```
let name = expr
```

```
let name = expr1 in expr2
```

The `name` binding is only active in `expr2`.

Instances of shadowing, creating a hole in the previous binding's scope.

```
fun param1 ... paramN -> expr
```

By using the `begin` and `end` keywords.

As in Haskell, give the function name followed by the arguments, space delimited.

```
funcname arg1 ... argN
```

As in Haskell:

```
param1Type -> ... -> paramNType -> resultType
```

They associate right, so the following two are equivalent:

```
X -> Y -> Z
X -> (Y -> Z)
```

Simply leave off arguments, at the end of the parameter list. The result will be a new function.

```
let name param1 ... paramN = expr
```

Values in their definition environment, not the environment of the function call.

... the `rec` keyword after `let`.

```
let rec name1 paramlist1 =
    expr1
and name2 paramlist2 =
    expr2
```

Operator names and identifiers.

Regular identifiers are made of letters, digits, apostraphes and underscores. They must start with a letter or an underscore. An underscore by itself is not a legal name.

They are made up of only operator symbols.

Operators default to infix usage.

By enclosing them in parens.

The comment syntax. Use spaces around the `*`.

Labeled parameters are a way to give formal parameters names so arguments can be provided to them in any order.

For each parameter instead of just giving it a name use `~label:name`. Then in a function call `~label:arg` can be used.

... positionally or with labels.

`~label` where `label` serves as both the name and label for the parameter.

```
?(label = expr)
```

`expr` provides the default value in case no argument is provided for the parameter in the call. To provide the argument the label must be used.

If an optional parameter comes last and no argument is provided, it could be to use the default argument or to partially apply the function.

The same label. In the call, argument order will determine the binding of arguments to parameters.

```
?someLabel:int

?(someLabel = initialVal)
```

Otherwise at call site the compiler will prefer to assume the invoked function has a labeled argument not an optional argument, regardless of which was intended.