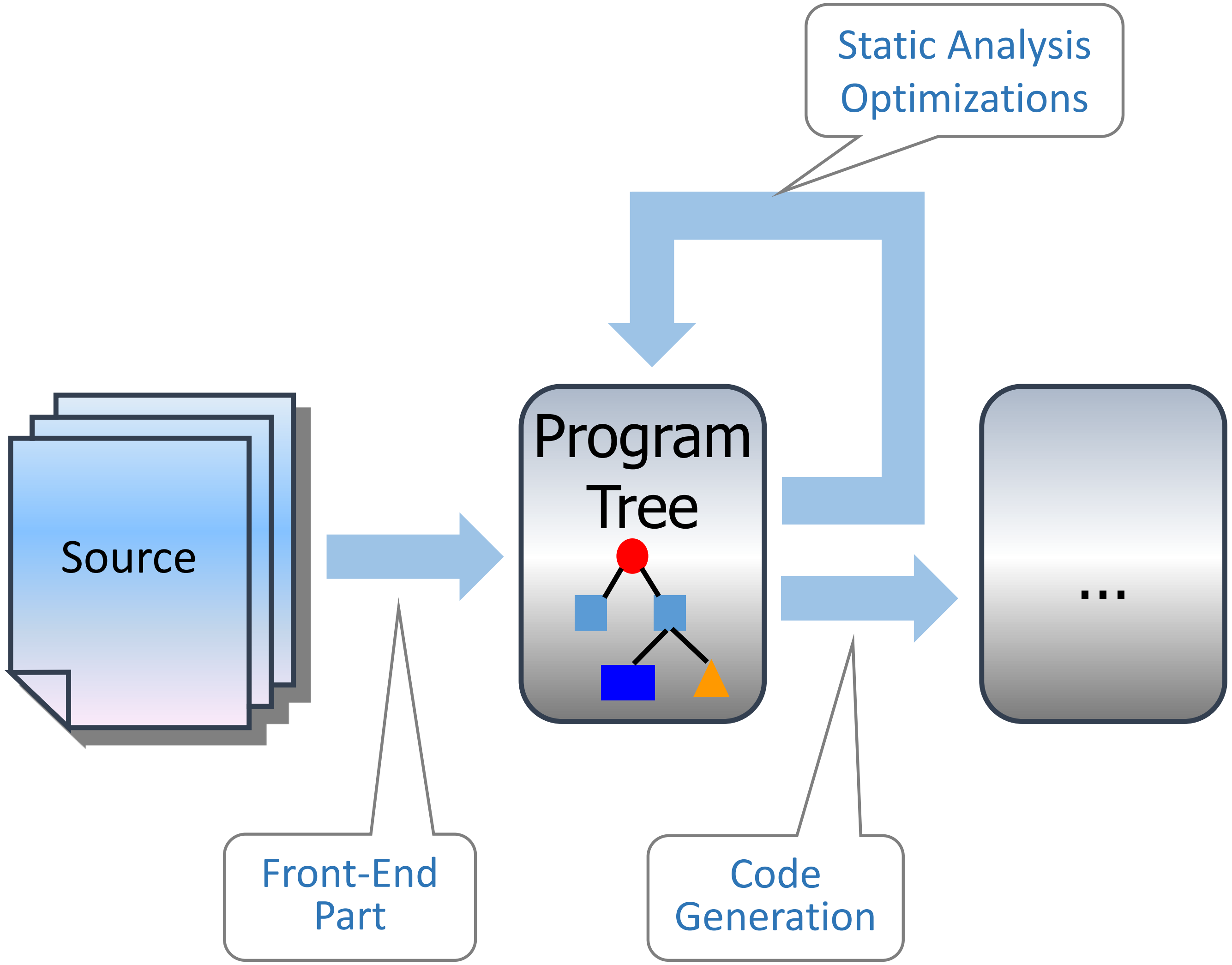
Compiler Construction: Practical Introduction

Lecture 4
Compilation Structures

Eugene Zouev

Spring Semester 2022
Innopolis University

Compilation structures



Compilation structures

What for:

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

Compilation structures

What for:

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

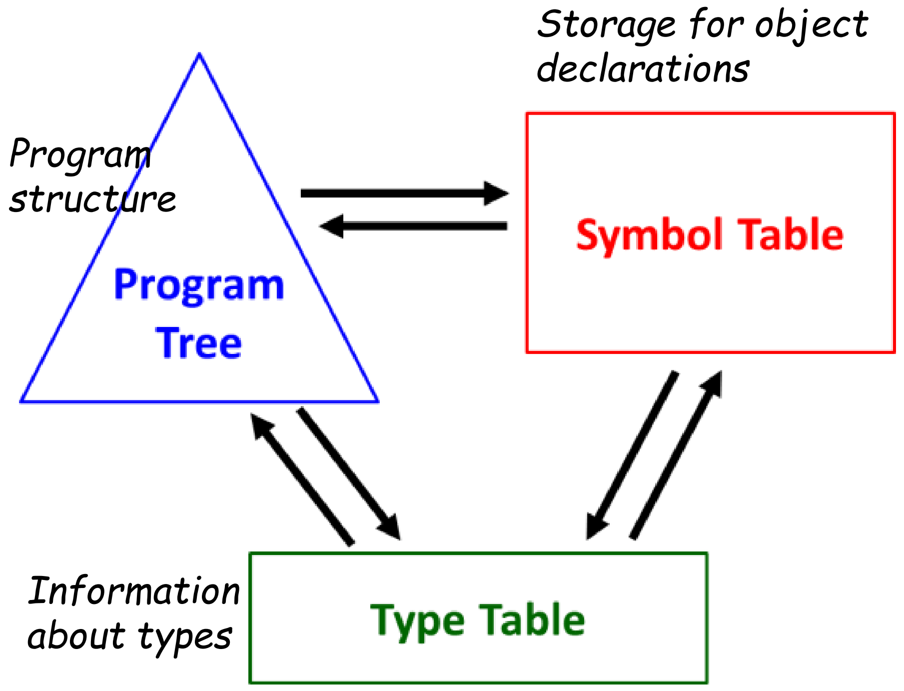
Compilation structures

What for:

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

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

- Old compiler example
- Old language (Ada)
- Old implementation platform
- *…But still good as an example* ☺

Symbol table

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

- Old compiler example
- Old language (Ada)
- Old implementation platform
- *…But still good as an example* ☺

Symbol Table for Swap

1	3		0	**int** (index to TT)	**a** (index to LT)	hash link
1	3		0	**int** (index to TT)	**b** (index to LT)	hash link
1	4		Link to AST	**int** (index to TT)	**Temp** …	hash link

initializer

in / out / local

used / not in use

Nested blocks & symbol tables

```
double f(int x, int y)
{
    float a = 1.0, b = 2.0;
    if ( x > y )
    {
        float tmp = a+b;
        for (int i=0; i<10; i++)
            tmp += i;
        return tmp/y;
    }
    else
    {
        double m = a%b;
        return x*m;
    }
}
```

Nested blocks & symbol tables

```
double f(int x, int y)
{
    float a = 1.0, b = 2.0;
    if ( x > y )
    {
        float tmp = a+b;
        for (int i=0; i<10; i++)
            tmp += i;
        return tmp/y;
    }
    else
    {
        double m = a%b;
        return x*m;
    }
}
```

