Yesterday’s lecture was all about the big picture, exploring lots of new ideas and getting confused. Today’s lecture is going to be all about the details. I outlined the four things that I want to talk about in this class yesterday; now we’re going to start establishing a common language that we can use to talk about those things.

The main ideas of today’s lecture are:

1. expressions versus values
2. type checking and evaluation
3. errors
4. declarations
5. functions

1 Expressions, Types, Values

The basic unit of an ML program is an expression. Here are some simple expressions:

15
15 - 150
(7 + 8) * (1000 + 10)
"15-"
"15-" ^ "150"
intToString 15150
"15" + 150
1515 div 0

Note that parens are used for grouping. Operator precedence and grouping works (mostly) like in regular math: \((1 + 2) * (3 + 4)\) is different than \(1 + 2 * 3 + 4\), which by convention is the same as \(1 + (2 * 3) + 4\).

*Based on drafts by Brandon Bohrer and others
1.1 Computing by calculation

The way we run an ML program is to calculate it down to a value, which is the result of a computation. Simple values include numeric constants (written like 15150) and string constants (written like "15-150").

The operations on int and real and string are all primitives that “do the expected thing” in one step of calculation. For example, we’re assuming that something called intToString transforms an integer into a string.

We write $e_1 \mapsto e_2$ to mean that “$e_1$ steps to $e_2$ in exactly one step of evaluation”.\(^1\) For example,

\[
(1 + 2) \ast (3 + 4) \\
\mapsto 3 \ast (3 + 4) \quad \text{(because 1 + 2 \mapsto 3)} \\
\mapsto 3 \ast 7 \quad \text{(because 3 + 4 \mapsto 7)} \\
\mapsto 21
\]

We write $e_1 \Longrightarrow e_2$ to mean “the expression $e_1$ reduces (in one or more steps) to the value $e_2$.\(^2\) If $e_2$ is a value, we might say $e_1$ evaluates to it. This is mostly how we will use the notation in lecture, though you will see it used for partial evaluations sometimes\(^3\). For example,

\[
\begin{align*}
15 & \Longrightarrow 15 \\
15 - 150 & \Longrightarrow ^\sim 135 \\
(7 + 8) \ast (1000 + 10) & \Longrightarrow 15150 \\
"15-" & \Longrightarrow "15-" \\
"15-" \ast "150" & \Longrightarrow "15-150" \\
\text{intToString 15150} & \Longrightarrow "15150"
\end{align*}
\]

The value of an expression is determined by calculation. Values are expressions that have no calculation left to do; they are done. For each operation, we calculate the values of its subexpressions, and then apply the operation to those results.

The last two expressions above, "15" + 150 and 1515 div 0, don’t have values. We’ll talk more about them later.

Note that it would be wrong to say

\[
(1 + 2) \ast (3 + 4) \\
\mapsto 3 \ast 7 \\
\mapsto 21
\]

because there are two steps that are happening at the same time.

1.2 Types

In the list of syntactically correct expressions above, there were two that we weren’t really sure what to do with. We can guess what the other ones are supposed to do, but it’s not at all clear what "15" + 150 means or is supposed to mean or what it might step to next.

\(^1\)Note that $\mapsto$ is not part of the syntax of ML; it is mathematical notation that we use to talk about programs. In ASCII, we will sometimes write $\mid \rightarrow$.

\(^2\)Again, $\Longrightarrow$ is not ML syntax. We will write $\Longrightarrow$ in an ASCII context.

\(^3\)There’s some disagreement about whether this is acceptable use of the notation.
The world of ML expressions is divided up into types. The type of an expression is a prediction about the value it will yield, should it yield a value at all. For example, if an expression has type \texttt{int}, then its value will be a numeral (if it has a value at all).

An expression is \textit{well-typed} if it has at least one type; otherwise it is \textit{ill-typed}.

The \textit{type checker} determines whether or not an expression is well-typed, and rejects ill-typed programs at \textit{compile-time} (when you’re writing the program). This helps you catch mistakes at compile-time, which is better than finding them at \textit{run-time} (when someone runs the program). E.g. it’s much better if you find the bug at compile-time, than if the program’s user finds the bug after you ship it to them. Remember what we said yesterday about software running in cars and airplanes...

