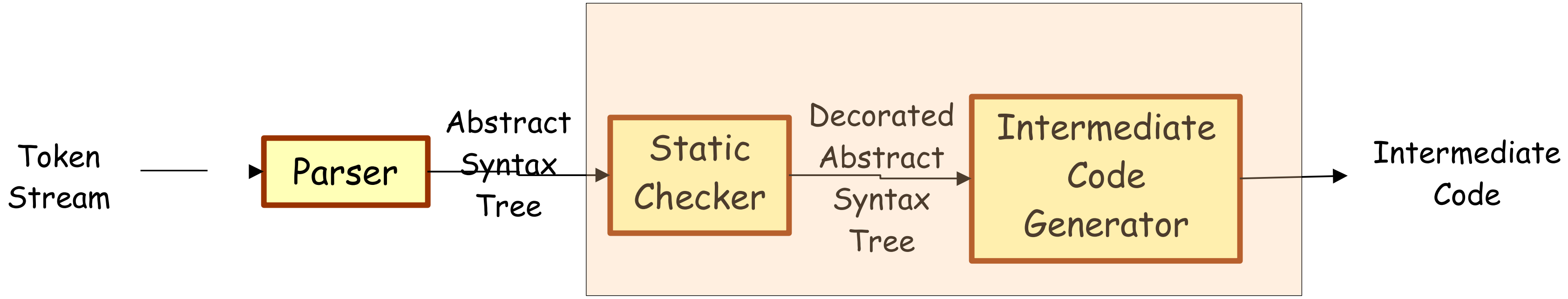
Type checking 6th sem cse

Type Checking

Static Checking



- Static (Semantic) Checks
  - Type checks: operator applied to incompatible operands?
  - Flow of control checks: break (outside while?)
  - Uniqueness checks: labels in case statements
  - Name related checks: same name?

Type Checking

- Problem: Verify that a type of a construct matches that expected by its context.
- Examples:
  - mod requires integer operands (PASCAL)
  - * (dereferencing) – applied to a pointer
  - a[i] – indexing applied to an array
  - f(a1, a2, …, an) – function applied to correct arguments.
- Information gathered by a type checker:
  - Needed during code generation.

Type Systems

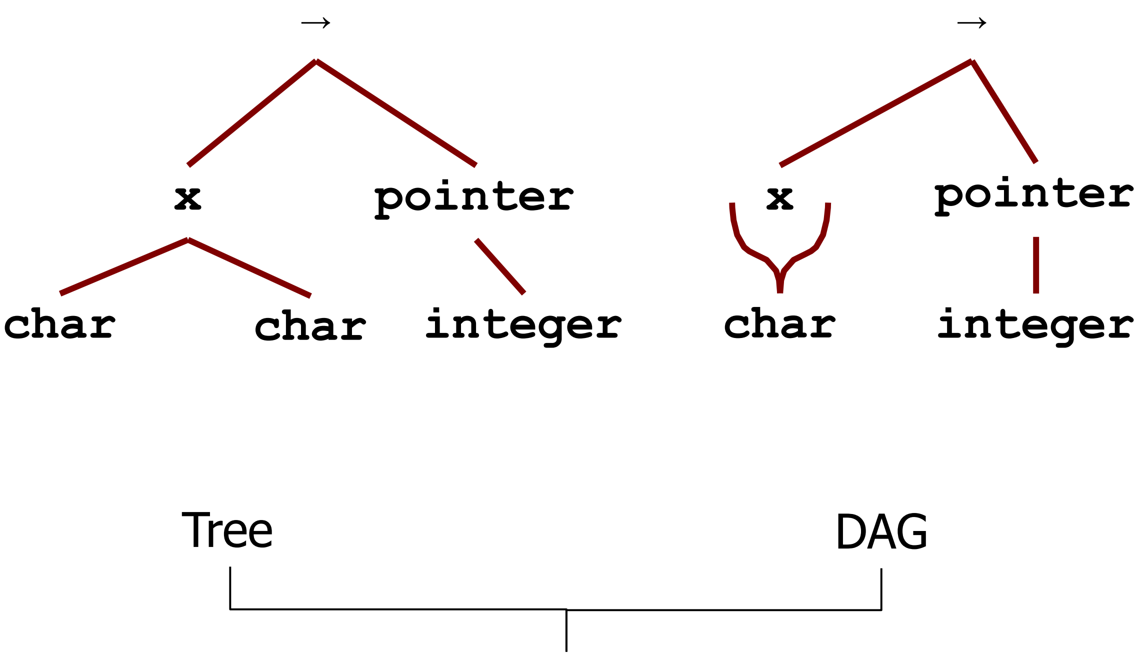
- A collection of rules for assigning type expressions to the various parts of a program.
- Based on: Syntactic constructs, notion of a type.
- Example: If both operators of "+", "-", "*" are of type integer then so is the result.
- Type Checker: An implementation of a type system.
  - Syntax Directed.
- Sound Type System: eliminates the need for checking type errors during run time.

