CSE 130 : Winter 2006
Programming Languages

Lecture 2: Simple Expressions

Ranjit Jhala
UC San Diego

Begin at the beginning ...

Exec-time
“Dynamic”

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 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: real

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

Base Type: string

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>False</th>
<th>b: bool</th>
<th>b -&gt; b</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 &lt; 3</td>
<td>true</td>
<td>e1 &lt; e2</td>
<td>a1 = v1, e2 = v2</td>
</tr>
<tr>
<td>not (2&lt;3)</td>
<td>false</td>
<td>not e</td>
<td>e = v1, e = not v</td>
</tr>
<tr>
<td>(&quot;ab&quot;=&quot;cd&quot;)</td>
<td>false</td>
<td>e1 = e2</td>
<td>a1 = v1, e2 = v2</td>
</tr>
<tr>
<td>not (2&lt;3) and also</td>
<td>false</td>
<td>e1 = e2</td>
<td>a1 = v1, e2 = v2</td>
</tr>
</tbody>
</table>

**Type Errors**

```
"pq" ^ "45";
(2 + "a") ;
```

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

```
(1,2,3,"ab","cd",7>8) ;
```

**Equality testing is built-in for all expr, values, types**
- but compared expressions must have same type
- ...except for?
  - function values ... why?

```
("ab"="cd") ;
```

**Tuple operators: “Select”**

- Selects an element of a tuple

```
3 : (1,2,3,"ab","cd",7>8) ;
```

- “Selector” must be an integer constant
  - between 1 ... tuple size

```
5 : (1,2,3,"ab","cd",7>8) ;
```

```
(1,2,3) : T1 * T2 ;
```

```
(1,2,3) : T1 ;
```

```
(1,2,3) : T1 * T2 ;
```

```
(1,2,3) : T1 ;
```

```
(1,2,3) : T1 * T2 ;
```

```
(1,2,3) : T1 ;
```

```
(1,2,3) : T1 * T2 ;
```

```
(1,2,3) : T1 ;
```

```
(1,2,3) : T1 * T2 ;
```

```
(1,2,3) : T1 ;
```

```
(1,2,3) : T1 * T2 ;
```

```
0 : (1,2,3) ;
```

```
(1,2,3) ;
```

```
(1,2,3) ;
```

```
(1,2,3) ;
```

```
(1,2,3) ;
```
Complex types: Records

```
{name="ranjit",
  age=27,
  pass=false}
```

Records are tuples with named elements...
- Names instead of numbers in selector

```
#age
{name="ranjit",age=28,pass=false}  28  int
```

```
{ name = "ranjit" ,
  age = 27 ,
  pass = false }
```

Complex types: Lists

```
[] : 'a list
[] => []
```

All elements have the same type

```
(1, "pq");
```

Complex types: list ..construct

Cons “operator”
```
1::[2,3]  [1,2,3]
```

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

Complex types: list ...deconstruct

Reading the elements of a list:
- Two “operators”: hd (head) and tl (tail)

```
[1,2,3,4]  [1,2,3,4,5]
```

Can only append lists of the same type
### List: Heads and Tails

**Head**
- \( e : \text{T list} \)
- \( \text{hd} : e \Rightarrow T \)
- \( \text{hd} e \Rightarrow e \)

**Tail**
- \( e : \text{T list} \)
- \( \text{tl} : e \Rightarrow \text{T list} \)
- \( \text{tl} e \Rightarrow e \)

\[
(\text{hd } \langle [], [1,2,3] \rangle) = (\text{hd } \langle [], ["a"] \rangle)
\]

**`list`**
- `e : T list`
- `e : T : bool`

**`string list`**
- `e : T list`
- `e : e : T : bool`

### List: Testing emptiness

**null**
- `e : T list`
- `null e` : `true`
- `null e` : `false`

\[
\begin{align*}
\text{null } ["a", "b", "c"] & \equiv \text{false} \\
\text{null } [] & \equiv \text{true} \\
\end{align*}
\]

**null 0**

### Recap

**Expressions (Syntax)**
- Dynamic

**Values (Semantics)**
- Compile-time
- Static

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

### If-then-else expressions

If-then-else is also an expression!

Can use any expression in then, else branch

\[
\begin{align*}
\text{if } e_1 \text{ then } \text{e2 else e3} \\
\text{if } e_1 : \text{bool} \text{ then } e_2 : \text{T} \text{ else } e_3 : \text{T} \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \text{T} \\
\end{align*}
\]

- Then-subexp, Else-subexp must have same type!
- __which is the type of resulting expression__

\[
\begin{align*}
\text{if } e_1 : \text{bool} \text{ then } e_2 \text{ else } e_3 : \text{T} \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \text{T} \\
\end{align*}
\]
If-then-else expressions

\[
\text{if } a \text{ then } b \text{ else } c \\]

- Then-subexp, Else-subexp must have same type!
- Equals type of resulting expression

```plaintext
if 1>2 then [1,2] else []  if 1<2 then [] else "a" []
```