

Compilers

Semantic Analysis

LEIC

FEUP-FCUP

2022

This lecture

Semantic analysis

Scopes

Symbol table

Type Systems

Type Checking

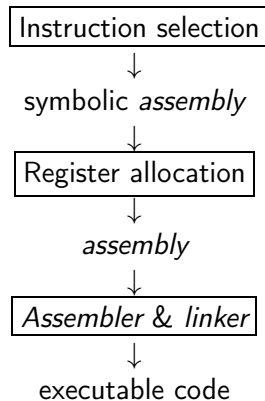
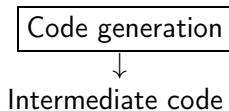
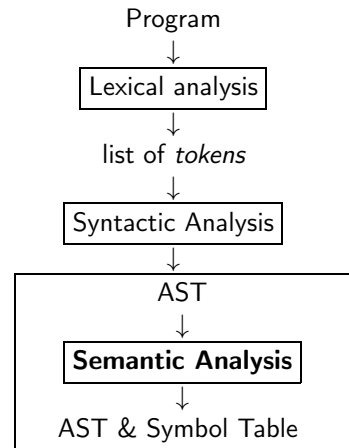
- Expressions

- Type Declarations

- Type Checking for Functions

Extras

Compiler



This lecture

Semantic analysis

Scopes

Symbol table

Type Systems

Type Checking

Expressions

Type Declarations

Type Checking for Functions

Extras

Context dependent static verifications.

Examples:

- ▶ undeclared variables
- ▶ type checking
- ▶ type inference

Semantics analysis (cont.)

- ▶ This can be done while parsing takes place
- ▶ Usually it is done after parsing - AST is the input of semantic analysis
- ▶ Needs information about **identifiers** (variables, function names, etc.) stored in a **symbol table**
- ▶ The symbol table is also used later for code generation

This lecture

Semantic analysis

Scopes

Symbol table

Type Systems

Type Checking

Expressions

Type Declarations

Type Checking for Functions

Extras

Scopes

- ▶ Variable declarations have a **limited scope**
 - ▶ e.g. we may declare a local variable with the same name as a global variable

Example

Scopes.

```
int i, j;  
int f(int size)  
{ char i, temp;  
  ...  
  { double j;  
    ...  
  }  
  ...  
  { char * j;  
    ...  
  }  
}
```

Example

Scopes.

```
int i, j;           // global: i, j
int f(int size)     // global: f
{ char i, temp;
  ...
  { double j;
    ...
  }
  ...
  { char * j;
    ...
  }
}
```

Example

Scopes.

```
int i, j;           // global: i, j
int f(int size)     // global: f
{ char i, temp;     // function: size, i, temp
  ...               // new declaration for i
  { double j;
    ...
  }
  ...
  { char * j;
    ...
  }
}
```

Example

Scopes.

```
int i, j;           // global: i, j
int f(int size)     // global: f
{ char i, temp;     // function: size, i, temp
  ...               // new declaration for i
  { double j;       // block 1: j hides the global declaration
    ...
  }
  ...
  { char * j;
    ...
  }
}
```

Example

Scopes.

```
int i, j;           // global: i, j
int f(int size)     // global: f
{ char i, temp;     // function: size, i, temp
  ...              // new declaration for i
  { double j;       // block 1: j hides the global declaration
    ...
  }
  ...
  { char * j;       // block 2: j hides the global declaration
    ...
  }
}
```

Lexical scope (also known as *static scope*) each use of a variable corresponds to the **closest declaration** in the AST

- ▶ Used in Pascal, C, C++, Java, Haskell, SML, etc.
- ▶ Dynamic scope: first versions of LISP

Function scopes

- ▶ Functions may be used in any scope:
 - ▶ Pascal, Modula e Ada,...
 - ▶ Scheme, ML, Haskell,...
 - ▶ JavaScript, Ruby, Python,...
- ▶ C and Java limit the scope for functions to the global scope (C) or classes (C++/Java)

Function scopes (cont.)

```
// Pascal
function E(x: real): real;
    function F(y: real): real;
    begin
        F := x + y
    end;
begin
    E := F(3) + F(4)
end;
```

```
-- Haskell
e :: Float -> Float
e x = f 3 + f 4
    where f y = x + y
```

```
// GNU C (not standard C)
float e(float x) {
    float f(float y) {
        return x + y;
    }
    return f(3) + f(4);
}
```


Function scopes (cont.)