Nested blocks & symbol tables

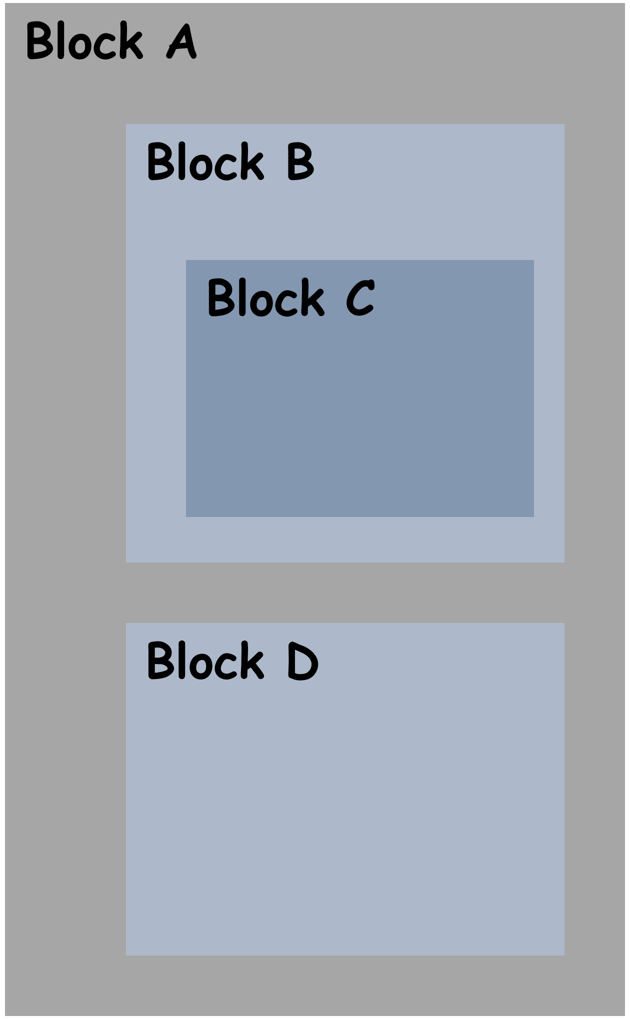
A

```
double f(int x, int y)
{
    float a = 1.0, b = 2.0;
    if ( x > y )
    {
        float tmp = a+b;
        for (int i=0; i<10; i++)
            tmp += i;
        return tmp/y;
    }
    else
    {
        double m = a%b;
        return x*m;
    }
}
```

