Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

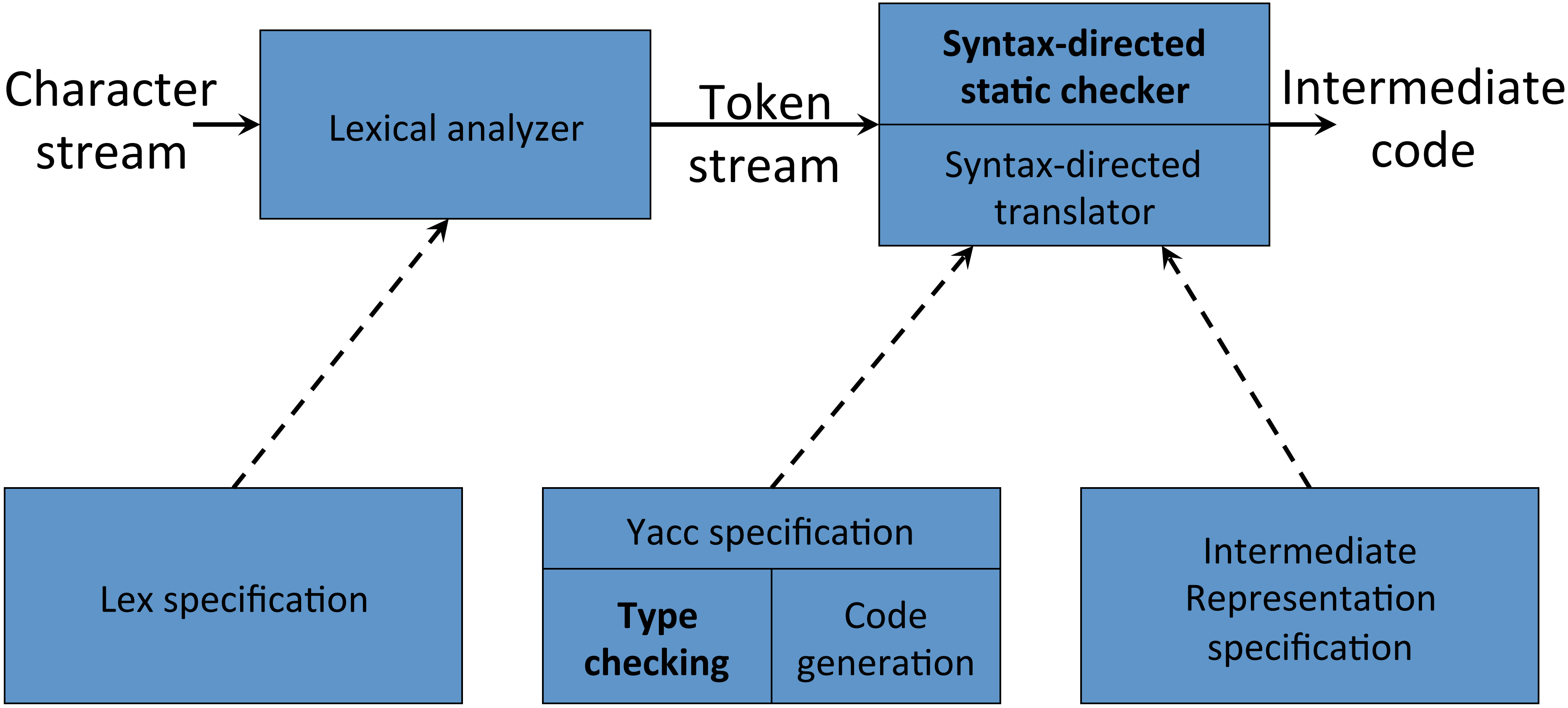
Lesson 14

- Static versus Dynamic Checking
- Type checking
- Type conversion and coercion

Recap (last lecture)

- Intermediate code generation for:
  - Multi-dimensional arrays
    - Translation scheme for computing type and width
    - Generation of three address statement for addressing array elements
  - Translating logical and relational expressions
  - Translating short-circuit Boolean expressions and flow-of-control statements with backpatching lists

Revisited Structure of Typical Compiler Front End



Static versus Dynamic Checking

- *Static checking*: the compiler enforces programming language's *static semantics*
  - Program properties that can be checked at compile time
  - Usually not expressible with CFG
- *Dynamic semantics*: checked at run time
  - Compiler generates verification code to enforce programming language's dynamic semantics