Expressions that are not well-typed are generally expressions that make no sense. Some languages will let you do something like "15" + 150, and they will do, well, something. Because it’s not generally clear what is meant by such a thing, what happens is very language specific—and often implementation or version specific as well. That makes it very hard to reason about programs and figure out what they mean. We introduce types as a way to get out of this mess.

In particular, we write \( e : t \) to mean that “the expression \( e \) has type \( t \)”.\footnote{When can an expression have more than one type? This is called \textit{polymorphism}. We’ll talk about it more next week.}

For example:

\[
\begin{align*}
15 : & \text{ int} \\
15 - 150 : & \text{ int} \\
(7 + 8) \times (1000 + 10) : & \text{ int} \\
"15-" : & \text{ string} \\
"15-" \ast "150" : & \text{ string} \\
\text{intToString 15150} : & \text{ string} \\
"15" + 150 & \text{ is ill-typed} \\
1515 \text{ div 0} & : \text{ int}
\end{align*}
\]

In general, a type is specified by a collection of \textit{values}, which are the possible results of an expression of that type, as well as a collection of \textit{operations}, which are how you use things of that type.

For example:

- The type \texttt{int} has
  - Values: 0, 27, ~82, ...
  - Operations: \texttt{+,*,-, intToString}, ...

  Note that negatives are written like 
  \texttt{"-3}, while subtraction is written as \texttt{-}.

- The type \texttt{string} has
  - Values: "I am", "the walrus", ...
  - Operations: \texttt{\^, size}, ...

- The type \texttt{real} (read: float) has

\footnote{Unlike the arrows above, this colon is actually SML syntax, which we’ll see a little bit later on.}
Values: $0.0, 3.14, 2.17, \ldots$
Operations: $+, *, -, /, \ldots$ (note that these reuse the same names as for int; they are disambiguated based on context; if the context provides insufficient info, they default to int).

- The type bool has
  - Values: true and false
  - Operations: if <exp> then <exp> else <exp>

1.3 Type checking

So what’s missing from this description of a type? Well, we told you what the values and operations are, but not what the expressions are. This is a little odd, in that we just spent a lot of time talking about expressions. Type checking is the process of going from knowing what the values of types are, and what the operations on those types are, to figuring out what the type of a large expression is.

The type of an expression is determined compositionally by looking (only) at the types of the expressions inside it (“subexpressions”):

- each of the values listed above has the type indicated. E.g.
  
  42 : int
  
  "The answer" : string

  Because both expressions are values, these are axioms, which are unconditionally true.

- each of the operations is well-typed if its subexpressions have the “right” types. For example:
  
  - $e_1 + e_2 : int$ if $e_1 : int$ and $e_2 : int$
  
  - $e_1 \land e_2 : string$ if $e_1 : string$ and $e_2 : string$
  
  - intToString $e : string$ if $e : int$

These rules can be used to derive the type of a compound expression:

$(3 + 7) \times 5 : int$ because

  3 + 7 : int because
  
  3 : int is an axiom
  7 : int is an axiom
  
  the addition of integers is an integer
  
  5 : int is an axiom
  
  the multiplication of integers is an integer

Some expressions have no type:

"15" + 150 would have type int if

"15" has type int (but it doesn’t!)

150 has type int (check)

Exercise. Derive a type for "The answer is " ^ (intToString 42).
1.4 Exceptions

Allowing ourselves to think about the types of expressions took care of one of the two edge case expressions we had at the beginning.

What about 1515 div 0? Here, div means integer division. This expression is well-typed, but can't have a value, because division by 0 is undefined. So, it signals an error at run-time, when you're running the program. This is called raising an exception:

\[
5 \div 0 \\
\rightarrow \text{raise Div}
\]

The difference between evaluating to a value and raising an exception is that exceptions propagate up to the top of your program. So if some expression somewhere in your program produces an error, that error will be the final result of the computation. If a well-typed subexpression failed to produce a value of the predicted type, then its parent expression can only do the same. For example:

\[
(5 \div 0) + 1 \\
\rightarrow (\text{raise Div}) + 1 \\
\rightarrow \text{raise Div}
\]

Later in the semester, we will talk about recovering from exceptions; for now, you should only raise them in cases where you want that to be the final result of your program.

