Type Checking

- **Environments, continued**
  - **Name spaces**
- **Type checking**
  - **Expressions and variable declarations**
  - **Functions**
  - **New types**

Name Spaces

- **Programs need different name spaces for types versus functions +variables**
  - **Flexible, if have 2 separate environments**
  - **Different symbol tables for Type environment and Variable environment**

```
let type a = int
        var a : a = 5;
        var b : a = a;
in b+a
end
```

Types in Tiger

- **Primitive: *int, string***
- **Record: fields with names and types**
  - **Address of object itself**
  - **Tiger does not use *structural equivalence***
  - **Uses *name equivalence***

```
let type a = {x: int, y: int}
    type c = a
```

- **Array: type of entries**
  - **Address of object itself**
- **VOID - return type for procedures**

Type Equivalence

- **Structural equivalence**
  - **Implies "matching" that only checks "shape" of aggregate type subpiece by subpiece (used in C except for records)**
  - **More formally (Sethi, Programming Languages and Concepts, 1989, Addison-Wesley)**

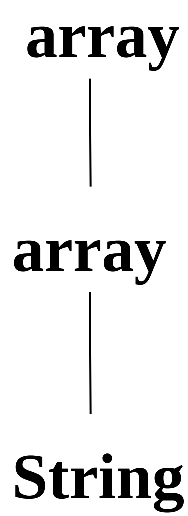
A type name is structurally equivalent to itself

Two types are structurally equivalent if they are formed by applying the same type constructor to structurally equivalent types

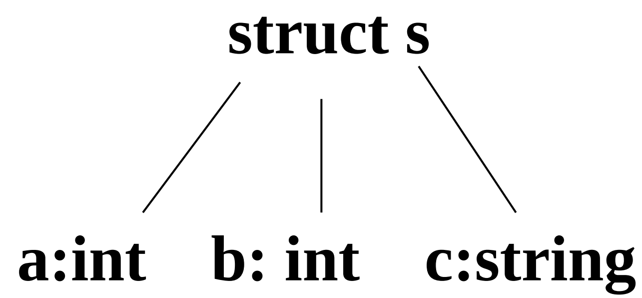
After *type n = T*, then *n* and *T* are structurally equivalent.

Examples

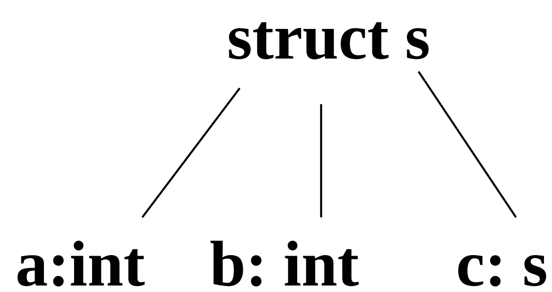
a C array



a Java 2 diml array



a C struct



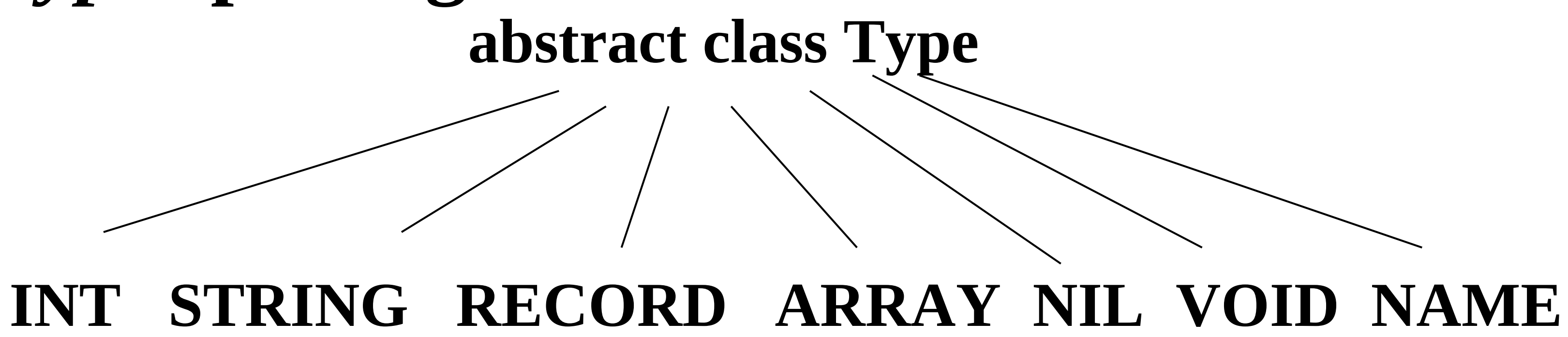
a recursive (self-referential) type

Type Equivalence

- **Name equivalence**
  - **One type declared to be same as another named type**
  - **More restrictive than structural equivalence; used in Ada**
- **Interesting questions about equivalence for arrays, depending on whether or not array bounds are considered part of the type**