B

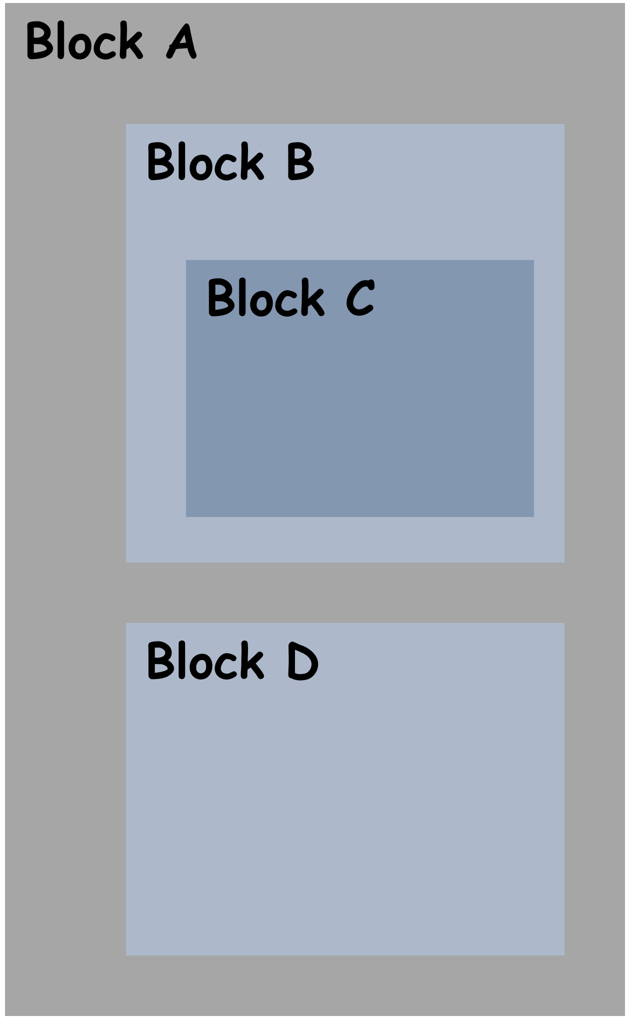
C

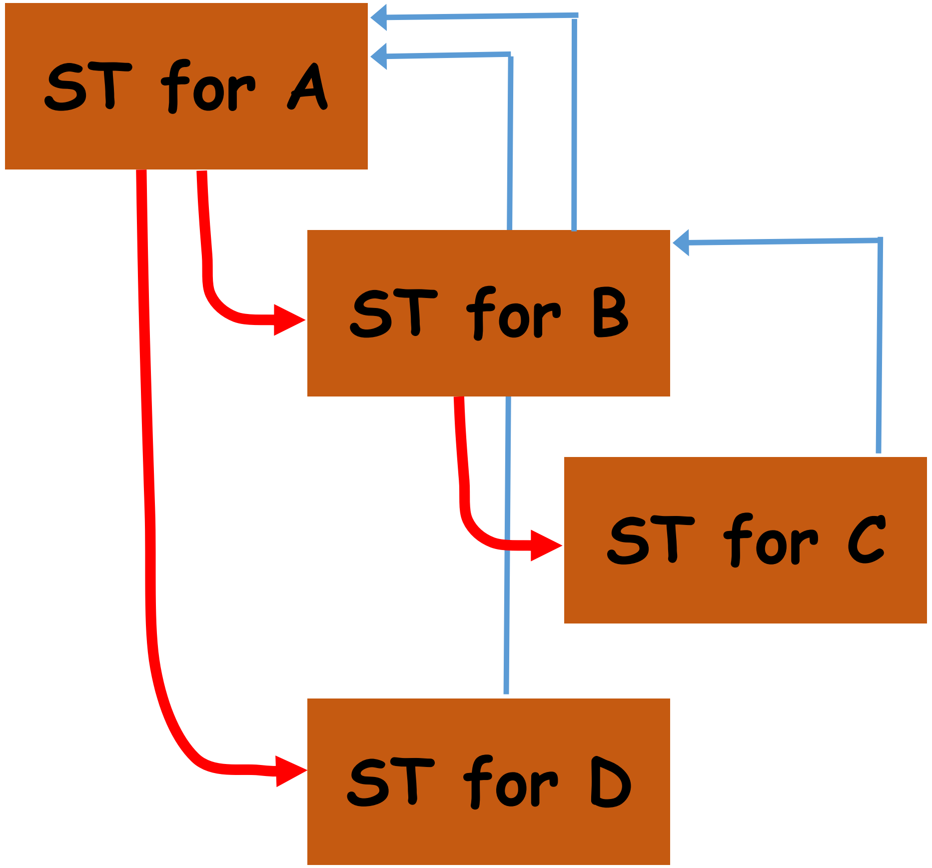
D



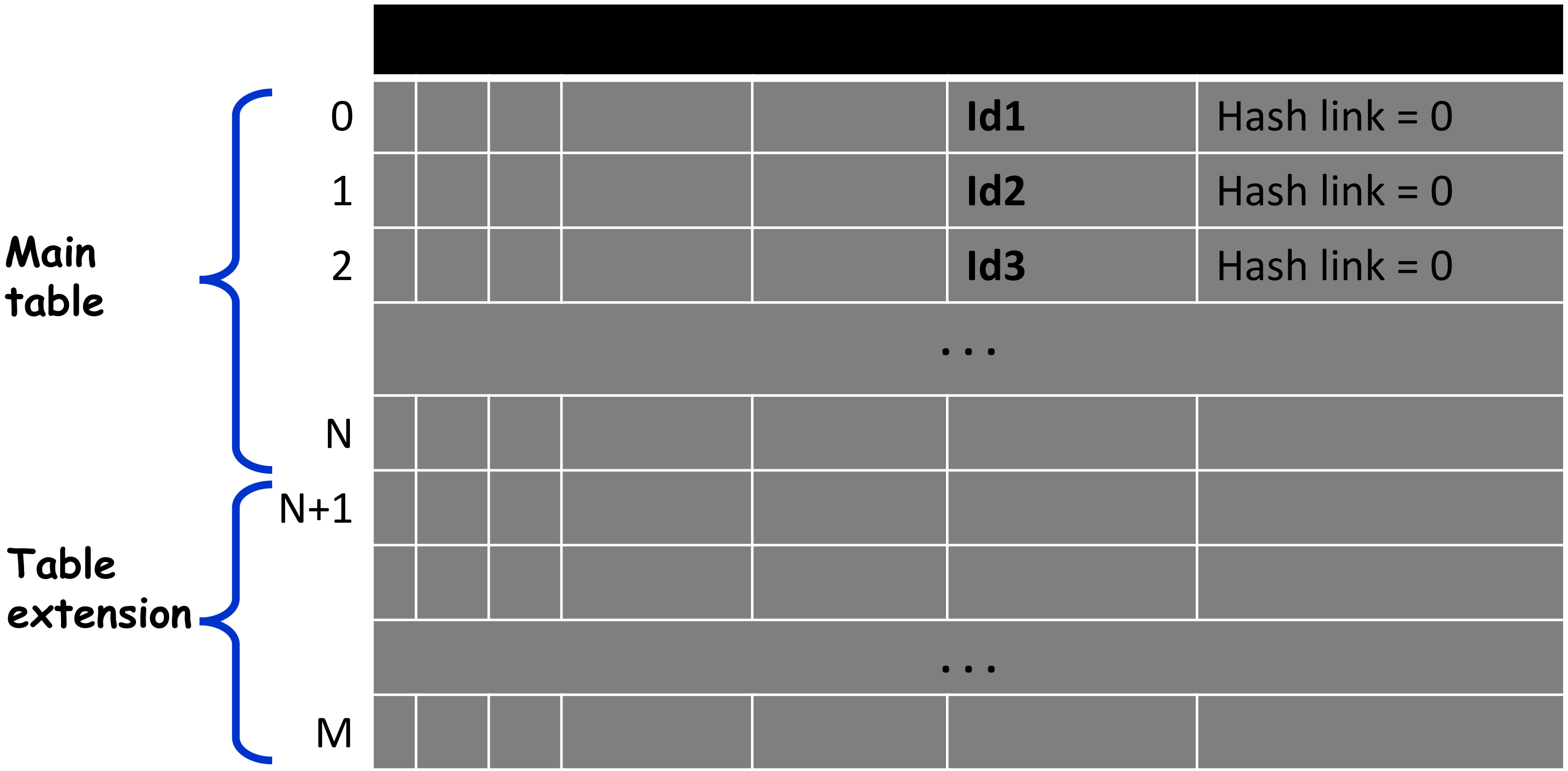
Nested blocks & symbol tables

```
double f(int x, int y)                     A
{
    float a = 1.0, b = 2.0;
    if ( x > y )
    {                                      B
        float tmp = a+b;
        for (int i=0; i<10; i++)
            tmp += i;                      C
        return tmp/y;
    }
    else
    {                                      D
        double m = a%b;
        return x*m;
    }
}
```



