POLITECNICO DI TORINO

(01JEUHT) Formal Languages and Compilers

Laboratory N°6

Stefano Scanzio
mail: stefano.scanzio@polito.it

Type checking

- *Type expressions*
- *Symbol tables*
- *Implementation of a type-checker*

Type Checking

- *Type Checking* is the process used for the verification of types constraints:
  - Can be performed at compilation time (static check) or at execution time (dynamic check)
  - Dynamic types appear more often in interpreted languages, whereas compiled languages favor static types
  - Static checking is one of the main semantic tasks performed by a compiler

- Example of static check:

```
int a;
float b;

a = 2.5; /* Correct in c and c++, not correct in Java */
b = 1.5; /* Correct in c and c++, not correct in Java ( b=1.5f; )
```

Type Systems

- *Base types*
  - Programming languages typically include base types for:
    - numbers (int, float), characters, booleans
- *Compound and constructed types*
  - Programmers need higher level abstractions than the base types,
    - such as lists, graphs, trees, tables, etc.
  - Programming languages provide mechanisms to combine and aggregate objects and to derive types for the resulting objects
  - arrays, structures, enumerated sets, pointers
- A type system consists of a set of base types and a set of *type constructors*
  - array, function, pointer, struct (class, list, hash)
- Using base types and type-constructors each expression in a program can be represented with a *type expression*

Type-expressions

- Generally, types can be divided in :
  - Primitive types (*int, float, char*)
  - Composite types (*struct, union, pointer, array*)

types defined in C language

- Primitive types comprises all the types necessary to the formalization of a given language (*int, float, char,...*) together with 2 special ones:
  - **void** : stating the absence of a type,
  - **type_error** : stating an error found during the type checking phase.

- **Type expressions**
  - **A type-expression is either a base type or is formed by applying a type constructor to a type-expression**

Type Constructors

- Array: `array( I , T )`
  - I : size of the array
  - T : type expression
- Pointers: `pointer( T )`
- Product: T_1 X T_2
- Structure: `struct( T )`

Type constructor examples referred to the C language

Examples:

Declaration:	Type expression:
`char v[10]`	`array(10,char)`
`struct { int i; char s[5]; }`	`struct((i x int) x (s x array(5,char)))`

Type Constructors: Functions

- A function maps an element of its own domain to an element in its own range.

- Functions: $\mathbf{T_1} \rightarrow \mathbf{T_2}$
  - T_1 : domain type
  - T_2 : range type

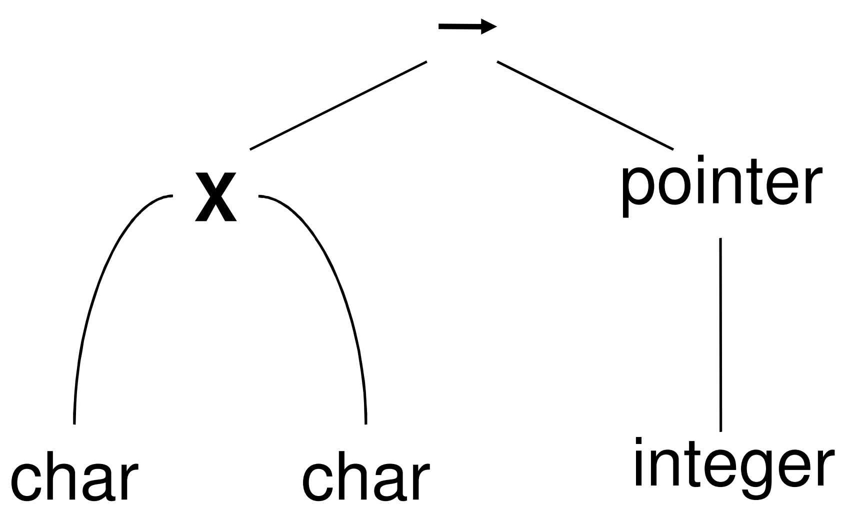
- The function int* f(char a, char b) is represented using the following type expression:

  (char x char) → pointer(int)

