Begin at the beginning ...

Expressions (Syntax) ➝ Values (Semantics)
Compile-time “Static” ➝ Types

1. Programmer enters expression
2. ML checks if expression is “well-typed”
   • Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expression
3. ML evaluates expression to compute value
   • Of the same “type” found in step 2

Style exercise
• Pattern matching style exercise

Base Types

Base Type: int

<table>
<thead>
<tr>
<th>Expression</th>
<th>2</th>
<th>2</th>
<th>( _ )</th>
<th>( _ ) : \text{int}</th>
<th>( _ = \text{int} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 3 ( _ )</td>
<td>5</td>
<td>( a_1 + a_2 )</td>
<td>( a_1 : \text{int}, a_2 : \text{int} )</td>
<td>( a_1 = \text{int}, a_2 = \text{int} )</td>
<td>( a_1 + a_2 )</td>
</tr>
<tr>
<td>7 - 4 ( _ )</td>
<td>3</td>
<td>( a_1 - a_2 )</td>
<td>( a_1 : \text{int}, a_2 : \text{int} )</td>
<td>( a_1 = \text{int}, a_2 = \text{int} )</td>
<td>( a_1 - a_2 )</td>
</tr>
<tr>
<td>( (2+3) \times (7-4) )</td>
<td>15</td>
<td>( a_1 \times a_2 )</td>
<td>( a_1 : \text{int}, a_2 : \text{int} )</td>
<td>( a_1 = \text{int}, a_2 = \text{int} )</td>
<td>( a_1 \times a_2 )</td>
</tr>
</tbody>
</table>

Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions

Base Type: float

<table>
<thead>
<tr>
<th>Expression</th>
<th>2.0</th>
<th>2.0</th>
<th>( _ )</th>
<th>( _ ) : \text{float}</th>
<th>( _ = \text{float} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0 + 3.0 ( _ )</td>
<td>5.0</td>
<td>( a_1 + a_2 )</td>
<td>( a_1 : \text{float}, a_2 : \text{float} )</td>
<td>( a_1 = \text{float}, a_2 = \text{float} )</td>
<td>( a_1 + a_2 )</td>
</tr>
<tr>
<td>7.0 - 4.0 ( _ )</td>
<td>3.0</td>
<td>( a_1 - a_2 )</td>
<td>( a_1 : \text{float}, a_2 : \text{float} )</td>
<td>( a_1 = \text{float}, a_2 = \text{float} )</td>
<td>( a_1 - a_2 )</td>
</tr>
<tr>
<td>( (2.0 + 3.0) / (7.0 - 4.0) )</td>
<td>1.66</td>
<td>( a_1 / a_2 )</td>
<td>( a_1 : \text{float}, a_2 : \text{float} )</td>
<td>( a_1 = \text{float}, a_2 = \text{float} )</td>
<td>( a_1 / a_2 )</td>
</tr>
</tbody>
</table>

Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions
### Base Type: string

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ab&quot;</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>&quot;ab&quot;</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>string</td>
<td>s → s</td>
</tr>
</tbody>
</table>

Expressions built from sub-expressions

Types computed from types of sub-expressions

Values computed from values of sub-expressions

### Base Type: bool

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>b</td>
<td>b → b</td>
</tr>
</tbody>
</table>

2 < 3: true

not (2 < 3): false

("ab" = "cd"):
false

b: bool

b: b → b

0 * (2 + "a");

### Complex types: Tuples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2+2, 7&gt;8);</td>
<td></td>
</tr>
<tr>
<td>(4,false)</td>
<td>int * bool</td>
</tr>
</tbody>
</table>

Expressions built from sub-expressions

Types computed from types of sub-expression

Elements can have different types

Tuples can be nested in other tuples
Complex types: Records

{name="sarah" ;
  age=31;
  pass=false}

Records are tuples with named elements...

{name="sarah";age=31;pass=false}.age

int 31

{name=31;name="sarah";pass=false}.age

int 31

{name=31;name="sarah";pass=false}.pass

false bool

Complex types: Lists

• Unbounded size
• Can have lists of anything (e.g. lists of lists)

\[
\text{["a";"b";"cd"]};
\]
\[
\text{["a";"b";"cd"]}
\]
\[
\text{int list}
\]

Complex types: list ..construct

Cons “operator”

\[
1::\text{[2;3]}
\]
\[
\text{int list}
\]

Can only “cons” element to a list of same type

Complex types: list ...deconstruct

Reading the elements of a list:
• Two “operators”: \( \text{hd} \) (head) and \( \text{tl} \) (tail)
Recap

1.  Programmer enters expression
2.  ML checks if expression is “well-typed”
   - Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expr
3.  ML evaluates expression to compute value
   - Of the same “type” found in step 2

Expressions (Syntax)  Values (Semantics)

Expressions (Syntax)  Values (Semantics)

•  Integers: +,-,*
•  floats: +,-,*
•  Booleans: =,<, andalso, orelse, not
•  Strings: ^

•  Tuples, Records: #i
  - Fixed number of values, of different types
•  Lists: ::,@,hd,tl,null
  - Unbounded number of values, of same type