Record Types
A set of field names and their associated types, surrounded by curly braces.

Example: The triples that we needed for Project 2 in CS109A (cash if out of the market and stock/cash if in) could be given the type

```ml
type stockRec =
    {cash_out: int, stock_in: int, cash_in: int};
```

- But different fields could have different types.

Record Values
A set of field names, each followed by = and a value of appropriate type. All are surrounded by curly braces.

Example: A possible value of type stockRec is

```ml
val myRec =
    {cash_out=500, stock_in=15, cash_in=10};
```

Extracting Field Values
The expression $#f(r)$ returns the value of field $f$ of record $r$.

Example:

```ml
#stock_in(myRec);
val it = 15 : int
```

Tuples and Records
An ugly little secret: tuples are just a shorthand for a record structure in which the fields are named 1, 2, etc.

- That’s why $i(t)$ extracts the $i$th component from a tuple $t$.

Deducing a Record Type
ML cannot assume that the only fields a record has are the ones it sees. Thus, the following attempt to decide whether we are better off in or out is
erroneous.

(* decide(r,v) determines if the cash-if-out in record 
  r exceeds the value of the stock-and-cash-in, assuming 
  v is the stock price *)
fun decide(r,v) =
  #cash_in(r) > v**#stock_in(r) + #cash_in(r);

Error: unresolved flex record in let pattern
  more errors . . .

• The problem is that ML doesn’t know these 
  are the only fields of r.

• Fix by declaring the type of r somewhere, e.g.,
  fun decide(r:stockRec,v) =
    #cash_in(r) > v**#stock_in(r) + #cash_in(r);
  val decide = fn : stockRec * int → bool

Ellipses

When writing patterns involving record types, we 
may specify the fields in any order. We may also 
omit some fields by using the ellipsis or wildcard 
symbol, . . . .

• But remember that ML must be able to figure 
  out the full set of fields somehow.

Example: A function that tests if an “in” posi-
tion leaves no cash left over:

fun noCash({cash_in=0,...}:stockRec) = true
  | noCash(_) = false;

• The type stockRec is sufficient to tell ML 
  what the fields are.

Matches

A match is an expression consisting of one or more 
subexpressions

    pattern => expression

separated by bars.

• A match M is applied to a value v. The first 
  pattern that matches the value determines 
  the result as follows:
First, any variables in the pattern are bound to values they match in \( v \).

Then the associated expression, which may involve variables of the pattern, is evaluated, yielding the value of \( M \) applied to \( v \).

Example: Here is a match that tells if a list has zero, one, two, or many elements:

\[
\begin{align*}
nil &\Rightarrow "zero" \mid \\
[x] &\Rightarrow "one" \mid \\
[x,y] &\Rightarrow "two" \mid \\
_ &\Rightarrow "many"
\end{align*}
\]

Warning: the above is not an expression; it is used within expressions.

Using Matches

A match \( M \) can be used to:

1. Define anonymous functions. \texttt{val f = fn M}
defines \( f \) to be a function that applies \( M \) to its argument.

OK; so now \( \texttt{fn M} \) is no longer “anonymous,” but the point is you can use \( \texttt{fn M} \) any place a function of the appropriate type is expected, without first calling it \( f \).

Example:

\[
\begin{align*}
\texttt{val f = fn } \\
&\{ \\
nil &\Rightarrow "zero" \mid \\
[x] &\Rightarrow "one" \mid \\
[x,y] &\Rightarrow "two" \mid \\
_ &\Rightarrow "many"
\}
\end{align*}
\]

\texttt{f([1,2]);}
\texttt{val it = "two" : string}

2. Match \( M \) can be used in a case statement. 
\texttt{case v of M} causes \( M \) to be applied to \( v \).
Example:

fun f(x) = case x of
    nil => "zero" |
    [x] => "one" |
    [x, y] => "two" |
    _ => "many"