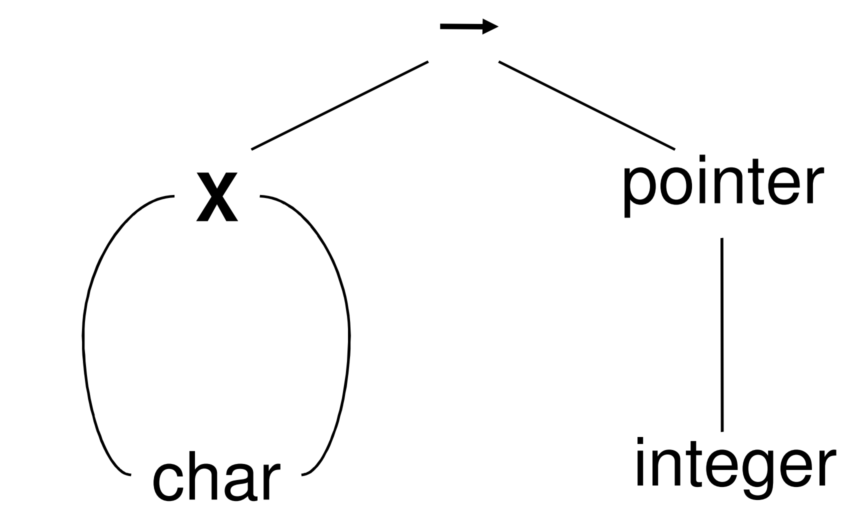
Types Graph

- An effective way of representing type expressions consists in the use of graphs (trees or DAGs).

