Compiler Construction: Practical Introduction

Samsung Compiler Bootcamp

Samsung Research Russia
Moscow 2019

Lecture 3
Compilation Data Structures

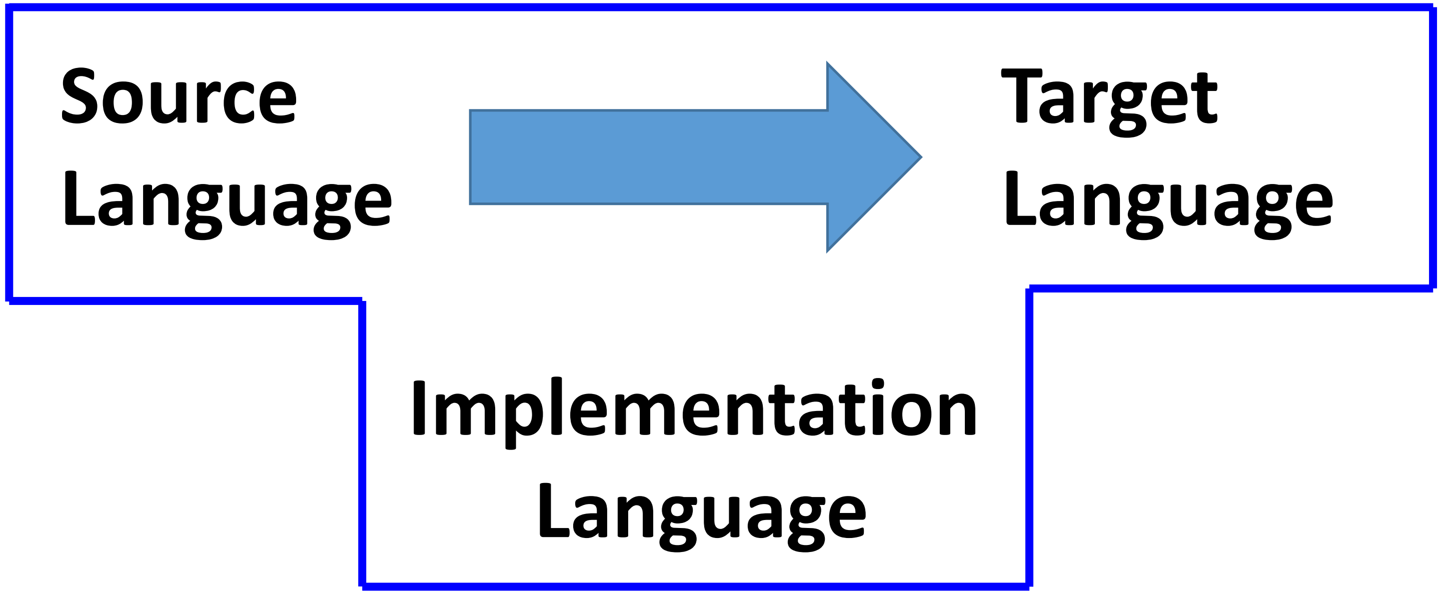
- Language entities: trees, tables, types
- Symbol tables
- Program tree
- Tables vs. Trees
- Type representation

Compilation Structures

Before we start…

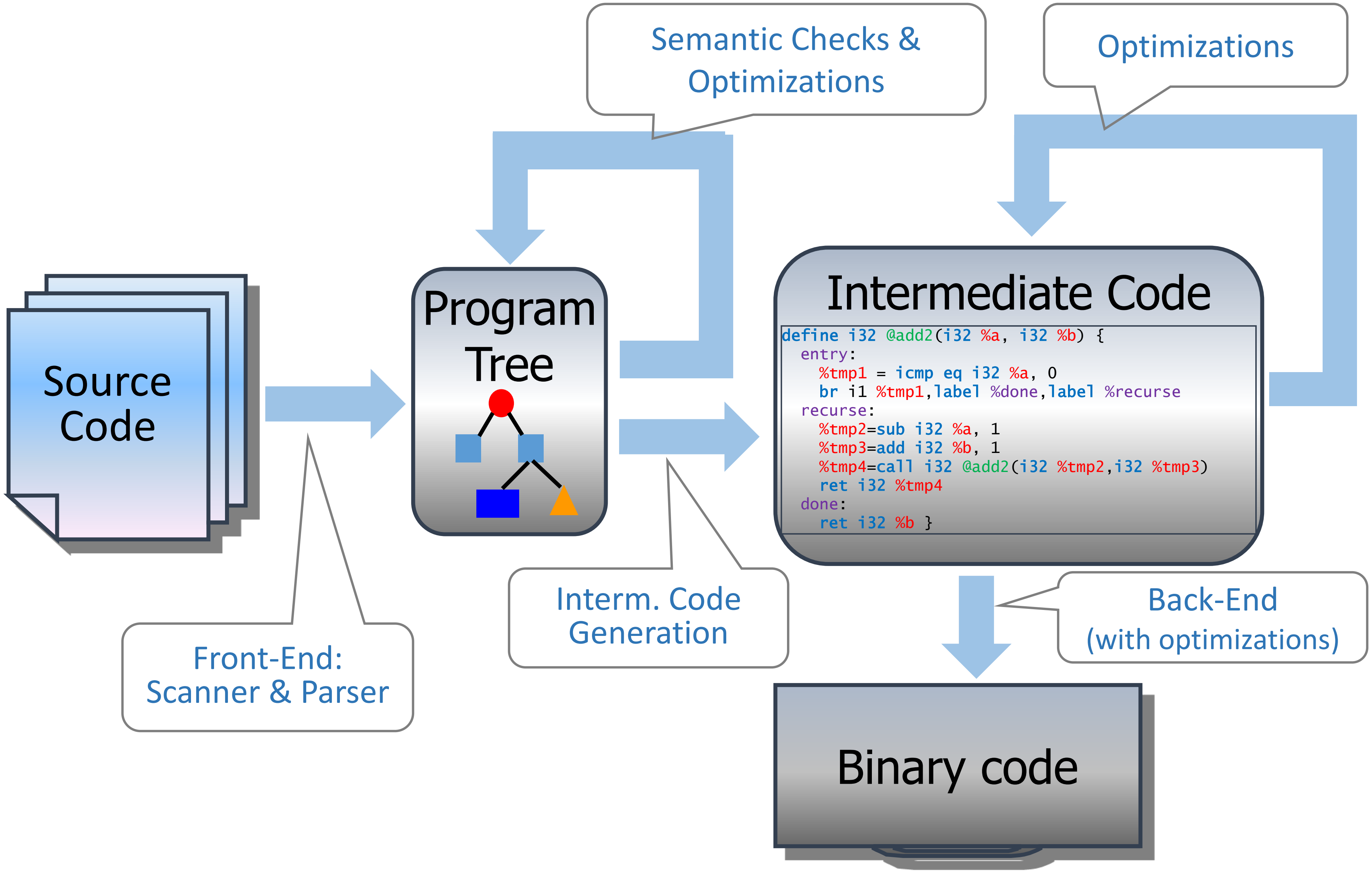
Graphical notation ("T Notation")

