Type Checking

What is it

- Its semantic Aspect of compilation
- Allows programmer to limit what types may be used in certain circumstances
  - What operator is used for what types of operands
- Assigns types to values
- Determines whether these types are used in an appropriate manner
- Simplest Situation: check types of objects and report a type error in case of violation
  - Type mismatch error
  - x(int)=y(string))
- More Complex: incorrect types may be corrected (type coercing)
  - x(int)=(int)[y(float)]

```
int x[5];     x[7]=10;
```

Static Checking	Dynamic Checking
Type checking done at compile time	Performed during program execution
Properties can be verified before program run	Permits programmers to be less concerned with types
Can catch many common errors	Mandatory in some situations, such as array bounds check
Desirable when faster execution is important	More robust and clearer code

Type Expressions

- Used to represent types of language constructs
- A type expression can be
  - Basic type: integer, float, char, Boolean and other atomic types that do not have internal structure. A special type, type-error is used to indicate type violations
  - Type name
    - typedef myint int myint I =int J
  - Type constructor applied to a list of type expressions
    - int a[10], b[5]; a=b is type error
    - struct a{int x; float y;};

Type Expressions

- Arrays are specified as array(I,T), where T=type and I=integer or range of integers
  - int a[5] := type of **a** to be an array(5,integer)
  - a x array(5,integer)

- If T1 and T2 are type expressions, T1 x T2 represents “anonymous records”
  - An argument list passed to a function with first argument integer and second float, has type integer x float
    - fun(a,b); T=T1 x T2 T=int x float
    - void fun(int x, float y){} T’=T1 x T2 T’=int x float

Type Expressions

- Named records are products with named elements. For a record structure with two named fields – length(an integer) and words (of type array(10,char)

  record((length x integer) x (words x array(10, character)))

struct record{int length; char words[10];};

- If T is a type expression, pointer(T) is also type expression representing objects that are pointers to objects of type T

- Function maps a collection of types to another, represented by D→R, where D is the domain and R is the range of the function
  - int fun(int, char, float){return;}
  - domain{integer x character x float}→{integer}
  - Range{ integer }

Type Expressions

- Function maps a collection of types to another, represented by D→R, where D is the domain and R is the range of the function
  - int fun(int, char, float){return;}
  - domain{integer x character x float}→integer
  - Range{ integer }

- integer x integer → character ?

- Integer→(float→character) ?

Type Systems

- Collection of rules depicting the type expression assignments to program objects
- Usually done with syntax directed definition
- ‘Type Checker’ is an implementation of a type system

Strongly typed language

- Compiler can verify that the program will execute without any type errors
- All checks are made statically
- Also called a sound type system
- Completely eliminated necessity of dynamic type checking
- Most programming languages are weakly typed
- Strongly typed languages put lot of restrictions
- Some type errors are caught dynamically only
- Many languages allow user to override the system

Strongly typed language

- Compiler can verify that the program will execute without any type errors
- All checks are made statically
- Also called a sound type system
- Completely eliminated necessity of dynamic type checking
- Most programming languages are weakly typed
- Strongly typed languages put lot of restrictions
  - 10+10.5 (not allowed)
- Some type errors are caught dynamically only
- Many languages allow user to override the system
  - int y=(int)10.5;

Type checking of Expressions

- Use synthesized attribute ‘type’ for non terminal

Expression	Action
$E \rightarrow id$	E.type $\leftarrow$ lookup(id.entry)
$E \rightarrow E_1$ op E_2	E.type $\leftarrow$ E_1.type==E_2.type then E_1.type else type.error
$E \rightarrow E_1$ relop E_2	E.type $\leftarrow$ E_1.type==E_2.type then boolean else type.error
$E \rightarrow E_1[E_2]$	E.type $\leftarrow$ E_2.type==integer and E_1.type == array(s,t) then t else type.error

Type checking of statements

- Statements normally do not have any value hence void

Expression	Action
S→id=E	S.type←id.type == E.type then void else type.error
S→if E then S_1	S.type←E.type==Boolean then S_1.type else type.error
S→while E do S_1	S.type←E.type== Boolean then S_1.type else type.error
S→S_1;S_2	S.type←S_1.type==void and S_2.type == void then void else type.error

Type checking of function

- A function call is equivalent to the application of one expression to another

Expression	Action
$E \rightarrow E_1(E_2)$	$E.type \leftarrow E_2.type == s$ and $E_1.type == s \rightarrow t$ then t else type.error

Type equivalence

- It is often needed to check whether two type expressions ‘s’ and ‘t’ are same or not

