System Software Crash Couse

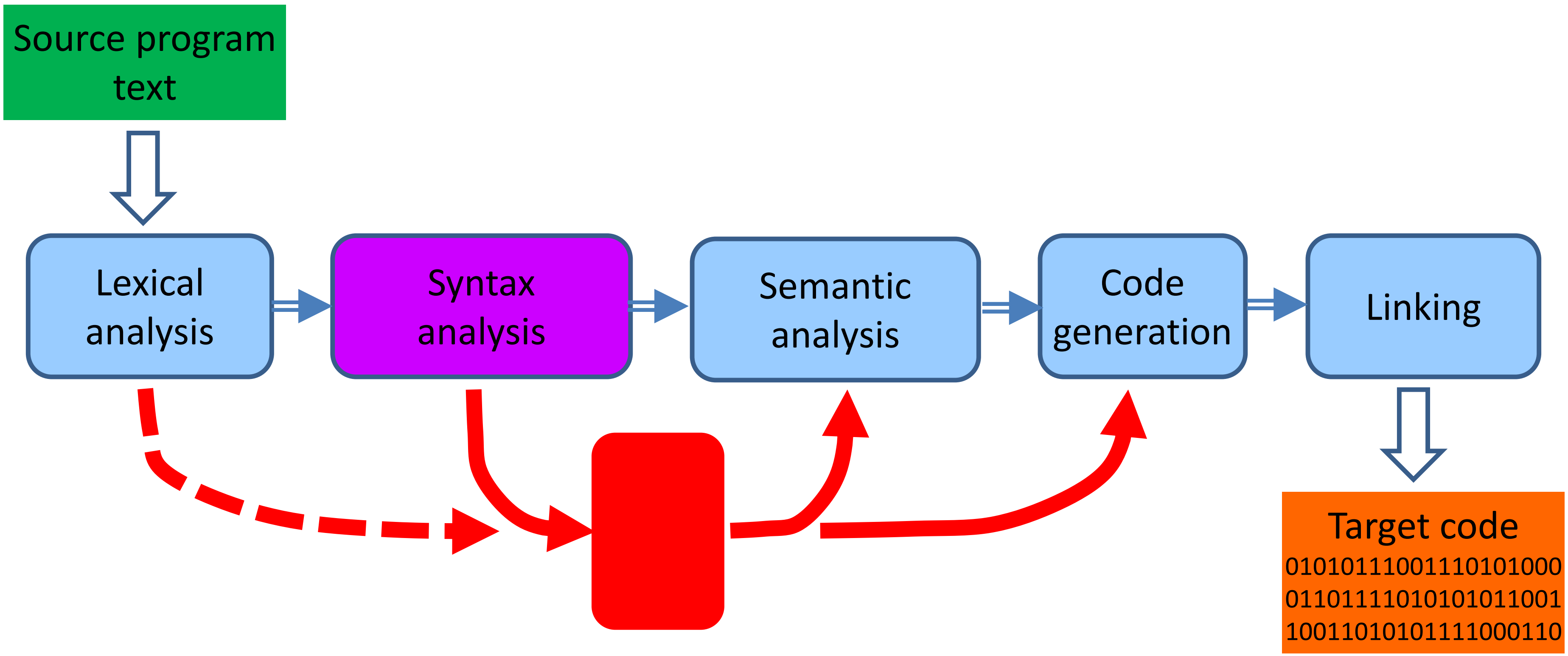
Samsung Research Russia
Moscow 2019

Block C Compiler Construction
4-5. Syntax Analysis & Compilation Structures

Eugene Zouev

Compilation: An Ideal Picture

A program written by a human
(or by another program)



A program binary image
suitable for immediate
execution by a machine

(c) Copyright 2018, E.Zouev

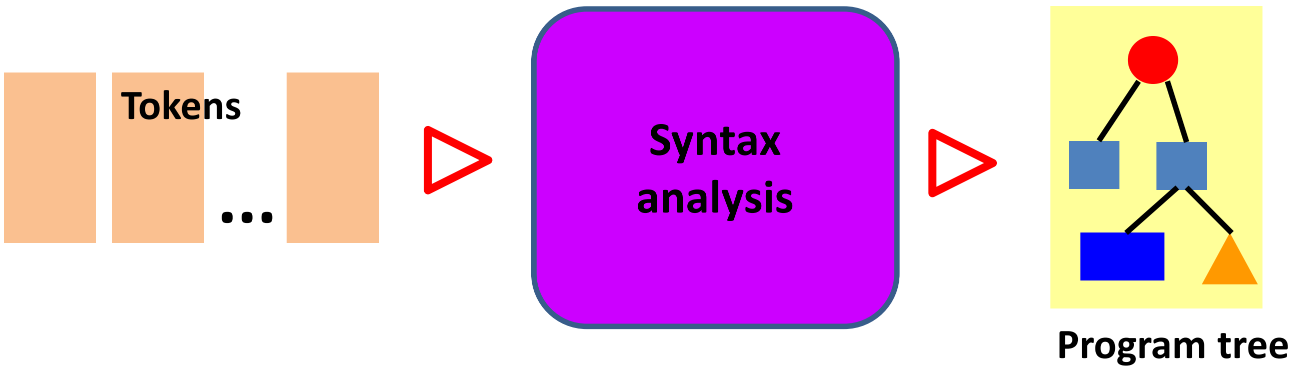
Syntax Analysis

Syntax analysis: why & what for?

- To **check correctness** of the syntactic structure of the source program in accordance with the language grammar.
- To **convert** the source program to an intermediate regular form (representation) which is suitable for subsequent processing (semantic analysis, optimization, code generation).

    The intermediate representation must be **semantically equivalent** to the source program.

- Syntax analysis can be done (completely or partially) *together with semantic analysis* (for simple languages).



(c) Copyright 2018, E.Zouev

Syntax Analysis

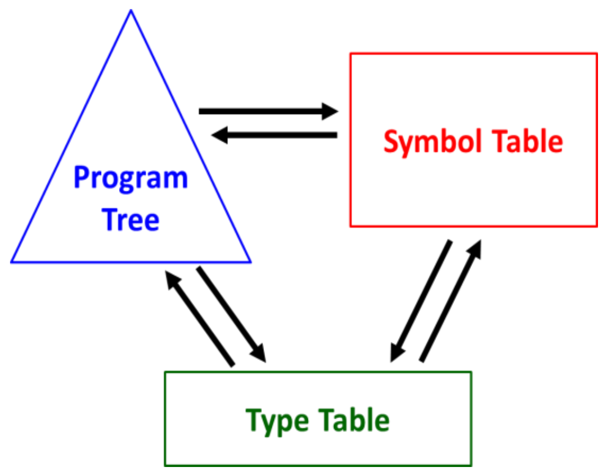
The result of syntax analysis:

an **internal program representation.**

Example:

a **tree structure**, whose nodes and sub-trees correspond to structure elements of the source program.

