Thanks to Dan Grossman, Josiah Adams, and Cody A. Schroeder for the substantial majority of this content.
Hi, I’m Konstantin
Hi, I’m Konstantin

fun \textbf{min}(x, y) =
  \textbf{if } x \leq y \textbf{ then } x \textbf{ else } y.
Hi, I’m Konstantin

Function \( \text{min} \ x \ y \ := \ if \ x \ \leq \ y \ then \ x \ else \ y \).
Hi, I’m Konstantin

Function \( \text{min} \ x \ y := \begin{cases} x & \text{if } x \leq y \text{ then } x \text{ else } y. \end{cases} \)

Goal \( \forall x \ y, \ \text{min} \ x \ y \leq x \land \text{min} \ x \ y \leq y. \)
Hi, I’m Konstantin

Function \texttt{min} \ x \ y :=
\begin{align*}
& \text{if } x \leq y \text{ then } x \text{ else } y. \\
\end{align*}

Goal $\forall x \ y, \ \texttt{min} \ x \ y \leq x \land \texttt{min} \ x \ y \leq y.$

Proof.
\begin{align*}
\text{intros; unfold min.} \\
\text{destruct } (x \leq y) \text{ eqn:e.} \\
- \text{rewrite } \texttt{Nat.leb_le} \text{ in e.} \\
\quad \text{omega.} \\
- \text{rewrite } \texttt{leb_iff_conv} \text{ in e.} \\
\quad \text{omega.} \\
\text{Qed.}
\end{align*}
Hi, I’m Konstantin
Hi, I’m Konstantin
Today’s Agenda

• ML Development Workflow
  – Emacs
  – Using use
  – The REPL (Read–Eval–Print Loop)

• More ML
  – Shadowing Variables
  – Debugging Tips
  – Boolean Operations
  – Comparison Operations
Emacs Demo

- Recommended (not required) editor for this course

- Powerful, but the learning curve can at first be intimidating
Using \textit{use}

\begin{verbatim}
use "foo.sml";
\end{verbatim}

- Enters bindings from the file \texttt{foo.sml}
  - Like typing the variable bindings one at a time in sequential order into the REPL (more on this in a moment)

- Result is () bound to variable \texttt{it}
  - Ignorable
The REPL

• Read-Eval-Print-Loop is well named

• Conveniently run programs
  – Useful to quickly try something out
  – Save code for reuse by moving it into a persistent .sml file

• Expects semicolons

• For reasons discussed later, it’s dangerous to reuse use without restarting the REPL session
Debugging Errors

Your mistake could be:

- **Syntax**: What you wrote means nothing or not the construct you intended

- **Type-checking**: What you wrote does not type-check

- **Evaluation**: It runs but produces wrong answer, or an exception, or an infinite loop

Keep these straight when debugging even if sometimes one kind of mistake appears to be another
Play around

Best way to learn something: Try lots of things and don’t be afraid of errors

Work on developing resilience to mistakes
   – Slow down
   – Don’t panic
   – Read what you wrote very carefully

*Maybe watching me make a few mistakes will help*…
Shadowing of Variable Bindings

\[
\begin{align*}
\text{val } a &= 1; \hspace{1em} (* a \rightarrow 1 *) \\
\text{val } b &= a; \hspace{1em} (* a \rightarrow 1, b \rightarrow 1 *) \\
\text{val } a &= 2; \hspace{1em} (* a \rightarrow 2, b \rightarrow 1 *)
\end{align*}
\]

1. Expressions in variable bindings are evaluated “eagerly”
   – Before the variable binding “finishes”
   – Afterwards, the expression producing the value is irrelevant

1. Multiple variable bindings to the same variable name, or “shadowing”, is allowed
   – When looking up a variable, ML uses the latest binding by that name in the current environment

2. Remember, there is no way to “assign to” a variable in ML
   – Can only \textit{shadow} it in a later environment
   – After binding, a variable’s value is an immutable constant
Try to Avoid Shadowing

val x = "Hello World";
val x = 2; (* is this a type error? *)
val res = x * 2; (* is this 4 or a type error? *)

• Shadowing can be confusing and is often poor style

• Why? Reintroducing variable bindings in the same REPL session may..
  – make it seem like wrong code is correct; or
  – make it seem like correct code is wrong.
Using a Shadowed Variable

- Is it ever possible to use a shadowed variable? Yes! And no…
- It can be possible to uncover a shadowed variable when the latest binding goes out of scope

```plaintext
val x = "Hello World";
fun add1(x : int) = x + 1; (* shadow x in func body *)
val y = add1 2;
val z = x"!!"; (* "Hello World!!" *)
```
Use **use** Wisely

- **Warning:** Variable shadowing makes it dangerous to call **use** more than once without *restarting* the REPL session.

- It *may* be fine to repeatedly call **use** in the same REPL session, but unless you know what you’re doing, *be safe!*
  - Ex: loading multiple distinct files (with independent variable bindings) at the beginning of a session
  - **use**’s behavior is well-defined, but even expert programmers can get confused

- Restart your REPL session before repeated calls to **use**
Boolean Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Type-checking</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>andalso</td>
<td>e1 andalso e2</td>
<td>e1 and e2 must have type bool</td>
<td>Same as Java’s e1 &amp;&amp; e2</td>
</tr>
<tr>
<td>orelse</td>
<td>e1 orelse e2</td>
<td>e1 and e2 must have type bool</td>
<td>Same as Java’s e1</td>
</tr>
<tr>
<td>not</td>
<td>not e1</td>
<td>e1 must have type bool</td>
<td>Same as Java’s !e1</td>
</tr>
</tbody>
</table>

- **not** is just a pre-defined function, but **andalso** and **orelse** must be built-in operations since they cannot be implemented as a function in ML.
  - Why? Because **andalso** and **orelse** “short-circuit” their evaluation and may not evaluate both **e1** and **e2**.

- Be careful to always use **andalso** instead of **and**.
- **and** is completely different. We will get back to it later.
Style with Booleans

Language does not need \texttt{andalso}, \texttt{orelse}, or \texttt{not}

\footnotesize
\begin{itemize}
\item \begin{verbatim}
(* el andalso e2 *)
if el
then e2
else false
\end{verbatim}
\item \begin{verbatim}
(* el orelse e2 *)
if el
then true
else e2
\end{verbatim}
\item \begin{verbatim}
(* not el *)
if el
then false
else true
\end{verbatim}
\end{itemize}

Using more concise forms generally much better style

And definitely please do not do this:

\footnotesize
\begin{verbatim}
(* just say e (!!!) *)
if e
then true
else false
\end{verbatim}
Comparisons

For comparing `int` values:

\[
= \ < > \ > \ < \ \geq \ \leq
\]

You might see weird error messages because comparators can be used with some other types too:

- `> < \geq \leq` can be used with `real`, but not `1 int` and `1 real`
- `= <>` can be used with any “equality type” but not with `real`
  - Let’s not discuss equality types yet