- Can be answered by deciding equivalence between the two types

- Two categories of equivalence
  - Name equivalence
  - Structural equivalence

Name equivalence

- Two types are name equivalent if they have same name or label

  typedef int Value

  typedef int Total

  …

  Value var1,var2

  Total var3, var4

- Variable var1 and var2 are name equivalent, so are var3 and var4
- Variables var 2 and var3 are not name equivalent, as their type names are different

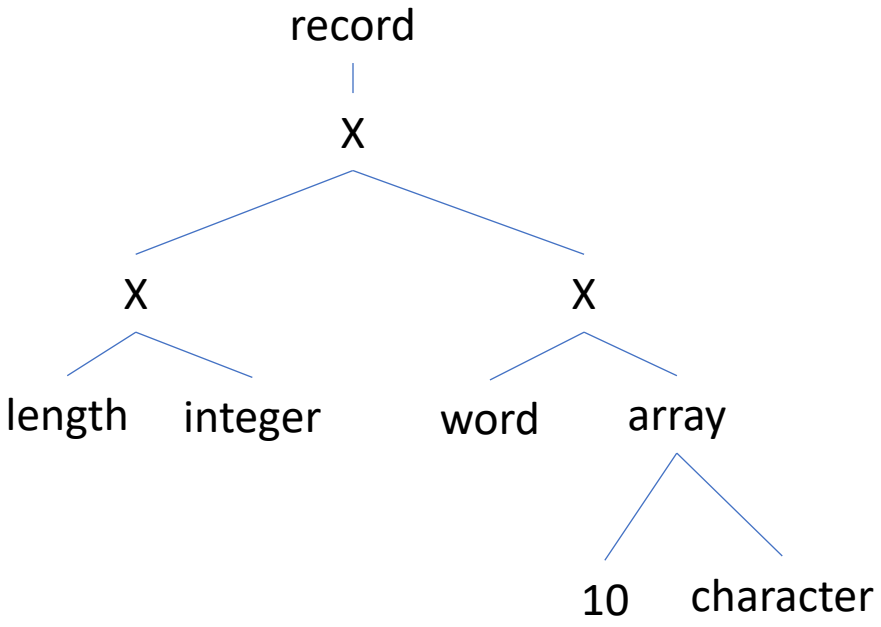
Structural equivalence

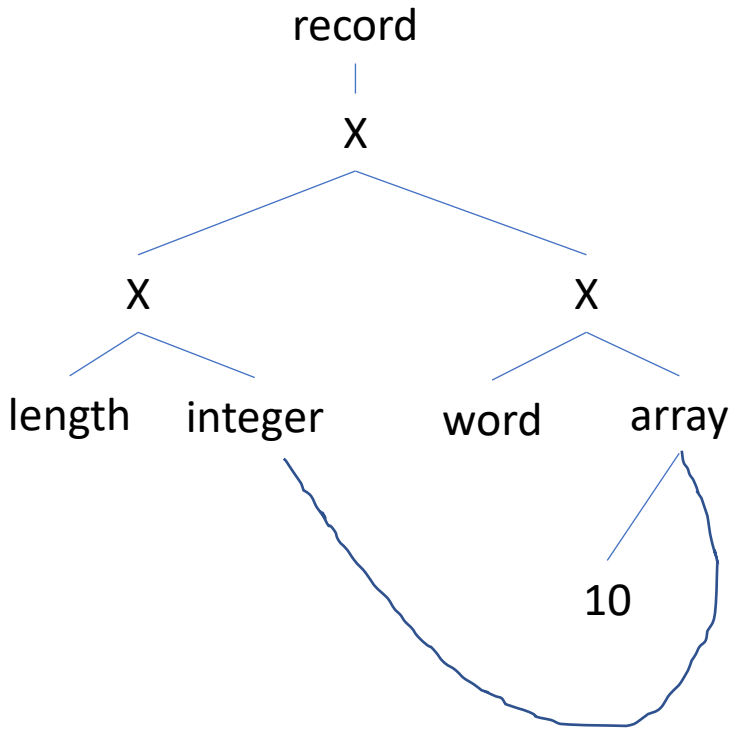
- Checks the structure of the type
- Determines equivalence by checking whether they have same constructor applied to structurally equivalent types
- Checked recursively
- Types array(I1,T1) and array(I2,T2) are structurally equivalent if I1 and I2 are equal and T1 and T2 are structurally equivalent

Value var1,var2	Value and Total are structurally equivalent
Total var3, var4	
X:array(100,int)	X and Y are not structurally equivalent
Y:array(5,int)	

