Lists

Accessing Lists

Accessing Lists

If we want to take apart a list into its component pieces, we have to say what to do with the list if it's empty, and what to do if it's non-empty (that is, a cons of one element onto some other list). We do that with a language feature called *pattern matching*.

Accessing Lists

If we want to take apart a list into its component pieces, we have to say what to do with the list if it's empty, and what to do if it's non-empty (that is, a cons of one element onto some other list). We do that with a language feature called *pattern matching*.

Here's an example of using pattern matching to compute the sum of a list:

```
let rec sum lst =
  match lst with
  | [] -> 0
  | h :: t -> h + sum t
```

```
val sum : int list -> int = <fun>
```

Pattern Matching

Syntax.

```
match e with
| p1 -> e1
| p2 -> e2
| ...
| pn -> en
```

Each of the clauses `pi -> ei` is called a *branch* or a *case* of the pattern match.