Tree is not mandatory and not the only possible program representation (*details follow later today*).



Even if there is **no tree** while syntax analysis (simplest cases), it exists **implicitly** ("the parser reflects the tree while running").

(c) Copyright 2018, E.Zouev

Syntax analysis: theory

- Theoretical basis for syntax analysis:
  formal grammar theory.
  Main results were in 60-80s.
- Powerful and efficient algorithms were designed and implemented based on the theory.
- *Any* parsing program implements (implicitly or explicitly) a particular algorithm of syntax analysis.

(c) Copyright 2018, E.Zouev

Syntax analysis: theory

Свердлов С.З.
Языки программирования и методы трансляции
Издательство ПИТЕР, 2007
ISBN 978-5-469-00378-6

The chapter «Теоретические основы трансляции»:
- carefully selected and small amount of information related to the theory;
- simple and clearly written explanations.

(c) Copyright 2018, E.Zouev

Syntax analysis: implementation techniques

- Despite of good theoretical basis, each language requires **individual approach**. The common theory rarely works for 100% (almost never ☺).
- There are powerful and convenient **tools** for automated parser generation (YACC/Bison – will be considered in details on the next lecture).
- Anyway, the basis for any parser is a formal (or semi-formal) **language grammar definition**.

(c) Copyright 2018, E.Zouev

Syntax analysis: implementation techniques

- Almost any language definition contains its (more or less) **formal grammar specification**.
- Typically, the grammar is just for information, but not for compiler writers. Therefore, to create a parser using the grammar from the language definition, we have **to transform** it.
  (Russian translation of the 1st edition of the "Dragon book": examples of C++ & C# grammars ready for implementation.)
- Some syntactical details just **cannot be represented** in the grammar – we have to take them into account in implementation.
  (Example: variable initialization in Ada declarations.)

(c) Copyright 2018, E.Zouev

Compiler Development Technologies

(Syntax analysis as an example.)

- Top-down or bottom-up parsing?
- «Hand-made» or automated development?

	By hand	By tools
Top-down parsing	Most often: **Recursive descent parser**	ANTLR COCO etc.
Bottom-up parsing	Rarely (too complicated)	Yacc, Bison, Jay, GPPG, etc.

- **The most of real compilers are "hand made" ☺.**

(c) Copyright 2018, E.Zouev

Syntax analysis: implementation techniques

- **Top-down parsing**

  The common parsing direction – starting from more common language notions (non-terminals) to more concrete, down to tokens.

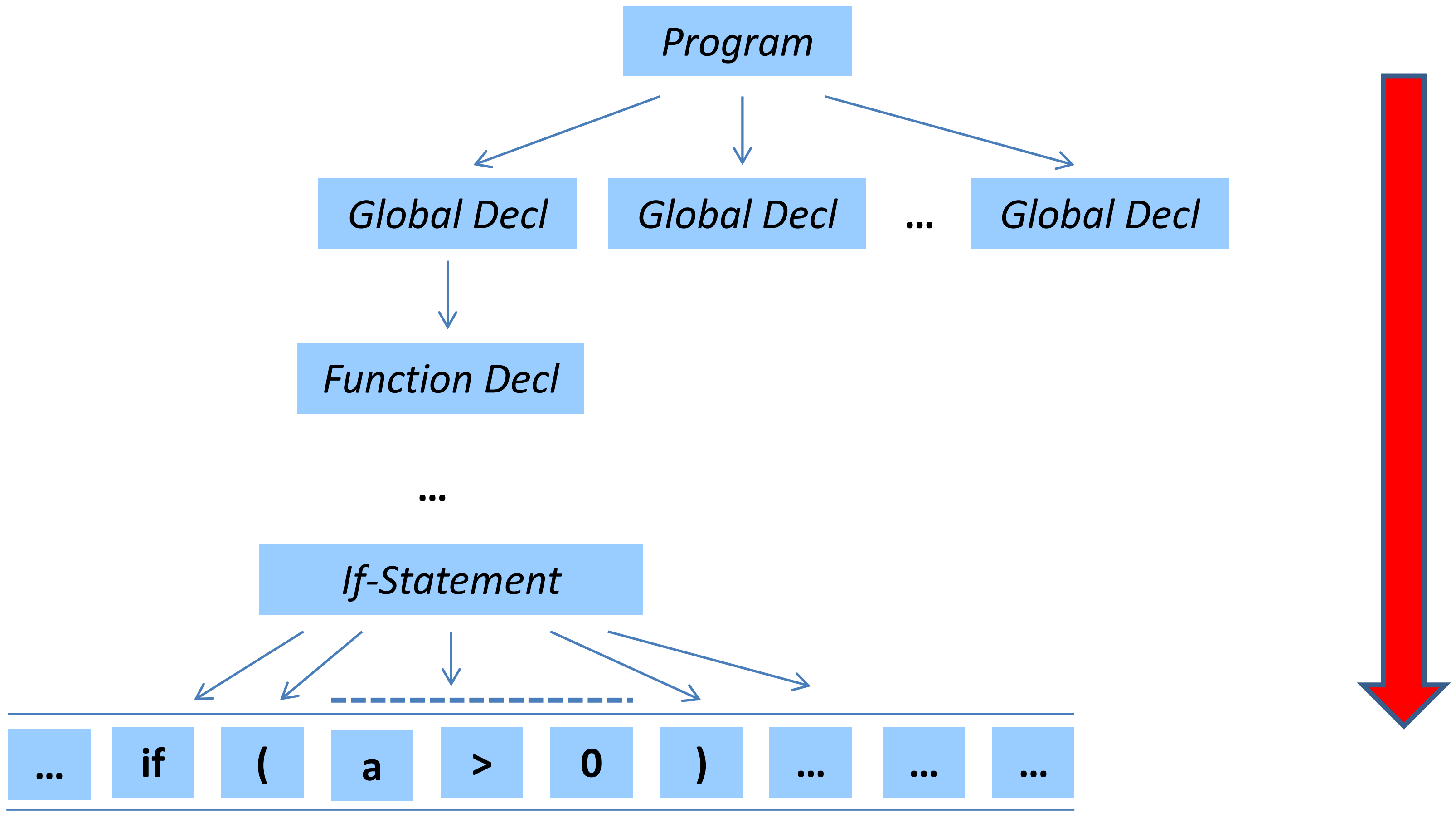
  Parsing algorithm is organized in accordance with language grammar rules: syntax-directed (syntax-driven) parsing.