Static Checking

- Typical examples of static checking are
  - Type checks
  - Flow-of-control checks
  - Uniqueness checks
  - Name-related checks

Type Checking, Overloading, Coercion, Polymorphism

```
class X { virtual int m(); } *x;
int op(int), op(float);
int f(float);
int a, c[10], d;

d = c + d;        // FAIL
*d = a;        // FAIL
a = op(d);        // OK: static overloading (C++)
a = f(d);         // OK: coercion of d to float
a = x->m();       // OK: dynamic binding (C++)
vector<int> v; // OK: template instantiation
```

Flow-of-Control Checks

```
myfunc()
{ …
  break; // ERROR
}
```

```
myfunc()
{ …
  while (n)
  { …
    if (i>10)
      break; // OK
  }
}
```

```
myfunc()
{ …
  switch (a)
  { case 0:
      …
      break; // OK
    case 1:
      …
  }
}
```

Uniqueness Checks

```
myfunc()
{ int i, j, i; // ERROR
   …
}
```

```
cnufym(int a, int a) // ERROR
{    …
}
```

```
struct myrec
{ int name;
};
struct myrec // ERROR
{ int id;
};
```

Name-Related Checks

```
LoopA: for (int i = 0; i < n; i++)
       { …
         if (a[i] == 0)
           break LoopB; // Java labeled break
         …
       }
```

One-Pass versus Multi-Pass Static Checking

- *One-pass compiler*: static checking in C, Pascal, Fortran, and many other languages is performed in one pass while intermediate code is generated
  - **Influences design of a language**: placement constraints
    - Declarations
    - Function prototypes
- *Multi-pass compiler*: static checking in Ada, Java, and C# is performed in a separate phase, sometimes by traversing a syntax tree multiple times

Towards Type Checking: Type Expressions

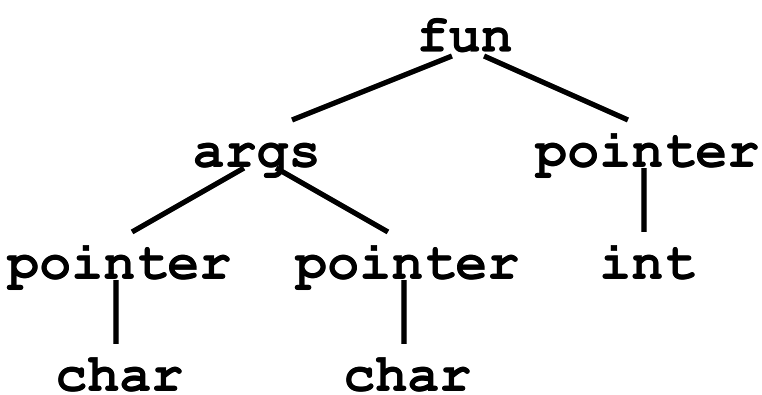
- *Type expressions* are used in declarations and type casts to define or refer to a type

Type ::= **int** | **bool** | … | X | Tname |pointer-to(Type) |
array(*num*, Type) | record(Fields) | class(…) |
Type → Type | Type x Type

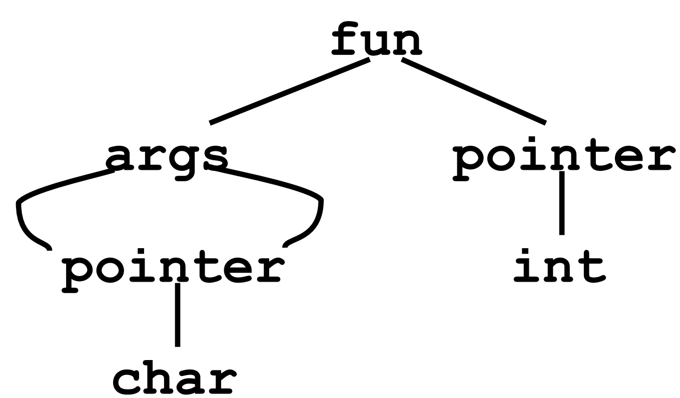
  - *Primitive types*, such as **int** and **bool**
  - *Type constructors*, such as pointer-to, array-of, records and classes, and functions
  - *Type names*, such as typedefs in C and named types in Pascal, refer to type expressions

Graph Representations for Type Expressions

- Internal compiler representation, built during parsing
- Example: `int *f(char*,char*)`