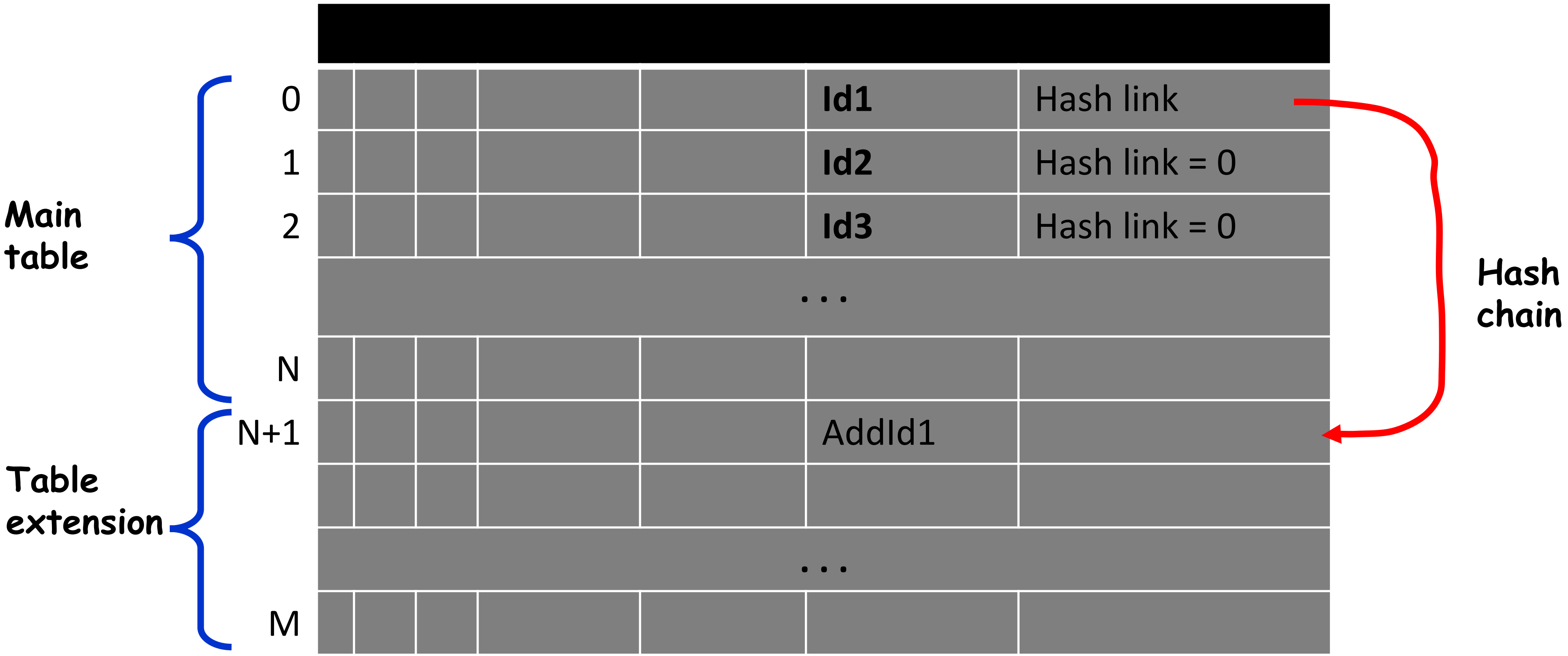


Hash tables



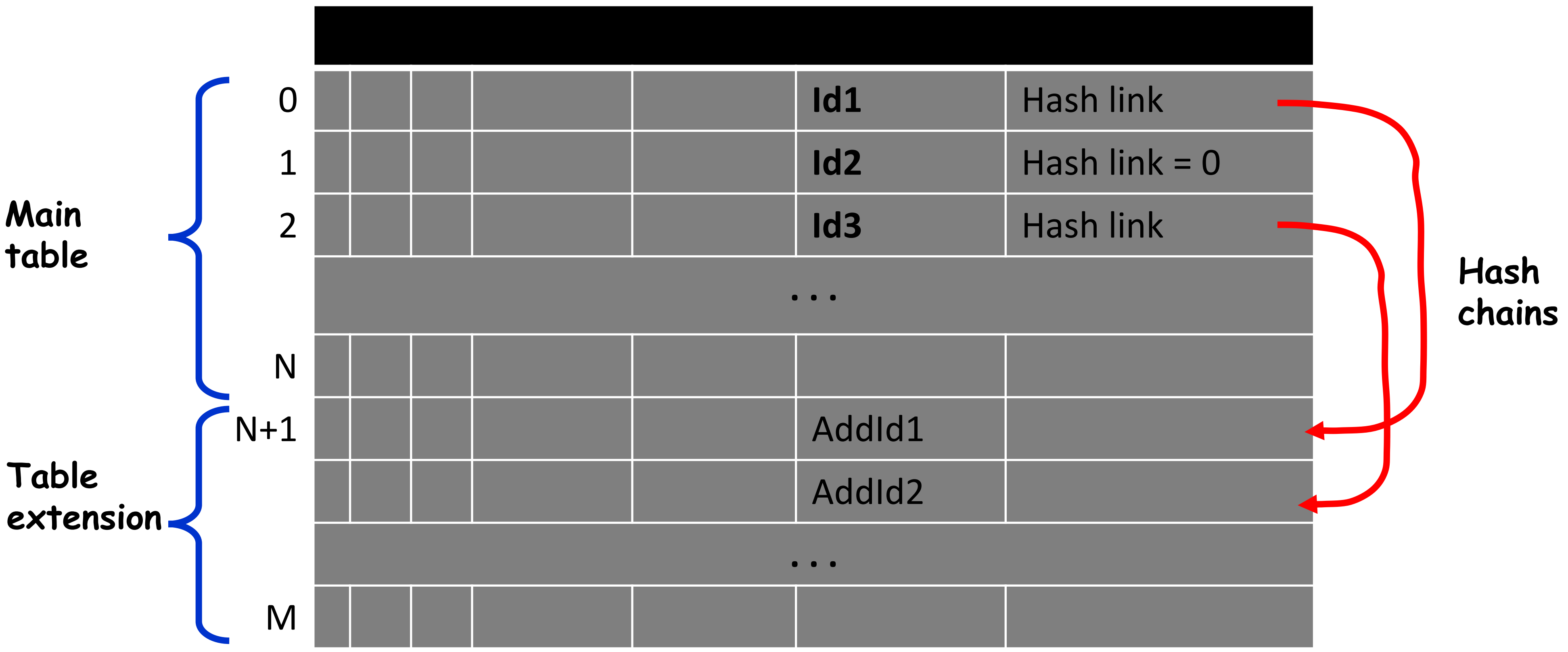
Main table
Table extension
0
1
2
N
N+1
M
Id1
Hash link = 0
Id2
Hash link = 0
Id3
Hash link = 0
. . .
. . .

Hash tables



Hash("Id1") = Hash("AddId1")

Hash tables



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"
    }
}
```

Tables AND/OR trees? (1)

Symbol Table:

- Each ST is filled while processing declarations.
- Each 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**.

Tables AND/OR trees? (1)

Symbol Table:

- Each ST is filled while processing declarations.
- Each 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.

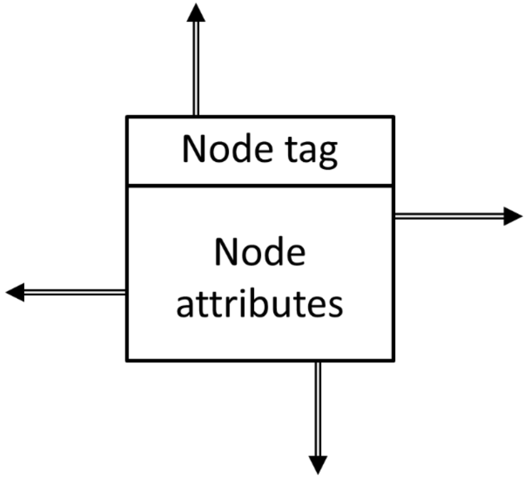
Tables AND/OR trees? (2)

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 the single structure instead of two: **join tables and trees**.

Program tree: Interstron C++ implementation (1)

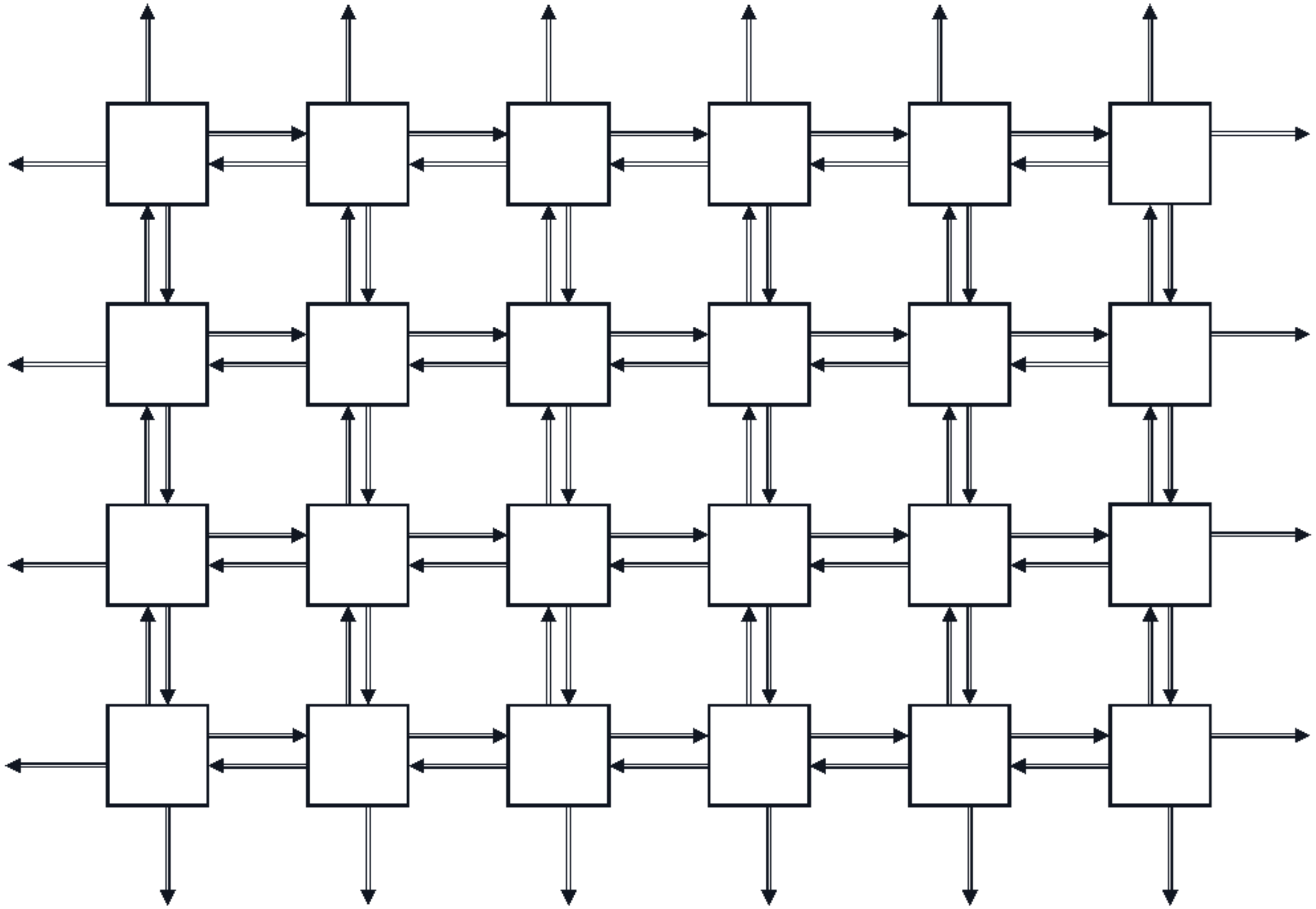



```cpp
struct Node {
    int Tag;
    Node* left;
    Node* right;
    Node* up;
    Node* down;
    void* semantics;
};
```

The tree node is a small structure:

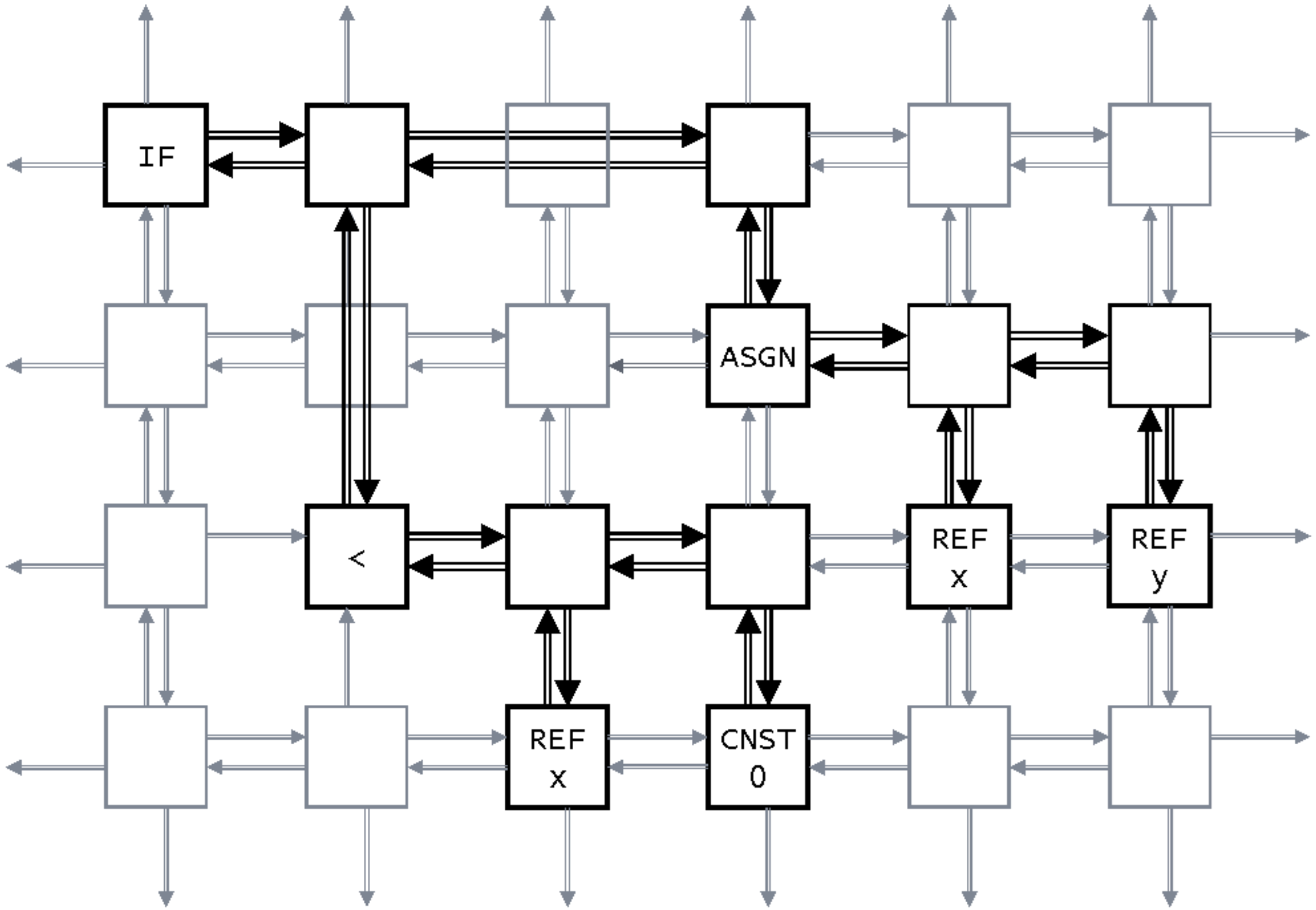
- The unique node tag.
- Each tag represents a particular language construct.
- There are four pointers to make links between nodes: «up», «down», «left», «right».
- Each node has a set of attributes; attributes depend on node's tag.
- There are "empty" nodes (without semantics) for organizing complex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

Program tree: Interstron C++ implementation (2)



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

CCI – Common Compiler Infrastructure

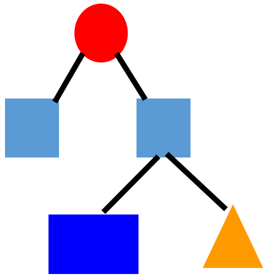
- Developed in Microsoft
- The author: Herman Venter (*now in Facebook* ☺)
- Used in experimental Microsoft projects: e.g., Cω, Spec#, Xen languages are implemented using CCI.

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

Tree structure:
a «pure» tree

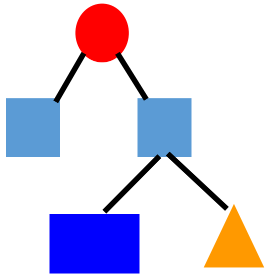


```
public class If : Statement
{
    Expression condition;
    Block      falseBlock;
    Block      trueBlock;
}
```

Pure subtree structure

AST implementation: CCI approach

Tree structure:
a «pure» tree



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

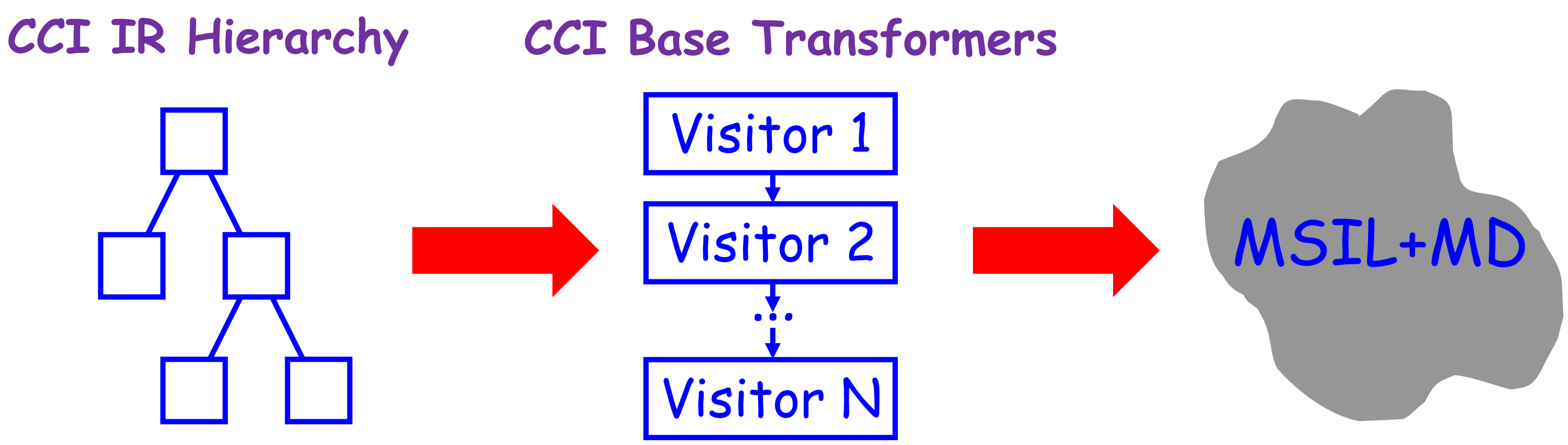
Pure subtree structure

Traversing algorithms
(Visitor pattern)

```
namespace System.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);
       ...
     }
  }
}
```

Pure functionality on the subtree

AST implementation: CCI approach



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.

AST implementation: an integral approach (1)

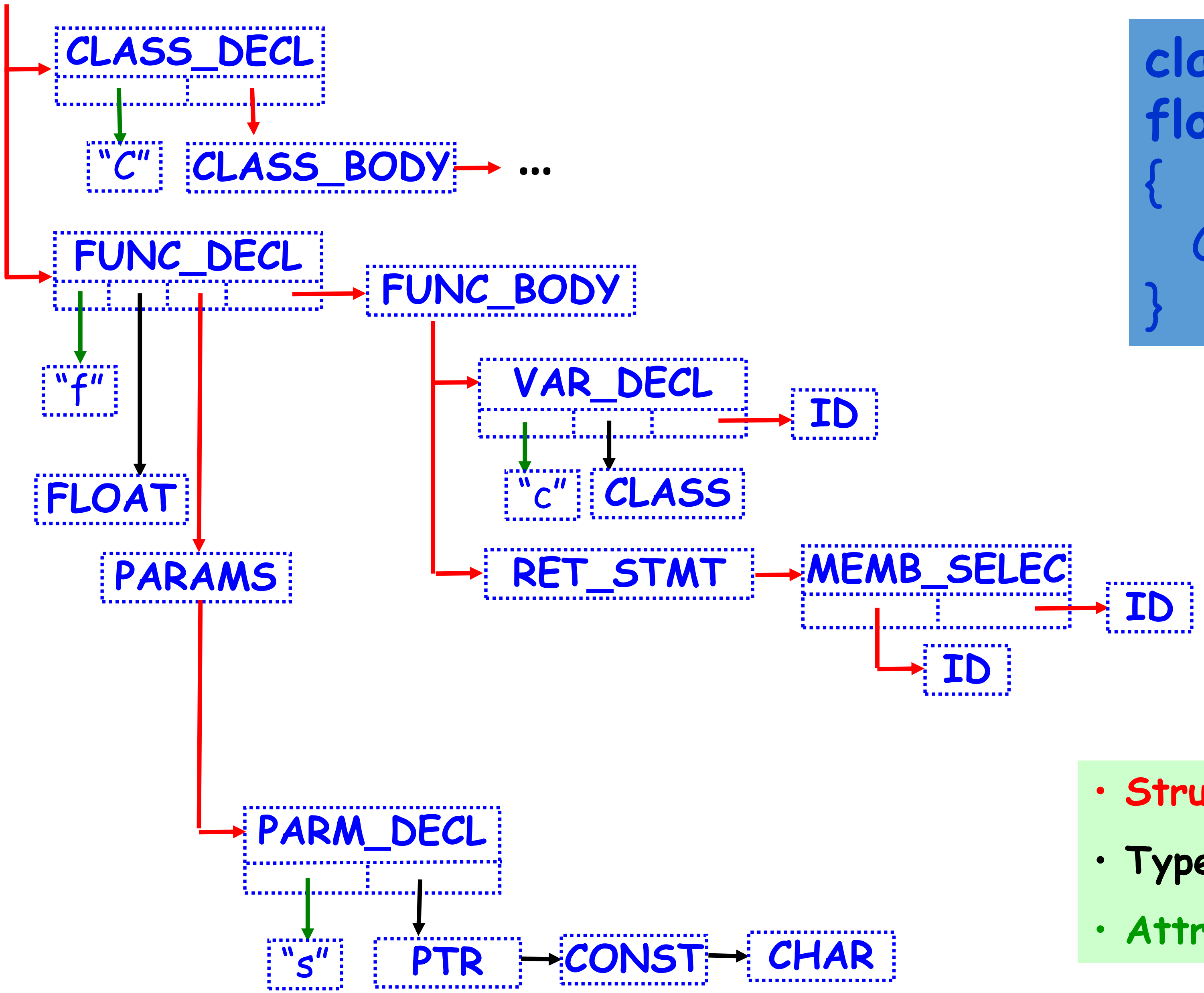
Main project decision:

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

AST implementation: an integral approach (2)

```
public class If : Statement
{
    // Sub-tree structure
    Expression condition;
    Block      falseBlock;
    Block      trueBlock;
    // 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();
        ...
    }
}
```

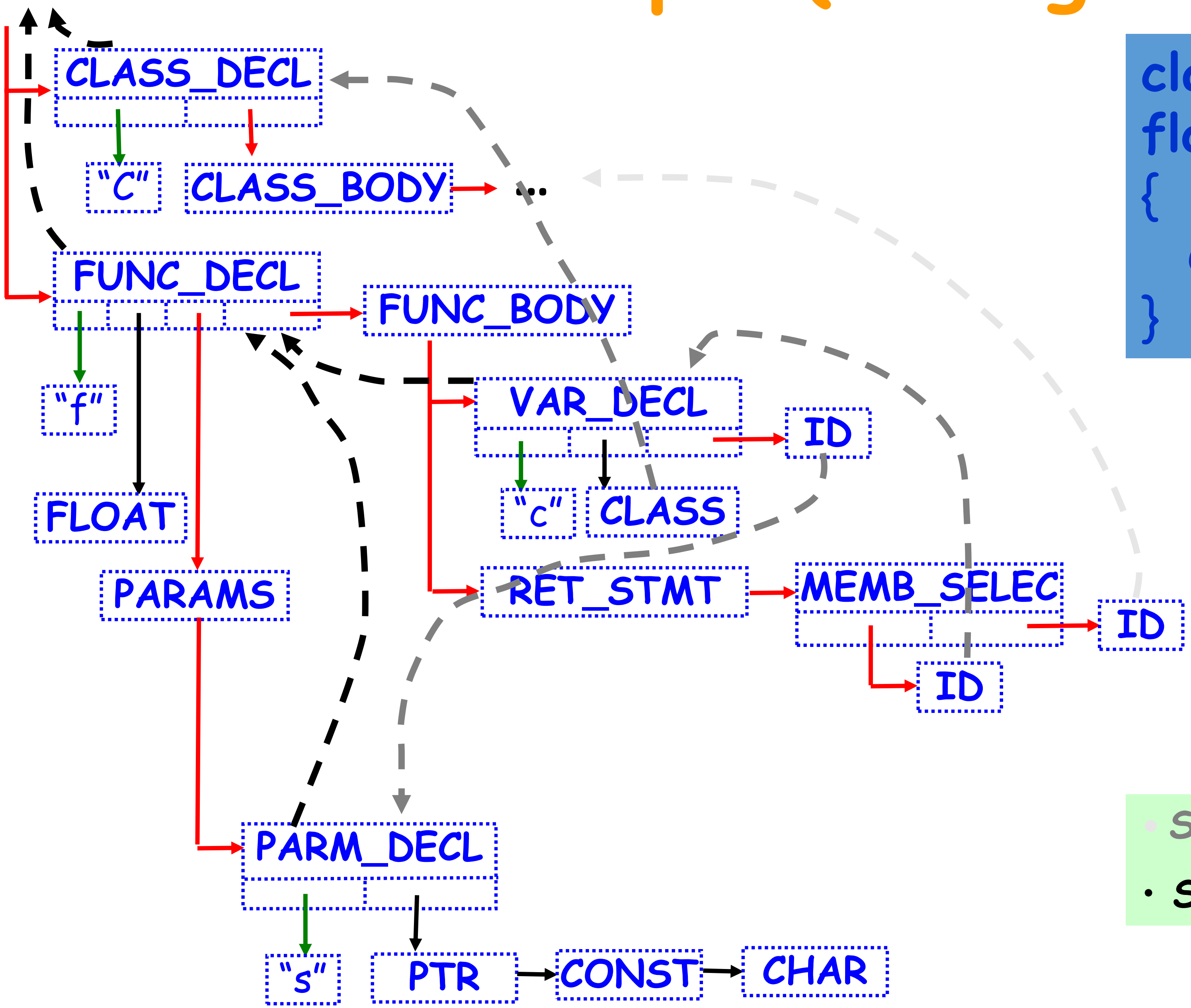
AAST example (a fragment)




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

- Structural links
- Type information
- Attributes

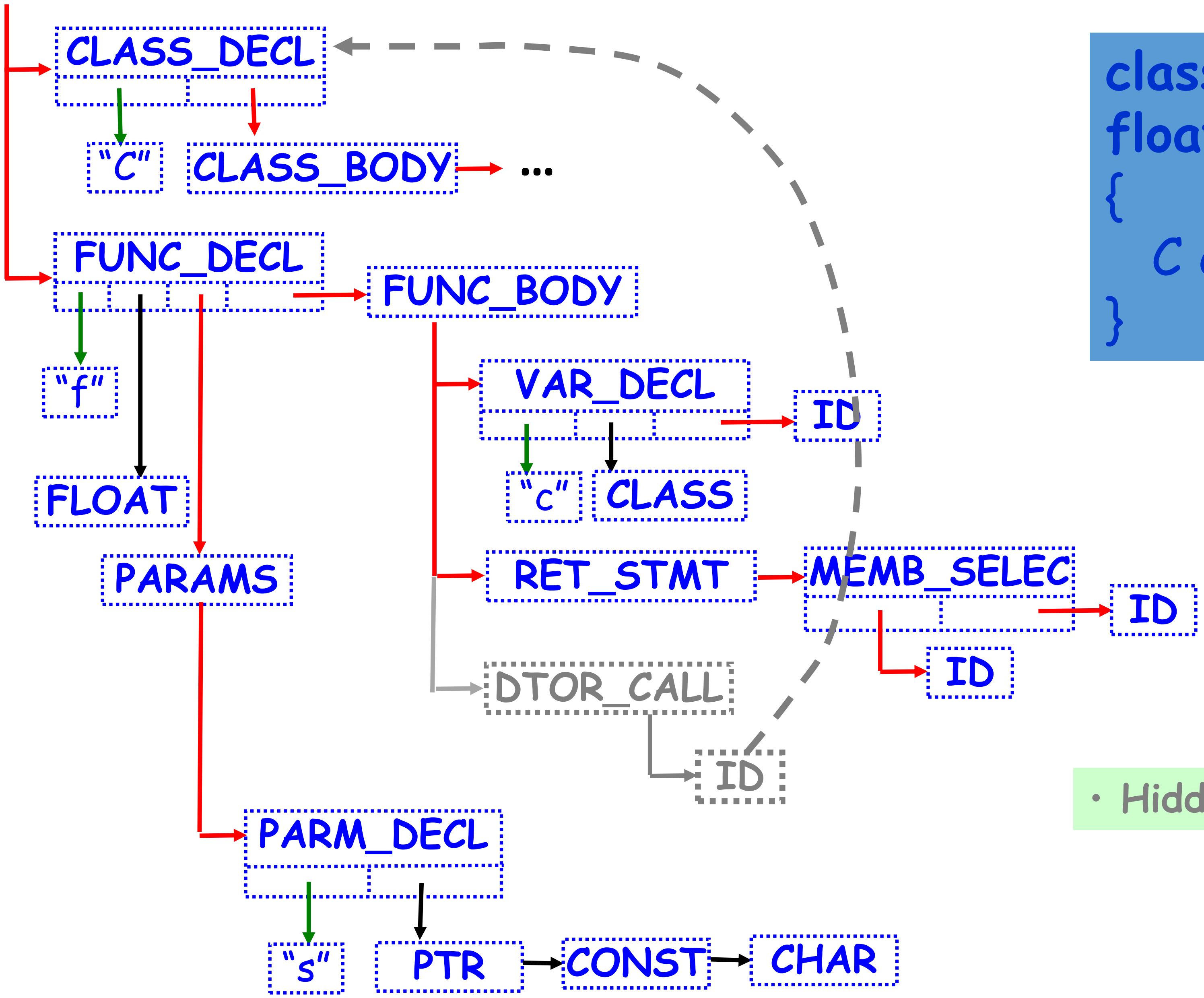
AAST example (a fragment)