- **Bottom-up parsing**

  The common parsing direction – from source tokens to grammar rules; the algorithm tries to reduce token sequences to more common non-terminal grammar symbols.

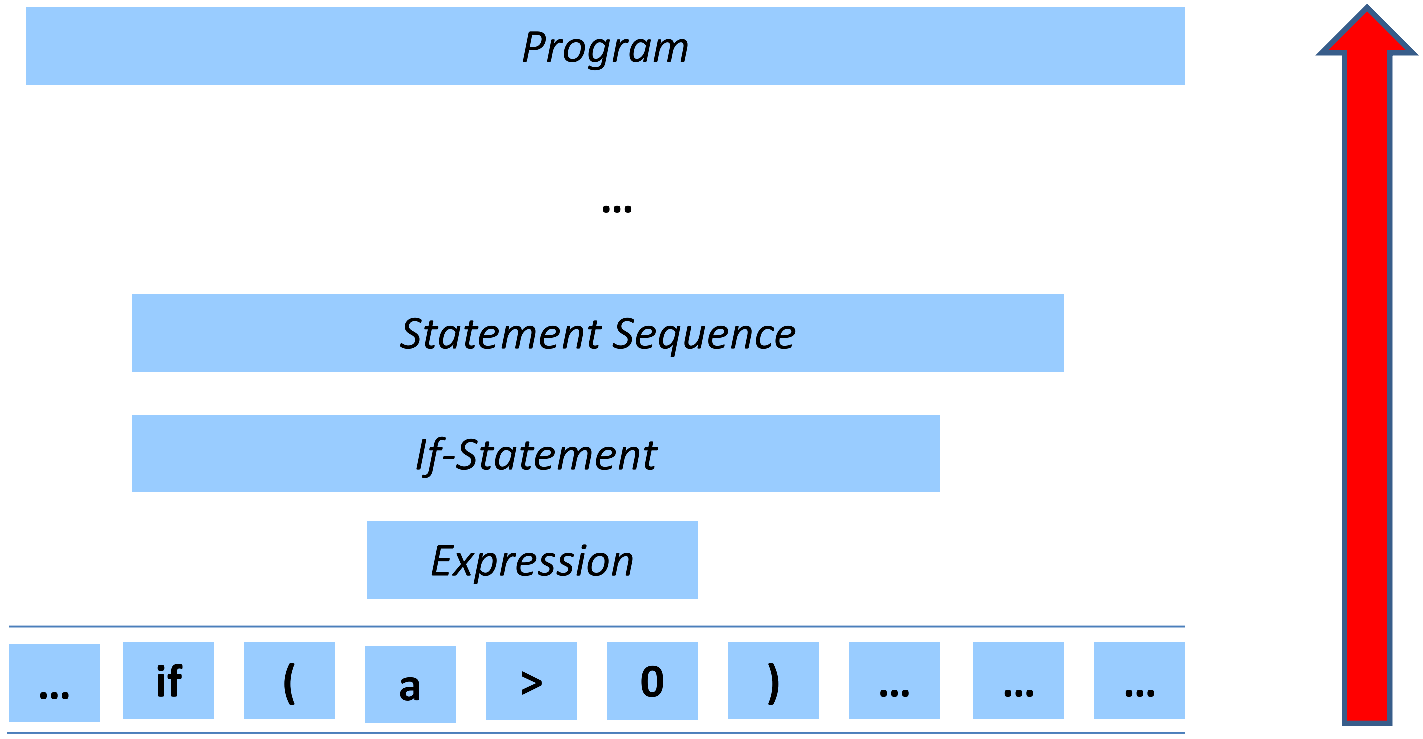
(c) Copyright 2018, E.Zouev

Top-down parsing



(c) Copyright 2018, E.Zouev

Bottom-up parsing



(c) Copyright 2018, E.Zouev

Syntax analysis: implementation techniques

- **Top-down OR bottom-up?**

T>b Top-down algorithm is much easier to implement. Bottom up requires much more efforts (typically it's table-driven).

t<B Top-down algorithm is less stable in case of syntax errors. Error recovery techniques are much harder to implement than for bottom-up..

t<B Top-down algorithm is less refactored: "syntax part" of the compiler is typically not a separate part of the compiler but spread over its source text. It's much harder to modify & maintain.

T>b Interface between the top-down parser & other compiler components (passes) can be organized in a more clear & convenient way than for the bottom-up parser.

(c) Copyright 2018, E.Zouev

Syntax analysis: implementation techniques

- «Hand made» development OR automation tools?

h<A Tools significantly **speed up** parser development (if you know how to use them ☺).

H>a Significant effort are required **to adapt the language grammar** to conform the requirements of a particular tool.

H>a The interface between automatically generated parser & other compiler components is typically **rather restricted**.

h<A Error recovery mechanism is easier to implement (at least for Yacc/Bison).

(c) Copyright 2018, E.Zouev

Syntax analysis: a grammar for expressions

The first approach:

```
Expr -> Expr + Expr
Expr -> Expr – Expr
Expr -> Expr * Expr
Expr -> Expr / Expr
Expr -> Id
Expr -> ( Expr )
```

- The grammar correctly defines expression structure.
- **The problem**: using this grammar we can interpret an expression by **more than one way**.

```
   a   +   b     *     c
   |   |   |     |     |
 Expr  + Expr    |     |
                 |     |
 _______________ |     |
       Expr      *   Expr
     _____________________
           Expr
```

```
 a     +     b     *     c
 |     |     |     |     |
 |     |   Expr    *   Expr
 |     |   _________________
Expr   +         Expr
 ______________________
         Expr
```

The grammar is <u>ambiguous</u>.