Tree forms

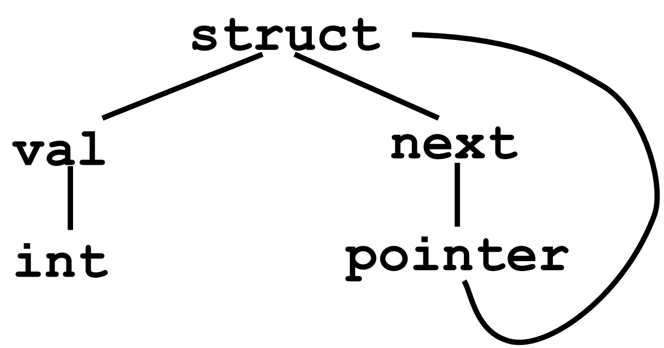


DAGs

Cyclic Graph Representations

Source program

```
struct Node
{ int val;
  struct Node *next;
};
```



Internal compiler representation
of the **Node** type: cyclic graph

Equivalence of Type Expressions

- Important for type checking, e.g. in assignments
- Two different notions: **name equivalence** and **structural equivalence**
  - Two types are ***structurally equivalent*** if
    1. They are the same basic types, or
    2. They have the form **TC(T1,…, Tn)** and **TC(S1, …, Sn)**, where **TC** is a type constructor and **Ti** is structurally equivalent to **Si** for all 1 <= i <= n, or
    3. One is a type name that denotes the other.
  - Two types are ***name equivalent*** if they satisfy 1. and 2.

On Name Equivalence

- Each *type name* is a distinct type, even when the type expressions that the names refer to are the same
- Types are identical only if names match
- Used by Pascal (inconsistently)

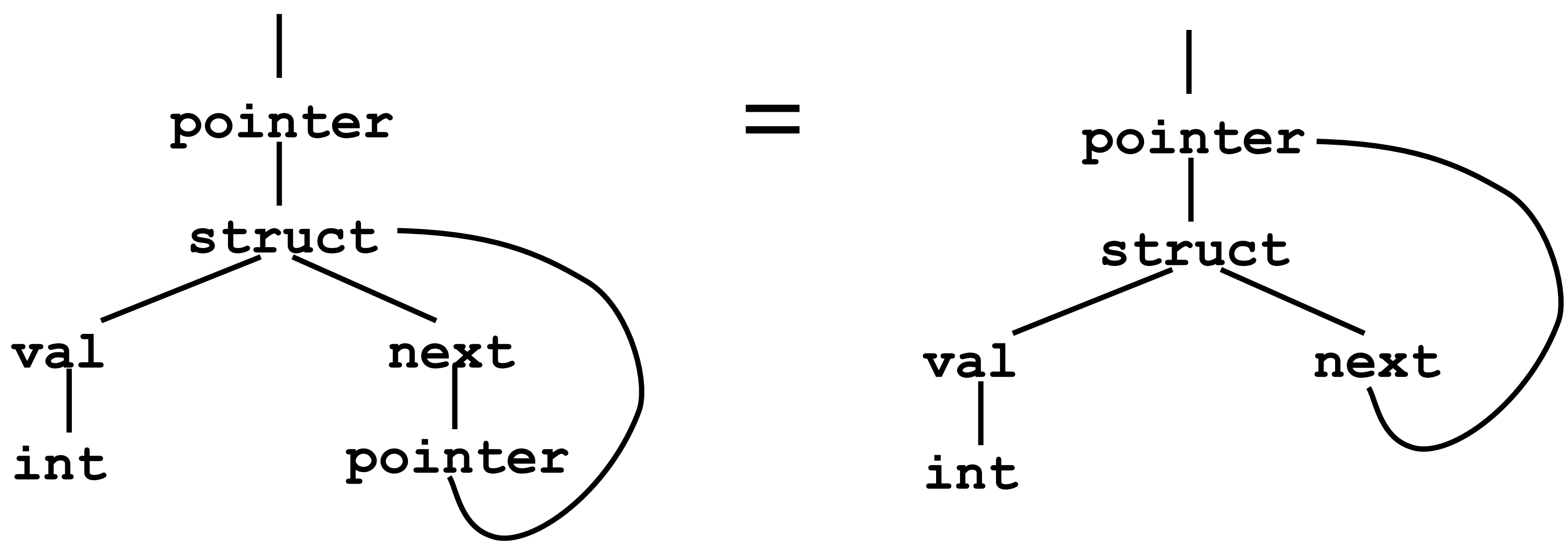
```
type link = ^node;
var next : link;
    last : link;
       p : ^node;
    q, r : ^node;
```