Type Checking in Tiger

- **Will build the *Semant* package**
- **Will use two different *Table* objects to implement type environment and value environment - *tenv, venv***
- ***Types* package**



Implementation

- ***Entry*** **abstract class**
  - ***VarEntry*** **subclass for encapsulating variable type**
  - ***FcnEntry*** **subclass for encapsulating function signature**
    - **Types. RECORD of formals --> Types.Type result**
  - **More instance variables to be added later**
- **Include predefined functions in value environment *Table venv***

Type Checking

- **Type checking involves a recursive walk of the abstract syntax tree of an expression**
- **Need to define functions for different AST nodes**
- **Appel in section 5.3, suggests an organization for type checking code, that can be extended later to do other tasks**

Type Checking

- **Essentially, have to type check each construct with a separate analysis**
  - ***transVar(Absyn.Var e), transExp(Absyn.Exp e), transDec(Absyn.Dec e), transTy(Absyn.Ty e)***
- **Result of a type check is a *ExpTy* object which encapsulates the expression object plus its type**

Type Checking - Expressions

- **Type checking a binary expression involves checking the type of each operand for consistency with the operator (Appel, p121)**

```
ExpTy transExp (Absyn.OpExp e){
    ExpTy left = transExp(e.left);
    ExpTy right= transExp(e.right);
    if (e.oper == Absyn.Op.PLUS) {
      if (! (left.ty instanceof Types.INT))
                  error(e.left.pos, "integer required");
      if (! (right.ty instanceof Types.INT))
                  error(e.right.pos, "integer required");
      return new ExpTy(null, new TYPES.INT() );
}  }
```

Type Checking - Variables

- **Need to lookup declared type and return *ExpTy* object for it or *undeclared variable error* (Appel p121)**
  - **Needs to lookup variable in symbol table for value environment, check it is a *VarEntry*, and then return its declared type**
  - **If find a NAME type, need to translate to its actual type(s) to return as type of variable**

Type Checking - Declarations

- **Declarations only appear in *let* expressions (Appel, p 123)**
- **When processing *let* expression, have to keep track of entering and leaving a new scope for *venv* and *tenv***
- **Then call *transDec()* to process declarations, building the augmented environment**
- **Finally, type check the body of the *let* expression**

Declarations

```
ExpTy transExp( Absyn.LetExp e) {
    env.vevn.beginScope();
    env.tenv.beginScope();
    for (Absyn.DecList p = e.decs; p!= null; p=;.tail)
        transDec( p.head);//augment envs
    ExpTy et = transExp(e.body);//type check body expression
    env.venv.endScope();
    env.tenv.endScope();
    return new ExpTy(null, et.ty);// returned type of let expr
}
```

Declarations

- **Of *variables* - with and without initialization**
  - **Need to check initializing expression is right type**
- **Of *types* (nonrecursive) -**
  - **Need to turn Absyn types into Types types**
  - **May need to handle named types through lookup in *tenv***
- **Of *functions* (nonrecursive) - need to form a *FcnEntry*, then define new scope and add params one by one, then type check fcn body**

Recursive Declarations

- E.g., record types, arrays of array, recursive functions
- Naïve approach will find undefined type in function body
  - (Appel p 126) For mutually recursive functions, process fcn declarations twice;
    - Form headers from fcn name and types of params
    - Rescan params and enter them into environment as new scope; then type check function body

Example

type a = b;

type b = d;

type c = a;

type d= a;

Cycle of types here a → b → d → a is illegal! because there is no record or array declaration corresponding to any of these types; this should be detected as such by type checker

Type Checking - Calls

- **For function calls, need to type check function name and all arguments**
  - **Need to lookup function entry in value environment, check each argument type versus the parameter type in signature, and return the result type of call expression**