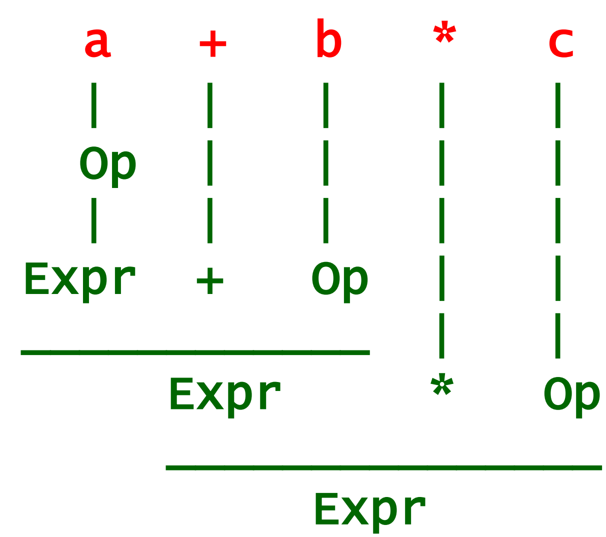
(c) Copyright 2018, E.Zouev

Syntax analysis: a grammar for expressions

The second approach:

```
Expr -> Expr – Op
Expr -> Expr + Op
Expr -> Expr * Op
Expr -> Expr / Op
Op -> Id
Op -> ( Expr )
```

- The grammar correctly defines expression structure.
- The grammar is **unambiguous**.
- **The problem: operator preferences** are not taken into account.



(c) Copyright 2018, E.Zouev

Syntax analysis: a grammar for expressions

The third approach:

```
Expr -> Term
Expr -> Expr + Term
Expr -> Expr - Term
Term -> Factor
Term -> Term * Factor
Term -> Term / Factor
Factor -> Id
Factor -> ( Expr )
```

- The grammar correctly defines expression structure.
- The grammar is **unambiguous**.
- **Operator preferences** are taken into account.

```
  a      +     b      *     c
  |      |     |      |     |
Factor   |   Factor   |     |
  |      |     |      |     |
Term     |   Term     *   Factor
  |      |   ___________________
  |      |            |
Expr     +           Term
___________________________
           Expr
```

So far so good... ☺☹

(c) Copyright 2018, E.Zouev

Recursive descent parsing: Common rules

- For each non-terminal (grammar production) a **corresponding parsing function** is declared.
- Each function sequentially **reads tokens** composing the given non-terminal, or reports an error.
- Each non-terminal in the right part of the rule is treated as **the call to the corresponding function**.

- Why parsing is "recursive"?- because any non-trivial grammar has rules with direct or indirect "self-declarations" which is actually a recursion.
- Why parsing is "descent"?- because the parsing process starts with the very common production down to more concrete ones.

(c) Copyright 2018, E.Zouev

Recursive descent parsing: History

To program recursive descent parser is as fast as just write ☺. (David Gries)

David Gries
Compiler Construction for Digital Computers
John Wiley and Sons, New York, **1971**, 491 pages.

Грис, Д.
Конструирование компиляторов
для цифровых вычислительных машин
Издательство: М.: Мир
544 страниц; **1975** г.

Написать рекурсивный нисходящий парсер можно, не отрывая пера от бумаги ☺. (David Gries)

Recursive descent parser: example

```
Expr -> Term
Expr -> Expr + Term
Expr -> Expr – Term
```

```
Term -> Factor
Term -> Term * Factor
Term -> Term / Factor
```

```
Factor -> Id
Factor -> ( Expr )
```

```
void parseExpr()
{
    parseExpr(); // !!!!!!!
    if ( tk=get(), tk==tkPlus || tk==tkMinus )
    {
        parseTerm();
    }
}
void parseTerm()
{
    parseTerm(); // !!!!!!!
    if ( tk=get(), tk==tkStar || tk==tkSlash )
    {
        parseFactor();
    }
}
void parseFactor()
{
    if ( tk=get(), tk==tkLParen )
    {
        parseExpr();
        get(); // skip ‘)’
    }
    else
        parseId();
}
```

LEFT RECURSION!! ☹☹

(c) Copyright 2018, E.Zouev

Recursive descent parser: example

A grammar with the left recursion always can be transformed to an equivalent grammar with the right recursion.

The fourth approach

```
Expr -> Term
Expr -> Expr + Term
Expr -> Expr - Term

Term -> Factor
Term -> Term * Factor
Term -> Term / Factor

Factor -> Id
Factor -> ( Expr )
```

```
Expr -> Term
Expr -> Term + Expr
Expr -> Term - Expr

Term -> Factor
Term -> Factor * Term
Term -> Factor / Term

Factor -> Id
Factor -> ( Expr )
```

(c) Copyright 2018, E.Zouev

Recursive descent parser: example

«Programming» solution:

- Use EBNF format for more clarity;
- Replacing recursion <u>for iteration</u>.

```
Expr -> Term
Expr -> Expr + Term
Expr -> Expr – Term

Term -> Factor
Term -> Term * Factor
Term -> Term / Factor

Factor -> Id
Factor -> ( Expr )
```

The fifth (final) approach:

```
Expr -> Term
           { +|- Term }

Term -> Factor
           { *|/ Factor }

Factor -> Id
Factor -> ( Expr )
```

(c) Copyright 2018, E.Zouev

Recursive descent parser: example

```
Expr ->
  Term { +|- Term }
```

```
Tree parseExpr()
{
    Tree left = parseTerm();
    while ( tk=get(), tk==tkPlus||tk==tkMinus )
        left = mkBinTree(tk,left,parseTerm());
    return left;
}
```

```
Term ->
 Factor { *|/ Factor }
```

```
Tree parseTerm()
{
    Tree left = parseFactor();
    while ( tk=get(), tk==tkStar||tk==tkSlash )
        left = mkBinTree(tk,left,parseFactor());
    return left;
}
```

```
Factor -> Id
Factor -> ( Expr )
```

```
Tree parseFactor()
{
    Tree res;
    if ( tk=get(), tk==tkLParen )
    {
        res = parseExpr();
        get(); // skip ‘)’
    }
    else
        res = mkUnaryTree(parseId());
    return res;
}
```

(c) Copyright 2018, E.Zouev

Bottom-up parsing:
an example of a table-driven parser

```
Factor -> Id
Factor -> ( Expr )
```

Finite state automaton with the stack memory

Детерминированный конечный автомат с магазинной памятью

Input token

Next state

Read the next token?