With name equivalence in Pascal:

```
p := next       FAIL
last := p       FAIL
q := r          OK
next := last    OK
p := q          FAIL !!!
```

Structural Equivalence of Type Expressions

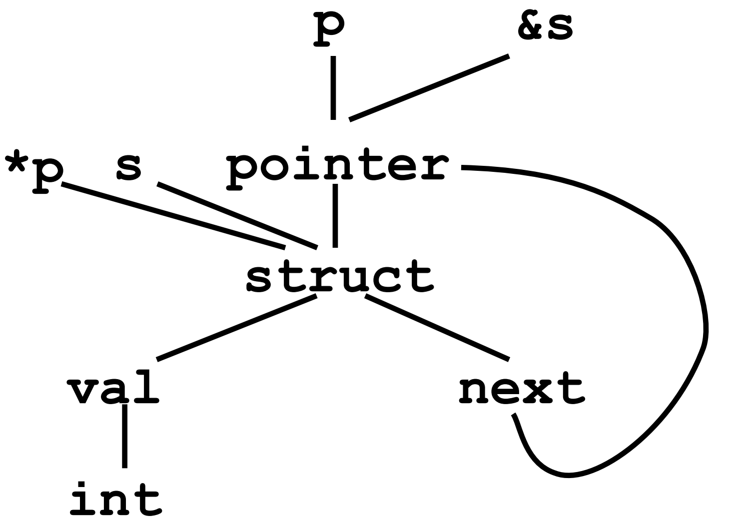
- Two types are the same if they are *structurally identical*
- Used in C/C++, Java, C#



Structural Equivalence of Type Expressions (cont'd)

- Two structurally equivalent type expressions have the same pointer address when constructing graphs by (maximally) sharing nodes

```
struct Node
{ int val;
  struct Node *next;
};
struct Node s, *p;
p = &s; // OK
*p = s; // OK
p = s;  // ERROR
```



Type Systems

- A *type system* defines a set of types and rules to assign types to programming language constructs
- Informal type system rules, for example “*if both operands of addition are of type integer, then the result is of type integer*”
- Formal type system rules: **Post systems**

Type Rules in Post System Notation

$$\frac{\rho(v) = \tau}{\rho \vdash v : \tau}$$

$$\frac{\rho(v) = \tau \qquad \rho \vdash e : \tau}{\rho \vdash v \mathbin{\mathbf{:=}} e : \mathit{void}}$$

$$\frac{\rho \vdash e_1 : \mathit{integer} \qquad \rho \vdash e_2 : \mathit{integer}}{\rho \vdash e_1 \mathbin{\mathbf{+}} e_2 : \mathit{integer}}$$

Type judgments

$e : \tau$

where e is an expression and τ is a type

Environment ρ maps objects v to types τ:

$\rho(v) = \tau$

Type System Example

Environment ρ is a set of $\langle \mathit{name}, \mathit{type} \rangle$ pairs, for example:

$\rho = \{ \langle \mathbf{x}, \mathit{integer} \rangle, \langle \mathbf{y}, \mathit{integer} \rangle, \langle \mathbf{z}, \mathit{char} \rangle, \langle 1, \mathit{integer} \rangle, \langle 2, \mathit{integer} \rangle \}$

From ρ and rules we can check the validity of typed expressions:

type checking = theorem proving

The proof that **x := y + 2** is typed correctly:

$$\dfrac{\rho(\mathbf{x}) = \mathit{integer} \qquad \dfrac{\dfrac{\rho(\mathbf{y}) = \mathit{integer}}{\rho \vdash \mathbf{y} : \mathit{integer}} \qquad \dfrac{\rho(\mathbf{2}) = \mathit{integer}}{\rho \vdash \mathbf{2} : \mathit{integer}}}{\rho \vdash \mathbf{y + 2} : \mathit{integer}}}{\rho \vdash \mathbf{x := y + 2} : \mathit{void}}$$

A Simple Language Example

$P \rightarrow D$ **;** S
$D \rightarrow D$ **;** D
  | **id :** T
$T \rightarrow$ **boolean**
  | **char**
  | **integer**
  | **array [num] of** T
  | **^** T
$S \rightarrow$ **id :=** E
  | **if** E **then** S
  | **while** E **do** S
  | S **;** S

