ian's random thoughts :(

Type checking is hard when you have mutually recursive types.

CASE 1

```
type r = {B:b, A:a}
type a = b
type b = c
type c = int
```

This is legal. But how do we differentiate from this case:

CASE 2

```
type r = {B:b, A:a}
type a = b
type b = c
```

has no base type (assuming c is not defined elsewhere) and is illegal.

So when we declare a record type, the types in the fields can be: 1) in the tenv already, 2) defined within the same TypeDec (mutually recursive) 3) in neither (illegal)

We also need to check for illegal cycles, which are cycles with no record or array type.

CASE 3

```
type r = {A:a}
type a = b
type b = a
```

On the other hand, this cycle is valid:

CASE 4

```
type r = {A:a}
type a = b
type b = r
```

could also have

CASE 5

```
type r = {A:a}
```

a already in the outer tenv but not in the typedec.

Idea

So how bout we add everything in a typedec into a symbol table first.
…wait but how do we add things like:

CASE 6

```
type a = b
type b = int
```

in this case we cannot add 'a' as we do not yet know b's type is int.

another idea

ok so how about this, in the function get_fields we:
1. Loop over the entire typedec.
2. If we see a record type, we add it to the local_tenv first. record types can be added as `Types.RECORD(get_fields, ref() )`.
2. If we see a name type, we call our special recursive function *R*, which will either report an error or return the base type of the name type.
3. get_fields will then search local_tenv and tenv when it is eventually called.

So what is R?

```
R(type_symbol:Absyn.Symbol)
```

- R has closure over typedec and tenv so it only takes one argument.
- looks for what type_symbol maps to in typedec.
- If it finds type_symbol maps to none of: int, string, record or array, it calls itself again. At some point, the recursion will bottom out.
- In this case, either we have a type_symbol that is one of the base types, OR we cannot find the symbol in the typedec.
- At this point, the recursive function looks at the outer tenv. If the symbol is in the outer tenv, it succeeds else we have something like **CASE 2** where we should rightly return a failure.
- In successful cases we will add the mapping to the local_tenv

Let us test this idea on **CASE 7**.

CASE 7

```
type r = {A:a, RR:rr, C:c}
type rr = {R:r}
type a = b
type b = int
```

we assume c was defined to be a string already.

the typedec is

```
[{name=r, ty=RecordTy(...)}, {name=r, ty=RecordTy(...)}, {name=a,ty=NameTy(...)}, {name=b,ty=NameTy(...)}]
```

the tenv is:

```
c|->string
```

So after two iterations, the local_tenv looks like:

```
r |-> Types.RECORD (get_fields, ref() ), rr |-> Types.RECORD (get_fields, ref() )
```

now look at the third iteration of the 'loop' over the typedec. We find the first NameTy, a|->b. b is not one of the base types, so we call `R(typedec, b)`. R finds b|->`int`. Thus, we put the mapping a|->`int` into the local_tenv.
local_tenv:

```
r |-> Types.RECORD (get_fields, ref() ), rr |-> Types.RECORD (get_fields, ref() ), a|->int
```

to the second iteration of the outer loop, we find the second NameTy b->int. This is added directly to local_tenv.
local_tenv:

```
r |-> Types.RECORD (get_fields, ref() ), rr |-> Types.RECORD (get_fields, ref() ), a|->int, b|->int
```

we are done till the get_fields() is called. get_fields has a closure over local_tenv and tenv.
the tenv is:

```
c|->string
```

:

```
r |-> Types.RECORD (get_fields, ref() ), rr |-> Types.RECORD (get_fields, ref() ), a|->int, b|->int
```

if `get_fields()` is called for `r`, it will find `a` in `local_tenv`, `rr` in `local_env` and `c` in `tenv`.

Okay

So this seems to work, but there are no double for loops in SML. We need to do something like:

```
foldl
    (fn(dec,localtenv => let
    val sym = #name dec
    val ty = #ty dec
    fun R = ....
    in
        case ty of NameTy(type_sym, pos) => S.enter(localtenv, sym, R(type_sym))
        | RecordTy(fieldlist) => S.enter(localtenv, sym, Types.RECORD(get_fields, ref()))
        | ArrayTy(type_sym*pos) => S.enter(localtenv, sym, Types.ARRAY(R(type_sym), ref())
        | (_) => ???
        )
    end)
    localtenv, typedec
```

Now the problem is that between two get_fields functions created in the same typedec, the enclosed ref() for arrays and record are different because they are generated fresh for every TypeDec.