State number

Factor

State	Input	Go to	Read next	Comment
f1	Id	0		Return
f2	∀ - (			Error
f3	(	call e1	+	Goto e1 with return
f3	)	0		Return
f4				Error

(c) Copyright 2018, E.Zouev

Bottom-up parsing: an example of a table-driven parser

```
Term ->
 Factor { *|/ Factor }
```

Term

State	Input	Go to	Read next	Comment
t1		call f1		Goto f1 with return
t2	*	call f1	+	Goto f1 with return
t3		t2		
t4	∀ - /			Error
t4	/	call f1	+	Goto f1 with return
t5		t2		

(c) Copyright 2018, E.Zouev

Bottom-up parsing: an example of a table-driven parser

Bottom-up parsing:

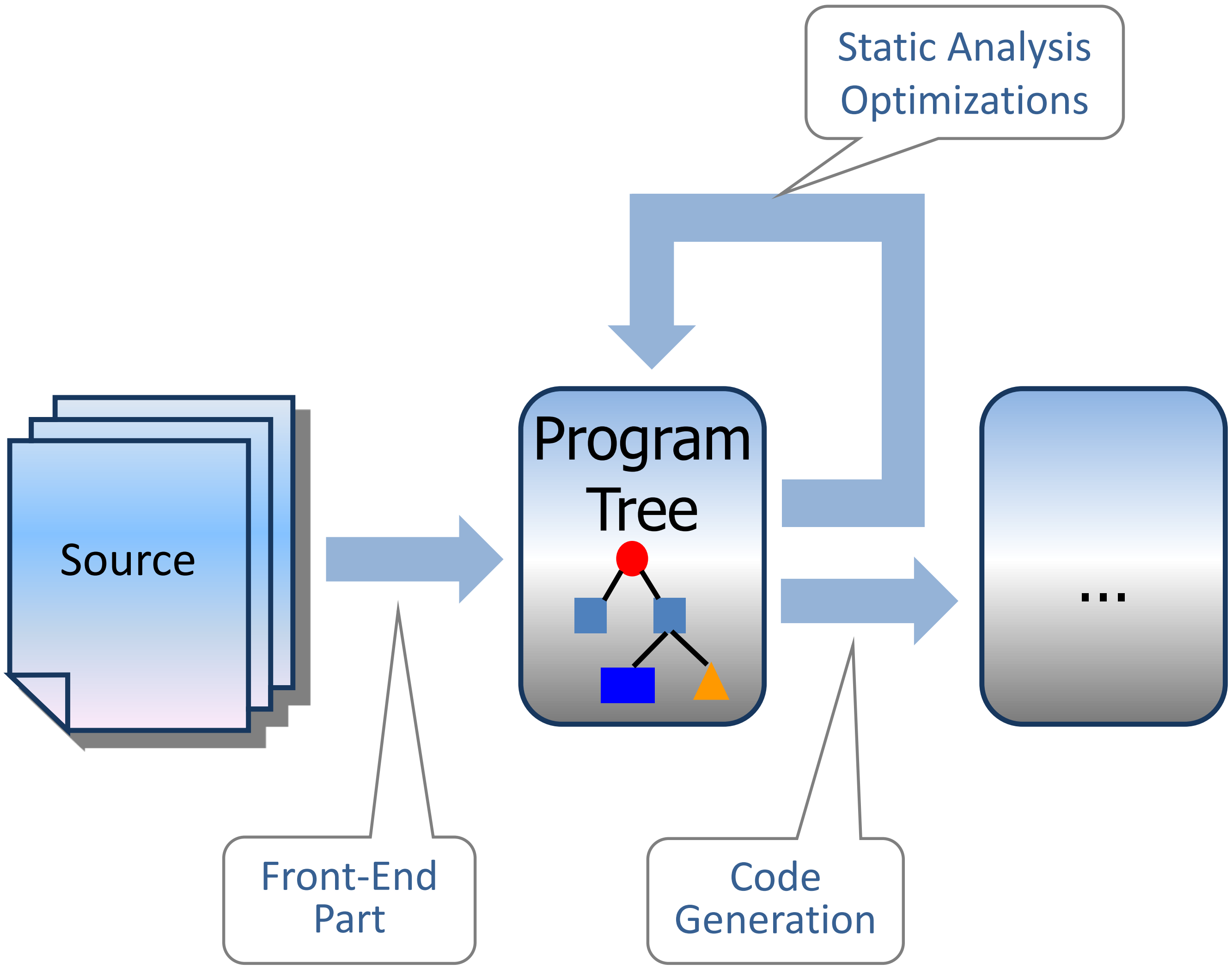
- Is controlled by the input token stream
- Uses its own stack memory for keeping return states
- Tables can be generated automatically (by a tool) from the grammar
- The single algorithm can work with tables for the whole grammar category.

Not shown:

- Error processing – some ideas in yacc/bison
- Semantic actions! – will see later for yacc/bison

(c) Copyright 2018, E.Zouev

Compilation structures



(c) Copyright 2018, E.Zouev

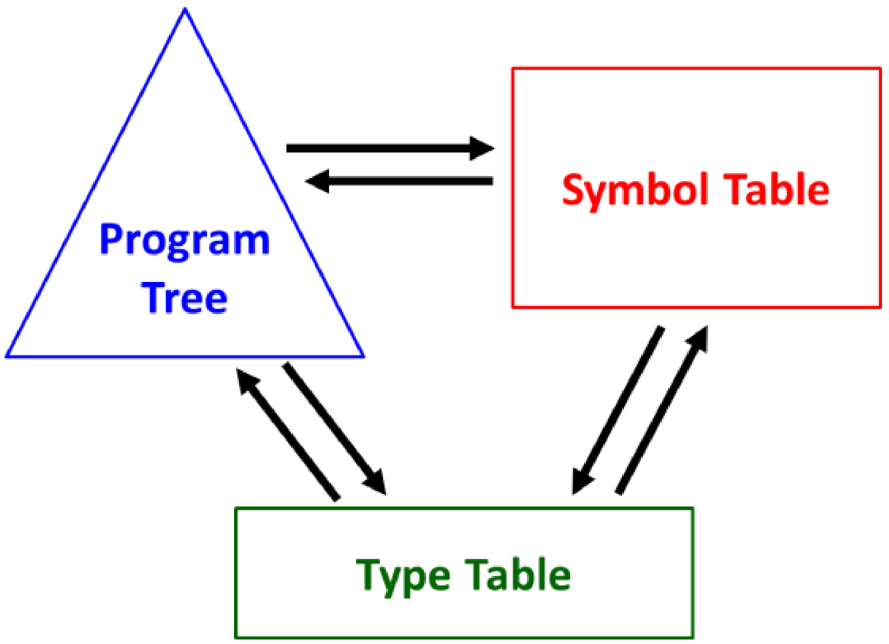
Compilation structures

The task:

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

Three language entity categories:

