Item 4: Type checking & symbol tables

Mari Paz Guerrero Lebrero

Grado en Ingeniería Informática

Curso 2018/2019

Type checking

Type system

- In computer science, a **type system** defines how a programming language classifies values and variables into types, how it can manipulate those types and how they interact.
- A type indicates a set of values that have the same sort of generic meaning or intended purpose
- Type systems vary a lot between languages, with perhaps the biggest variations in their compile-time syntactic and run-time operational implementations

Type checking

- The process of verifying and enforcing the constraints of types
- ***type checking*** - may occur either at
  - compile-time (a static check)
  - run-time (a dynamic check).
- Static type-checking becomes a primary task of the semantic analysis carried out by a compiler.
- If a language enforces type rules strongly (that is, generally allowing only those automatic type conversions which do not lose information), one can refer to the process as *strongly typed*,
- Else, as *weakly typed*

Inference vs. synthesis

- A checker types usually handle information from two processes:
  - **Inference**: to try deduce the type of data from it use in an expression.
    - It is widely used in functional languages.
  - **Synthesis of types**: build the type of an expression from the types of its subexpressions.
    - Uses variable declarations and types provided by the user.

Type theory

- A compiler may use the static type of a value in order to optimize the storage it needs and the choice of algorithms for the operations on the value.
- The depth of type constraints and the manner of their evaluation affects the *typing* of the language.
- Further, a programming language may associate an operation with varying concrete algorithms on each type in the case of type polymorphism.
- ***Type theory*** studies type systems, although the concrete type systems of programming languages originate from practical issues of computer architecture, compiler implementation and language design.

Major functions that type systems provide include

- **Safety** - Use of types may allow a compiler to detect meaningless or provably invalid code. For example, "Hello, World" / 3 is invalid because one cannot divide a string literal by an integer. Strong typing offers more safety, but it does not necessarily guarantee complete safety.

- **Optimization** - Static type-checking may provide useful information to a compiler.

Major functions that type systems provide include

- **Documentation** - In more expressive type systems, types can serve as a form of documentation, since they can illustrate the intent of the programmer. For instance, timestamps may be a subtype of integers -- but if a programmer declares a function as returning a timestamp rather than merely an integer, this documents part of the meaning of the function.

- **Abstraction** (or **modularity**) - Types allow programmers to think about programs at a higher level. For example, programmers can think of strings as values instead of as a mere array of bytes. Or types can allow programmers to express the interface between two subsystems. This localizes the definitions required for interoperability of the subsystems and prevents inconsistencies when those subsystems communicate

Type system

- Typically a program associates each value with one particular type (although a type may have more than one subtype).
- Other entities, such as objects, modules, communication channels, dependencies, or even types themselves, can become associated with a type.
- For example:
  - datatype - a type of a value
  - class - a type of an object
  - kind - a type of a type
- A ***type system***, specified in each programming language, stipulates the ways typed programs may behave and makes behaviour outside these rules illegal.
- More formally, type theory studies type systems.

In a compiler (in practice)

- There is a language to describe the types forming type expressions. It is different that language for describing value expressions.

```
int *x[4][6];

struct nodo {
  int etiqueta;
  struct nodo * sig;
} y[2];

list<int> z;
int mcd(int m, int n);
```

Examples in C and Haskell

```
longitud :: [a] -> Int

longitud [ ] = 0
longitud (x:xs) = 1 + longitud xs
```

Internal representation of types by graphs

- The following is needed:
  - The function type
  - The tuple type (for example for the parameters)
  - Due to overloading (ad-hoc polymorphism) should be considered type sets
  - Traders also have targeted its type in the symbol table (at first)
  - Type variables (for parametric polymorphism) universally quantified
  - In the above case, when used to make new copies

```
new tipoEntero();
new tipoPuntero(tipo*);
Etc.
```

Static and dynamic typing

- In dynamic typing, type checking often takes place at runtime because variables can acquire different types depending on the execution path.
- In static typing, type checking takes place at compile time
- Dynamic typing often occurs in "scripting languages" and other rapid application development languages.
  - Dynamic types appear more often in interpreted languages,
- Whereas compiled languages favour static types.
- The term *duck typing* refers to a form of dynamic typing implemented in languages which "guess" the type of a value.

A type checking example

- In this example, (1) declares the name x;
- (2) associates the integer value 5 to the name x;
- and (3) associates the string value "hi" to the name x.
- In most statically typed systems, this code fragment would be illegal, because (2) and (3) bind x to values of inconsistent type.
- By contrast, a purely dynamically typed system would permit the above program to execute, since types are attached to values, not names.
- Dynamic typing catches errors *during program execution*.

```
var x;          // (1)
x = 5;          // (2)
x = "hi";       // (3)
```

A type checking example

- A typical implementation of dynamic typing will keep all program values "tagged" with a type, and check the type tag before using any value in an operation.
- For example:
  - `var x = 5;      // (1)`
  - `var y = "hi";   // (2)`
  - `var z = x + y; // (3)`
- In this code fragment, (1) binds the value 5 to x; (2) binds the value "hi" to y; and (3) attempts to add x to y.

