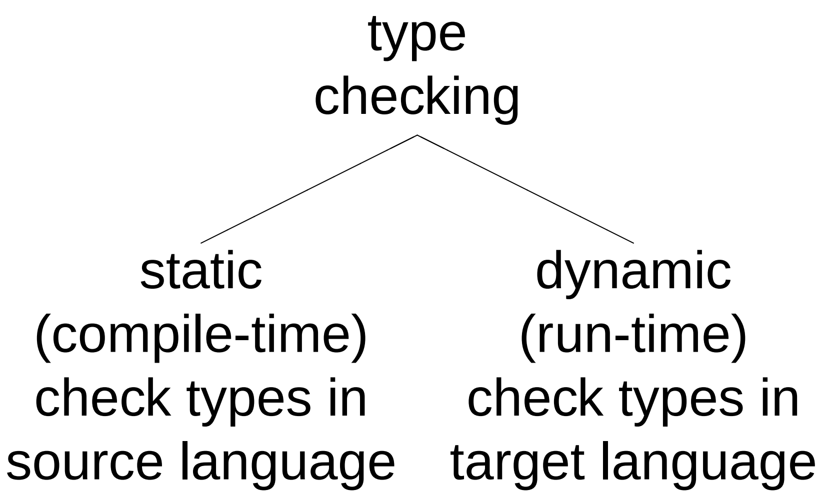
- Semantics checks:

  Type consistency and/or equivalence

  Flow of control (e.g., break must be in a loop)

  Uniqueness checks (e.g., declare once)

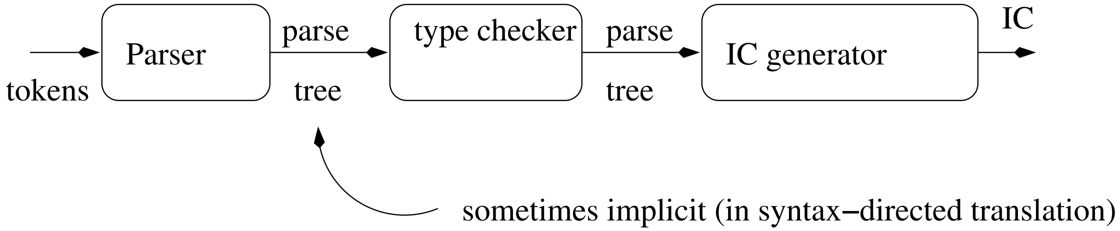
  Declaration checks (e.g., declare before use)