A Compiler



Reference: **Terence Parr**

Compilation Structures



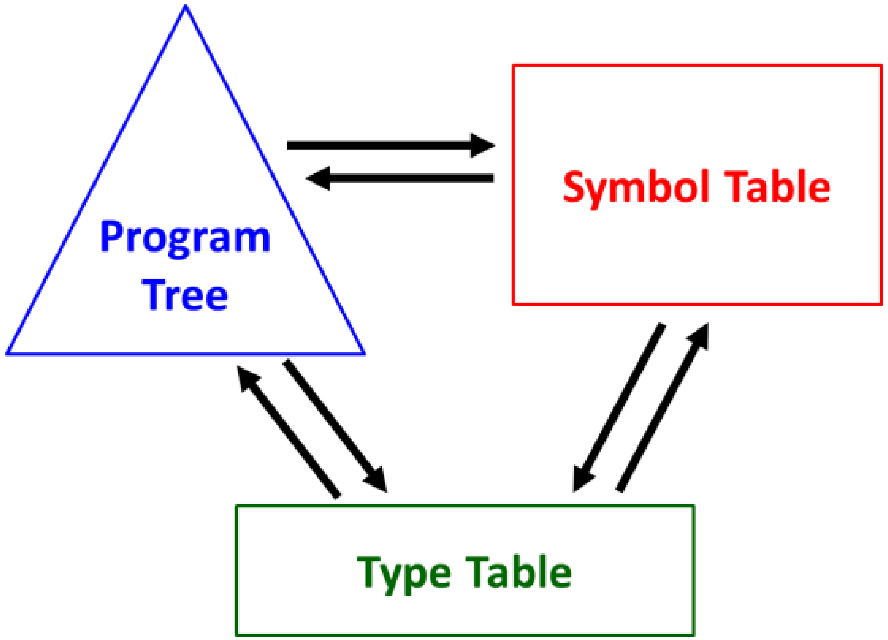
Compilation Structures

The task:

Represent all information from the source program (lexical, syntactical, semantic) in a way convenient for further analysis and processing.

Three entity categories of any language:

- Objects/declarations
- Executable parts: expressions, statements
- Types



Symbol Table: Low-Level Case

```
procedure Swap ( a, b : in out Integer )
is
   Temp : Integer := a;
begin
   a := b;
   b := Temp;
end Swap;
```

Symbol Table

Таблица символов, таблица имен

- Keeps all information about <u>named entities</u> from the program.

What exactly do we need to keep:

- Entity name
- Entity type (if any)
- Entity initializer (if any)
- Entity status
- Usage sign
- …

Swap

49	48		39 … 30	29 … 20	19 … 10	9 … 0
1	3		--	**int** (index to TT)	**a** (index to LT)	hash link
1	3		--	**int** (index to TT)	**b** (index to LT)	hash link
1	4		Initializer	**int** (index to TT)	**Temp** …	hash link

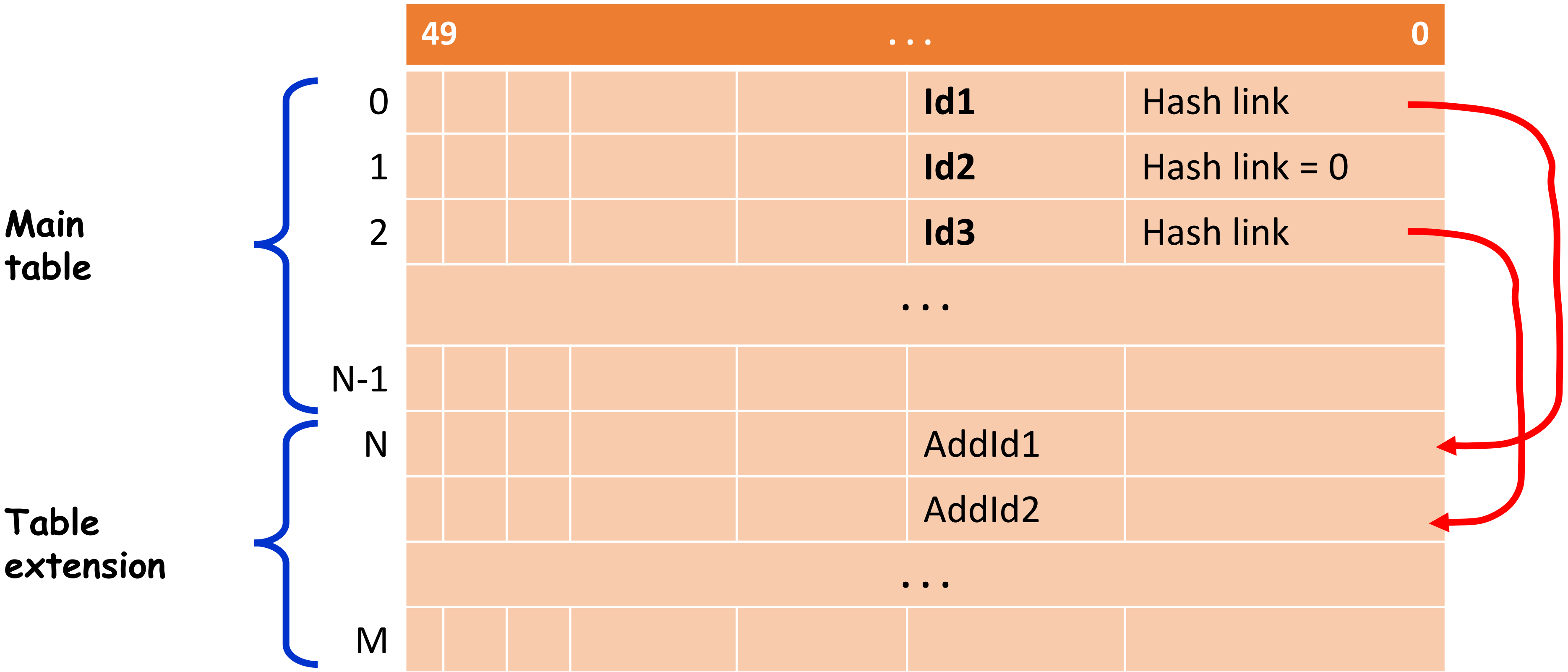
in / out / local

used / not in use

Link to Program tree

Link to Type Table

Hash Table



Hash("Id1") = Hash("AddId1")
Hash("Id3") = Hash("AddId2")

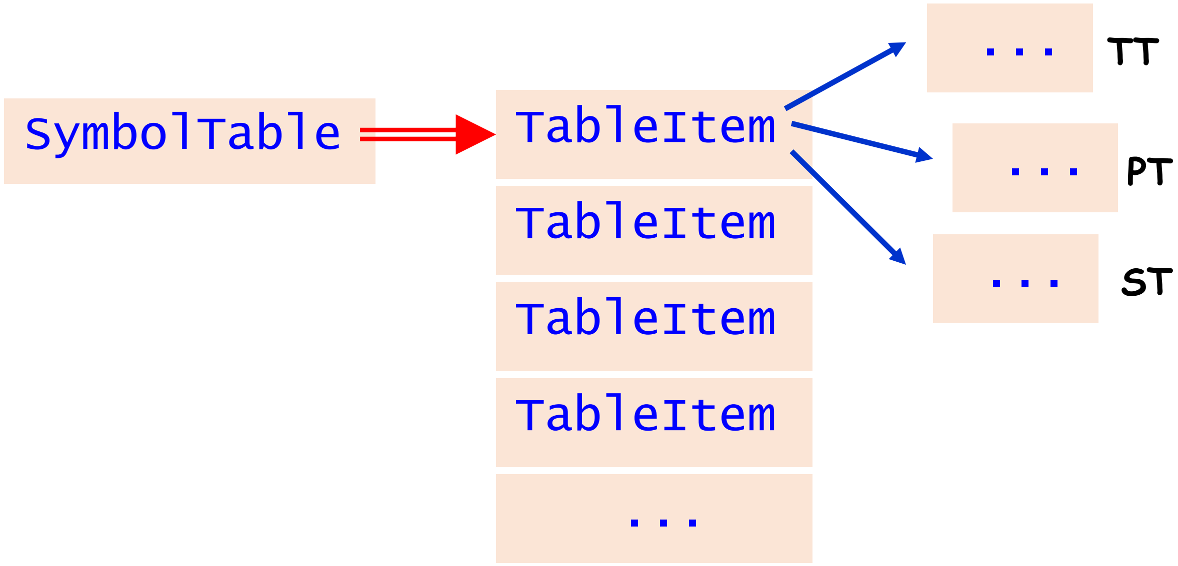
Hash Function Example

```
class Hash_Holder {
    private static readonly uint hash_module = 211;

    public static uint Hash ( string identifier ) {
        uint g;     // for calculating hash
        const uint hash_mask = 0xF0000000;

        uint hash_value = 0;
        for ( int i=0; i<identifier.Length; i++ )
        {
            // Calculating hash: see Dragon Book, Fig. 7.35
            hash_value = (hash_value << 4) + (byte)identifier[i];
            if ( (g = hash_value & hash_mask) != 0 )
            {
                hash_value = hash_value ^ (hash_value >> 24);
                hash_value ^= g;
            }
        }
        return hash_value % hash_module;   // the final hash value for "identifier"
    }
}
```