```
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)

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];

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**

integer

pointer to integer

pointer to pointer …

constant integer

pointer to constant integer

constant pointer to constant integer

```
int
int*
long unsigned int**
const int
const int*
const int *const
const C*[10]
int& (*f)(float)const
C::*int
…
f
```

Type representation (2)

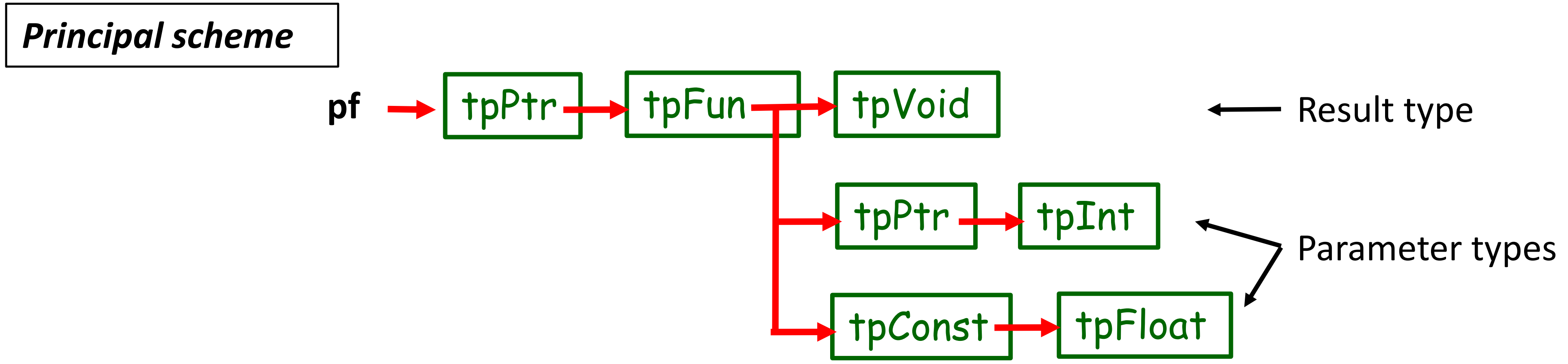
Solution for C++:

- Represent types as **type chains**

integer	int	tpInt
pointer to integer	int*	tpPtr, tpInt
pointer to pointer …	long unsigned int**	tpPtr, tpPtr, tpULI
constant integer	const int	tpConst, tpInt
pointer to constant integer	const int*	tpPtr, toConst, tpInt
constant pointer to constant integer	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

