Next: Let’s get cosy with Recursion

Recursive Code Mirrors Recursive Data

Representing Trees

```
type tree =
  | Leaf of int
  | Node of tree * tree

let rec sum t =
```

“Sum up the leaf values”. E.g.

```
# let t0 = sum (Node(Node(Leaf 1, Leaf 2), Leaf 3));;
- : int = 6
```

```
let rec sum_leaf t =
```

```
let rec sum_leaf t =
```

```
let rec sum_leaf t =
```

```
let rec sum_leaf t =
```
Recursive Code Mirrors Recursive Data

Code almost writes itself!

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- $4.0 + 2.9 \Rightarrow 6.9$
- $3.78 - 5.92 \Rightarrow -2.14$
- $(4.0 + 2.9) \times (3.78 - 5.92) \Rightarrow -14.766$

What's a ML TYPE for REPRESENTING expressions?

```ml
type expr =
| Num of float
| Add of expr*expr
| Sub of expr*expr
| Mul of expr*expr
```

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- $4.0 + 2.9 \Rightarrow 6.9$
- $3.78 - 5.92 \Rightarrow -2.14$
- $(4.0 + 2.9) \times (3.78 - 5.92) \Rightarrow -14.766$

What's a ML FUNCTION for EVALUATING expressions?

```ml
type expr =
| Num of float
| Add of expr*expr
| Sub of expr*expr
| Mul of expr*expr
```
Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:
- 4.0 + 2.9 ===> 6.9
- 3.78 - 5.92 ===> -2.14
- (4.0 + 2.9) * (3.78 - 5.92) ===> -14.766

What's a ML FUNCTION for EVALUATING expressions?

```ml
let rec eval e = match e with
| Num f -> f
| Add (e1, e2) -> eval e1 + eval e2
| Sub (e1, e2) -> eval e1 - eval e2
| Mul (e1, e2) -> eval e1 * eval e2
```

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:
- 4.0 + 2.9 ===> 6.9
- 3.78 - 5.92 ===> -2.14
- (4.0 + 2.9) * (3.78 - 5.92) ===> -14.766

What's a ML FUNCTION for EVALUATING expressions?

```ml
let rec eval e = match e with
| Num f -> f
| Add (e1, e2) -> eval e1 + eval e2
| Sub (e1, e2) -> eval e1 - eval e2
| Mul (e1, e2) -> eval e1 * eval e2
```

Begin at the beginning ...

Expressions (Syntax) \(\xrightarrow{\text{Types}}\) Values (Semantics)

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

Base Types

```
Base Type: int
```

Expressions built from sub-expressions

Types computed from types of sub-expressions

Values computed from values of sub-expressions
### Base Type: int

<table>
<thead>
<tr>
<th>2</th>
<th>2</th>
<th>1</th>
<th>i: int</th>
<th>i = j</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 3;</td>
<td>5</td>
<td>a + e</td>
<td>e1: int</td>
<td>e2: int</td>
</tr>
<tr>
<td>2 - 3;</td>
<td>3</td>
<td>a - e</td>
<td>e1: int</td>
<td>e2: int</td>
</tr>
<tr>
<td>(2 + 3) * (7 - 4);</td>
<td>15</td>
<td>a * e</td>
<td>e1: int</td>
<td>e2: int</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>2.0</th>
<th>2.0</th>
<th>e</th>
<th>r: float</th>
<th>r = r</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0 + 3.0</td>
<td>5.0</td>
<td>a + e</td>
<td>e1: float</td>
<td>e2: float</td>
</tr>
<tr>
<td>1.0 - 4.0</td>
<td>3.0</td>
<td>a - e</td>
<td>e1: float</td>
<td>e2: float</td>
</tr>
<tr>
<td>(2.0 + 3.0) / (7.0 - 4.0)</td>
<td>1.66</td>
<td>a / e</td>
<td>e1: float</td>
<td>e2: float</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>“ab”</th>
<th>“ab”</th>
<th>s</th>
<th>s: string</th>
<th>s = s</th>
</tr>
</thead>
<tbody>
<tr>
<td>“ab”</td>
<td>“cd”</td>
<td>a + e</td>
<td>e1: string</td>
<td>e2: string</td>
</tr>
<tr>
<td>“abcd”</td>
<td>“abcd”</td>
<td>a + e</td>
<td>e1: string</td>
<td>e2: string</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>true</th>
<th>true</th>
<th>b</th>
<th>b: bool</th>
<th>b = b</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 &lt; 3</td>
<td>true</td>
<td>a &lt; b</td>
<td>e1: int</td>
<td>e2: int</td>
</tr>
<tr>
<td>not (2 &lt; 3)</td>
<td>true</td>
<td>a &lt; b</td>
<td>e1: int</td>
<td>e2: bool</td>
</tr>
<tr>
<td>“ab” = “cd”</td>
<td>false</td>
<td>a = b</td>
<td>e1: string</td>
<td>e2: string</td>
</tr>
<tr>
<td>“ab” = “cd”</td>
<td>false</td>
<td>a = b</td>
<td>e1: bool</td>
<td>e2: bool</td>
</tr>
</tbody>
</table>

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

### Type Errors

- Expressions built from sub-expressions
- Types computed from types of sub-expression
- If a sub-expression is not well-typed then whole expression is not well-typed

0 * (2 + “a”);
Complex types: Tuples

- Can be of any fixed size
  - (9,3,"ab","cd",7>8) (6, "abcd", false)
  - (int * string * bool)

- Elements can have different types
- Tuples can be nested in other tuples

But wait...

- All evaluation rules look like:

Complex types: Records

Records are tuples with named elements...

<table>
<thead>
<tr>
<th>name: string</th>
<th>age: int</th>
<th>pass: bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>name=&quot;sarah&quot;</td>
<td>age=31</td>
<td>pass=false</td>
</tr>
<tr>
<td>age=31</td>
<td>name=&quot;sarah&quot;</td>
<td>pass=false</td>
</tr>
<tr>
<td>age=31</td>
<td>name=&quot;sarah&quot;</td>
<td>pass=false</td>
</tr>
</tbody>
</table>

- Can be of any fixed size

  - (2+2, 7>8) ; (4,false)
  - int * bool

- Elements can have different types
- Tuples can be nested in other tuples

- All evaluation rules look like: