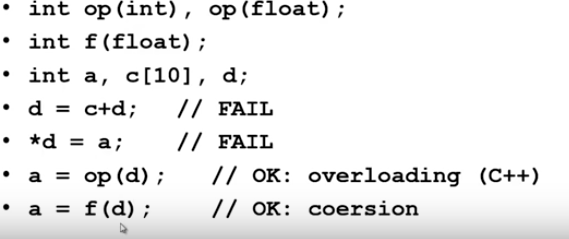
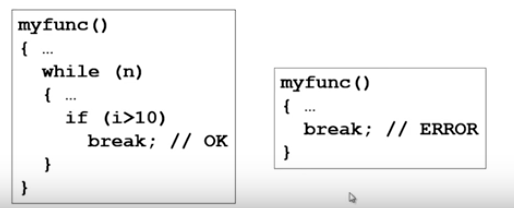
**Type Checking**

**Introduction**:-  
  
The compiler must perform static checking (checking done at compiler time).This ensures that certain types of programming errors will be detected and reported.  Example of static checks include.

 **Type** **checks**:- A compiler should report an error if an operator is applied to an incompatible operand.  
 **Flow-of-control checks:-** Statements that cause flow of control to leave a construct must have some place to which to transfer flow of control. For example, branching to non-existent labels.  
 **Uniqueness** **checks: -** Objects should be defined only once. This is true in many languages.  
 **Name-related checks: -** Sometimes, the same name must appear two or more times. For example, in Ada the name of a block must appear both at the beginning of the block and at the end.

Type information gathered by a type checker may be needed when code is generated. For example, arithmetic operators may be different at the machine level for different types of operands (real and integer).



****

**Type system** is a collection or rules for assigning type expression to the various parts of a program. Type checker implements type system

The type analysis and type checking is an important activity done in the semantic analysis phase. The need for type checking is-

-To detect the errors arising in the expression due to incompatible operand.

-To generate intermediate code for expressions and statements. Typically language supports two types of data types- basic and constructed.

Type Expression:- Type of a language construct. It is either a **basic type** or is formed by applying an operator called a **type constructor** to other type expressions.

The **basic data** type are- integer, character, and real, Boolean, enumerated data type. And Arrays, record (structure),set and pointer are the **constructed types**. The constructed data types are build using basic data types.

Type constructor

* *Arrays*. If *T* is a type expression and *I* is the type expression of an index set then *array* (*I*, *T* ) denotes an array of elements of type *T*.
* *Products*. If *T*1 and *T*2 are type expressions, then their Cartesian product, *T*1 x *T*2, is a type expression.

For example if the arguments of a function are two reals followed by an

integer then the type expression for the arguments is: **real** x **real** x **integer**.

* *Records*. The fields in a record (or structure) have names which should be included in the type expression of the record. The type expression of a record with *n* fields is:

*record* (*F*1 x *F*2 x ... x *F*n )

where if the name of field *i* is namei and the type expression of field *i* is *T*i

then *F*i is: (namei x *T*i ).

* Pointers. If T is a type expression then pointer (T ) denotes a pointer to an object of type T.
* Functions. A function maps elements from its domain to its range. The type expression for a function is: D --> R where D is the type expression for the domain of the function and R is the type expression for the range of the function. For example, the type expression of the mod operator in Pascal is: integer x integer --> integer because it divides an integer by an integer and returns the integer remainder.
* The type expression for the domain of a function with no arguments is void and the type expression for the range of a function with no returned value is void: e.g., void --> void is the type expression for a procedure with no arguments and no returned value

Here we use type expressions formed from the following rules:

* A basic type is a type expression. Other basic type expressions are **type-error** to signal the presence of a type error and **void** to signal the absence of a value.

If a type expression has a name then the name is also a type expression.

A convenient way to represent the type expression is to use graph. We can construct a tree or DAG for type expression, with interior node for type constructors and leaves for basic type, type name and type variables.

Example function f (a,b:char): ↑ integer

Say the domain type of f is denote by char x char and rang type by pointer(integer)

char x char → pointer(integer)

A source type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs.

**Rules for Type Checking**

Type checking can take on two forms: **synthesis** and **inference**.

-Type synthesis builds up the type of an expression from the types of its subexpressions. It requires names to be declared before they are used.

-Type inference determines the type of a language construct from the way it

is used. Looking ahead to the following examples

fun length(x) =

if null(x) then 0 else length(tl(x)) + 1;

, let null be a function that tests whether a list is empty. Then, from the usage null(x), we can tell that x must be a list. The type of the elements of x is not known; all we know is that x must be a list of elements of some type that is presently unknown.

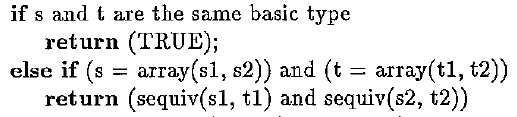
**Equivalence of type expression**

Two type expressions are **structural equivalence** either they have the same basic type or they are formed by applying the same constructor to the structurally equivalent types. For example the type expression integer is equivalent only to integer because they are the same basic type. Similarly, pointer (integer) is equivalent only to pointer (integer) because they are formed by applying the same constructor pointer to the equivalent type.

For example look at the following assignment statement

x=y

here the value of object y is copied into the memory locations for variable x. However, before an operation such as an assignment can be accepted by the translator, usually the types of the two operands must be the same (or perhaps compatible in some other specified way). Thus a language translator must decide whether two types are equal in some cases.



**Type Conversions**

Consider expressions like x + i, where x is of type float and i is of type integer.

Since the representation of integers and floating-point numbers is different

within a computer and different machine instructions are used for operations

on integers and floats, the compiler may need to convert one of the operands of + to ensure that both operands are of the same type when the addition occurs.

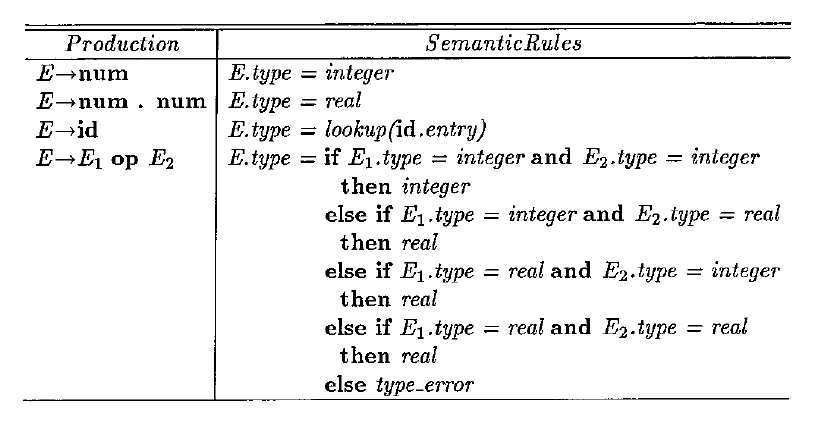
Suppose that integers are converted to floats when necessary, using a unary

operator ( float ) . For example, the integer 2 is converted to a float in the code

for the expression 2 \* **3** .14:

tl = (float) 2

t2 = tl \* 3.14



Type conversion rules vary from language to language. The rules for Java

in Fig.1 distinguish between widening conversions, which are intended to

preserve information, and narrowing conversions, which can lose information.

The widening rules are given by the hierarchy in Fig. 2 (a): any type lower

in the hierarchy can be widened to a higher type. Thus, a char can be widened

to an int or to a float, but a char cannot be widened to a short. The narrowing

rules are illustrated by the graph in Fig. 2 (b): a type s can be narrowed to a

type t if there is a path from s to t. Note that char, short, and byte are pairwise

convertible to each other.

Conversion from one type to another is said to be implicit if it is done

automatically by the compiler. Implicit type conversions, also called coercions

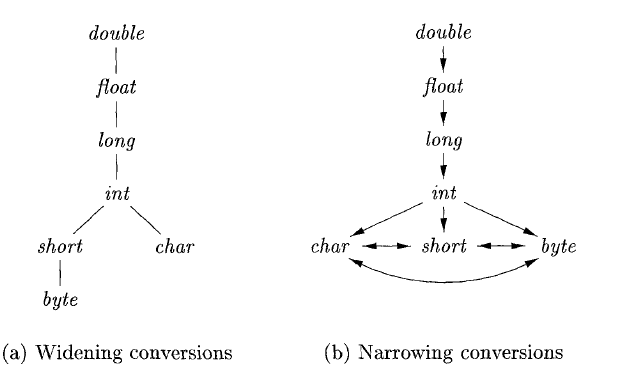


Figure 2

The semantic action for checking E 🡪 El + *E2* uses two functions:

1. max(t1, t2) takes two types t1 and t2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t1 or t2 is not in the hierarchy; e.g., if either type is an array or a pointer type.

2. widen(a, t, w) generates type conversions if needed to wide conversion and place the result in a temporary t, which is returned as the result. Pseudocode for widen, assuming that the only types are .integer and float, appears in Fig.3.

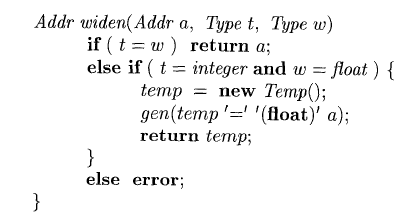
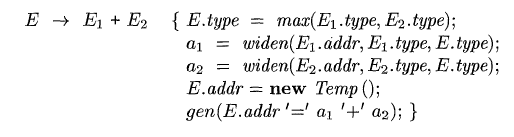


Figure 3



**Overloading of Functions and Operators**

Overloading is a feature in programming languages that allows creating several methods that have the same name but differ from each other in terms of type of input and output.

An overloaded symbol has different meanings depending on its context. Overloading is resolved when a unique meaning is determined for each occurrence

of a name. We restrict attention to overloading that can be resolved by looking only at the arguments of a function, as in Java.

Example : The + operator in Java denotes either string concatenation or addition, depending on the types of its operands.

User-defined functions can be overloaded as well, as in

void err() {…..}

void err(String s) {…..}

Note that we can choose between these two versions of a function err by looking

at their arguments.

**Polymorphic Functions**

The term "polymorphic" refers to any code fragment that can be executed with arguments of different types. In this section, we consider parametric polymorphism, where the polymorphism is characterized by parameters or type

variables. The running example is the ML program below, which defines

a function length. The type of length can be described as, "for any type a,

length maps a list of elements of type a to an integer."

fun length(x) =

if null(x) then 0 else length(tl(x)) + 1;

The function length determines the length or number of elements of a list

x. All elements of a list must have the same type, but length can be applied to

lists whose elements are of any one type. In the following expression, length is

applied to two different types of lists (list elements are enclosed within "[" and

“]”):



The list of strings has length 3 and the list of integers has length 4, so the above expression evaluates to 7.

Standard type checking

Int f(intx) { return x+1; };

Int g(inty) { return f(y+1)\*2;};

•Look at body of each function and use declared types of identifies to check agreement.

Type inference

f( x) { return x+1; };

g( y) { return f(y+1)\*2;};

•Look at code without type information and figure out what types could have been declared

Type checking: The process of checking whether the types declared by the programmer “agrees”with the language constraints/ requirement

Type inference: The process of determining the type of an expression based on information given by (some of) its symbols/sub-expre