`(char x char) → pointer(int)`



TREE



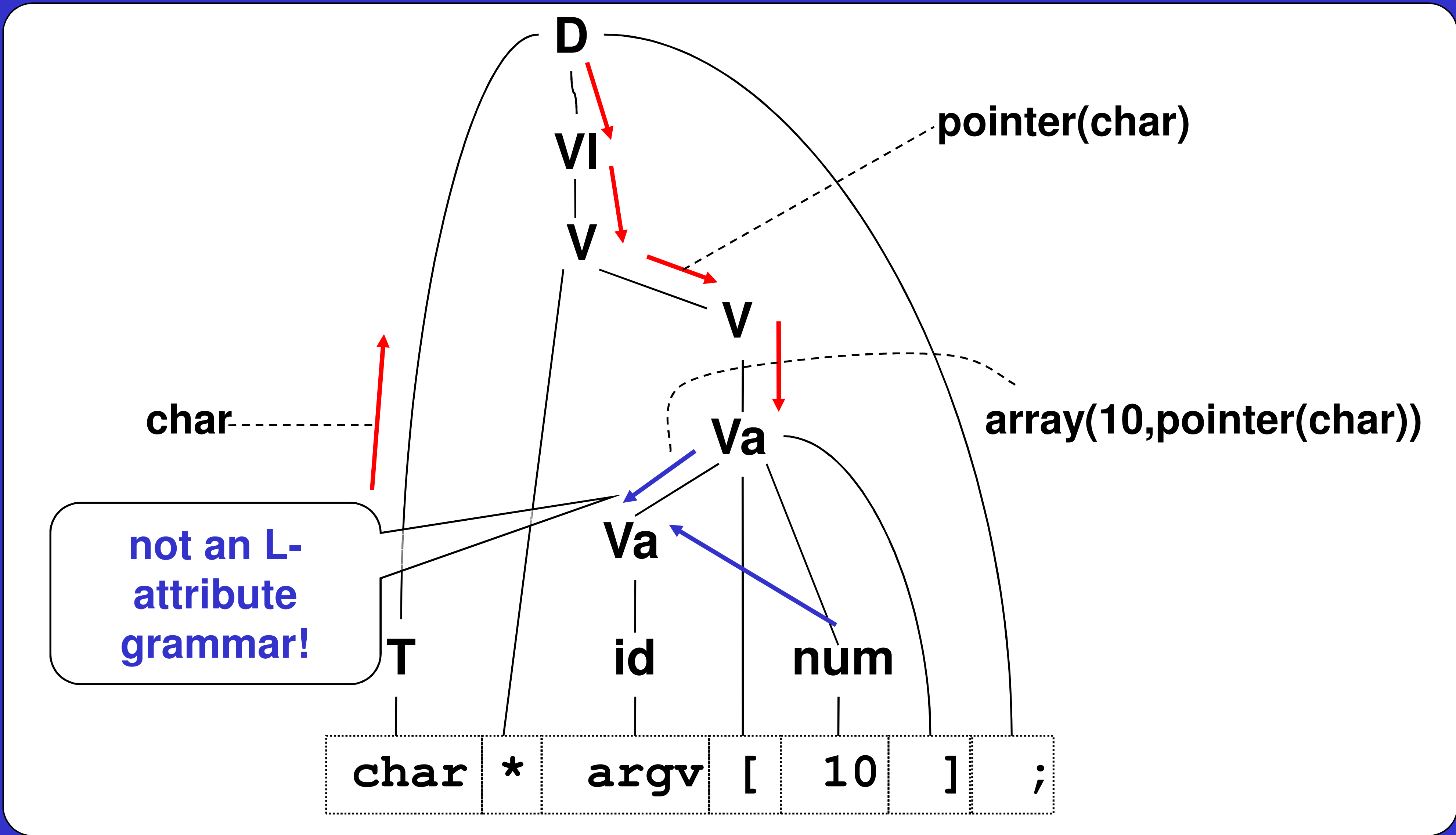
DAG

Construction of type-expressions

- A type checker for the C language could be implemented by means of the following grammar and semantic rules:

D $\rightarrow$ T Vl ';'	`Vl.type=T.type`
Vl $\rightarrow$ V	`V.type=Vl.type`
Vl $\rightarrow$ Vl_1 ',' V	`V.type=Vl.type`
V $\rightarrow$ '*' V_1	V_1`.type=pointer(V.type)`
V $\rightarrow$ Va	`Va.type=V.type`
Va $\rightarrow$ Va_1 '[' num ']'	Va_1`.type=array(num.val,Va.type)`
Va $\rightarrow$ id	`put(id.name,Va.type);`

Construction of type-expressions (2)



Construction of type-expressions (rewriting of the prev. example)

D → T Vl ';'	`Vl.type=T.type`
Vl → V	`V.type=Vl.type`
Vl → Vl_1 ',' V	`V.type=Vl.type`

V → P id A	`P.base=V.type`
	`A.base=P.type`
	`put(id.value,A.type)`

P → ε	`P.type=P.base`
P → P_1 '*'	P.type=pointer(P_1.type) P_1.type=P.base
A → ε	`A.type=A.base`
A → A_1 '[' num ']'	A.type=array(num.val,A_1.type) A_1.type=A.base

Types names

- In many languages it is possible to assign explicit names to types.

- Example:

```
typedef cell* link;        type link = ^cell;
link p;                    var p : link;
cell* q;                   var q : ^cell;
```

- Do variables **p** and **q** belongs to the same type? Answer depends on the approach used for checking it.
  - Structural equivalence.
  - Names equivalence
- In C structural equivalence is used while other languages (e.g., pascal) use names equivalence.

