**Introduction:**

The basic expression type is
type expr
    This contains the basics of: variables, integers, booleans,
    binary operators, let expressions, and if expressions of the form:

```
| Var of string
| Int of int
| Bool of bool
| Binop of bop * expr * expr
| Let of string * expr * expr
| If of expr * expr * expr
```

    As well as two special types, which I will explain later, of the form:

```
| Vector2 of expr * expr
| TaggedExpr of expr * tag
```

The typechecker runs on an expression and results in a "failwith" or a type of the form:
type typ
    This contains the types

```
| TInt
| TBool
```

    and

```
| TVector2 of typ * typ
| TTagged of typ * tag
```

**Run instructions:**
To make the project type "make build" to compile, and "make" to compile and run.
And "make clean" removes all compiled files.

You may need to run:
    opam install ocamlfind
    opam install ocamlbuild

**Utop:**
This project currently runs as an extension of the ocaml command line using utop.
Typing "make" will enter utop, where we can execute the examples listed below.
Each command listed below will be of the format
    #command;;
Results will be of the format:
    - : type  = value (or similar)
Please copy the "command;;" portion, ignoring the demarking #.
To exit utop at any point type #exit 0;; or similar command.

**Parsing and typechecking:**
We lex and parse an expression from a string, for example, run:
    #let intExpr = parse "1";;
    which yields "val intExpr : expr = Int 1", a basic integer expression.
Then check that the type is valid and return the type if it is by running:
    #typecheck intExpr;;
    Which gives "typ = TInt".

Now for some more interesting examples.
As described in the writeup, the goal of this project is to reduce subtle bugs in graphics programs
caused when calculations occur in reference to two different spaces or coordinate systems.
In line with this I implemented a Vector2 object, and a TaggedExpression system.

An example vector is:
    #parse "vector2(1,2)";;

Now let us explore tags, the main function of this project.
Let us instantiate two vectors wrapped Tags.
For this example lets say one is in cartesian coordinates, and the other in spherical.
The format for Tags is "(expr) tagged {tagVame":"tagValue}"
    #let tagExpr1 = parse "(vector2(1,2)) tagged {coordinateSpace:cartesian2D}";;
    #let tagExpr2 = parse "(vector2(3,4)) tagged {coordinateSpace:spherical}";;

These give you an expression of the form

```
    TaggedExpr (Vector2 (Int 1, Int 2), Tag ("coordinateSpace", "cartesian2D"))
    TaggedExpr (Vector2 (Int 3, Int 4), Tag ("coordinateSpace", "spherical"))
```

**Example - Catching a bug via incorrect tags:**
Now if we try to typecheck an operation with those two vectors, it should fail, since the tags do not match.

For simplicity let us manually setup an addition of the two vectors
(this could also be written out as one string).

```
    #let badAddExpr = Binop (Add, tagExpr1, tagExpr2);;
```

which yields an expr of:

```
    Binop (Add,
    TaggedExpr (Vector2 (Int 3, Int 4), Tag ("coordinateSpace", "cartesian2D")),
    TaggedExpr (Vector2 (Int 3, Int 4), Tag ("coordinateSpace", "spherical")))
```

Now if we typecheck that addition:

```
    #typecheck badAddExpr;;
```

It will result in

```
    Exception: Failure "Tags of same space must have matching values".
```

**Example - Correct tags:**
To fix this we just just change the tags to match, lets make them both cartesian2D.

```
    #let tagExpr1 = parse "(vector2(1,2)) tagged {coordinateSpace:cartesian2D}";;
    #let tagExpr2 = parse "(vector2(3,4)) tagged {coordinateSpace:cartesian2D}";;
```

Then

```
    #let goodAddExpr = Binop (Add, tagExpr1, tagExpr2);;
    #typecheck goodAddExpr;;
```

Which tells us that the type is:

```
    TTagged (TVector2 (TInt, TInt), Tag ("coordinateSpace", "cartesian2D"))
```

**Further examples with Tags:**
Tags can be wrapped around any expression, and the tag "space" and "value" can be defined by any two strings.
For shorter code let us use a more abstract example.

```
    #let taggedInt1 = parse "(42) tagged {numberType:cool}";;
```

Now let us create another int, with a mismatching tag value but in a different tag space.

```
    #let taggedInt2 = parse "(69) tagged {oddOrEven:odd}";;
    #let okMultiplyExpr = Binop(Mult, taggedInt1, taggedInt2);;
    #typecheck okMultiplyExpr;;
```

As discussed in the writeup, this succeeds because we declared tags to be inclusive not exclusive.

In the example above, since there were two tagged expressions, the first tag stays while the other is discarded.
If there is ever only one tag it will be the one that remains. For example:

```
    #let taggedInt = parse "(1) tagged {reference:world}";;
    #let untaggedInt = parse "1";;
    #typecheck (Binop(Leq, taggedInt, untaggedInt));;
    #typecheck (Binop(Leq, untaggedInt, taggedInt));;
```

And they should have the same resulting tag.

Notes:
*TaggedExpressions can be nested inside each other for some interesting effects.
But since this demo is already pretty long, feel free to try it out on your own, or read about it in the writeup.

*if you want previously instantiated variables to work in "typecheck" function you must call it with the first parameter of a generated context to the "typeof" function. The default is the empty context for simplicity.

*all types in this project can be evaluated under eval_small or eval_big.
However since the focus of this project is on the typechecker instead of evaluation there may be bugs