Pointer to *T*

$E \rightarrow$ **true**
  | **false**
  | **literal**
  | **num**
  | **id**
  | E **and** E
  | E **+** E
  | E **[** E **]**
  | E **^**

Pascal-like pointer dereference operator

Simple Language Example: Declarations

D → **id :** *T* { *addtype*(**id**.entry, *T*.type) }
T → **boolean** { *T*.type := *boolean* }
T → **char** { *T*.type := *char* }
T → **integer** { *T*.type := *integer* }
T → **array [num] of** T_1 { *T*.type := *array*(1..**num**.val, T_1.type) }
T → **^** T_1 { *T*.type := *pointer*(T_1)

Parametric types:
type constructor

Simple Language Example: Checking Statements

$$\frac{\rho(v) = \tau \qquad \rho \vdash e : \tau}{\rho \vdash v \mathbin{\mathbf{:=}} e : \mathit{void}}$$

S → **id :=** *E* { *S*.type := (**if id**.type = *E*.type **then** *void* **else** *type_error*) }

Note: the type of **id** is determined by scope's environment:
id.type = *lookup*(**id**.entry)

Simple Language Example: Checking Statements (cont'd)

$$\frac{\rho \vdash e : boolean \qquad \rho \vdash s : \tau}{\rho \vdash \textbf{if } e \textbf{ then } s : \tau}$$

$S \rightarrow$ **if** E **then** S_1 { S.type := (**if** E.type = *boolean* **then** S_1.type **else** *type_error*) }

Simple Language Example: Statements (cont’d)

$$\frac{\rho \vdash e : boolean \qquad \rho \vdash s : \tau}{\rho \vdash \textbf{while}\ e\ \textbf{do}\ s : \tau}$$

S → **while** *E* **do** S_1 { *S*.type := (**if** *E*.type = *boolean* **then** S_1.type
else *type_error*) }

Simple Language Example: Checking Statements (cont’d)

$$\frac{\rho \vdash s_1 : \mathit{void} \qquad \rho \vdash s_2 : \mathit{void}}{\rho \vdash s_1 \mathbf{;}\, s_2 : \mathit{void}}$$

$S \rightarrow S_1$ **;** S_2 { S.type := (**if** S_1.type = *void* **and** S_2.type = *void* **then** *void* **else** *type_error*) }

Simple Language Example: Checking Expressions

$$\frac{\rho(v) = \tau}{\rho \vdash v : \tau}$$

$E \rightarrow$ **true**	{ *E*.type = *boolean* }
$E \rightarrow$ **false**	{ *E*.type = *boolean* }
$E \rightarrow$ **literal**	{ *E*.type = *char* }
$E \rightarrow$ **num**	{ *E*.type = *integer* }
$E \rightarrow$ **id**	{ *E*.type = *lookup*(**id**.entry) }

…

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e_1 : integer \qquad \rho \vdash e_2 : integer}{\rho \vdash e_1 \mathbf{+} e_2 : integer}$$

$E \rightarrow E_1$ **+** E_2 { E.type := (**if** E_1.type = *integer* **and** E_2.type = *integer* **then** *integer* **else** *type_error*) }

Simple Language Example: Checking Expressions (cont’d)

$$\frac{\rho \vdash e_1 : boolean \qquad \rho \vdash e_2 : boolean}{\rho \vdash e_1 \textbf{ and } e_2 : boolean}$$

$E \rightarrow E_1$ **and** E_2 { E.type := (**if** E_1.type = *boolean* **and** E_2.type = *boolean* **then** *boolean* **else** *type_error*) }

Simple Language Example: Checking Expressions (cont'd)

$$\frac{\rho \vdash e_1 : array(s, \tau) \qquad \rho \vdash e_2 : integer}{\rho \vdash e_1[e_2] : \tau}$$

$E \rightarrow E_1$ **[** E_2 **]** { E.type := (**if** E_1.type = *array*(*s*, *t*) **and** E_2.type = *integer*
then *t* **else** *type_error*) }

Note: parameter *t* is set with the unification of
E_1.type = *array*(*s*, *t*)

Simple Language Example: Checking Expressions (cont’d)

$$\frac{\rho \vdash e : pointer(\tau)}{\rho \vdash e\,\text{^} : \tau}$$

$E \rightarrow E_1$ **^** { E.type := (**if** E_1.type = *pointer*(t) **then** t
else *type_error*) }

Note: parameter t is set with the unification of
E_1.type = *pointer*(t)