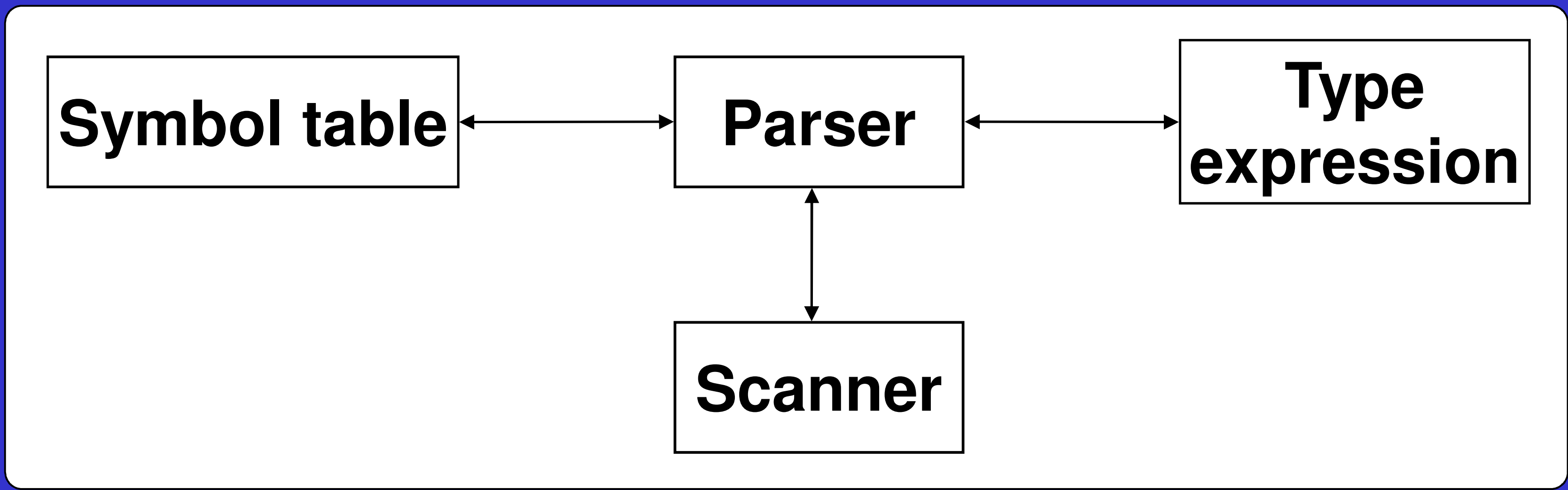
Structural Equivalence

- Two expression are equals if:
  - They belong to the same primitive type
  - They are based on the application of the same types constructors to equivalent types.

- Using a tree based representation for type expression it is possible (and convenient) to use a recursive visit algorithm in order to verify the equivalence.

Type checker

- A type checker comprises a set of interoperating modules:
  - scanner: lexicon recognition
  - parser: checks the syntax and adds the semantic,
  - type-expression manager,
  - symbol table manager.



Symbol table

- Symbol tables associates values to names in order to make accessible the semantic information related to an identifier outside of the context where it has been declared.

- Information related to each name are used in order to verify the semantic correctness of identifiers usage within a program.

Designing a Symbol Table

- A symbol table can be implemented using different data structures:
  - Unordered Lists
  - Ordered Lists
  - Binary Tree
  - Hash Table
  - BTree …
- This choice is based on the number of symbols to store, on the required performances and on the complexity of the code to be produced.

Symbol table: HashMap

```
import java.util.HashMap;
// Initializing the table
HashMap<String, String> symTable = new HashMap<String, String> ();

// Inserting entries: int a; float b;
symTable.put("a", "int");
symTable.put("b", "float");

// Get the value related to key "a"
String tipo = (String) symTable.get("a");
System.out.println(tipo);

// Deleting entry
symTable.remove("a");

// Deleting all entries
symTable.clear();
```

Type expression

- Type expressions (naturally represented by means of trees of types) can be transformed into a different internal representation (i.e., a Class).

- The management of type expressions requires
  - The definition of the data structure associated to each graph node
  - The definition of primitives that operate on nodes

- Nodes should be capable of representing the different type constructor and the base types as well.

- Primitives are required in order to hide the internal representation of nodes thus allowing the user to produce the easiest code possible.

Implementing type expressions

- Each node of a types graph comprises:
  - a tag, representing the type of node;
  - A set of different fields depending on the type of data to be stored

```
public class te_node {
  public int tag;    // BASE, ARRAY, POINTER, …
  public int size;   // Number of elements in array
  public int code;   // Base type: INT, CHAR, FLOAT, …

  // Only for structs
  public String name;// Struct name

  // Left and right children
  private te_node left, right;
}
```