Raising an exception is considered different from returning a value. An expression is valuable iff there exists some value that it evaluates to:

\[
e \text{ is valuable iff } \exists \text{ a value } v \text{ such that } e \Rightarrow v.
\]

So a valuable expression doesn't raise an exception. For example, 150 div 15 is valuable, but 1515 div 0 is not.

Note that, in general, this is the sort of property that requires a proof. You can't check for this statically; we need to do some reasoning at a higher level than the program source code.

You might wonder why 1515 div 0 isn't a type error: why wait until run-time to signal a problem? The reason is that, to check whether 1515 div e is permissible, you'd need to know whether e evaluates to 0 or not. In general, you can't write an algorithm to decide this, because of something called the halting problem.

There are programming languages with fancy type systems where you can rule out some such programs at compile-time. But they're really complicated, so that's why we're programming in SML instead.

If you think about it, it's a bit unfair to blame the type checker for not catching our divide-by-zero error, since as I just said, this is actually impossible. What other options do we have?

This is a perfect example of something we can put in the spec (specification) of a function. We know m div n doesn't make any sense when n = 0, because division by zero is undefined. We should express values that don't make sense as part of our specification. For example, this would be a perfectly good spec for div
It’s worth noting that there’s a trade-off between types and specs. The advantage of types is that they’re checked at compile-time by the type-checker, so SML will prevent us from making a mistake. Specs aren’t checked at all, so the responsibility is on the programmer to make sure they always fulfill the spec.

While it’s obviously better to have the type-checker catch our mistakes for us, some properties such as \( n \neq 0 \) are difficult or even impossible to express through types. Since we can put anything we want in a spec, those properties are best written as specs.

This tradeoff is one that every programming language makes in its design. Dynamically-typed languages like Python and Ruby are one extreme of the tradeoff, the fancy languages I alluded to are another extreme, and SML is somewhere in the middle.

In Homework 1, you will work with specs. In particular we can have more than one spec for the same function, and some are better than others. A strong spec is a spec that REQUIRES as little as possible and ENSURES as much as possible, because that means we can use it lots of situations and still get to know a lot about the output. Your job will be find the strongest possible spec for a function.

### 1.5 Classes of Expressions

We can summarize this by identifying several classes of expressions, each of which is a strict superset of the next. For each, we give an example expressions that is in that class, but not the next one down.

- Every expression we could possibly write down, including ones that are total nonsense.
- Syntactically correct expressions. These expressions make some basic level of sense; all of the above expressions are syntactically correct.
- Well-typed expressions. These expressions pass the type checker.
- Valuable expressions. These expressions compute to a value.
- Values. These expressions already are values.

For example:

- \((1+2)\) is not even syntactically correct, so if you say
  - \((1+2)\;

  you will get a syntax error at compile-time.

- "Ishmael"+1 is syntactically correct, but not well-typed, so you will get a type error at compile-time.

- \(5 \div 0\) is well-typed, but not valuable.

- \(5 \div 1\) is valuable, because it evaluates to 5. But it is not already a value.

- 5 is a value.
2 Declarations

The top level of an ML program is a sequence of *declarations*.

For now, we will consider three kinds of declarations:

2.1 Val Bindings

The first is a *val* binding, such as

```
val x : int = 2 + 3
```

This means that the *variable* x stands for the *value* of the expression \(2 + 3\) (which must have type *int*) in the subsequent program. The general form of a val binding is `val <var> : <type> = <exp>`. These declarations are used to name intermediate steps in a program.

**Typing** The declaration is well-typed iff \(<exp> : <type>\). In the scope of the declaration (“below”, modulo the caveat mentioned soon), we can write the variable \(<var>\) and it will be an expression with type \(<type>\). E.g. in the scope of the above val binding, \(x\) can be used as an expression with type *int*.

**Evaluation** To evaluate a sequence of val declarations, you evaluate the first expression, and then *substitute* its value in for the variable in the subsequent declarations (replace occurrences of the variable with the value).