Type Expressions

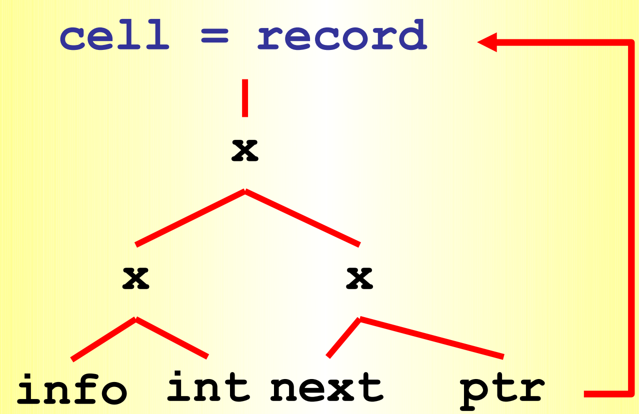
- Implicit Assumptions:
  - Each program has a type → Expressions, Statements
  - Types have a structure

Basic Types		Type Constructors	
Boolean	Character	Arrays	(strings)
Real	integer	Records	
Enumerations	Sub-ranges	Sets	
Void	Error	Pointers	
Variables	Names	Functions	

Representation of Type Expressions



(char x char)→ pointer (integer)




```
struct cell {
        int info;
        struct cell * next;
};
```

Type Expressions Grammar

Type → int | float | char | …
| void
| error
| name
| variable
| array(size, Type)
| record((name, Type)*)
| pointer(Type)
| tuple((Type)*)
| fcn(Type, Type) (Type →Type)

Basic Types: int, float, char, …, void, error, name, variable

Structured Types: array, record, pointer, tuple, fcn

A Simple Typed Language

Program →Declaration; Statement

Declaration →Declaration; Declaration

| id: Type

Statement →Statement; Statement

| id := Expression

| <u>if</u> Expression <u>then</u> Statement

| <u>while</u> Expression <u>do</u> Statement

Expression →literal | num | id

| Expression <u>mod</u> Expression

| E[E] | E ↑ | E (E)

Type Checking Expressions

$E \rightarrow int\_const$ { E.type = int }

$E \rightarrow float\_const$ { E.type = float }

$E \rightarrow id$ { E.type = sym_lookup(id.entry, type) }

$E \rightarrow E_1 + E_2$ {E.type = if E_1.type $\notin$ {int, float} | E_2.type $\notin$ {int, float})

then error

else if E_1.type == E_2.type == int

then int

else float }

Type Checking Expressions

$E \rightarrow E_1 [E_2]$ {E.type = if E_1.type = array(S, T) $\wedge$ E_2.type = int then T else error}

$E \rightarrow *E_1$ {E.type = if E_1.type = pointer(T) then T else error}

$E \rightarrow \& E_1$ {E.type = pointer(E_1.type)}

$E \rightarrow E_1(E_2)$ {E.type = if (E_1.type = fcn(S, T) $\wedge$ E_2.type = S, then T else error}

$E \rightarrow (E_1, E_2)$ {E.type = tuple(E_1.type, E_2.type)}

Type Checking Statements

$S \rightarrow id := E$ {S.type := <u>if</u> id.type = E.type <u>then</u> void <u>else</u> error}

$S \rightarrow if\ E\ then\ S_1$ {S.type := <u>if</u> E.type = boolean <u>then</u> S1.type <u>else</u> error}

$S \rightarrow while\ E\ do\ S_1$ {S.type := <u>if</u> E.type = boolean <u>then</u> S_1.type}

$S \rightarrow S_1;\ S_2$ {S.type := <u>if</u> S_1.type = void ∧ S_2.type = void <u>then</u> void <u>else</u> error}

Equivalence of Type Expressions

Problem: When in E_1.type = E_2.type?

- We need a precise definition for type equivalence
- Interaction between type equivalence and type representation

Example: type vector = array [1..10] of real type weight = array [1..10] of real var x, y: vector; z: weight

Name Equivalence: When they have the same name.