Implementing type expressions

- The module charged to manage Type Expressions should offer the following primitives:

```
public class te_node {
  public int tag;
  ...
  public static te_node te_make_base(int code);
  public static te_node te_make_pointer(te_node base);
  public static te_node te_make_array(int size, te_node base);

  // Only for structs and functions
  public static te_node te_make_product(te_node l, te_node r);
  public static te_node te_make_name(String name);
  public static void te_cons_struct(te_node str, te_node flds);
  public static te_node te_make_fwdstruct(String name);
  public static te_node te_make_struct(te_node flds, String n);
  public static te_node te_make_function(te_node d, te_node r);
}
```

Type checker: complete grammar

```
S   ::= /* empty */
      | S Decl ';'
;

Decl  ::= T Vlist
        | TYPEDEF T ID
;

T   ::= TYPE
      | STRUCT ID '{' SFL '}'
      | STRUCT '{' SFL '}'
      | STRUCT ID
;

SFL   ::= Field
      | SFL Field
;
```

```
Field ::= T Vlist

;
Vlist ::= V
| Vlist ',' V

;
V   ::= Ptr ID Array

;
Ptr    ::= /* empty */
       | Ptr '*'

;
Array ::= /* empty */
       | Array SO NUM SC

;
```

Type checker: Semantic

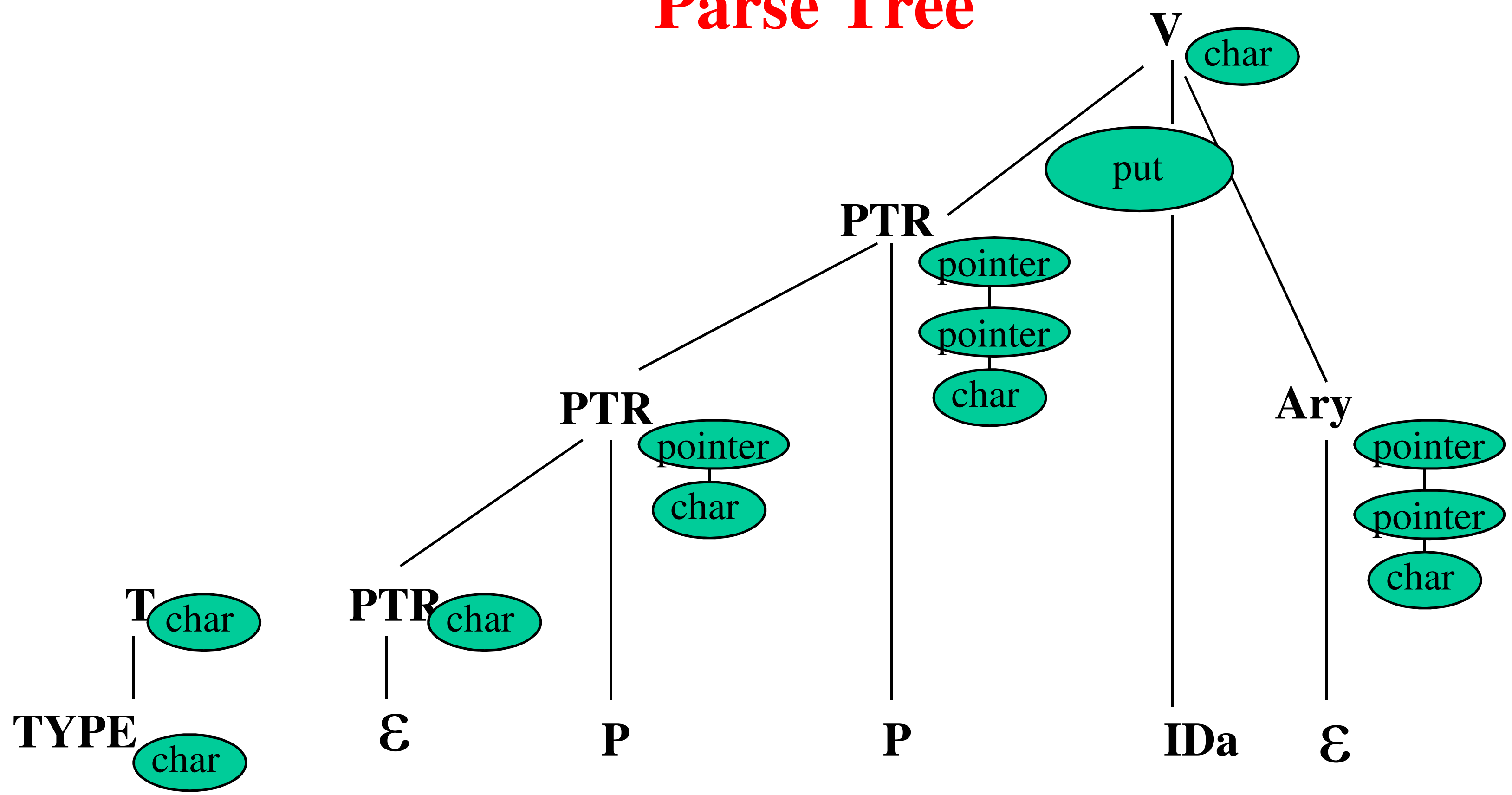
```
Decl      ::=  T Vlist S;
T   ::=   TYPE:t;            {: RESULT=te_make_base(t); :}

Vlist::=  V:t                {: RESULT=(te_node)t; :}
          | Vlist:t ',' NT0 V {: RESULT = t; :}
;
NT0 ::= /* empty */ {: RESULT=(te_node)stack[top-1]; :}
;
V   ::= Ptr ID:a Ary:t       {: add_var(a,t);
                                RESULT=(te_node)stack[top-3]; :}
;
Ptr ::=        /* empty */   {: RESULT=(te_node)stack[top]; :}
               | Ptr:p '*'   {: RESULT=te_make_pointer(p); :}
;
Ary  ::= /* empty */              {: RESULT=(te_node)stack[top-1]; :}
         | Ary:a SO NUM:b SC {: RESULT=te_make_array(b,a); :}
;
```