GCC (extention to the *standard*):

```
float E(float x)
{
    float F(float y)
    {
        return x + y;
    }
    return F(3) + F(4);
}
```

Function scopes (cont.)

Advantages of functions inside functions

- ▶ local auxiliary functions
- ▶ simplifies the control flow
- ▶ enable the use of local auxiliary functions instead of cycles

Function scopes (cont.)

Example: test primes (C vs. Haskell).

```
int is_prime(int n) {  
    int d = 2;  
    if(n<=1)  
        return FALSE;  
    while(d*d <= n) {  
        if (n%d == 0)  
            return FALSE;  
        d++;  
    }  
    return TRUE;  
}
```

```
isPrime :: Int -> Bool  
isPrime n = n>1 && checkDivs 2  
    where checkDivs d  
            | n`mod`d==0 = False  
            | d*d <= n    = checkDivs (d+1)  
            | otherwise  = True
```

Function scopes (cont.)

Non-standard C extension:

```
int is_prime(int n) {  
    int checkDivs() {  
        int d = 2;  
        while(d*d <= n) {  
            if(n%d == 0)  
                return FALSE;  
        }  
        return TRUE;  
    }  
    return n>1 && checkDivs();  
}
```

This lecture

Semantic analysis

Scopes

Symbol table

Type Systems

Type Checking

Expressions

Type Declarations

Type Checking for Functions

Extras

Symbol table

- ▶ Relates **identifiers** with semantic information **informação**; some examples:
 - ▶ *values* of variables (interpreters);
 - ▶ *types*;
 - ▶ *location* (registers or memory during code generation)

Name spaces

- ▶ Multiple name spaces
 - ▶ e.g. in Haskell we may have the same name for a *type* and a *constructor*

```
data Expr = Int Int | Add Expr Expr
```

- ▶ **modules** or *packages* usually have different name spaces
- ▶ We may use **different symbol tables** for different name spaces
- ▶ Or: use a global symbol table and **prefixes**

```
Prelude.lookup
```

```
Data.Map.lookup
```

Symbol table

Functions:

`initialize` an *empty table*;

`insert` a pair *identifier e information*;

`lookup` given an *identifier* returns the *information*

Use *tables* built-in in your programming language. Examples:

- ▶ Map in Java
- ▶ Data.Map in Haskell
- ▶ dict in Python
- ▶ `std::map` in C++

Symbol table (cont.)

New operations to deal with scopes:

`open` a new scope (i.e. in the beginning of a function/method)

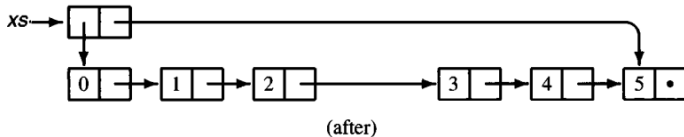
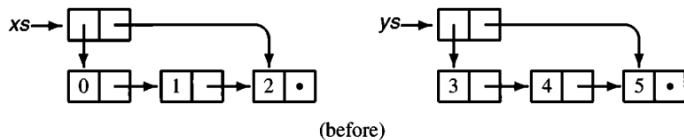
`close` finishes the actual scope

Note: **it is not sufficient to delete information** when closing the scope (we need to reconstruct the table as it was before).

Implementation

C, Java, etc.

```
xs = new List(0,1,2);  
ys = new List(3,4,5);  
xs.append(ys);
```



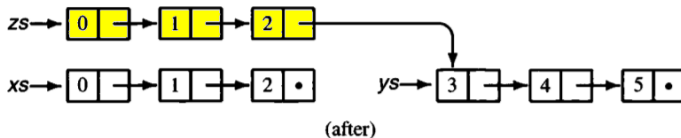
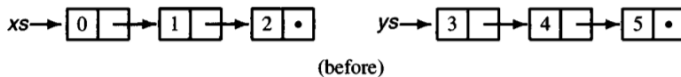
Implementation (cont.)

Haskell, SML, etc.

`xs = [0,1,2]`

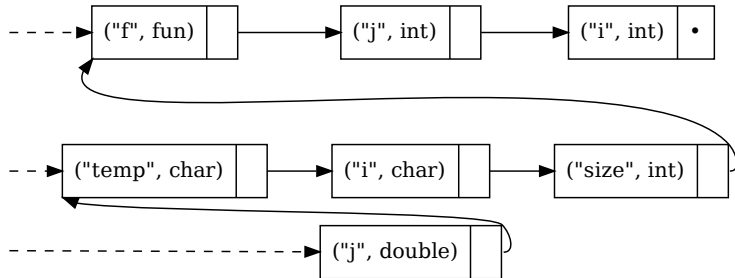
`ys = [3,4,5]`

`zs = xs ++ ys`



Types (with lists)

```
int i, j;  
int f(int size)  
{ char i, temp;  
  ...  
  { double j;  
    ...
```



Types (with a stack)

Opera de forma similar às listas funcionais.

`init` an empty stack;

`insert` push (*name,info*)

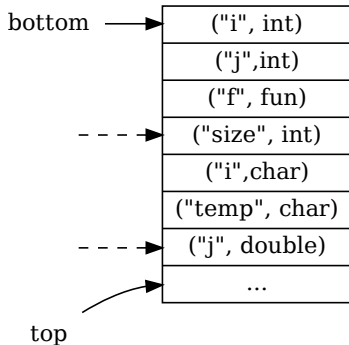
`lookup` pop;

`open scope` store the top of the stack;

`close scope` load the previous top of the stack.

Types (with a stack) (cont.)

```
int i, j;  
int f(int size)  
{ char i, temp;  
  ...  
  { double j;  
    ...
```



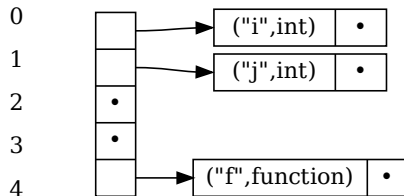
- ▶ Lists and stacks use **sequential search**: $O(n)$ for a table with n elements
- ▶ Not efficient!
- ▶ Alternatives: **search trees** or **hash tables**

- ▶ Based on a **total order on keys** ($>$, $<$, $==$)
- ▶ AVL or Red-Black trees are $O(\log n)$ in the worse case

Hash tables

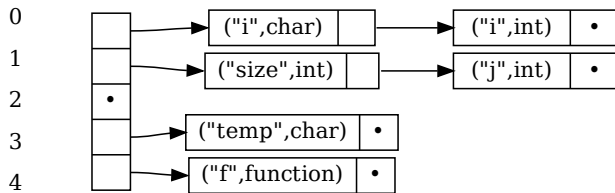
Exemplo:

- ▶ Hash table with $N = 5$ *buckets* and 3 entries (i,j,f)
- ▶ Suppose that $h(i) = 0$, $h(j) = 1$ e $h(f) = 4$
- ▶ No collisions: each *bucket* has one entry



Hash tables (cont.)

- ▶ Insert new entries for `i`, `size` and `temp`
- ▶ **Collisions** for (`i`) and suppose that: $h(\text{size}) = h(j) = 1$
- ▶ We solve collisions putting the new entries at the beginning:



- ▶ Search from the beginning of each list
- ▶ Closing the scope removes the entries

How to choose a good *hash function*?

$$x = x_0x_1 \dots x_{k-1}$$

- ▶ There are lots of different heuristics depending on the *trade-offs* (speed / collisions)
- ▶ One simple solution: sum the character codes multiplied by powers of α

$$h(x) = (x_0\alpha^{k-1} + x_1\alpha^{k-2} + \dots + x_{k-1}\alpha^0) \bmod N$$

Choose an α with a 2 exponent: multiplications may be implemented by *shifts*

Hash tables (cont.)

Example ($\alpha = 2^4 = 16$).

```
#define N ...
```

```
unsigned hash(char *ptr) {  
    unsigned h = 0;  
    while(*ptr) {  
        h = (h<<4) + (unsigned)(*ptr++);  
    }  
    return (h % N);  
}
```

Symbol Table - Example

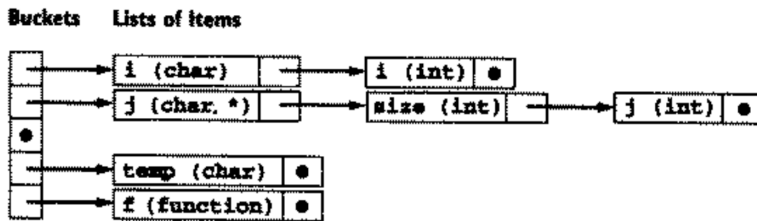
Consider the following C code fragment illustrating nested scopes:

```
int i,j;

inf f(int size)
{ char i, temp;
  ...
  { double j;
    ...
  }
  ...
  { char *j;
    ...
  }
}
```

Symbol Table - Example

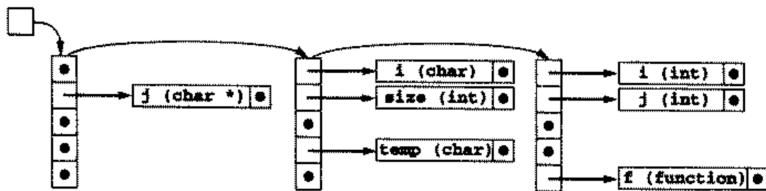
Symbol table contents:



After processing the declaration of the second nested compound statement within the body of f

Separate Symbol Tables

Symbol table structure:



Using separate tables for each scope - corresponding to the same example)

This lecture

Semantic analysis

Scopes

Symbol table

Type Systems

Type Checking

Expressions

Type Declarations

Type Checking for Functions

Extras

Type System A set of **logical rules** which programs must respect.

Exemples:

- ▶ $+$, $-$, $*$, $/$ are only applied to numbers
- ▶ the condition in an `if` command must be a *boolean*
- ▶ On an assignment `var = expr` the *variable* `var` must have the same type as the *expression* `expr`

Type Systems

Classification of type Systems:

Static type checking is done at **compile time**

vs.

Dinamic type checking is done at **run-time**

Strong type errors are **not allowed**

vs.

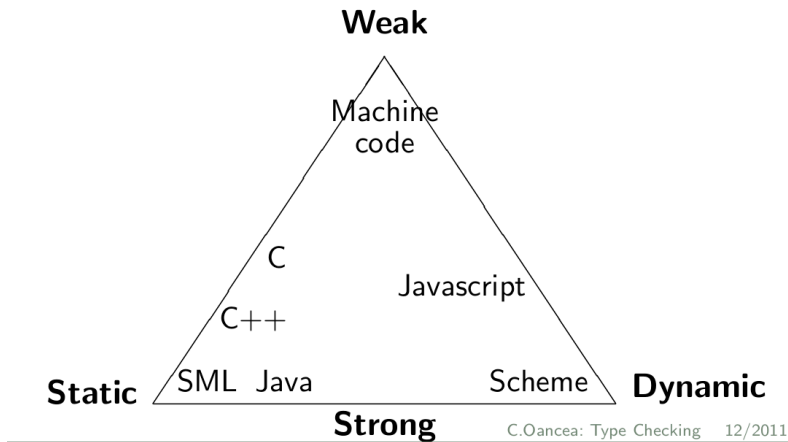
Weak some type errors **may be allowed**

Type Systems (cont.)

Examples:

- ▶ Haskell, SML, Java: **static and strong** typing
- ▶ Python, Scheme: **dynamic and strong** typing
- ▶ C: **static and weak** typing
- ▶ Note that these classifications are gradual and not mandatory
e.g. o Java type system is stronger than for C and weaker than Haskell

Type Systems (cont.)



(Image: *Introduction to Compiler Design*, Torben Mogensen.)

Why do we need types?

- ▶ avoids type errors at run-time
- ▶ helps the compiler to generate more efficient code
- ▶ helps the programmer to detect bugs (a first program verification technique)

This lecture

Semantic analysis

Scopes

Symbol table

Type Systems

Type Checking

Expressions

Type Declarations

Type Checking for Functions

Extras

Attribute Grammars

- ▶ Type checking may be made by traversing the AST (one or more times)
- ▶ As the AST is a recursive structure type checking uses recursive functions
- ▶ The compiler builds **node attributes**; examples:
 - ▶ Types;
 - ▶ Symbol Table (**context**)
- ▶ **Synthesized attributes**: bottom-up
- ▶ **Inherited attributes**: top-down
- ▶ Attributes maybe synthesized and inherited
(e.g. the symbol table is synthesized on the variable declarations and inherited on expressions)

Expressions

- ▶ A language with *variables*, *arithmetic expressions* and *boolean expressions*
- ▶ Types: `int` (integer numbers) e `bool` (boolean values)

Examples:

$1+2*3$

$(1+2)<3$

$1+x*3$ (*valid if x has type `int`*)

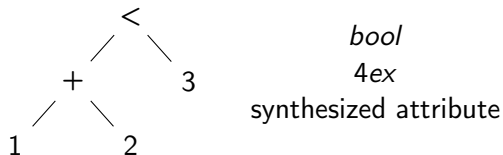
$(1+x)<y$ (*valid if x and y have type `int`*)

A **type error**:

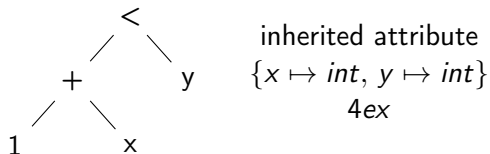
$1+(2<3)$

Inherited and Synthesized Attributes

We calculate the type of an expression from the type of its sub-expressions(**synthesized attribute**).



We use a symbol table with the types of variables (**inherited attribute**).



Type Checking

Grammar Rule	Semantic Rules
$decl \rightarrow type\ var\text{-}list$	
$type \rightarrow \mathbf{int}$	$dtype = integer$
$type \rightarrow \mathbf{float}$	$dtype = real$
$var\text{-}list_1 \rightarrow \mathbf{id}, var\text{-}list_2$	$insert(\mathbf{id}.name, dtype)$
$var\text{-}list \rightarrow \mathbf{id}$	$insert(\mathbf{id}.name, dtype)$

(Attribute Grammar for Type Declarations)

Type Checking

Grammar Rule	Semantic Rules
$\text{var-decl} \rightarrow \text{id} : \text{type-exp}$	$\text{insert}(\text{id.name}, \text{type-exp.type})$
$\text{type-exp} \rightarrow \text{int}$	$\text{type-exp.type} := \text{integer}$
$\text{type-exp} \rightarrow \text{bool}$	$\text{type-exp.type} := \text{boolean}$
$\text{type-exp}_1 \rightarrow \text{array} \quad [\text{num}] \text{ of type-exp}_2$	$\text{type-exp}_1.\text{type} :=$ $\text{makeTypeNode}(\text{array}, \text{num.size}, \text{type-exp}_2.\text{type})$
$\text{stmt} \rightarrow \text{if exp then stmt}$	if not $\text{typeEqual}(\text{exp.type}, \text{boolean})$ then $\text{type-error}(\text{stmt})$
$\text{stmt} \rightarrow \text{id} := \text{exp}$	if not $\text{typeEqual}(\text{lookup}(\text{id.name}), \text{exp.type})$ then $\text{type-error}(\text{stmt})$
$\text{exp}_1 \rightarrow \text{exp}_2 + \text{exp}_3$	if not ($\text{typeEqual}(\text{exp}_2.\text{type}, \text{integer})$ and $\text{typeEqual}(\text{exp}_3.\text{type}, \text{integer})$) then $\text{type-error}(\text{exp}_1)$; $\text{exp}_1.\text{type} := \text{integer}$

(Attribute Grammar for Types)

Type Checking

$exp_1 \rightarrow exp_2 \text{ or } exp_3$	if not ($typeEqual(exp_2.type, boolean)$ and $typeEqual(exp_3.type, boolean)$) then $type-error(exp_1)$; $exp_1.type := boolean$
$exp_1 \rightarrow exp_2 [exp_3]$	if $isArrayType(exp_2.type)$ and $typeEqual(exp_3.type, integer)$ then $exp_1.type := exp_2.type.child1$ else $type-error(exp_1)$
$exp \rightarrow num$	$exp.type := integer$
$exp \rightarrow true$	$exp.type := boolean$
$exp \rightarrow false$	$exp.type := boolean$
$exp \rightarrow id$	$exp.type := lookup(id.name)$

(Attribute Grammar for Types)

Implementation (two phases)

1. Fill the symbol table with type declarations (visitor which builds the symbol table)
 - 1.1 variables bound with their types
 - 1.2 method names bound to their parameters, result type and local variables
 - 1.3 class names bound to their variable and method declarations
2. Type check expressions and statements (visitor which type checks)