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



(c) Copyright 2018, E.Zouev

Symbol table

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

Swap

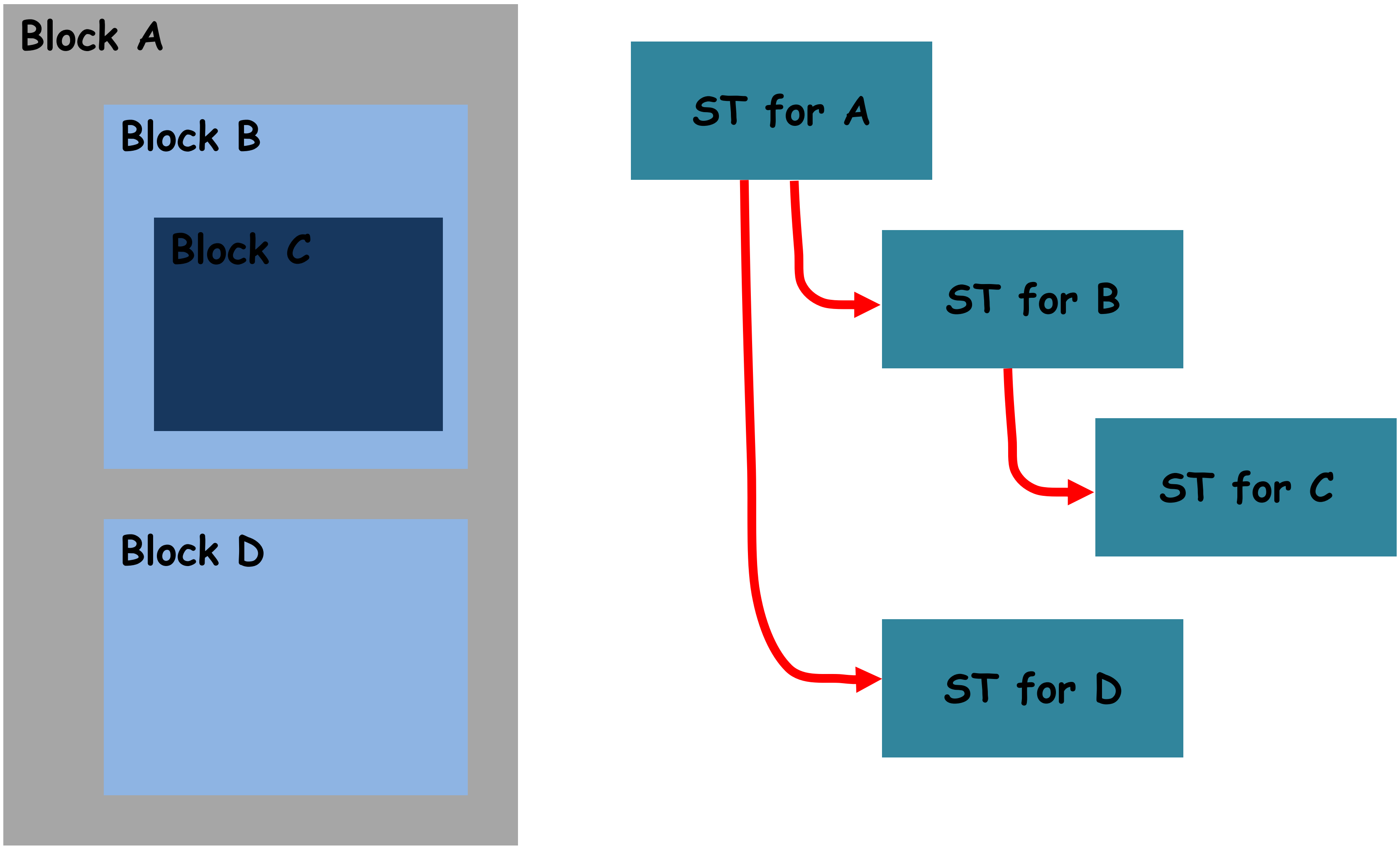
47	46.44		39 … 30	29 … 20	19 … 10	9 … 0
1	3		0	**int** (index to TT)	**a** (index to LT)	hash link
1	3		0	**int** (index to TT)	**b** (index to LT)	hash link
1	4		Link to AST	**int** (index to TT)	**Temp** …	hash link