Symbol Table: Mid-Level Case

Source code

```
procedure Swap ( a, b : in out Integer )
is
   Temp : Integer := a;
begin
   a := b;
   b := Temp;
end Swap;
```



C-based table implementation

```
struct TableItem
{
    char* name;
    int hashChain;
    TYPE* type;
    NODE* initializer;
    bool isUsed;
    int status;
    ...
};
```

```
struct TableItem* SymbolTable[];
```

Symbol Table: High-Level Case

Source code

```
procedure Swap ( a, b : in out Integer )
is
   Temp : Integer := a;
begin
   a := b;
   b := Temp;
end Swap;
```

C#-based table implementation

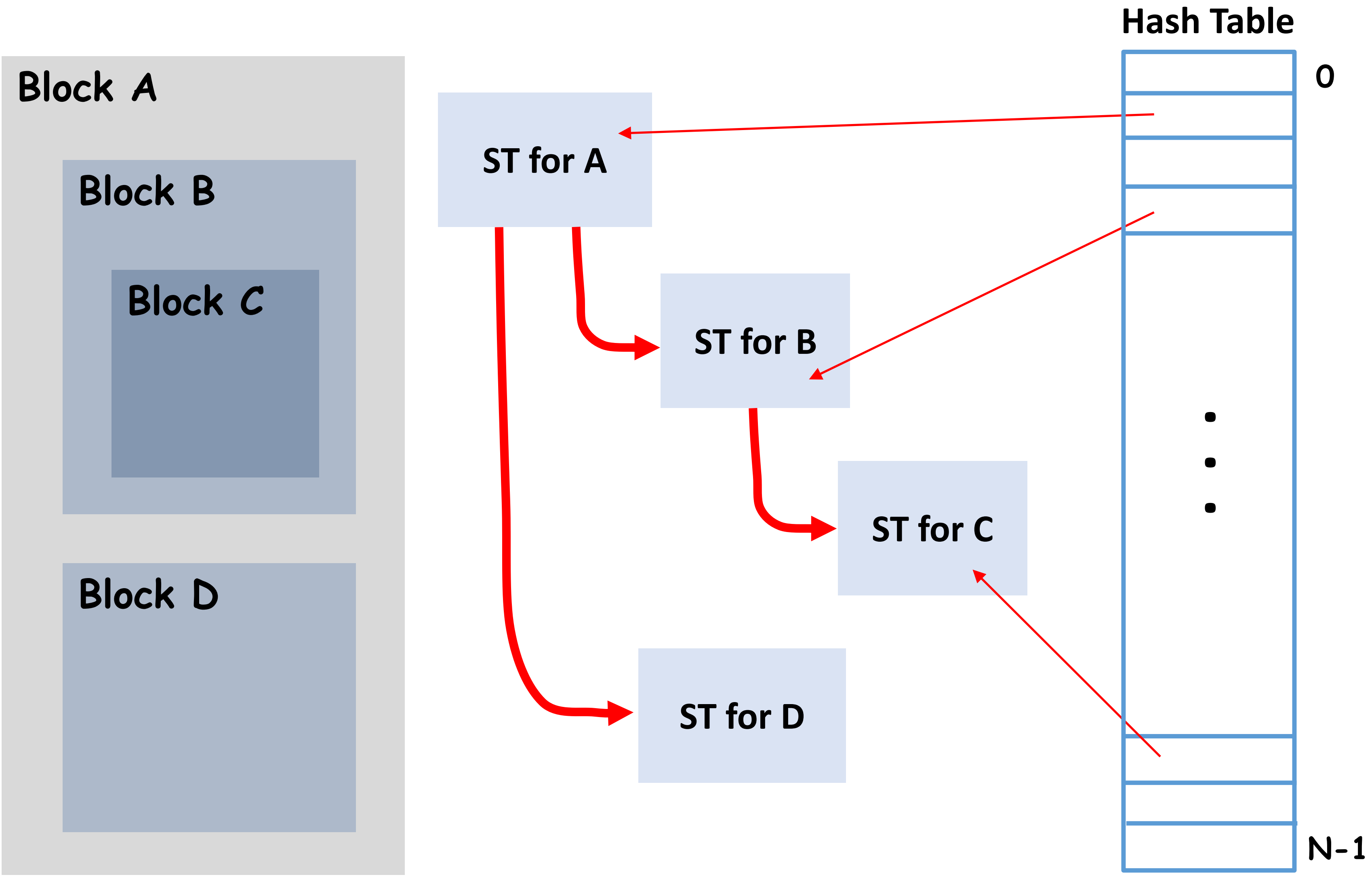
```
class TableItem
{
    string name;
    TYPE type;
    NODE initializer;
    bool isUsed;
    int status;
    ...
};
```

```
var SymbolTable = new Dictionary<string,TableItem>();
```

Symbol Tables & Nested Blocks

```
void F1(int a, int b)
{
  int local1;
  ...
  if ( condition )
  {
      double d;
      ...
  }
  int local2;
}
```

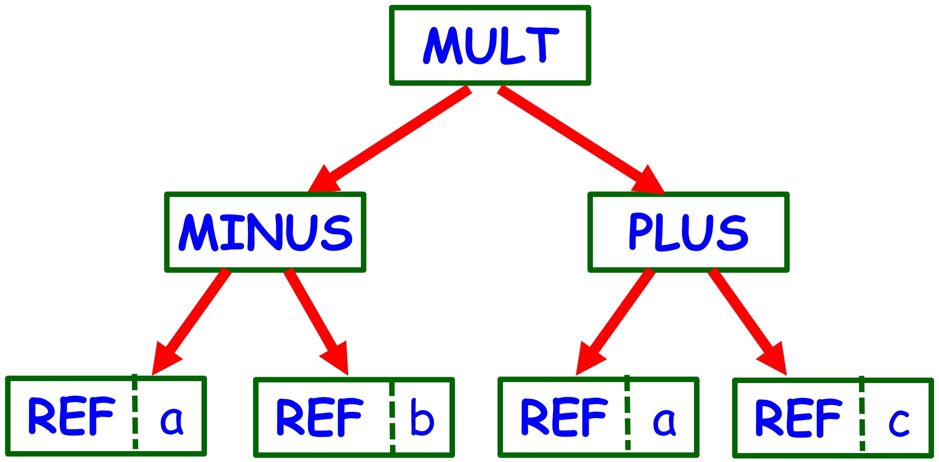
```
void F2(void)
{
   long local;
   ...
}
```