A Simple Language Example: Functions

$T \rightarrow T$ **->** T

Function type declaration

$E \rightarrow E$ **(** E **)**

Function call

Example:

```
v : integer;
odd : integer -> boolean;
if odd(3) then
    v := 1;
```

Simple Language Example: Function Declarations

$T \rightarrow T_1$ **->** T_2 { T.type := *function*(T_1.type, T_2.type) }

Parametric type: type constructor

Simple Language Example: Checking Function Invocations

$$\frac{\rho \vdash e_1 : function(\sigma, \tau) \qquad \rho \vdash e_2 : \sigma}{\rho \vdash e_1(e_2) : \tau}$$

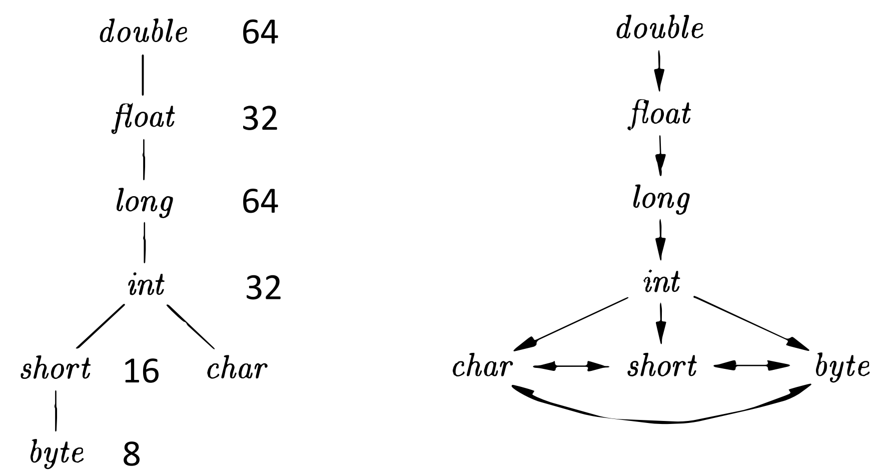
$E \rightarrow E_1$ **(** E_2 **)** { E.type := (**if** E_1.type = *function*(s, t) **and** E_2.type = s **then** t **else** *type_error*) }

Type Conversion and Coercion

- *Type conversion* is explicit, for example using type casts
- *Type coercion* is implicitly performed by the compiler to generate code that converts types of values at runtime (typically to *narrow* or *widen* a type)
- Both require a *type system* to check and infer types from (sub)expressions

Example: Type Coercion and Cast in Java among numerical types

- Coercion (implicit, widening)
  - No loss of information (almost…)
- Cast (explicit, narrowing)
  - Some information can be lost
- Explicit cast is always allowed when coercion is



(a) Widening conversions (b) Narrowing conversions

Handling coercion during translation

Translation of sum without type coercion:

$E \rightarrow E_1 + E_2$ { *E*.place := *newtemp*();
gen(*E*.place ':=' E_1.place '+' E_2.place) }

With type coercion:

$E \rightarrow E_1 + E_2$ { *E*. type = **max**(E_1.type,E_2.type);
a_1 = **widen**(E_1.addr, E_1.type, E.type) ;
a_2 = **widen**(E_2.addr, E_2.type, E.type);
E.addr = new Temp();
gen(E.addr '=' a_1 '+' a_2); }

where:

- **max(T_1,T_2)** returns the least upper bound of T_1 and T_2 in the widening hierarchy
- **widen(addr, T_1, T_2)** generate the statement that copies the value of type T_1 in addr to a new temporary, casting it to T_2

Pseudocode for widen

```
Addr widen(Addr a, Type t, Type w){
    temp = new Temp();
    if(t = w) return a;  //no coercion needed
    elseif(t = integer and w = float){
        gen(temp '=' '(float)' a);
    elseif(t = integer and w = double){
        gen(temp '=' '(double)' a);
    elseif ...
    else error;
    return temp; }
}
```

Type Inference and Polymorphic Functions

- Languages like ML are **strongly typed**, but declaration of type is not mandatory
- Type checking includes a **Type Inference** algorithm to assign types to expressions
- Type Inference is very interesting in presence of **polymorphic types**, which are type expressions with variables

For example, consider the *list length* function in ML:

fun *length*(*x*) = **if** *null*(*x*) **then** 0 **else** *length*(*tl*(*x*)) + 1

  - *length*(["sun", "mon", "tue"]) + *length*([10,9,8,7]) returns 7

- What is its type?