initializer

in / out / local

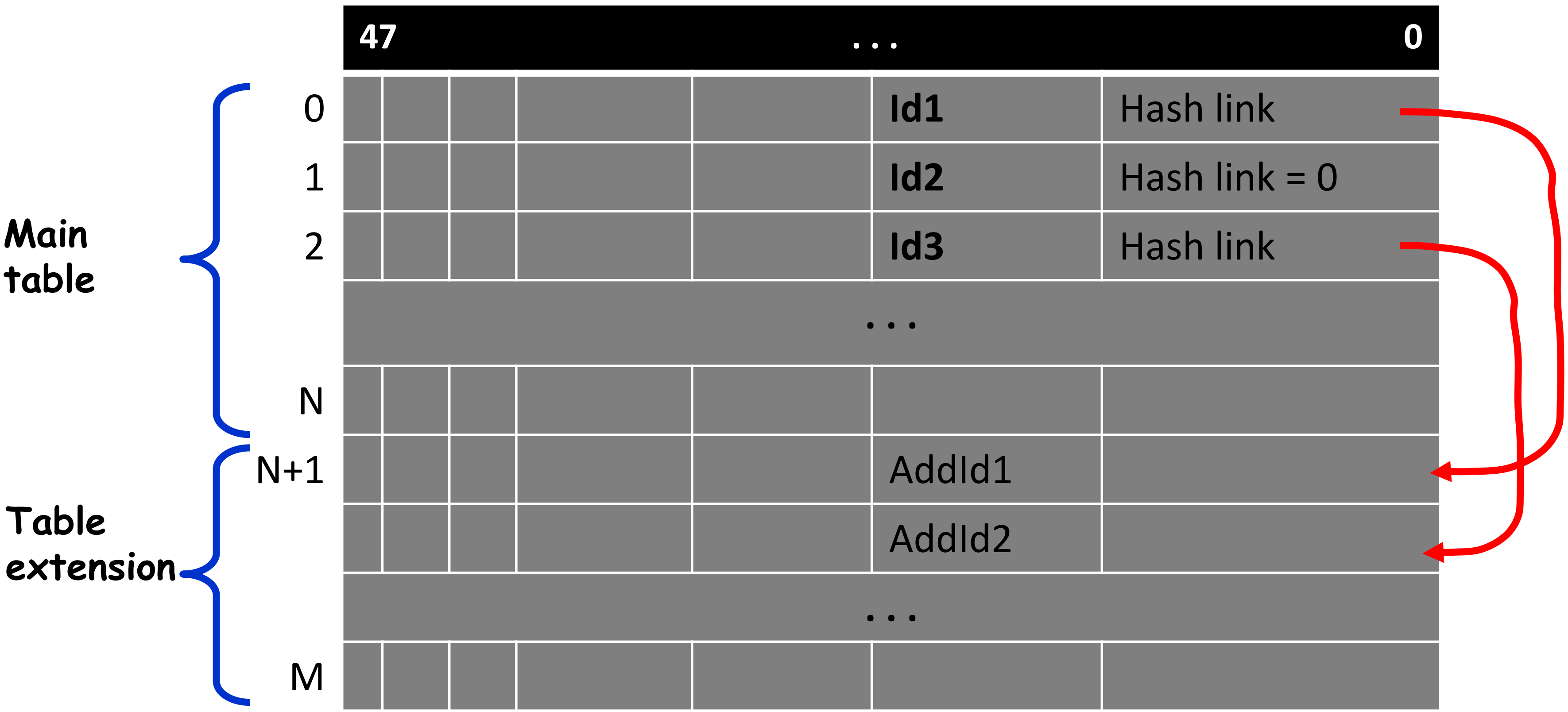
used / not in use

Nested blocks & symbol tables



(c) Copyright 2018, E.Zouev

Hash tables



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

(c) Copyright 2018, E.Zouev

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"
    }
}
```

(c) Copyright 2018, E.Zouev

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.

(c) Copyright 2018, E.Zouev

Tables AND/OR trees? (2)

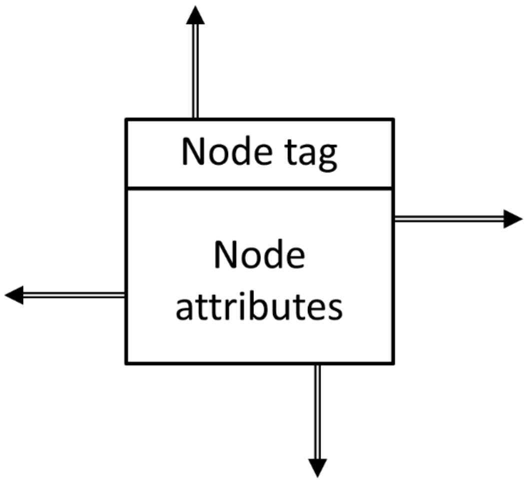
However:

- 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**.

(c) Copyright 2018, E.Zouev

Program tree: Interstron C++ implementation (1)

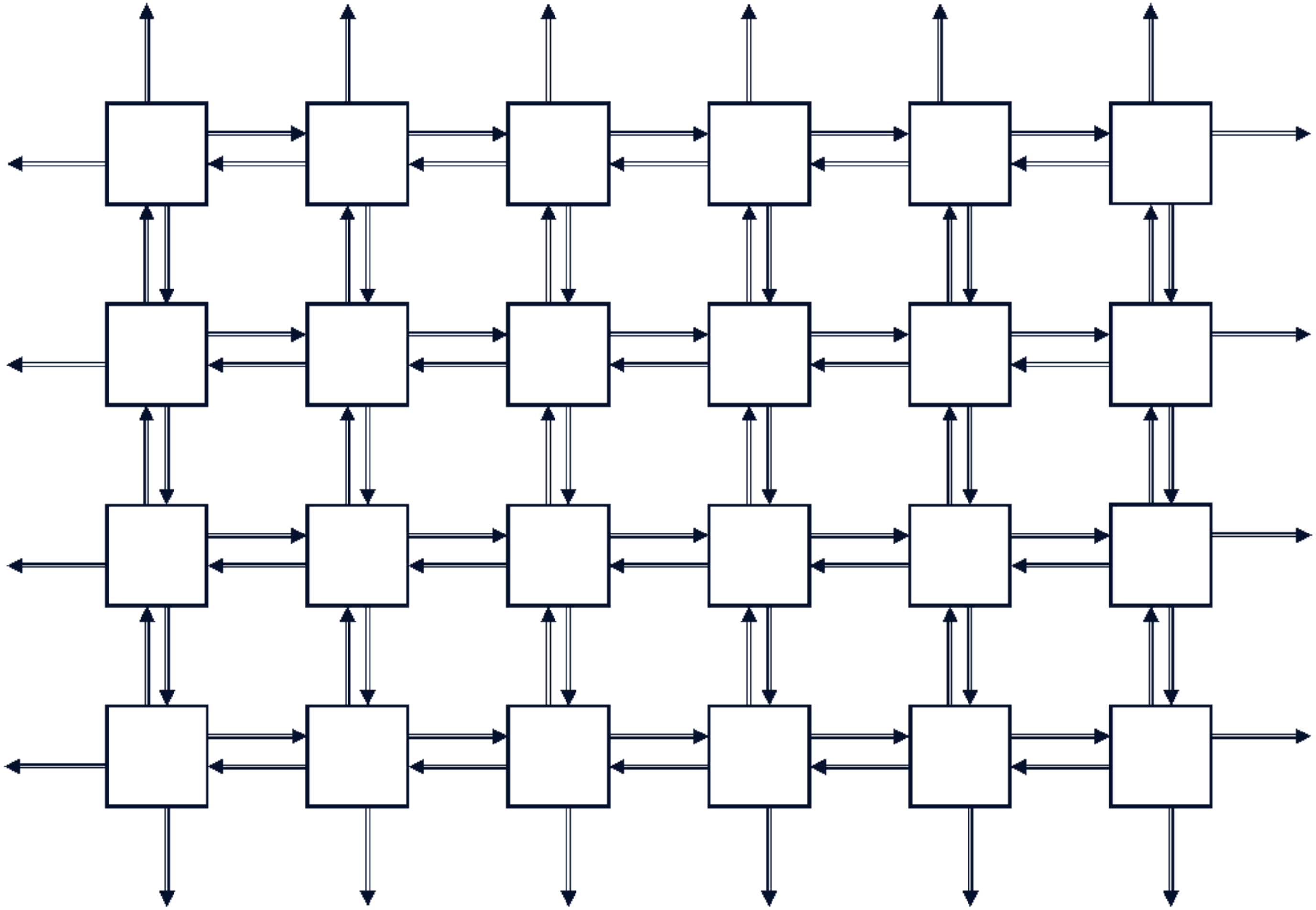


The tree node is a small structure:

- 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.

(c) Copyright 2018, E.Zouev

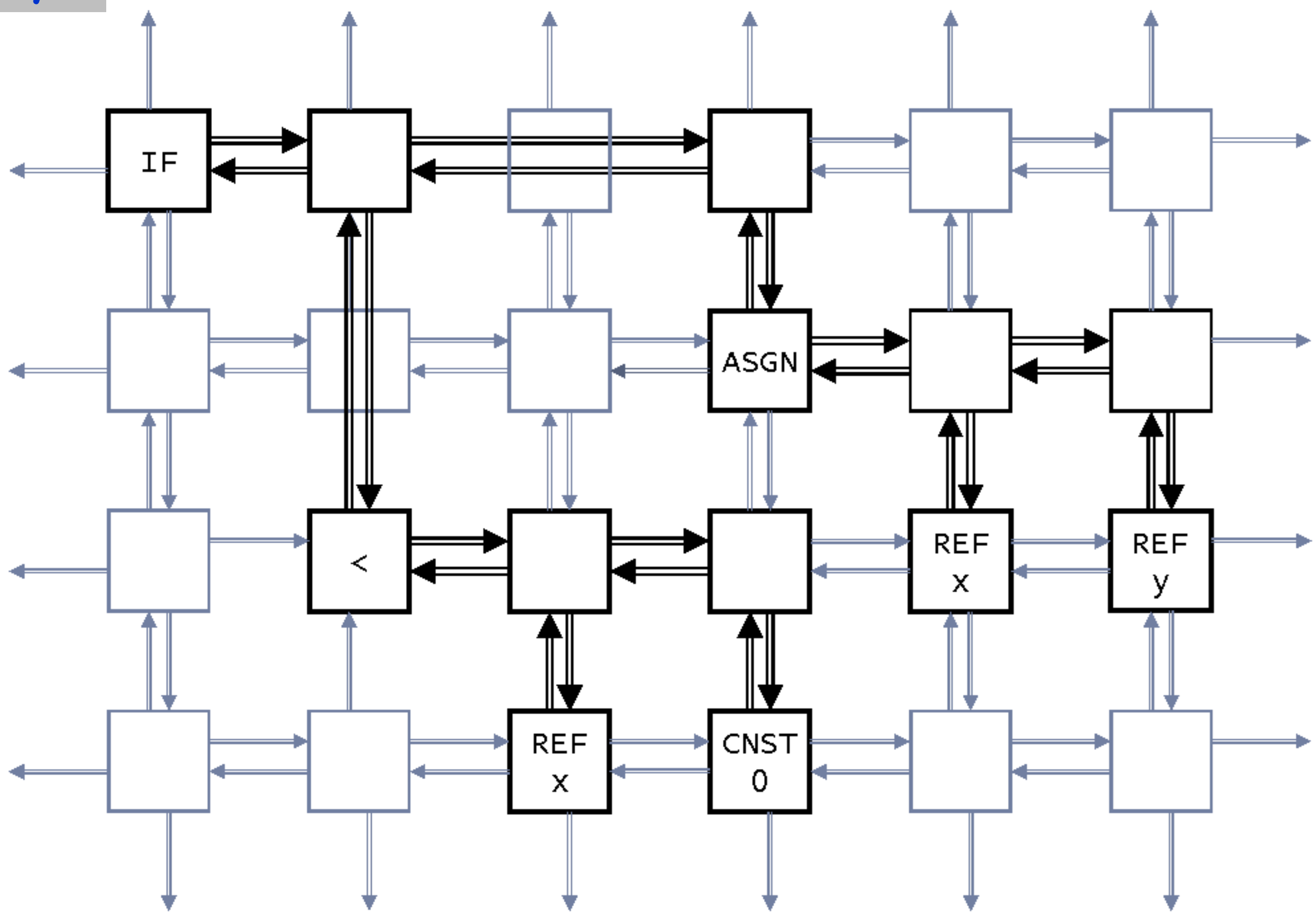
Program tree: Interstron C++ implementation (2)



(c) Copyright 2018, E.Zouev