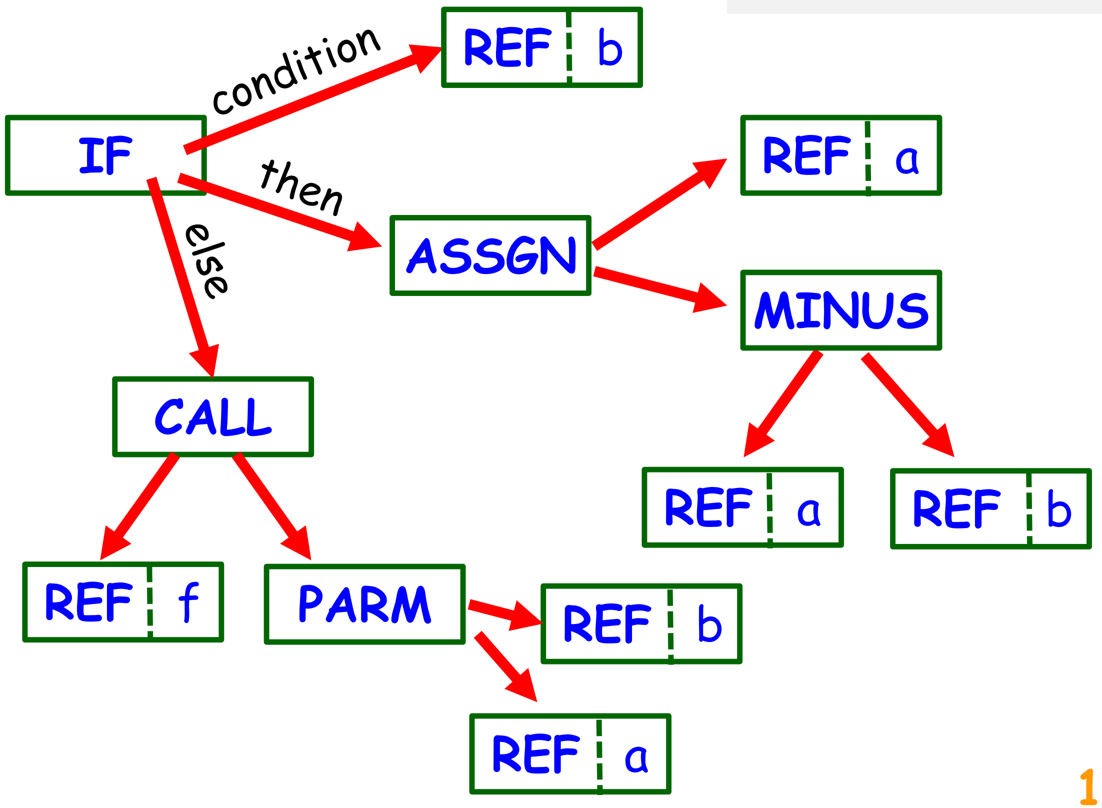
Program Tree

Each language construct can be represented as a (sub)tree

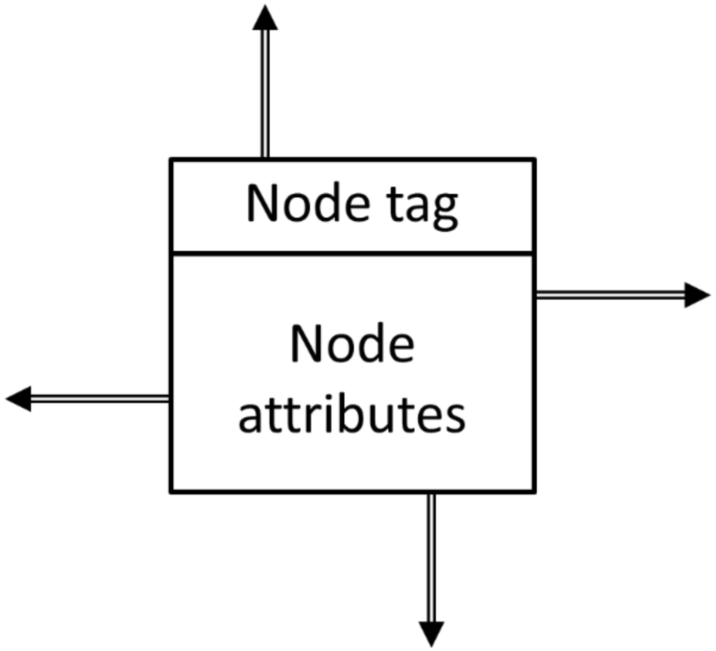
```
(a - b) * (a + c)
```




```
if ( b )
    a = a-b
else
    f(a,b)
```



Program Tree: Interstron C++ implementation (1)

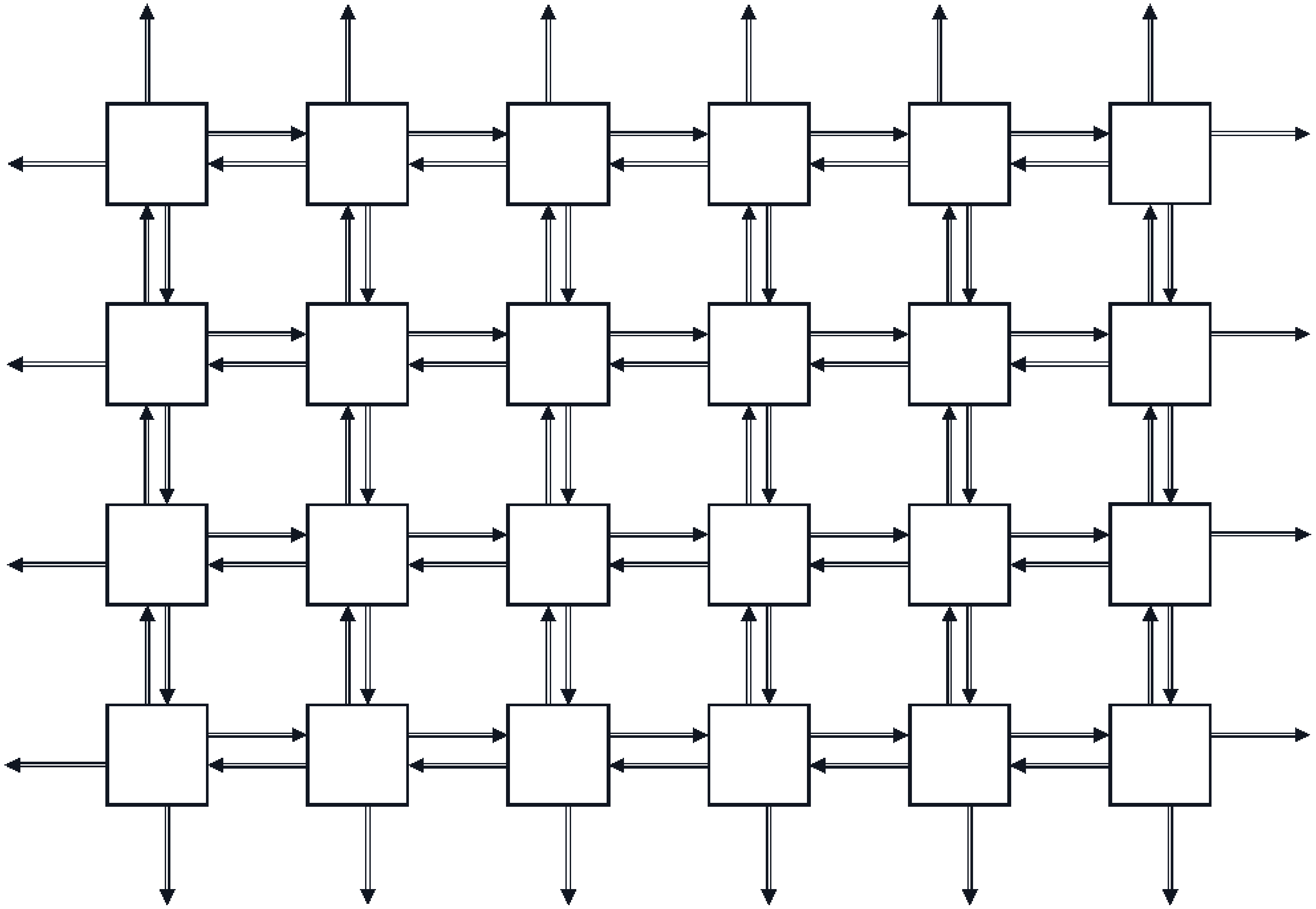


The tree node is a small structure:

- The unique node tag.
- Each tag represents a particular language construct.
- There are four pointers to make links between nodes: «up», «down», «left», «right».
- Each node has a set of attributes; attributes depend on node's tag.
- There are "empty" nodes (without semantics) for organizing complex configurations.