type checking

static (compile-time) check types in source language

dynamic (run-time) check types in target language



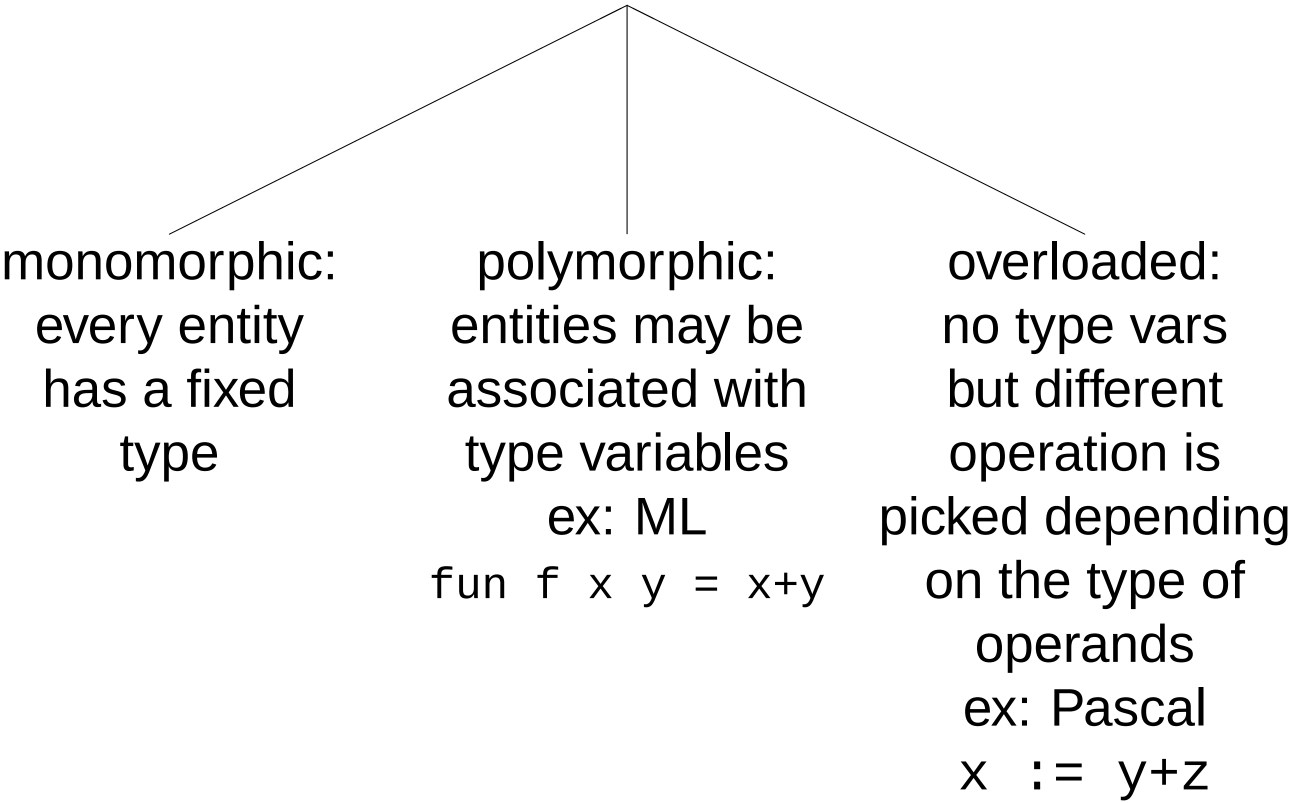
Type systems for programming languages

weakly-typed:
type checks do not
affect well-formedness
of programs
(need run-time support)
ex: C

strongly-typed:
language defines what
types can be combined.
Affects well-formedness
of programs
ex: Pascal, Ada

simple types
only atomic types

complex types
more types can be
built by
type constuctors
ex: array, list



monomorphic:
every entity
has a fixed
type

polymorphic:
entities may be
associated with
type variables
ex: ML

`fun f x y = x+y`

overloaded:
no type vars
but different
operation is
picked depending
on the type of
operands
ex: Pascal

`x := y+z`

- TYPE EXPRESSIONS:

basic: char, boolean..

named type: zoint= boolean

type constructor: a function of $type \rightarrow type$

- *arrays*: $I \times T \rightarrow arr(I,T)$
- *products*: if T_1 and T_2 are types, so is $T_1 \times T_2$
- ML ex: `val(a,b)=(0,2.5);`
- C ex: `struct {char c1; int c2}`
- *records*: named products
- *pointers*: if T_1 is a type, so is $p(T_1)$
- *functions*: if T_1 and T_2 are types,

$f(T_1)$: T_2 is of type $T_1 \rightarrow T_2$

type variables: if x is a type,

$type\_expr(x)$ is also a type

- TYPES in the symbol table

```
struct symtab
{   char *name;
    int type;
    int blockno;
    int offset;
};

in YACC

func : FUNC var ’(’ pars ’)’ body
             {struct symtab *s = search($2);
              s->type=T_FUNC;
              ..
             };

Type attribute in YACC

%union {int etype;
```

```
        int val;}
```

OR for lex/yacc common def:

```
typedef union {int etype; int val;} YYSTYPE;

a: a ’+’ b  {$$.etype=T_INT};
```

ex: type-checking of simple expressions

```
E -> E + E  {E.type= if E1.type=E2.type then E1.type
             else error}

E -> E mod E {E.type= if E2.type=int and E1.type=int
                then E1.type else error}

E -> NUM     {E.type=NUM.type}

E -> (E)     {E.type=E1.type}

E -> F(E)    {E.type=if E1.type=t1 and F.type=t1->t2
                then t2 else error}
```

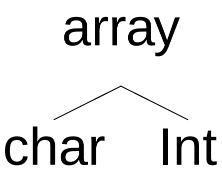
If the language allows complex types, a simple type attribute in the symbol table is not enough

Type trees.

