How are variables created?

```
let name = expr
```

Create a variable with limited, lexical scope.

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

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

Apparent re-assignments are actually what?

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

What is the anonymous function syntax?

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

How can parens be avoided, in general?

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

What is the syntax of function application?

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

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

How are types written?

Which way do they associate?

As in Haskell:

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

They associate right, so the following two are equivalent:

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

How are functions partially applied?

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

What syntactic sugar is provided for declaring named functions?

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

Variables in functions are bound to what?

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

Recursive functions require ...

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

How are mutually recursive functions written?

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

Name two kins of names.

Operator names and identifiers.

What are the rules of non-operator names?

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.

What are the rules of operator names?

They are made up of only operator symbols.

What is the syntactic difference between operators and regular identifiers?

Operators default to infix usage.

How can operators be used in the prefix position?

By enclosing them in parens.

The prefix definition of $*$ conflicts with what?

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

What are labeled parameters?

What is its syntax?

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.

Arguments to functions with labeled parameters can be provided either ...

... positionally or with labels.

What is the shorthand syntax for labeled parameters?

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

Give the syntax of optional parameters.

```
?(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.

Why should each optional parameter by followed by an mandatory one?

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.

Oddly, two labeled parameters can share what?

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

What is the type of a labeled `int`?

What is the type of an `int` with a default value?

```
?someLabel:int

?(someLabel = initialVal)
```

Why in higher-order function definitions should type annotations be given to parameters that are functions?

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.