Type checker: Semantic (II)

```
char **a, *b[2][3];
```

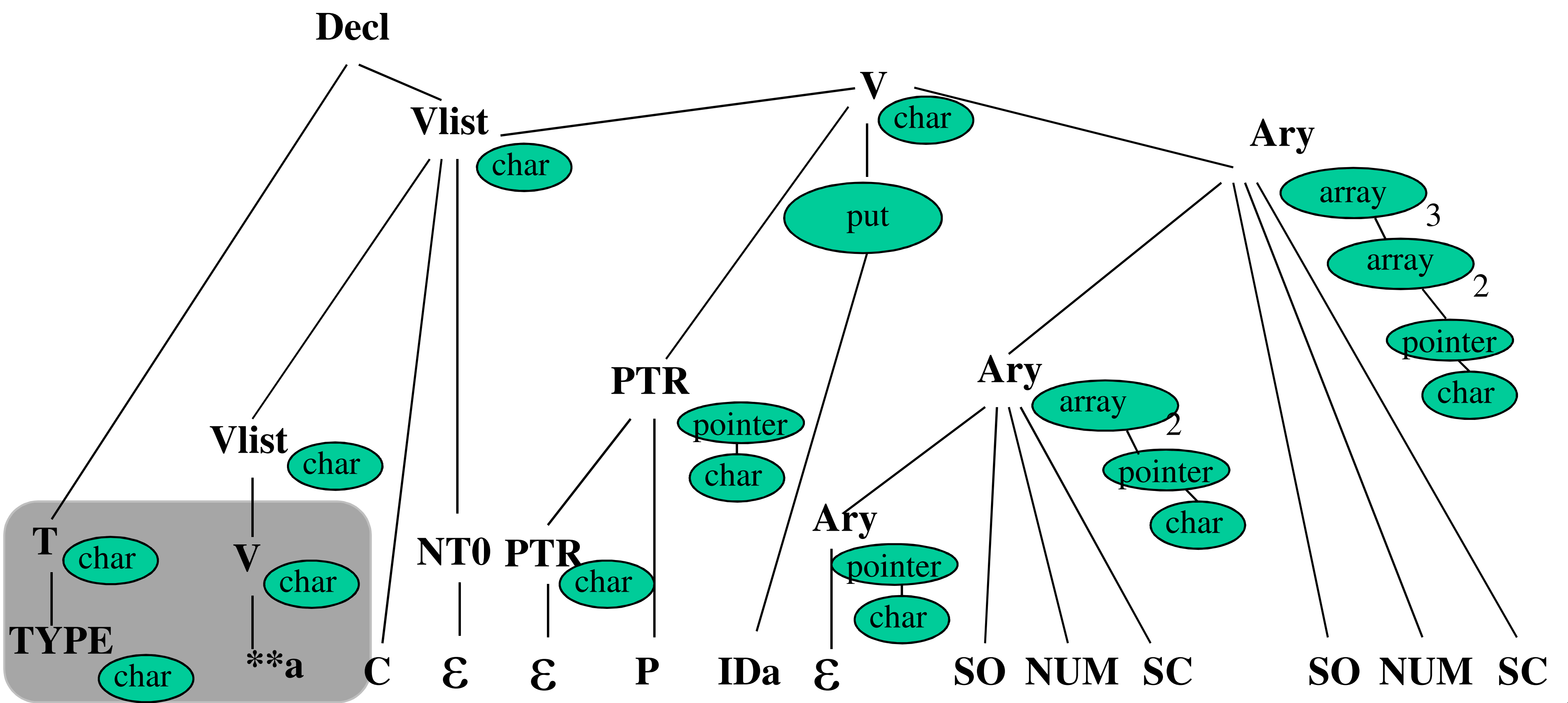
Parse Tree



Type checker: Semantic (III)

```
char **a, *b[2][3];
```

Parse Tree



Exam 1

```
// Definition of the product type:
( taste: 12 , perfume: 8 ) -> wine
( taste: 10, transparency:2 ) -> honey
.
// Description of the products
wine: * taste, + perfume = barbera DOC;
wine: * taste, * perfume = barolo di annata;
wine: - taste, / perfume = a stinky wine;
honey: * taste, * transparency = acacia honey;
```

Thesis

List: https://www.skenz.it/ss/theses
About myself: https://www.skenz.it/ss

About myself

- 2004 – Finished my studies at Politecnico di Torino in Computer Science at DAUIN
- 2008 – Ph.D. at DAUIN in Automatic Speech Recognition
  - In collaboration with Loquendo (now Nuance)
  - Specifically on Artificial Neural Networks and classification algorithms
- 2009 – Research Fellow at IEIIT (institute of the CNR)
  - CNR is the biggest Italian research organization
  - IEIIT institute is in Politecnico di Torino (near room 12, 4° floor)
- 2012 – Won a permanent position at CNR as a Researcher

Current research activities
Industrial Automation

- Communication protocols
  - Industrial networks require a high degree of determinism
  - Easy to obtain in wired networks, but in wireless ones ???
- Real-time operating system (Sometimes most of the indeterminism is inside the PC)
  - Use of real-time extensions of Linux kernel
  - Properly optimize the code (threads, kernel modules, communication between kernel and user spaces)
- Industrial Internet of Things (IIoT)

Current research activities
Industrial Automation

- Synchronization protocols
  - Nodes must have the same notion of time (μs precision or less)
  - It is a very complex argument that involves the network, operating system, hardware, control algorithms for clock correction, …

- Machine learning applied to industry

- Complete list of research activities:
  - https://www.skenz.it/ss/research

- Collaborations with: Comau and Ferrero

Programming languages

- C/C++ for the fastest parts of the code (i.e., applications with real-time requirements)

- Python for post analysis of experimental data or to coordinate experiments

- Linux operating system, and in particular:
  - Linux bash shell
  - Threads
  - Processes

For more details…

- Read my papers…

  https://www.skenz.it/ss/publications

  - Click on the paper
  - You are redirected to the relevant web page for download
  - REMEMBER: a PC inside the network of the “Politecnico di Torino” has to be used (otherwise you cannot access the paper)
  - (Citations: https://scholar.google.it/citations?user=yqyzGToAAAAJ&hl=en)

- Or better call myself (011 090 5438) or write an email
- Or even better...pass into my office
- More details regarding thesis: https://www.skenz.it/ss/theses