For example, consider

```
val x : int = 2 + 3
val y : int = x + 1
val z : int = x + y
```

We first calculate \((2 + 3) \Rightarrow 5\), and then proceed as if the program were

```
val x : int = 5
val y : int = 5 + 1
val z : int = 5 + y
```

You could also write an intermediate step showing the substitution explicitly, writing it as \([5/x]\):

```
val x : int = 5
val y : int = [5/x]x + 1
val z : int = [5/x]x + y
```

Here we have substituted the one occurrence of \(x\) in the expression \(x + 1\) with the value 5, to get the expression \(5 + 1\), and the occurrence in \(x + y\) to get \(5 + y\). Next, we calculate \(5 + 1 \Rightarrow 6\) and proceed with the program

```
val x : int = 5
val y : int = 6
val z : int = 5 + 6
```
which in one more step evaluates to

\[
\begin{align*}
\text{val } x &: \text{ int } = 5 \\
\text{val } y &: \text{ int } = 6 \\
\text{val } z &: \text{ int } = 11
\end{align*}
\]

To summarize, the value of a sequence of declarations is a sequence of declarations of values.

**Shadowing**  What happens when you have two different `val` bindings with the same variable?

\[
\begin{align*}
\text{val } x &: \text{ int } = 5 \\
\text{val } x &: \text{ int } = 3
\end{align*}
\]

The right way to think about this is that these are *two different variables that happened to be spelled with the same ASCII string*. The second is *not* an assignment that updates \(x\), like in an imperative language. Variables are like variables in math: placeholders that can be plugged in for. This difference will become sharper later, when we can declare variables inside of functions; at that point, we can construct an example to illustrate it.

So, when you evaluate

\[
\begin{align*}
\text{val } x &: \text{ int } = 5 \\
\text{val } x &: \text{ int } = 3
\end{align*}
\]

you do *not* substitute 5 for the \(x\) on the second line. The \(x\) on the second line is an entirely independent variable, not an occurrence of the first \(x\).

Similarly, if you have

\[
\begin{align*}
\text{val } x &: \text{ int } = 5 \\
\text{val } y &: \text{ int } = x + 1 \\
\text{val } x &: \text{ int } = 3 \\
\text{val } z &: \text{ int } = x + 1
\end{align*}
\]

then the \(x\) in the second line is an occurrence of the variable bound in the first line, where the \(x\) in the fourth line is an occurrence of the variable bound in the third line. This is because, by convention, a variable refers to the *nearest enclosing declaration*. So the value of this sequence of declarations is

\[
\begin{align*}
\text{val } x &: \text{ int } = 5 \\
\text{val } y &: \text{ int } = 6 \\
\text{val } x &: \text{ int } = 3 \\
\text{val } z &: \text{ int } = 4
\end{align*}
\]

We can make this apparent by *consistently renaming* the second \(x\) to \(x’\) in the declaration and at all occurrences.

\[
\begin{align*}
\text{val } x &: \text{ int } = 5 \\
\text{val } y &: \text{ int } = x + 1 \\
\text{val } x’ &: \text{ int } = 3 \\
\text{val } z &: \text{ int } = x’ + 1
\end{align*}
\]

This program has the same value as before, modulo the fact that the second \(x\) in the result is now called \(x’\).
2.2 Type Definitions

The second kind of declaration is a type definition. For example, last lecture, we saw

\texttt{type row = int sequence}

This declaration means that the type variable \texttt{row} stands for the type \texttt{int sequence} in the subsequent program. The general form of a type declaration is \texttt{type <tyvar> = <type>}. In the scope of a type definition, \texttt{<type>} can be used as a type.

The scoping rules (shadowing) for type variables are the same as for value variables.

3 Functions

Thus far, we have some basic values and operations, and the ability to name intermediate computations. However, a \texttt{val} binding introduces a variable that stands for the value of one specific expression. Thus far, we have no way to capture \textit{repeated patterns of computation}. That is where \textit{functions} come in.

To a first approximation, functions in ML are like functions in math. For example,

\[ f(x) = 2x + 6 \]

can be rendered as

\begin{verbatim}
fun f(x : real) : real = (2.0 * x) + 6.0
\end{verbatim}

This is a \texttt{fun} declaration, a third kind of declaration which introduces a function named \texttt{f}, with argument \texttt{x} of type \texttt{real}, and result, also of type \texttt{real}, given by the body expression \((2.0 \times x) + 6.0\).

