- Have signed and unsigned versions.
- Include a fixed precision point type.

You must use `pown` instead of `**`, or convert to a floating-point type.

They have the same names as the types being converted to.

There is no type coercion.

Language support. Literals ending with `I` are `System.Numerics.BigIntegers`.

`&&&`, `|||, `^^^`, `<<<`, `>>>`

```
"abcd".[0]
```

- One can end each line with a backslash while still in a string. Leading whitespace will be removed from the next line.
- With verbatim strings, but leading whitespace will not be removed.

An `@` before the string begins.

The suffice `B` after the close quote of a string literal.

`not` for unary negation.

= and <>.

```
let funcname arg1 arg2 ... argN = body
```

The value of the last expression in a given path. There is no `return` keyword.

As in Haskell:

```
type1 -> type2 -> ... -> typeN
```

... `int` parameters instead of type parameters.

As usual, `name : type`.

With a leading `'`, e.g., `'a`, `'b`.

In the same block. E.g., two consecutive lines can `let` the same identifier.

```
if   cond1 then res1
elif cond2 then res2
...
else res
```

The same type. Upcasting will not be done.

Comma separated values surrounded by round parens.

```
type1 * type2 * ... * typeN
```

- Use `fst` and `snd`.
- Unpack with `let`

```
let a, b, c = (1, 2, 3)
```

Semi-colon separated values surrounded by square brackets.

Basically arbitrary code within square brackets. Elements are selected using the `yield` keyword.

Eagerly, in memory.

```
for var = start-expr to end-expr do
    //body
```

`to` is inclusive, as in Scala.

```
for pattern in expr do
    //body
```

```
for pattern in expr -> expr
```

`reduce` uses the first element of the list as the initial accumulator value, `fold` takes the initial accumulator value explicitly.

`List.iter` is like Scala's `foreach`.

As in Scala, `Option` can have values `Some('a)` or `None`.

The arguments to the formatting string are type checked, contributing to type inference.

`%d, %i` - integer

`%s` - string

`%f` - floating-point number

`%c` - character

`%b` - boolean

`%O` - object

`%A` - anything

you are using the `anonymous module` which can nonetheless be accessed using the filename with the first letter capitalized.
For example, `MyFile.someValue`.

```
module MyModule
//whatever
```

Equal sign and indentation are mandatory:

```
module Outer
module Inner =
    //whatever
```

- they can contain declarations but not values.
- they cannot be nested

Namespaces: large object-oriented programs
Modules: rapid prototyping

Straight down the *last* code file.

Add the `[<EntryPoint>]` attribute to a method.
It must be in the last function in the last file of the program.
It must take a `string array` and return an integer.