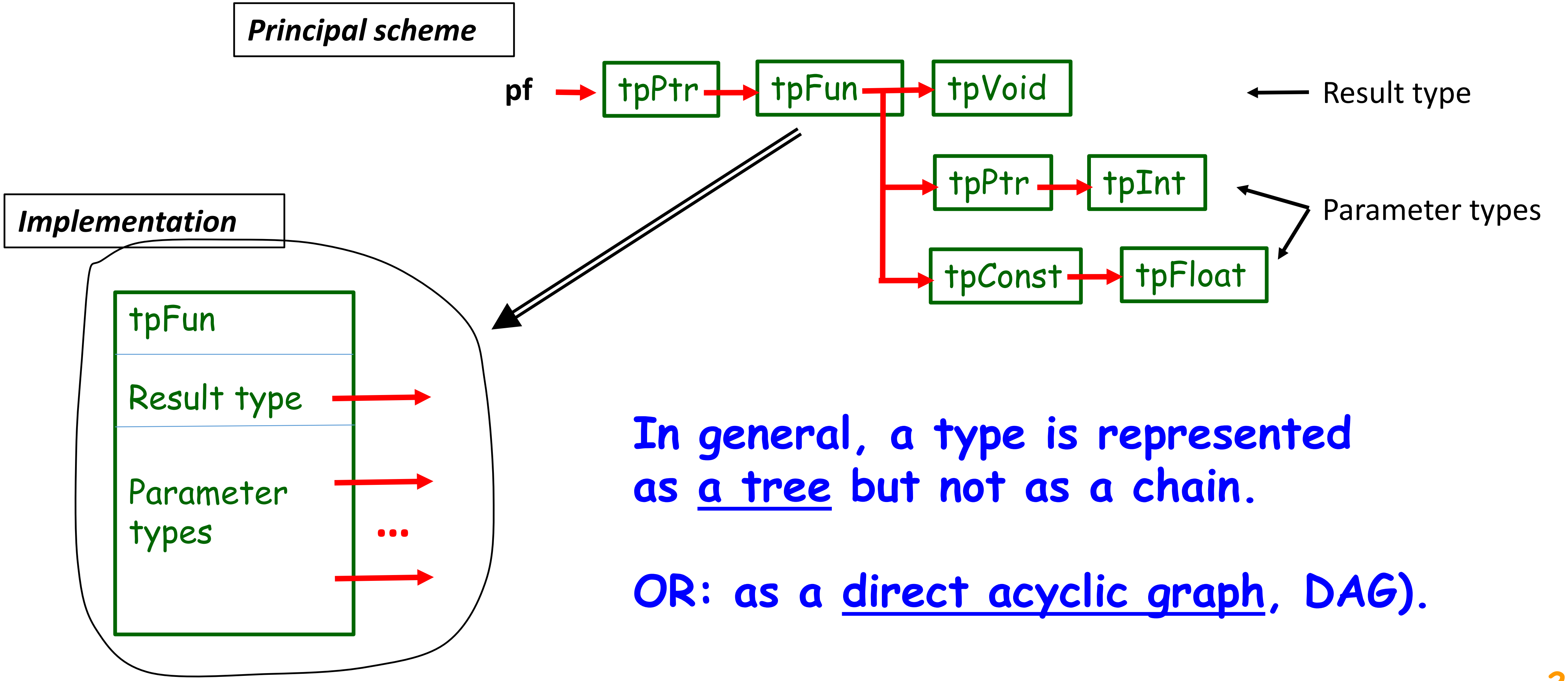
Type Representation: Example

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



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;
```

Taking address:
int -> int*

Type1 → tpInt

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

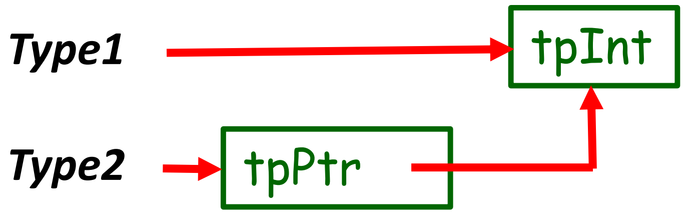
Dereferencing:
int* -> int

Type1 → tpPtr → tpInt

Operations on Types: Examples

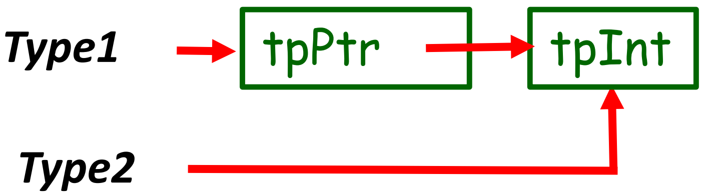
`int* p = &x;`

Taking address:
int -> int*



`int v = *p;`

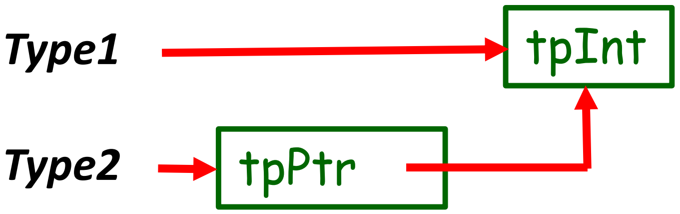
Dereferencing:
int* -> int



Operations on Types: Examples

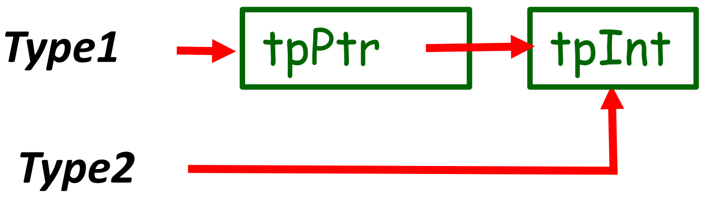
```
int* p = &x;
```

Taking address:
int -> int*



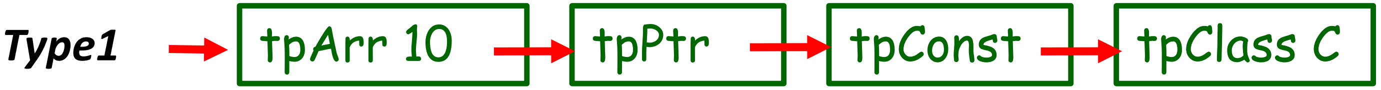

```
int v = *p;
```

Dereferencing:
int* -> int




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

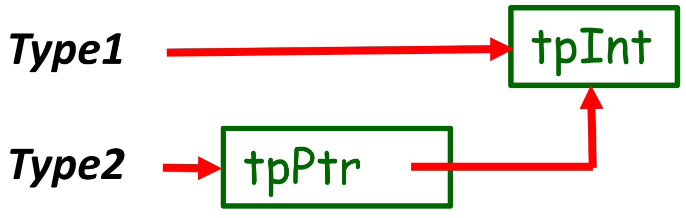
Access to a type of an array element:
tpArr,10,tpPtr,tpConst,tpClass,C ->
tpPtr,tpConst,tpClass,C



Operations on Types: Examples

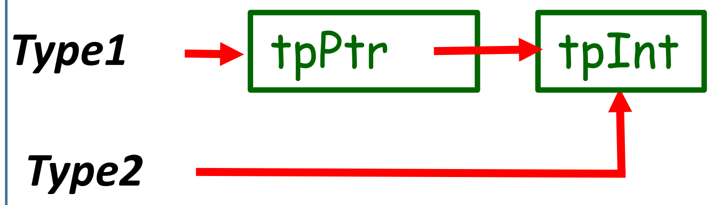
```
int* p = &x;
```

Taking address:
int -> int*



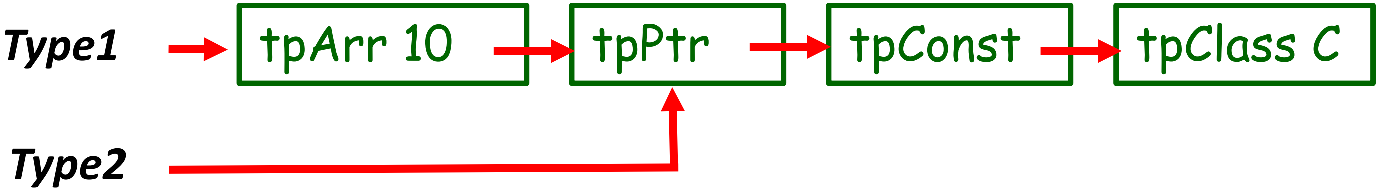

```
int v = *p;
```

Dereferencing:
int* -> int




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

Access to a type of an array element:
tpArr,10,tpPtr,tpConst,tpClass,C ->
tpPtr,tpConst,tpClass,C



Type Representation

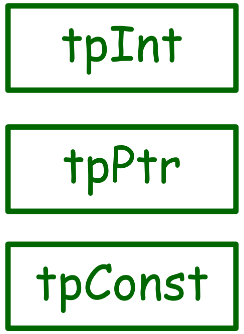
Fundamental types:

tpInt
tpPtr
tpConst

Just the single code

Type Representation

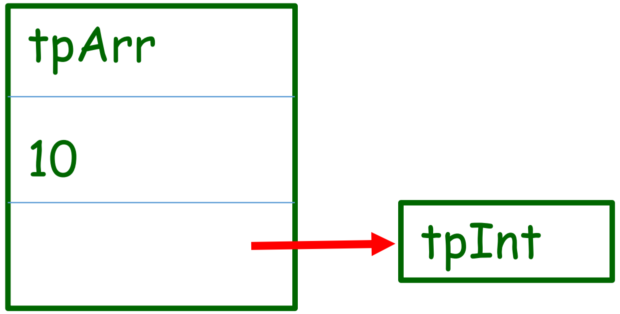
Fundamental types:



Just the single code

Compound type: array

int[10]



Compound type: class

class C