The main operation on functions is \textit{function application}. For example, we can write \texttt{f 3.0} to apply the function \texttt{f} to the argument \texttt{3.0}. A function application calculates by substitution: you plug in the value of the argument for the variable. In this case

\begin{verbatim}
|-> (2.0 * 3.0) + 6.0
|-> 6.0 + 6.0
|-> 12.0
\end{verbatim}

\textbf{Call-by-value} It’s important to know that you \textit{calculate a function’s argument down to a value before plugging in}. This is called \textit{call-by-value evaluation}. E.g.

\begin{verbatim}
|-> (2.0 * (1.0 + 2.0)) + 6.0
\end{verbatim}

Another choice would be to plug in the whole expression:

\begin{verbatim}
|-> (2.0 * (1.0 + 2.0)) + 6.0
\end{verbatim}
but this is **not** what ML does. How can you tell? Consider $f \ (5 \ \text{div} \ 0)$ where $f$ doesn’t use its argument.

Call-by-value makes it easy to to predict when an expressions is evaluated. This is helpful for time analysis and for reasoning about non-valuable expressions (like an expression that raises an exception).

**Scoping** Function bodies can refer to variables that are in scope, including other functions. Example:

```ml
fun g(x : real) : real = f (f x)
```

Function arguments take precedence over bindings further out:

```ml
val x : real = 4.0
fun g(x : real) : real = f (f x)
```

Here $x$ still refers to the function argument, not to the `val` binding above it.

Function arguments are **not** in scope below the function declaration:

```ml
fun g(a : real) : real = f (f a)
val y : real = a
```

Here $a$ is unbound. This is the only thing that makes sense: the $a$ doesn’t stand for any particular value at this point, but for any number of possible values that we may later choose to apply $f$ to.

Functions can also call themselves recursively:

```ml
fun h(x : int) : int = (h (x - 1)) + 1
```

We’ll talk more about this a bit later, and even more tomorrow.

**Functions are values**

Of course, we can have functions on types other than `real`:

```ml
fun repeatThreeTimes(s : string) : string = s ^ s ^ s
```

**Exercise.** Calculate the value of `repeatThreeTimes "hi"`.

Moreover, functions are not some special class of things, but regular old values of a type, just like everything else in ML. The type of $f$ is written `real -> real` and the type of `repeatThreeTimes` is written `string -> string`. In general, we have

- The type `<type1> -> <type2>` has
  - Values: functions introduced by `fun` bindings
  - Operations: function application, written `$f \ a$`. Here $f \ a$ has type `<type2>` if $a$ has type `<type1>`.

This means that functions can be passed as arguments to other functions, and returned from functions as results. We won’t exploit this much for a few weeks, but we’ve already seen a couple of examples: `map` and `reduce` in the previous lecture.

If we update our hierarchy of syntactic things, we can add something like `$f$` to the class of values, `$f \ 3.0$` to the class of valuable expressions, and `$f \ 5.0/0.0$` to the class of well-typed but not valuable expressions.
Cases in functions  Above, we wrote the function

\[
\text{fun } h(\text{x : int}) : \text{int} = (h(\text{x - 1})) + 1
\]

This would seem like not a very useful function! What happens if we call

\[h 5\]

We might only intend for \( h \) to be called on numbers greater than 0, which we could express in a spec:

\[\text{(* REQUIRES: x > 0 *)}\]
\[
\text{fun } h(\text{x : int}) : \text{int} = (h(\text{x - 1})) + 1
\]

but we might violate our own spec when calling it recursively! Instead, we might wish to do something different when \( h \) is called with zero. We can do this by giving several cases for the definition of \( h \) depending on the input.

\[
\text{fun } h(0 : \text{int}) : \text{int} = 0
\]
\[
| h(\text{x : int}) : \text{int} = (h(\text{x - 1})) + 1
\]

If \( h \) is called with 0, it will use the first case and return 0. Otherwise, evaluation will fall through to the second case and bind the input (which we now know is not 0, though it could be negative) to \( x \).