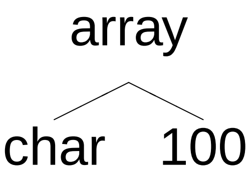
root: type constructor

children: argument types

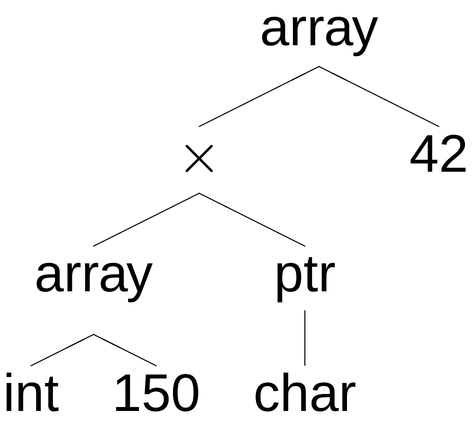
```
char x[100]
```



or

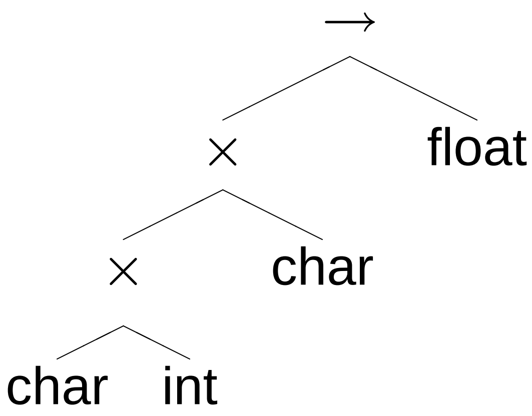


```
struct {int i[150];
        char *c;
       } y[42];
```



For a function

```
float f(x,y,z)
  char x,z;
  int y;
  {....}
```



- The symbol table must reflect the structure of the type

- Complex types constructed with type constructors require some notion of *type equivalence*:

  named equiv.: treat named types as simple types; just check that the types have the same name

  ```
  var a,b:  array[1..10] of char;
  c:  array[1..10] of char;
  ```

  structural equiv.: replace named types with their definitions and recursively check the type trees

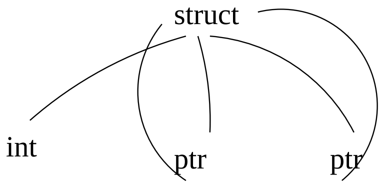
  ```
  typedef int[100] list;
  typedef int[100] vector;
  ```

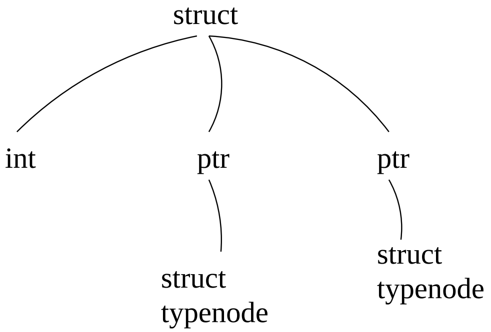
```
..

list a;   vector b;
```

What about recursive types?

```
struct typenode
  { int comp0;
   struct typenode *comp1;
   struct typenode *comp2;
  } ;
```





either mark the traversal of type tree or use structural equiv. everywhere except recursive types (C’s solution)

Some type check problems

- Declare variables before use (what about type info?)

  lex analyzer inserts IDs into symbol table with type `T-UNDEF`

  In *use* of a var, if type is not known, issue error

  example in Yacc

```
e :  e '+' e { ...  };
       | t { ...  };
t :  ID { struct symtab *s=search-sym($1);
```

```
      if (s->type != T-VAR || s->blockno != currblock)
error(..)};
```

- Uniqueness check (unique types)

```
decl :  vars ’:’  type { ...  }

vars :  vars ’,’ id { check if same ID with current block
no is in ST};
```

- What if procedure nesting is allowed (finite or indefinite length)?

- explicit type conversion : language provides type constructors for type casting

  ```
  float a; int i;

  a = (float) i;
  ```

- implicit type conversion: type checker must perform type coercion

- overloading: syntactically same operator denotes different operations semantically

  ```
  e :  e '+' e { either integer/real addition or set union
  in Pascal};
  ```

- Type polymorphism: use type variables and type inference