just saying "Hey SML, 2 binds to 2", which it has no objections to.

6.4 Now it's your turn!

For each of the following functions in `code/practice/playground.sml`:

1. write the function's spec (write the function's type, the REQUIRES, and the ENSURES)
2. implement the function
3. write some tests for the function (make sure each function passes its tests!)

Do all of this in the `code/practice/playground.sml` file; we've included some starter code for your convenience.

Task 17.

A function named `add3` which takes in a triple `(x,y,z)` of three integers and returns their sum $x + y + z$.

Task 18.

A function named `flip` which takes a tuple `(x,s)` where x is an integer and s is a string, and it returns the tuple `(s,x)`.

Task 19.

A function named `diff` which takes a tuple `(x,y)` of integers where $x < y$ and returns their difference $y - x$.

Task 20.

A function named `isZero` which takes an integer `x` and returns `true` if `x` is zero, otherwise `false`.

Note: We use `=` in SML to create bindings, but within an expression it's also an operator. There are two different ways to do this. See if you can figure out both!

Task 21.

A function named `detectZeros` which takes a tuple `(x,y)` and evaluates to `true` if either `x` is zero or `y` is zero.

Hint: You might find the built-in infix operator `orelse` helpful. Here's some examples for how `orelse` is used:

```
val true = true orelse true
val true = true orelse false
val true = false orelse true
val false = false orelse false
```