A type checking example

- In a dynamically typed language,
  - the value bound to x might be a pair `(integer, 5)`,
  - and the value bound to y might be a pair `(string, "hi")`.

- When the program attempts to execute line 3, the language implementation checks the type tags integer and string, discovers that the operation + (addition) is not defined over these two types, and signals an error.

Statically type languages

- Some statically typed languages have a "back door" in the language that enables programmers to write code that does not statically type check.
- For example, Java and C-style languages have "casts".
- The presence of static typing in a programming language does not necessarily imply the absence of dynamic typing mechanisms.
- For example, Java uses static typing, but certain operations require the support of runtime type tests, which are a form of dynamic typing.
- The choice between static and dynamic typing requires some trade-offs. Many programmers strongly favour one over the other.

Static and dynamic type checking in practice

- **Static typing** finds type errors reliably and at compile time. This should increase the reliability of the delivered program.
- However, programmers disagree over how commonly type errors occur, and thus what proportion of those bugs would be caught by static typing.
- Static typing advocates believe programs are more reliable when they have been type-checked,

Static and dynamic type checking in practice

- while **dynamic typing** advocates point to distributed code that has proven reliable.
- The value of static typing, then, presumably increases as the strength of the type system is increased.
- Advocates of strongly typed languages such as ML and Haskell have suggested that almost all bugs can be considered type errors, if the types used in a program are sufficiently well declared by the programmer or inferred by the compiler.

Static typing

- Static typing usually results in compiled code that executes more quickly. When the compiler knows the exact data types that are in use, it can produce machine code that just does the right thing.
- Some dynamically-typed languages such as Common Lisp allow optional type declarations for optimization for this very reason.
- Statically-typed languages which lack **type inference** – such as Java – require that programmers declare the types they intend a method or function to use.

Static typing

- This can serve as additional documentation for the program,
- However, a language can be statically typed without requiring type declarations, so this is not a consequence of static typing.
- Static typing allows construction of libraries which are less likely to be accidentally misused by their users. This can be used as an additional mechanism for communicating the intentions of the library developer.

Dynamic typing

- Allows constructs that some static type systems would reject as illegal.
  - For example, *eval* functions, which execute arbitrary data as code, become possible
- Furthermore, dynamic typing accommodates transitional code and prototyping, such as allowing a string to be used in place of a data structure.
- It allows debuggers greater functionality; in particular, the debugger can modify the code arbitrarily and let the program continue to run and, thus, have a shorter edit-compile-test-debug cycle.
  - However, some computer scientists consider the need to use debuggers as a sign of problems in the processes of design or of development.

Dynamic typing

- Typically makes metaprogramming more powerful and easier to use.
  - For example, C++ templates are typically more cumbersome to write than the equivalent Ruby or Python code.
  - More advanced constructs such as metaclasses and introspection are often more difficult to use in statically-typed languages.
- May allow compilers and interpreters to run more quickly, since changes to source code may result in less checking to perform and less code to revisit. This too may reduce the edit-compile-test-debug cycle.

Strongly-typed programming language

- A language that places many restrictions on how variables of different types may interact with each other is said to be **strongly typed**,
- while a language that has few such restrictions is said to be **weakly typed**.
- Whether a given language is viewed as being strongly or weakly typed often depends on the person making the judgement.
- Being strongly typed is often seen as a positive attribute of a programming language and supporters of a language may refer to it as being strongly typed for this reason.

Strong and weak typing

- One definition of *strongly typed* involves not allowing an operation to succeed on arguments which have the wrong type.
- A C cast gone wrong exemplifies the absence of strong typing; if a programmer casts a value in C, not only must the compiler allow the code, but the runtime should allow it as well. This allows compact and fast C code, but it can make debugging more difficult.
- Some *pundits* use the term *memory-safe language* (or just *safe language*) to describe languages that do not allow undefined operations to occur. For example, a memory-safe language will also check array bounds.
- *Weak typing* means that a language will implicitly convert (or cast) types when used.

Weak typing example

```
var x = 5;      // (1)
var y = "hi";   // (2)
x + y;          // (3)
```

- Writing the code above in a weakly-typed language, such as JavaScript, would produce runnable code which would yield the result **`"5hi"`**.
- The system would convert the number 5 into the string **`"5"`** to make sense of the operation (the language overloads the plus-sign operator '+' to express both addition and concatenation).
- However, problems can ensue with such conversions and with operators overloaded in this way.

Weak typing

- In contrast, the REXX language (a weakly-typed environment because it only has one type) does not overload the '+' operator, and hence '+' always denotes addition.
  - The equivalent of the first example would fail (with one operand not a number),
  - and the second would yield "9", unambiguously.
- Careful language design has also allowed other weakly-typed languages to appear
  - through type inference and other techniques
  - for usability
  - while preserving the type checking and protection offered by languages such as VB.Net, C# and Java

Safely and unsafely typed systems

- A language is "type-safe" if it does not allow operations or conversions which lead to erroneous conditions.
- Let us again have a look at the JavaScript examples:
  - `var x = 5;     // (1)`
  - `var y = "hi";  // (2)`
  - `var z = x + y; // (3)`
- Variable z in the example will acquire the value **`"5hi"`**. While the programmer may not have intended this,
- Nevertheless the language defines the result specifically and the program does not crash or assign an undefined value to z. In this respect the JavaScript appears "type-safe".

Safely and unsafely typed systems

- Now let us look at the same example in C:
  - `int  x  = 5;`
  - `char  y[] = "hi";`
  - `char* z  = x + y;`
- In this example z will point to a memory address five characters beyond y. The content of that location is undefined, and might lie outside addressable memory, and so dereferencing z at this point could cause termination of the program.
- We have a well-typed, but not memory-safe program — a condition that cannot occur in a type-safe language.

Polymorphism and types

- The term *polymorphism* refers:
  - to the ability of code (in particular, functions or classes) to act on values of multiple types,
  - or to the ability of different instances of the same data-structure to contain elements of different types.
- Type systems that allow polymorphism generally do so in order to improve the potential for code re-use:
  - in a language with polymorphism, programmers need only implement a data structure such as a list or a dictionary once,
  - rather than once for each type of element with which they plan to use it.
- For this reason computer scientists sometimes call the use of certain forms of polymorphism *generic programming*.
- The type-theoretic foundations of polymorphism are closely related to those of abstraction, modularity and (in some cases) subtyping

Explicit or implicit declaration and inference

- Many static type systems, such as C's and Java's, require *type declarations*: the programmer must explicitly associate each variable with a particular type.
- Others, such as Haskell's, perform *type inference*: the compiler draws conclusions about the types of variables based on how programmers use those variables.
  - For example, given a function *f(x, y)* which adds *x* and *y* together, the compiler can infer that *x* and *y* must be numbers -- since addition is only defined for numbers.
  - Therefore, any call to *f* elsewhere in the program that specifies a non-numeric type (such as a string or list) as an argument would signal an error.
- Numerical and string constants and expressions in code can, and often do, imply type in a particular context.
  - For example, an expression 3.14 might imply a type of floating-point;
  - while **[1, 2, 3]** might imply a list of integers; typically an array.

Types of types

- A *type of types* is a **kind**.
- In the type systems of most programming languages, there are only simple types in the sense that they all belong to the same kind.
- Kinds appear explicitly in typeful programming, such as a *type constructor* in the Haskell programming language, which returns a simple type after being applied to enough simple types.
- For example the type constructor ***Either*** :
  - has the kind **** -> * -> **** and its application:
  - ***Either String Integer*** is a simple type (kind *).

Compatibility: equivalence

- A type-checker must verify that the type of any expression is consistent with the type expected by the context.
  - For instance, in an assignment statement **x = e**, the inferred type of **e** must be consistent with the declared or inferred type of the variable **x**.
- This notion of consistency, called *compatibility*, is specific to each programming language.
  - Clearly, if the type of e and the type of x are the same, then the assignment should be allowed.
  - In the simplest type systems, whether two types are compatible reduces to that of whether they are *equal* (or *equivalent*).
- Different languages, however, have different criteria for when two type expressions are understood to denote the same type.
- These different *equational theories* of types vary widely.

Compatibility: and subtyping

- Two extreme cases being:
  - *structural type systems*, in which any two types are equivalent that describe values with the same structure,
  - and *nominative type systems*, in which types must have the same "name" in order to be equal.
- In languages with subtyping, the compatibility relation is more complex.
- In particular, if *A* is a subtype of *B*, then a value of type *A* can be used in a context where one of type *B* is expected, even if the reverse is not true.
- Like equivalence, the subtype relation is defined differently for each programming language, with many variations possible.
- The presence of parametric or ad hoc polymorphism in a language may also have implications for type compatibility.

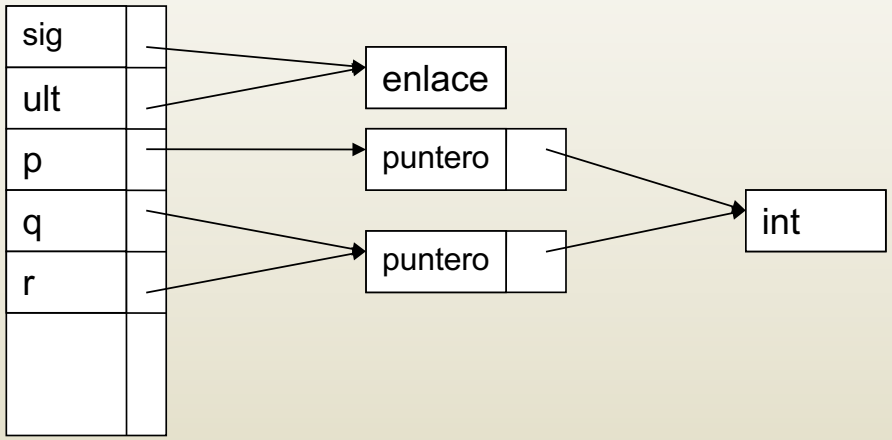
Structural and name equivalence

```
typedef int * enlace;
enlace sig, ult;
int * p;
int * q, * r;
```

- In the following program do they **`sig, ult, p, q, r`** the same type or different types?
- The answer in C is that they are the same type.
- The equivalent program in Pascal depends on the compiler.
- Under structural equivalence they do have the same type.
- Under equivalence name: **`sig = ult, q = r, but p != q != sig`**
- C uses structural equivalence but stops in the struct to allow circular definitions

Structural and name equivalence

- The reason is the construction of the symbol table and check only, equal pointers to type trees.
- The record types (struct) is usually stored with a table of private symbols.
- Type names and the names of struct are targeted at another table of special symbols.



In practice

- A type are represented as a structure:
  - IntegerType
  - FloatType
  - CharType
  - PointerType(elm_type)
  - ArrayType(#elms, elm_type)
- Statements form the corresponding structure and write it into the symbol table

```
-- int *a, ***b;
a => PointerType(IntegerType)
b => PointerType(PointerType(PointerType(IntegerType)))
-- char c[2][3][4];
c => ArrayType(2,ArrayType(3,ArrayType(4,CharType)))
-- float d;
d => FloatType
```

AST

- The expressions (and subexpressions) are represented by a syntax tree (Abstract Syntax Tree)
- An attribute to save the corresponding type is included in AST nodes

```
-- a;
The resulting type is: PointerType(IntegerType)
And the AST is: ID(a)
-- *a;
The resulting type is: IntegerType
And the AST is: pointer to ID(a)
-- *a = 23 + ***b;
The resulting type is: IntegerType
And the AST is: asignacion(puntero a
ID(a)=suma(CteEnt(23)+puntero a puntero a puntero a
ID(b)))
```

Annotation of AST

- AST labels can be annotated:
  - Through an **upstroke** each type is deducted from its basic components that read it in the symbol table (synthesized attributes)
  - By **downward** path that follows each type of use (context = inherited attributes)
  - The result of both rounds must match **to some extent!**
  - There may be errors and enter the **faultType**

Why do I say “to some extent”?

- There could be coercion of types, polymorphism, etc.
- This depends on the "type system" concrete language that we use
- The type system of C language is one of the simplest

```
-- *a = d;
hago conversion ftoi
El Tipo resultante de la expresion es: Entero
Y el arbol de nodos para la expresion es:
asignacion(puntero a ID(a)= funcion : ftoi(ID(d)))
```

Example

Input

- We have to write a program that supports an entry like the following and do check types

```
int *a, ***b;
char c[2][3][4];
float d;
a;
*a;
&a;
***b;
*a = 23 + ***b;
a = **b;
c[*a];
c[*a][14][***b];
**a;
*a = d;
```

Example

Output

```
-- ***b;
El Tipo resultante de la expresion es: Entero
Y el arbol de nodos para la expresion es: puntero a puntero a
puntero a ID(b)
-- *a = 23 + ***b;
El Tipo resultante de la expresion es: Entero
Y el arbol de nodos para la expresion es:
asignacion(puntero a ID(a)=suma(CteEnt(23)+puntero a puntero a
puntero a ID(b)))
-- a = **b;
El Tipo resultante de la expresion es: Puntero_a(Entero)
Y el arbol de nodos para la expresion es:
asignacion(ID(a)=puntero a puntero a ID(b))
-- c[*a];
El Tipo resultante de la expresion es:
array(3,array(4,Character))
Y el arbol de nodos para la expresion es:
array base(ID(c)) e indice(puntero a ID(a))
```

Example

Output

```
-- c[*a][14][***b];
El Tipo resultante de la expresion es: Character
Y el arbol de nodos para la expresion es:
array base(array base(array base(ID(c)) e indice(puntero
a ID(a))) e indice(CteEnt(14))) e indice(puntero a
puntero a puntero a ID(b))
```

```
-- *a = d;
hago conversion ftoi
El Tipo resultante de la expresion es: Entero
Y el arbol de nodos para la expresion es:
asignacion(puntero a ID(a)=funcion : ftoi(ID(d)))
```

Example

Solve unification

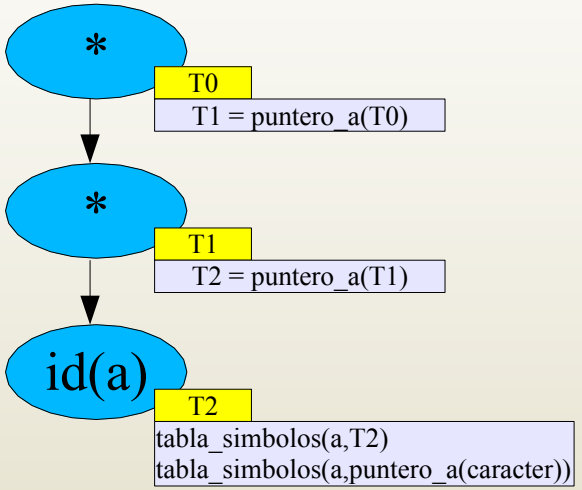
- Two passes to AST (up and down) can be given to check if both passes are compatible
- Also, a single pass can be given using **variables of types** and **unification**
  - Nodes with type variables (T0, T1, T2, ... etc.) are labeled
  - Equations linking these variables are generated
  - The equations are solved by unification

Example

Unification

- Nodes with type variables (T0, T1, T2, ... etc.) are labeled
- Equations linking these variables are generated
- The equations are solved by unification

```
-- **a;
Error al unificar en el nodoPuntero. Tengo: Entero Y
Puntero_a(variable libre(T0))
El Tipo resultante de la expresion es: variable
libre(T0)
Y el arbol de nodos para la expresion es: puntero a
puntero a ID(a)
```



Example

Grammar

```
ent : defs exps
defs : def ';' defs
     | épsilon
def : tipo lista
tipo : ENTERO     {tipo.s = new TipoInt;}
     | FLOTANTE   {tipo.s = new TipoFloat;}
     | CARACTER   {tipo.s = new TipoChar;}
lista : elm ',' {lista1.h = lista.h;} lista1
      | elm
elm : '*' {elm1.h = new TipoPuntero(elm.h);} elm1
    | ID {dims.h = elm.h;} dims
         {tabla[ID.lexval] = dims.s;}
dims : '[' CTEENT ']' {dims1.h := dims.h;} dims1
         {dims.s = new TipoArray(CTEENT.lexval,dims1.s);}
     | {dims.s = dims.h;}
```

Example

Grammar

```
exps : exp {exp.s->etiqueta();
            cout << "El arbol de nodos es: "
                 << exp.s->escribir() << endl;}
       ';' exps
     | épsilon

exp : lval '=' exp1  {exp.s = new NodoAsig(lval.s, exp1.s);}
    | sum            {exp.s = sum.s;}

sum : sum1 '+' factor
                     {sum.s = new NodoSuma(sum1.s, factor.s);}
    | factor         {sum.s = factor.s;}

factor : factor1 '*' term
                     {factor.s = new NodoProd(factor1.s, term.s);}
       | term        {factor.s = term.s;}
```

Example

Grammar

```
term :  lval          {term.s = lval.s;}
     | '(' exp ')'    {term.s = exp.s;}
     | CTEENT         {term.s = new NodoCteEnt(CTEENT.val);}
     | CTEFLOAT       {term.s = new NodoCteFloat(CTEFLOAT.val);}

lval : '*' lval       {lval.s = new NodoPuntero(lval1.s);}
     | '&' lval       {lval.s = new NodoAmp(lval1.s);}
     |  basico        {lval.s = basico.s;}

basico : basico1 '[' exp ']`
                 {basico.s = new NodoArray(basico1.s, exp.s);}
       |  ID     {basico.s = new NodoId(ID.lexval);}
```

Example

Syntax tree

```
class Nodo {
  public:
    Tipo * miTipo;
    virtual string escribir(void) = 0;
    virtual Tipo* etiqueta(void) = 0;
} ;

class NodoId : public Nodo {.....} ;

class NodoCteEnt : public Nodo {.....} ;

class NodoCteFloat : public Nodo {.....} ;

class NodoPuntero : public Nodo {.....} ;

class NodoAmp : public Nodo {.....} ;

class NodoFunc : public Nodo {.....} ;
```

```
class NodoArray : public Nodo {.....} ;

class NodoAsig : public Nodo {.....} ;

class NodoSuma : public Nodo {.....} ;

class NodoProd : public Nodo {.....} ;
```

Example

Method “Etiqueta”

- This method is making a tour "depth-first" the syntax tree
- Tagging the "miTipo" attribute of each node with the type that corresponds (synthesized mostly)
- It is able to insert some type conversions by modifying the tree
- Use sometimes unification

```
Tipo* NodoId::etiqueta(void) {
  if (tabla.find(nombre) != tabla.end())
    miTipo = tabla[nombre];
  else
    miTipo = new TipoError(string("Id desconocido"));
  return miTipo;
}

Tipo* NodoCteEnt::etiqueta(void) {
  miTipo = new TipoInt;
  return miTipo;}
```

Example

Method "Etiqueta"

```
Tipo* NodoAsig::etiqueta(void) {
  Tipo * miTipo = izq->etiqueta()->deref();
  Tipo * t      = der->etiqueta()->deref();
  if (! unifica(miTipo,t)) {
    if (miTipo->tipo() == TIPOINT
        && t->tipo() == TIPOFLOAT) {
      Nodo * n = new NodoFunc(string("ftoi"),der);
      der = n;
      cout << "hago conversion ftoi\n";
    }
    else if (miTipo->tipo() == TIPOFLOAT
             && t->tipo() == TIPOINT) {
      Nodo * n = new NodoFunc(string("itof"),der);
      der = n;
      cout << "hago conversion itof\n";
    }
    else  cout << "Error al unificar\n";
 }
  return miTipo;
}
```

Example

Method "Etiqueta"

```
Tipo* NodoPuntero::etiqueta(void) {
  Tipo * s = comp->etiqueta();
  miTipo = new TipoVar();
  Tipo * t = new TipoPuntero(miTipo);
  if (! unifica(s,t))
    cout << "Error al unificar" << endl;
  return miTipo;
}
```

- In practice it could make several passes to the syntax tree
- In the upstream past the type be synthesized from components
- In descending it would prove that the types match those required by the context and conversions would be inserted
- The "unified" may return the "supreme" type

Example

Findings

- Type checking is mechanical but very laborious
- In a "real" compiler there are many different nodes of the syntax tree
- Could you automate this? generating files: "nodo.h" and "tipo.h" automatically from a description of tree nodes
- How would the description of the tree be?
- We need to manipulate syntax trees: inserting nodes, removing nodes, transforming the tree (optimization) etc.
- Programming "object-oriented" gives us a very local vision of the tree
- It is great to have other languages or tools that allow us to reason and describe the tree-level processes and patterns

Symbol tables

What a symbol table (ST) is

- The symbol table (ST) is a global data structure used in the various stages of compilation
- The ST can be a global variable or be passed as an attribute inherited
- It contains an entry for each symbol defined in the source program:
  - Identifiers (variables, functions)
  - Constants (numeric, string, etc.)
  - Other: reserved words
- Information is stored for (almost) all symbols that is required during the compilation process : data type value (constant), storage class (static, local, ...) area, address in memory, ...

Stored information

- The attributes associated with a name in the TS depend on the type of declaration

Variables: (eg bool found;)

- Type (eg bool)
- Storage class (static, parameter, local dynamic var)
- Offset: in dynamic variables should store the "offset". For example 8 (% ebp)
- Name (eg 'found')

Constants: (Example: const double PI = 3.1415; "Hello World", "%d")

- Type
- Value

Arrays: (int m [10] [15];)

- Type elements.
- Number of items
- Lower and upper limits.
- Address or symbolic name

Functions: (void f (int a, float b))

- Number of parameters.
- Type parameters.
- Way of passing parameters.
- Return type.

Stored information

- It is important to store information about the scope
- Often, the physical structure of the symbol table reflects the scopes
- Type declarations, class, etc. keep other structures similar to symbol tables, but their scope rules are different and are used differently.
  - Type: (struct Complex {float re, im;}) ...

Operations on the symbol table

Insert: Inserts a symbol, following a statement.

- Usually a search is performed in the ST to ensure that this symbol is not previously inserted, thereby ensuring that that statement is unique (in the current scope)

Search: retrieve information associated with a symbol.

- The search can be restricted to the current ST(during analysis of statements) or, in the case that language allows nesting areas, performed on the set of all ST (for analysis of non-declarative sentences)

Delete

- Deletes the information of a symbol, when no longer used

Edit

- Updates the information of a symbol

Implementation of the symbol table

- The ST is represented abstractly as a dictionary data structure type:
  - `<Symbol, information symbol>`
- There are three data structures that are often used to implement a ST:

Single or doubly linked linear lists, usually resizable.	Hashing	Binary search trees, AVL trees and B.
• It is a simple but slow system when there are many entries	• Its effectiveness depends on the function chosen dispersion. It is most often used	• Not too helpful for the complexity of certain operations such as removing

Use of ST

Statement & ST

- In compiled languages, often mandatory to declare an identifier before you can reference it.
- Remember that this rule can not be expressed with a context-free grammar.
- The verification of this rule is carried out with the support of the ST:
  - During the parsing, recognizing a statement, the identifier is inserted into the ST
    - ST is completed as the source code is analysed.
  - During the semantic analysis, when you find a symbol in the source code searches the ST and if that search fails, a violation of the rule that an identifier must be declared before use is detected

Use of ST

Nested scopes & ST

- Block structured languages and nesting:
  - A nesting blocks are allowed
  - The scope of a statement in a block is limited to the block and its nested blocks.

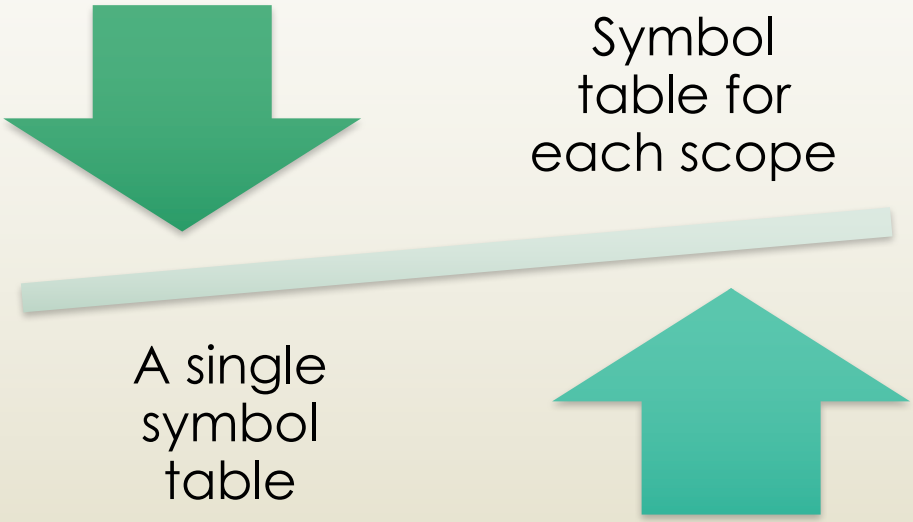
Rule nearest nesting: an identifier references always refers to the statement made in the next block

The reference identifier 'a' refers to the variable declared in block 4

```
{                       // Bloque 1
  int a, b, c, d;
  {                     // Bloque 2
    int e, f;
    L1:
  }
  {                     // Bloque 3
    int g, h;
    L2: {               // Bloque 4
          int a;
        }
    L3:
  }
}
```

Implementation on nested scopes using a ST

- There are two ways to implement symbol tables to handle nested scopes:



Implementation on nested scopes

Symbol table for each scope

- A list of scopes is created. The active scopes are placed in the list in order of opening, with the current first
- It is used in compiler of one step, in which you can discard the information regarding an scope when it is closed. Therefore, the list becomes a stack
- Disadvantages:
  - Searching into multiple symbol tables, with consequent loss of time.
  - Fragmentation of storage space of tables

Implementation on nested scopes

Symbol table for each scope

- Data structure for ST:
  - When nested scopes are processed, the symbol table behaves like a stack.
  - For each scope, a symbol table is created that is stacked on the symbol table of the container scope
- Operations on ST:

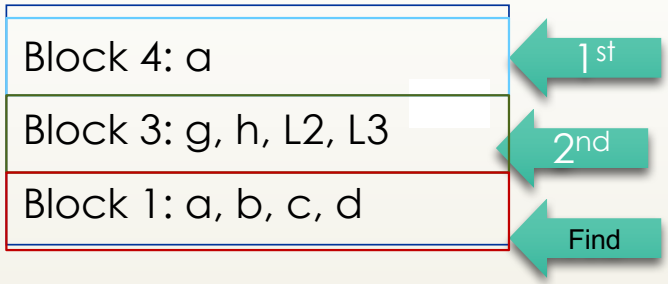
Insert	Search	Delete
• No to overwrite previous statements. • A statement on the current block results in the insertion of a symbol in the ST of that block	• The search is performed only in the current block if it is a statement, else, the search is performed in the current block and his scope containers • You must always be the latest statement • You start looking for a reference in the current ST (top of the stack); if it not found, it searches the ST corresponding to the container block, so on to the first.	• Only removes the most recent statement of a name

Implementation on nested scopes

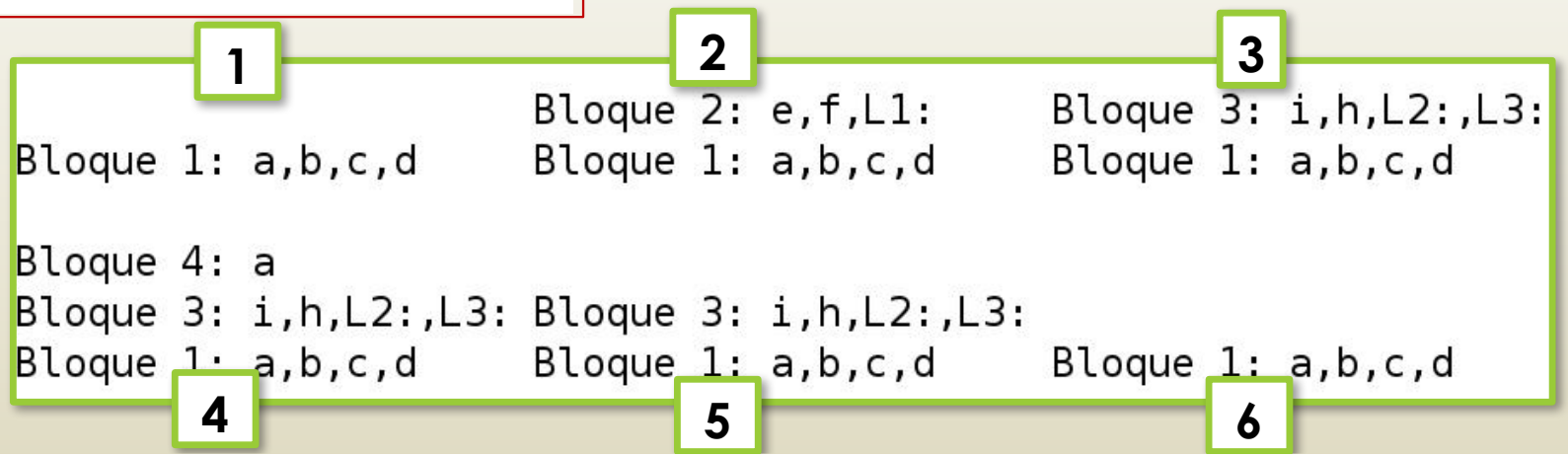
Symbol table for each scope

```
{                       // Bloque 1
  int a, b, c, d;
  {                     // Bloque 2
    int e, f;
    L1:
  }
  {                     // Bloque 3
    int g, h;
    L2: {               // Bloque 4
          int a;
        }
    L3:
  }
}
```

Searching of 'b':



Symbol table (stack)



Implementation on nested scopes

Symbol table for each scope

- If the compiler has more than one step, you will need to save the information on the scopes already closed for use in subsequent steps

```
                                                  Bloque 2: e,f,L1:
                         *Bloque 2: e,f,L1:      *Bloque 3: i,h,L2:,L3:
*Bloque 1: a,b,c,d        Bloque 1: a,b,c,d       Bloque 1: a,b,c,d

 Bloque 2: e,f,L1:        Bloque 2: e,f,L1:       Bloque 2: e,f,L1:
*Bloque 4: a              Bloque 4: a             Bloque 4: a
 Bloque 3: i,h,L2:,L3:*Bloque 3: i,h,L2:,L3:  Bloque 3: i,h,L2:,L3:
 Bloque 1: a,b,c,d        Bloque 1: a,b,c,d      *Bloque 1: a,b,c,d
```

  - The blocks marked with an asterisk and those located below it are active
  - New blocks are inserted into the table when analysing the symbol '{'. When the '}' symbol appears, the block is closed and the new blocks are inserted below it.

Implementation on nested scopes

A single symbol table

- All identifiers of all scopes are in the same table.
- Each identifier carries information about the scope to which it belongs.
- An identifier may appear several times, provided that each appearance bring a number of different levels.
- It can be implemented with a list:
  - List all the symbols are maintained, separating the symbols "available" (those belonging to the active scopes) of the "not available" (for compilers of more than one step)
  - Another list allows the identification of areas in the first list of symbols using pointers

Implementation on nested scopes

A single symbol table

- When '{' it appears a new node is added to the blocklist and the pointer is initialized at the end of the symbol list "available".
- When it comes to '}', they are disabled from the section available (or removed if the compiler is one step)
- Operations on ST:

Insert	Search
• The symbol at the end of the list of "available", the pointer that marks the beginning of the list and increases by 1 the number of elements of that block is advanced is added.	• It seeks in this single block (if statement) or he and his forefathers otherwise.

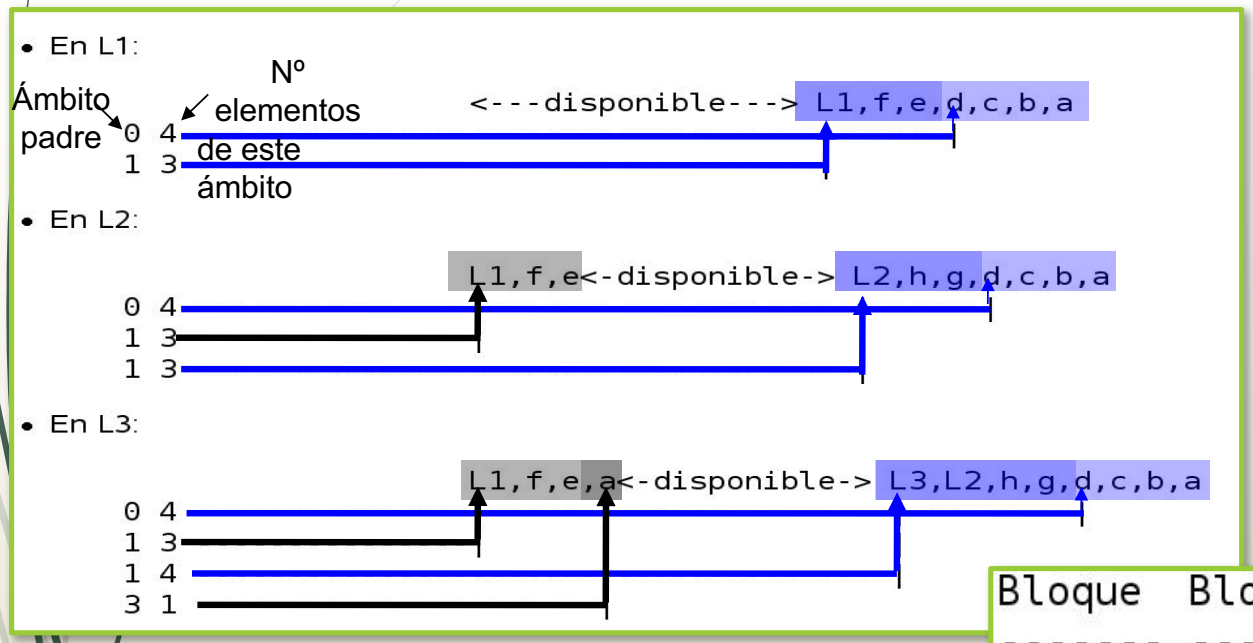
Implementation on nested scopes

A single symbol table

- Advantages of this implementation
  - The search is faster, since only need to look at a table.
  - The space occupied is less
  - The difference is offset by the fact that in this implementation are a scope

Implementation on nested scopes

A single symbol table




```
{                       // Bloque 1
  int a, b, c, d;
  {                     // Bloque 2
    int e, f;
    L1:
  }
  {                     // Bloque 3
    int g, h;
    L2: {               // Bloque 4
          int a;
        }
    L3:
  }
}
```

Bloque	Bloque superior	Nº elementos
1	0	4
2	1	3
3	1	4
4	3	1

- HASH TABLES: https://www.youtube.com/watch?v=MfhjkfocRR0