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 `Typess.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=rr, 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
```

the fourth iteration of the 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 we need to do something like:

```
fold1
    (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
```

A problem with our unit refs...

TransTy is called to generate types on TypeDecs. When we check array or record types in TransExp and TransDec, we will look up the symbol table to find the corresponding Types.RECORD or Types.ARRAY. These will have unique refs. When we do something like

```
var someRecordTypeInstance: someRecordType = {a=1, b="somestring", c=someRecordTypeInstance2}
```

, we will look up the types of someRecordType as well as its third field, which is supposed to be itself. Both unique refs should be equal.

In another case, we can have mutually recursive record types.

```
var a: someRecordType1 = {b:someRecordType2=nil}
var b: someRecordType2 = {a:someRecordType1=a}
```

from the first vardec, our type checker will first add `a|->someRecordType1` to the venv. It then looks up `someRecordType1` in the tenv to find its fields types. nil is a member of every record type so this works.

At the second vardec, we add `b|->someRecordType2` to the venv. We then look up `someRecordType2` in the tenv and see that it has one field with symbol a and type `someRecordType1`.The `someRecordType1` derived from the second line came from the closure of `someRecordType2`'s get_fields method. It does not have the same unit ref as the someRecordType1 in the tenv.

NOTE: when we repeatedly call transty on typedec, the typedec was pass in has to be truncated from the currently processed element onwards.

Why? Well, using **CASE 7** as an example again, if we have a typedec:

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

and lets say we *already* added `r|->RecordTy` to tenv.

Then when TransTy processes `rr`, we only pass in:

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

This is so that our function in, which has closure over the truncated typedec, won't find r in our truncated typedec. It will instead find it in the tenv.

Solving the unit ref problem...

CASE 8

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

last ingredient is to make get_fields handle unit refs properly. Now, when it finds a field in the tenv, it will return the unit ref from the tenv. If it doesn't, it can return the field from the local_tenv, which will have a newly initialized unit ref.

first tydec:

B and C are in localenv with new unit refs. [self]A added to tenv with new unit ref.

second tydec:

A in tenv, use that unit ref. [self]B in A.fields, use that unit ref. C in A.fields, use that unit ref.

third tydec:

B in tenv, use that unit ref. A in tenv, use that unit ref. [self]C in A.fields, use that unit ref.

so, every time we run get_fields, when find a field in the tenv and if the type of the field is a record, we need to add that field's fields to the search space. The search space is now, in order of priority, (related_search_space + tenv + localtenv =)