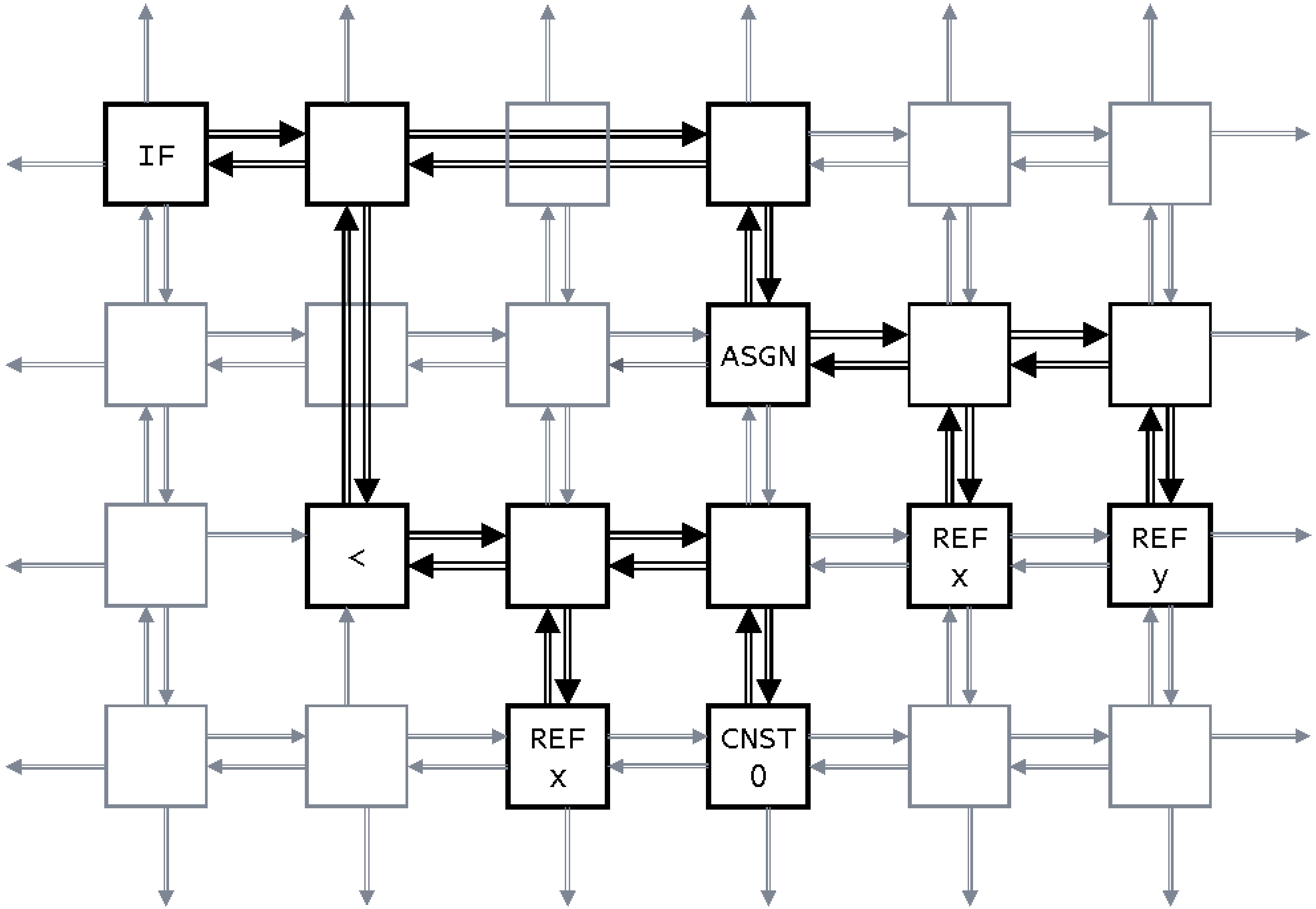
Program Tree:
Interstron C++ implementation (2)

Tree Lattice



Program Tree: Interstron C++ implementation (3)

```
if (x < 0) x = y;
```



Program Tree: Interstron C++ implementation (4)

Advantages

- High regularity, simple and obvious structure. It's quite easy to create a structure for any kind of language construct.
- Easy-to-use: all processing functions are written using the same pattern.

Disadvantages:

- Low level: no semantics – just structure.
- Low code reuse: for structurally similar sub-trees we have to write separate processing functions.
- A lot of empty nodes that connect significant nodes.

AST Implementation: CCI approach (1)

CCI – Common Compiler Infrastructure

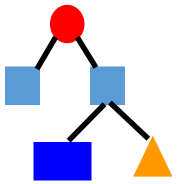
- Developed in Microsoft
- The author: Herman Venter (*now in Facebook* ☺)
- Used in experimental Microsoft projects: e.g., Cω, Spec#, Xen languages are implemented using CCI.

Main functions:

- Provides an extendable tree for C#-like languages' representation.
- The tree gets built as a **hierarchy of classes**, corresponding to the main language notions.
- Provides a few base tree traversers (walkers).
- Automates MSIL code generation: the last tree walker
- Supports compiler integration into Visual Studio.

AST implementation: CCI approach (2)

Tree structure: a «pure» tree, without attributes

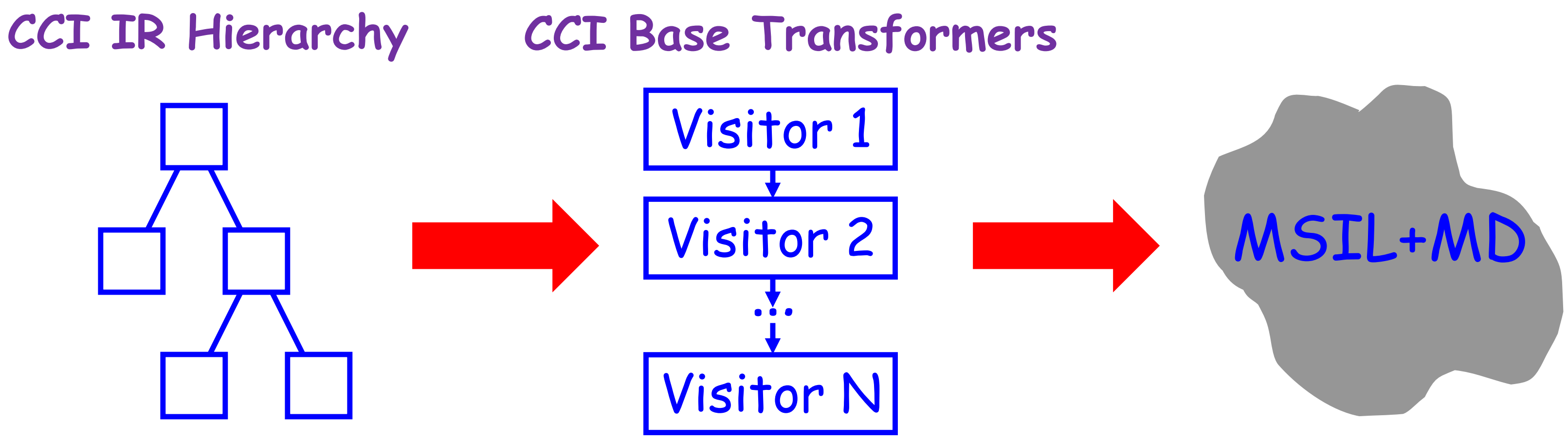


```
public class If : Statement
{
    Expression condition;
    Block      falseBlock;
    Block      trueBlock;
}
```

Traversing algorithms (Visitor pattern)

```
namespace System { namespace Compiler {
  public class Looker {
     public Node Visit ( Node node )
     {
       switch ( node.NodeType ) {
         case NodeType.If:
           // working with If node
           return SomeFunctionForIf(node);
        case NodeType.While:
           // working with While node
           return SomeFunctionForWhile(node);
        ...
      }
    }
  }
}
```

AST implementation: CCI approach (3)



Advantages:

- Flexibility: easily add and modify transformers, change their order without changing class hierarchy.

Disadvantages:

- Hard to refactor: if class hierarchy changes you have to modify all transformers correspondingly.

Tables AND/OR trees? (1)

Symbol Table:

- ST is filled while processing declarations.
- ST have a linear structure.
- After completing processing declarations ST *does not change*.
- While further processing ST *does not change*.
- Typical actions on ST: adding new element; **look up**.

Program Tree:

- It gets constructed in accordance with the static construct nesting (tree form)
- It is constructed while parsing "executable" parts of the source program.
- After creation it is *actively modified*.
- Typical actions: recursive traversing, re-constructing.

Tables AND/OR trees? (2)

However:

- In modern languages declarations & statements have the same status: they can be mixed.
- Tables reflect visibility scopes and therefore they are hierarchical – i.e., they compose *a tree*.
- Symbol table tree is *structurally identical* to the tree of "executable" program parts.
- Symbol tables & program tree are closely related. An example: initializers in declarations.

=> There are obvious reasons to create <u>the single structure</u> instead of two: **join tables and trees**.

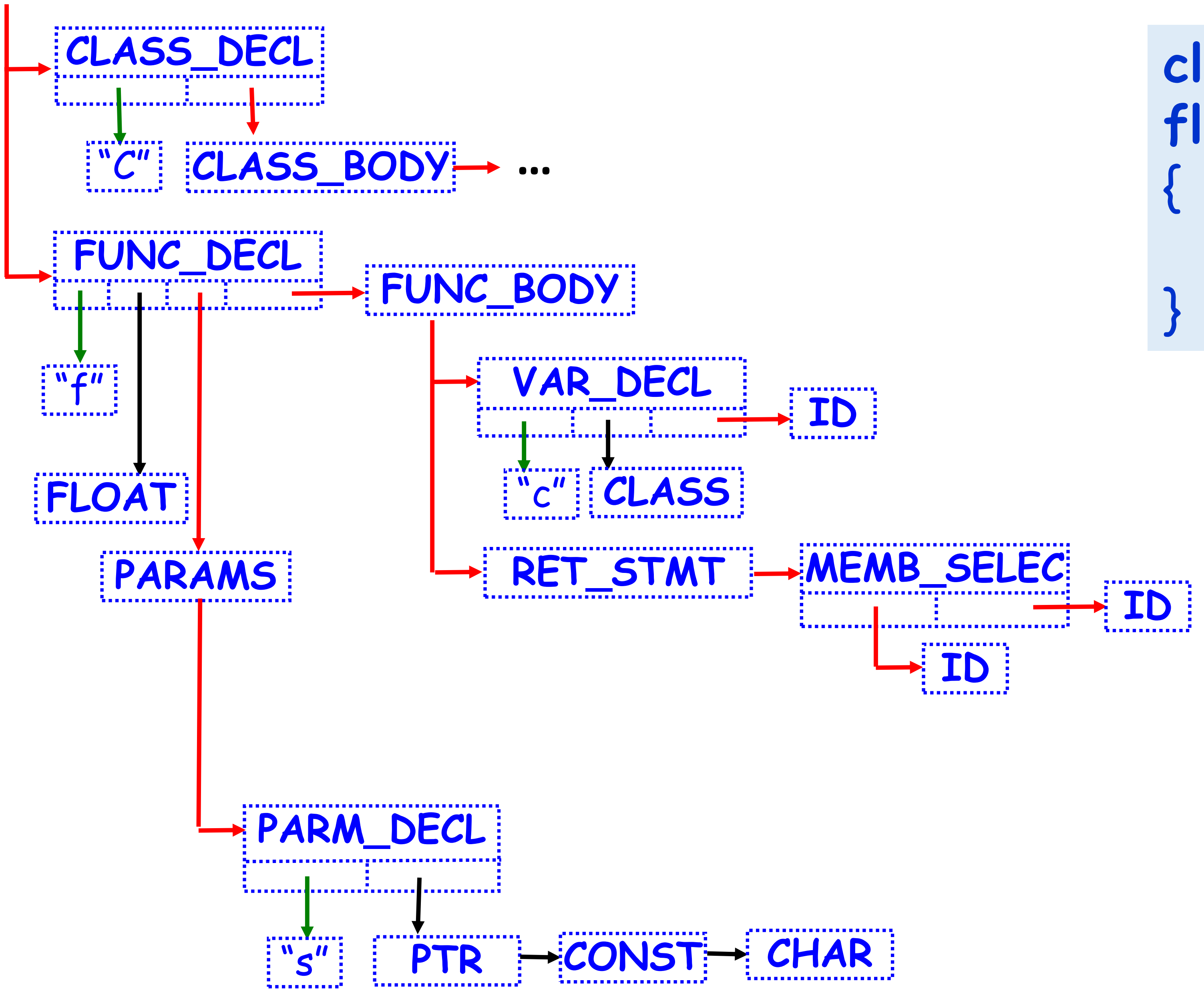
AST Implementation: an integral approach

Main project decision:

- Each program tree node contains **both** structure (its parts) **and full set or operators** on the given node (and its sub-trees).

```
public class If : Statement
{
    // Sub-tree structure
    Expression condition;
    Block      falseBlock;
    Block      trueBlock;
    // Attributes
    ...
    // Operations on sub-trees
    override bool validate()
    {
        if ( !condition.validate() ) return false;
        if ( falseBlock != null && !falseBlock.validate() ) return false;
        if ( !trueBlock.validate() ) return false;
        // Checking ‘condition’
        // Other semantic checks...
        return true;
    }
    override void generate()
    {
        condition.generate();
        trueBlock.generate();
        if ( falseBlock != null ) falseBlock.generate();
        ...
    }
}
```

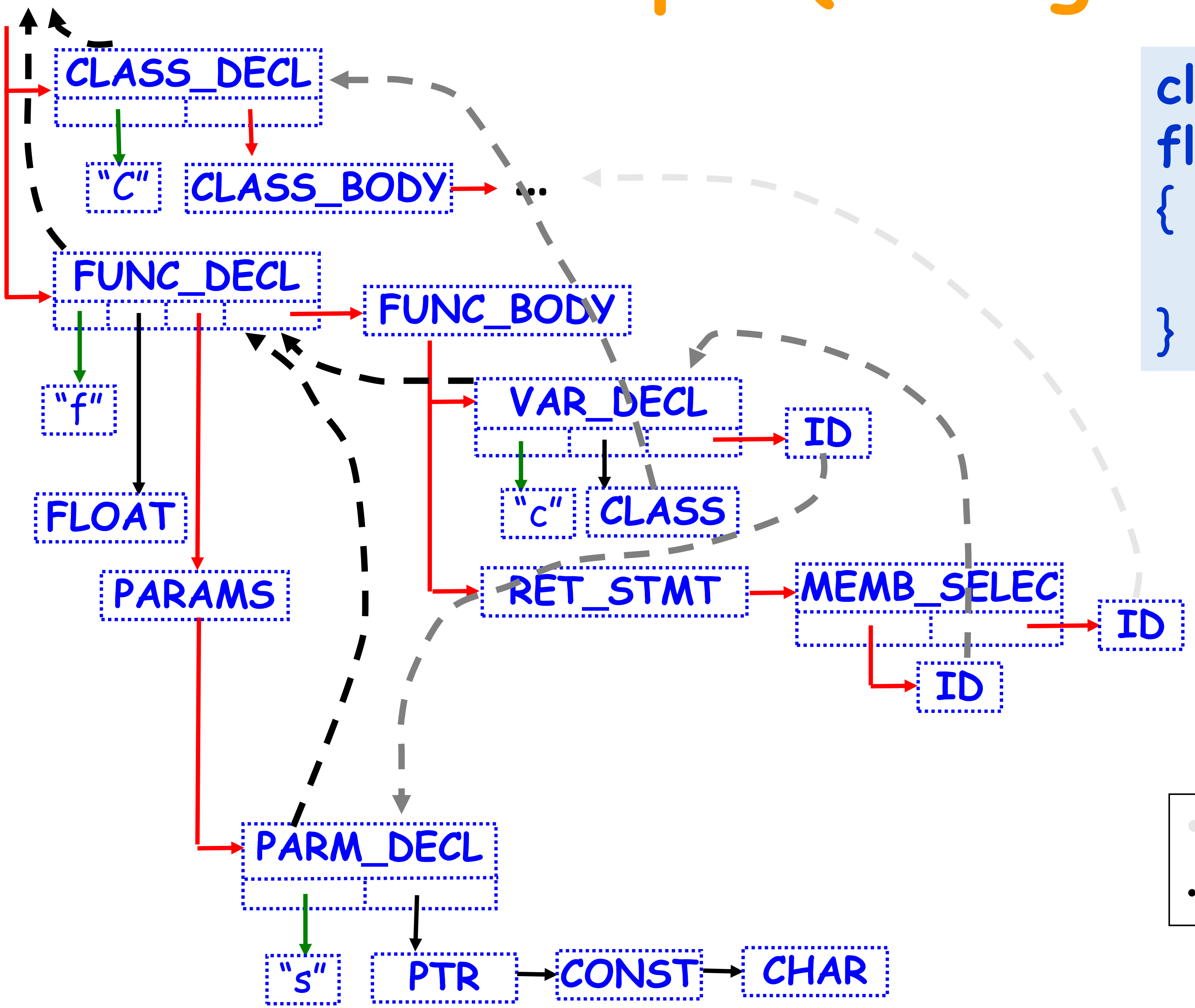
AAST Example (a fragment)




```
class C { … };
float f(const char* s)
{
   C c(s); return c.m;
}
```

- Structural links
- Type information
- Attributes

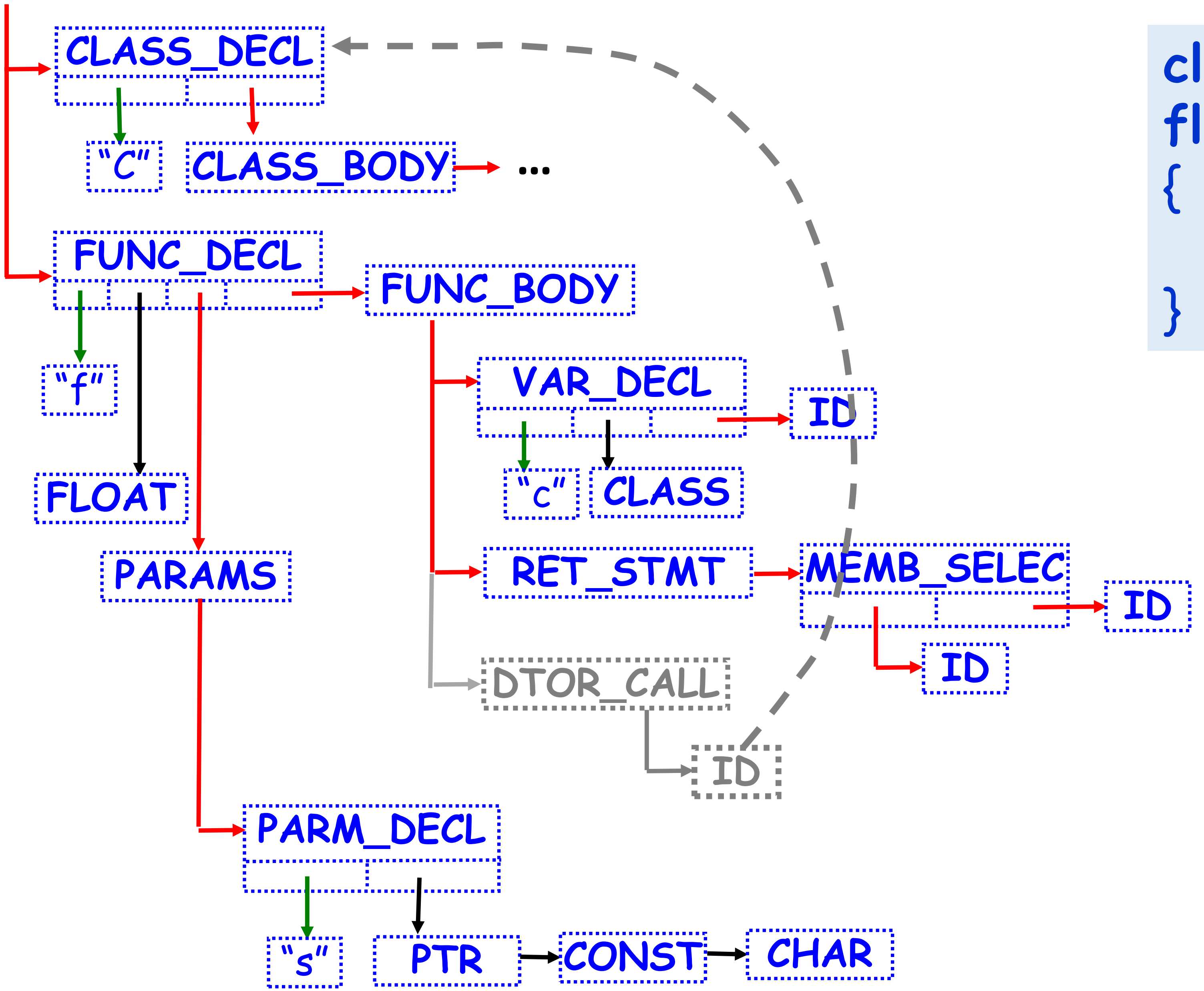
AAST Example (a fragment)




```
class C { … };
float f(const char* s)
{
   C c(s); return c.m;
}
```

- Semantic links
- Scopes

AAST Example (a fragment)




```
class C { ... };
float f(const char* s)
{
   C c(s); return c.m;
}
```

- Hidden semantics

Type Representation (1)

C++ type system:

- Fundamental types: integer, float, character, ...
- Class and enumeration types
- Type modifiers: constants, pointers, references, pointers to class members
- Functional types, arrays
- Families of types (templates)

Many ways for defining new types, for example:

- Reference to pointer `int*& rp = p;`
- Pointer to function `double& (*f)(const C*);`
- Array of pointers to pointers to class members

  `C<int,float>::*char A[10];`

Many complex & non-obvious conversion rules

Type Representation (2)

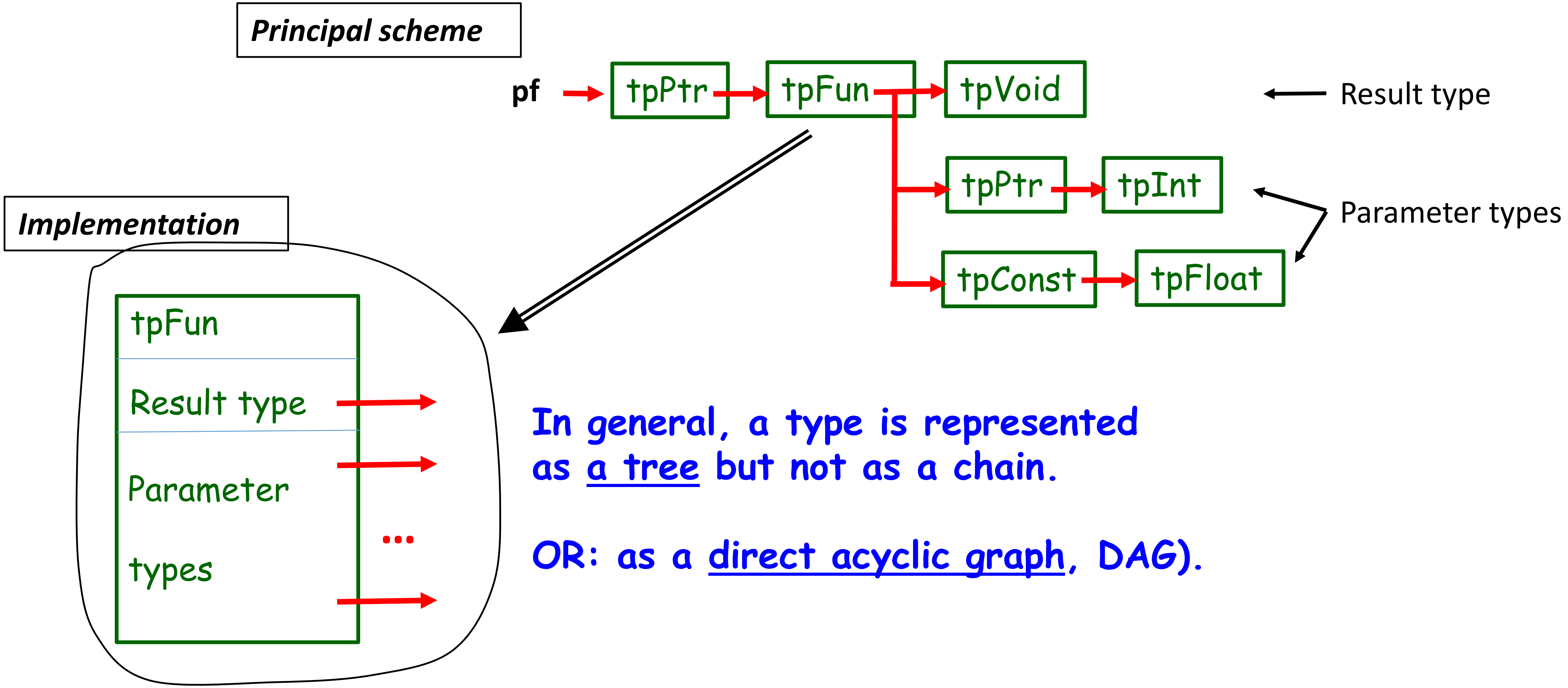
Solution for C++:

- Represent types as **type chains**

int	tpInt
int*	tpPtr,tpInt
long unsigned int**	tpPtr,tpPtr,tpULI
const int	tpConst,tpInt
const int*	tpPtr,toConst,tpInt
const int *const	tpConst,tpPtr,tpConst,tpInt
const C*[10]	tpArr,10,tpPtr,tpConst,tpClass,C
int& (*f)(float)const	tpPtr,f
C::*int	tpPtrMemb,C,tpInt
…	
f	tpMembFun,tpRef,tpInt,1,tpFloat

Type Representation: Example

```
typedef void (*pf)(int*, const float);
```



In general, a type is represented as a tree but not as a chain.

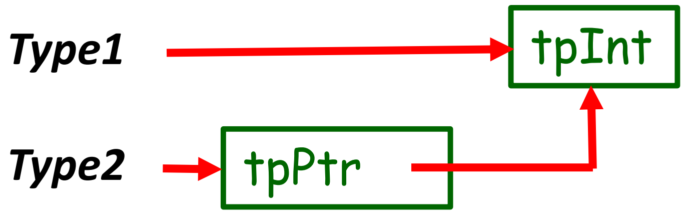
OR: as a direct acyclic graph, DAG).

Operations on Types: Examples

```
int* p = &x;
```

Взятие адреса:

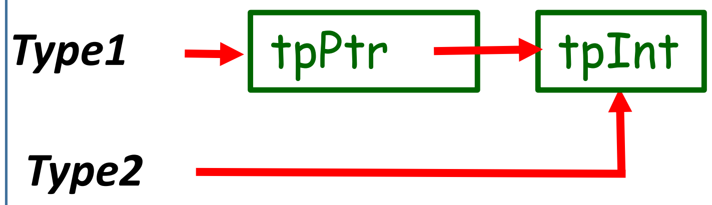
int -> int*




```
int v = *p;
```

Разыменование:

int* -> int

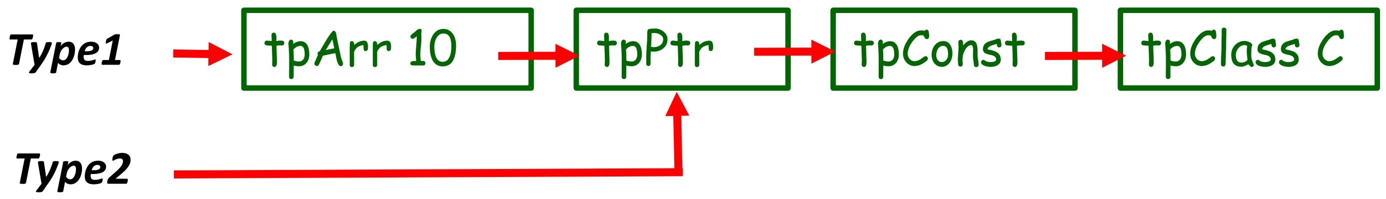



```
class C { … };
const C* A[10];
…
const C* a = A[3];
```

Доступ к типу элементов массива:

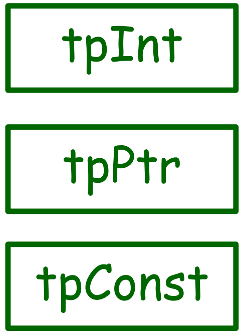
tpArr,10,tpPtr,tpConst,tpClass,C ->

tpPtr,tpConst,tpClass,C



Type Representation

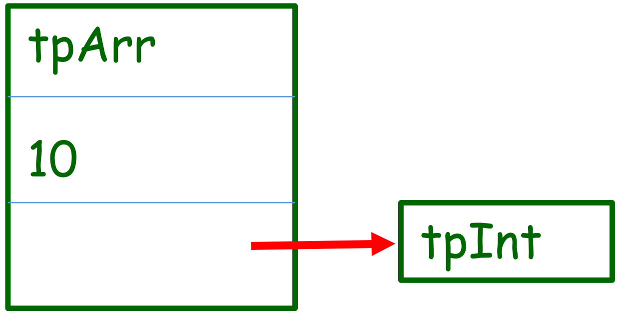
Fundamental types:



Just the single code

Compound type: array

int[10]



Compound type: class

class C