Directed Acyclic Graph representation

- Type Expressions can be represented in DGA or tree
- “record((length x integer) x (word x array(10,character)))”





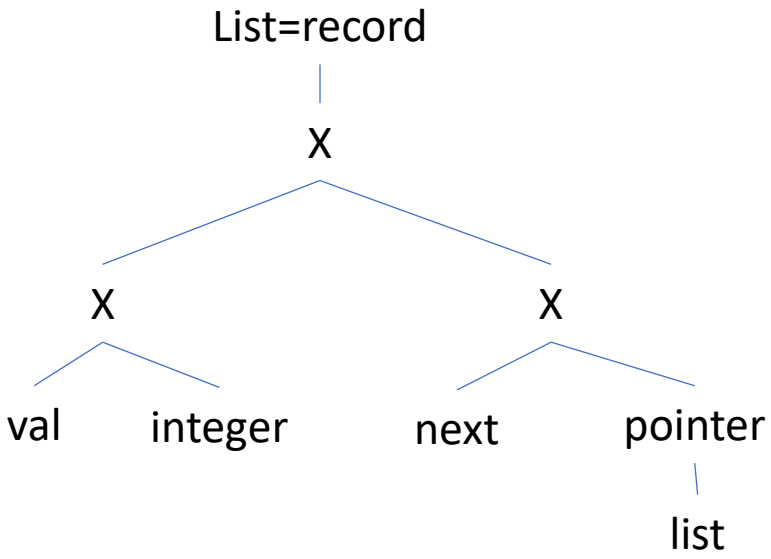
DAG equivalence

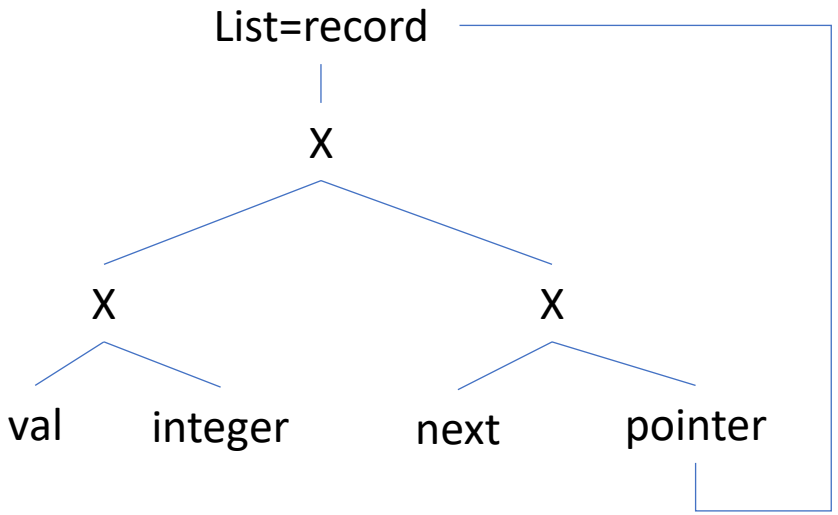
```
function dag-equivalence(s,t: type-DAGs): Boolean
begin
      if s and t represents the same basic type then return true
      if s represents array(I1,T1) and t represents array(I2,T2) then
            if I1=I2 then return dag-equivalence(T1,T2)
            else return false
      if s represents s1 x s2 and t represents t1 x t2 then
            return dag-equivalence(s1,t1) and dag-equivalence(s2,t2)
      if s represents pointer(s1) and t represents pointer(t1) then
             return dag-equivalence(s1,t1)
      ifs=s1→s2 and t=t1→t2 then
             return dag-equivalence(s1,t1) and dag-equivalence(s2,t2)
      return false
end
```

Cycles in Type representation

- Some language allows types to be defined in cyclical fashion

struct list{int no; struct list* next;}





Cycles in Type representation

- Most programming languages use acyclic one
- Type names are to be declared before using it, except pointer
- Name of the structure is also part of type
- Equivalence test stops when a structure is reached
- At this point, type expressions are equivalent if they point to the same structure name, non equivalent otherwise

Type Conversion

- Refers to local modification of type for a variable or subexpression
- For example, it may be necessary to add an integer quantity to a real variable, however, the language may require both the operands to be of same type
- Modifying integer variable to real will require more space
- Solution: to treat integer operand as real operand locally and perform the operation
- May be done implicitly or explicitly
- Implicit Conversion→Type coercion

Summary

- Compiler usually perform static type checking
- Dynamic type checking is costly
- Types are normally represented as type expressions
- Type checking can be performed by syntax directed techniques
- Type graphs may be compared to check type equivalence