- x, y have the same type; z has a different type.

Structural Equivalence: When they have the same structure.

- x, y, z have the same type.

Structural Equivalence

- Definition: by Induction
  - Same basic type (basis)
  - Same constructor applied to SE Type (induction step)
  - Same DAG Representation
- In Practice: modifications are needed
  - Do not include array bounds – when they are passed as parameters
  - Other applied representations (More compact)
- Can be applied to: Tree/ DAG
  - Does not check for cycles
  - Later improve it.

Algorithm Testing Structural Equivalence

```
function sequiv(s, t): boolean
{   if (s ∧t are of the same basic type) return true;
     if (s = array(s1, s2) ∧t = array(t1, t2))
           return sequiv(s1, t1) ∧sequiv(s2,t2);
     if (s = tuple(s1, s2) ∧t = tuple(t1, t2))
           return sequiv(s1, t1) ∧sequiv(s2, t2);
     if (s = fcn(s1, s2) ∧t = fcn(t1, t2))
           return sequiv(s1, t1) ∧sequiv(s2,t2);
     if (s = pointer(s1) ∧t = pointer(t1))
           return sequiv(s1,t1);
}
```

Recursive Types

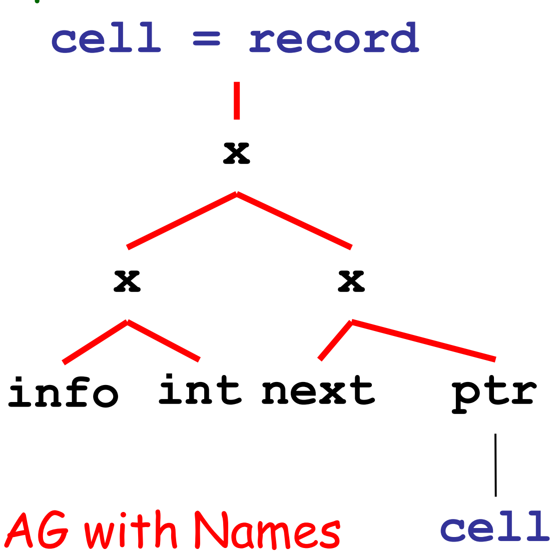
Where: Linked Lists, Trees, etc.

How: records containing pointers to similar records

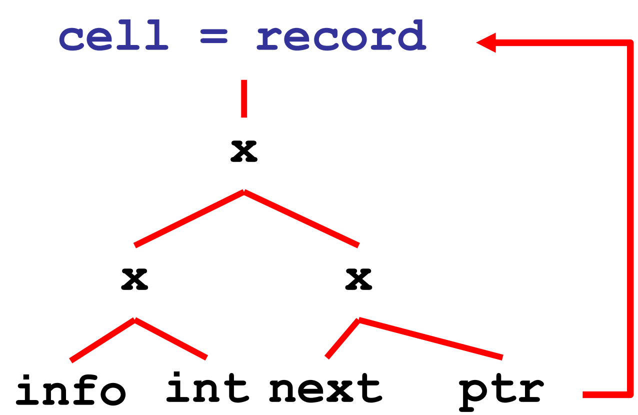
Example: type link = ↑ cell;

cell = record info: int; next = link end

Representation:



DAG with Names



Substituting names out (cycles)

Recursive Types in C

- C Policy: avoid cycles in type graphs by:
  - Using structural equivalence for all types
  - Except for records →name equivalence
- Example:
  - `struct cell {int info; struct cell * next;}`
- Name use: name cell becomes part of the type of the record.
  - Use the acyclic representation
  - Names declared before use – except for pointers to records.
  - Cycles – potential due to pointers in records
  - Testing for structural equivalence stops when a record constructor is reached ~ same named record type?

Overloading Functions & Operators

- Overloaded Symbol: one that has different meanings depending on its context
- Example: Addition operator +
- Resolving (operator identification): overloading is resolved when a unique meaning is determined.
- Context: it is not always possible to resolve overloading by looking only the arguments of a function
  - Set of possible types
  - Context (inherited attribute) necessary

Overloading Example

function "*" (i, j: integer) **return** complex;

function "*" (x, y: complex) **return** complex;

* Has the following types:

  fcn(tuple(integer, integer), integer)

  fcn(tuple(integer, integer), complex)

  fcn(tuple(complex, complex), complex)

int i, j;

k = i * j;