Implementation (example)

```
// PlusExp e1,e2;

public Type visit(Plus n) {
    if (! (n.e1.accept(this) instanceof IntegerType) )
        error.complain("Left side of Plus must be of type integer");
    if (! (n.e2.accept(this) instanceof IntegerType) )
        error.complain("Right side of Plus must be of type integer");
    return new IntegerType();
}
```

Type Declarations

- ▶ Type declarations add new information to the symbol table: (Identifier i, Type t)

```
public void visit(VarDecl n) {
    Type t = n.t.accept(this);
    String id = n.i.toString();

    if (currMethod == null) {
        if (!currClass.addVar(id,t))
            error.complain(id + "is already defined in " + currClass.getId());
        } else if (!currMethod.addVar(id,t))
            error.complain(id + "is already defined in "
                + currClass.getId() + "." + currMethod.getId());
    }
}
```

Type Declarations (cont.)

```
class ErrorMsg {  
    boolean anyErrors;  
    void complain(String msg) {  
        anyErrors = true;  
        System.out.println(msg);  
    }  
}
```


Type Checking for Commands

Assignment ($var := expr$) is well-typed if the type declaration for var is the type of $expr$

Conditional ($if\ cond\ then\ e1\ else\ e2$) is well-typed if:

1. $cond$ must have type `bool`
2. $e1$ and $e2$ must have the same type

Conditional ($if\ cond\ then\ expr$) is well-typed if:

1. $cond$ must have type `bool`
2. $expr$ must be well-typed

While ($while\ cond\ expr$) is well-typed if::

1. $cond$ must have type `bool`
2. $expr$ must be well-typed

Sequence ($e1; e2$) is well-typed if e_1 and e_2 are well-typed

Type Checking for Functions

- ▶ Let us extend the language with **function definitions and function calls**
- ▶ Functional types:

$$(t_1, t_2, \dots, t_n) \rightarrow r$$

(they can be trivially reconstructed by the type declarations for arguments and return values)

- ▶ t_1, t_2, \dots, t_n are the types of **function arguments**
- ▶ r is the **type of the result value**

Function calls

To type check a function call $f(e_1, \dots, e_n)$

1. lookup for f in the symbol table and get a functional type $(t_1, \dots, t_n) \rightarrow r$
2. type check e_1, \dots, e_n
3. if the types are equal to types t_1, \dots, t_n return type (r) ; otherwise return a type error

Function Definition

To type check $f(x_1 : t_1, \dots, x_n : t_n) : r = e$

1. Add the new binding $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ to the symbol table
2. Using this new symbol table type check e
3. If the type of e is equal to r return a new symbol table with

$$\{f \mapsto (t_1, \dots, t_n) \rightarrow r\}$$

Otherwise return a type error.

(Note: the symbol table in this case is an *inherited* and *synthesized* attribute)

Recursive Functions

- ▶ This type checking algorithm does not work with *recursive calls*
- ▶ The solution to this problem is simple: add an entry for f to the symbol table used to type check the body of the function.

This lecture

Semantic analysis

Scopes

Symbol table

Type Systems

Type Checking

Expressions

Type Declarations

Type Checking for Functions

Extras

Overloading and coercion

Overloading Use the *same operator* applied to different types (e.g. + for integers and reals)

```
1 + 2      -- int
1.0 + 2.5  -- float
```

Coercion Type conversion

```
int x = ...;
(float)x + 2.5  -- explicit coercion
x + 2.5         -- implicit coercion
```

Overloading and coercion (cont.)

- ▶ Type checking is trivial for built-in operators (e.g. C, Pascal)
- ▶ More (much more!) complex if the programmer may define operators for new types (e.g. C++, Haskell, ...)

Polymorfism

polymorphic: something which has several forms.

In programming languages: the possibility of writing programs which **work for different types**

Parametric Polymorphism SML/Haskell or “generics” in Java/C#

```
reverse :: [t] -> [t]           -- em Haskell  
void reverse(List<T> list);     // em Java
```

Ad-hoc Polymorphism *overloading*, interfaces, inheritance in OO languages...

Type Inference

- ▶ For some type systems it is possible to automatically **infer the types** instead of just doing **type checking**
- ▶ Strongly typed functional languages (Haskell, ML) have type inference (*Damas-Milner* type inference algorithm)