Program tree: Interstron C++ implementation (3)

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



(c) Copyright 2018, E.Zouev

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.

(c) Copyright 2018, E.Zouev

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#* languages are implemented using CCI.

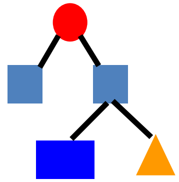
Main functions:

- 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.

(c) Copyright 2018, E.Zouev

AST implementation: CCI approach

Tree structure:
a «pure» tree



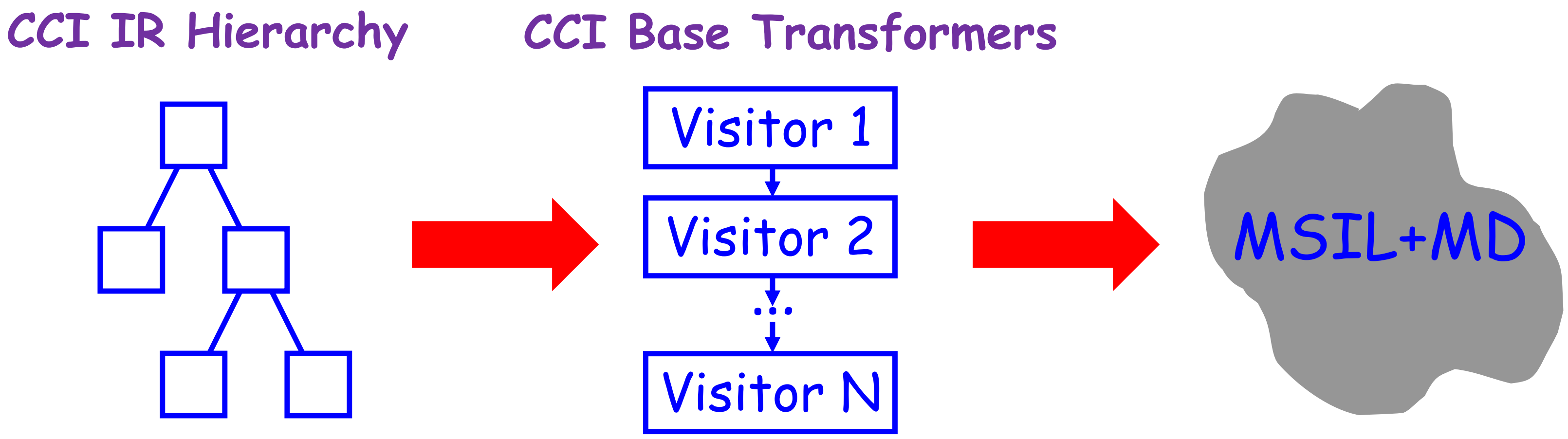
```
public class If : Statement
{
    Expression condition;
    Block      falseBlock;
    Block      trueBlock;
}
```

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);
        ...
       }
     }
  }
}
```

(c) Copyright 2018, E.Zouev

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.

(c) Copyright 2018, E.Zouev

AST implementation: an integral approach (1)

Main project decision:

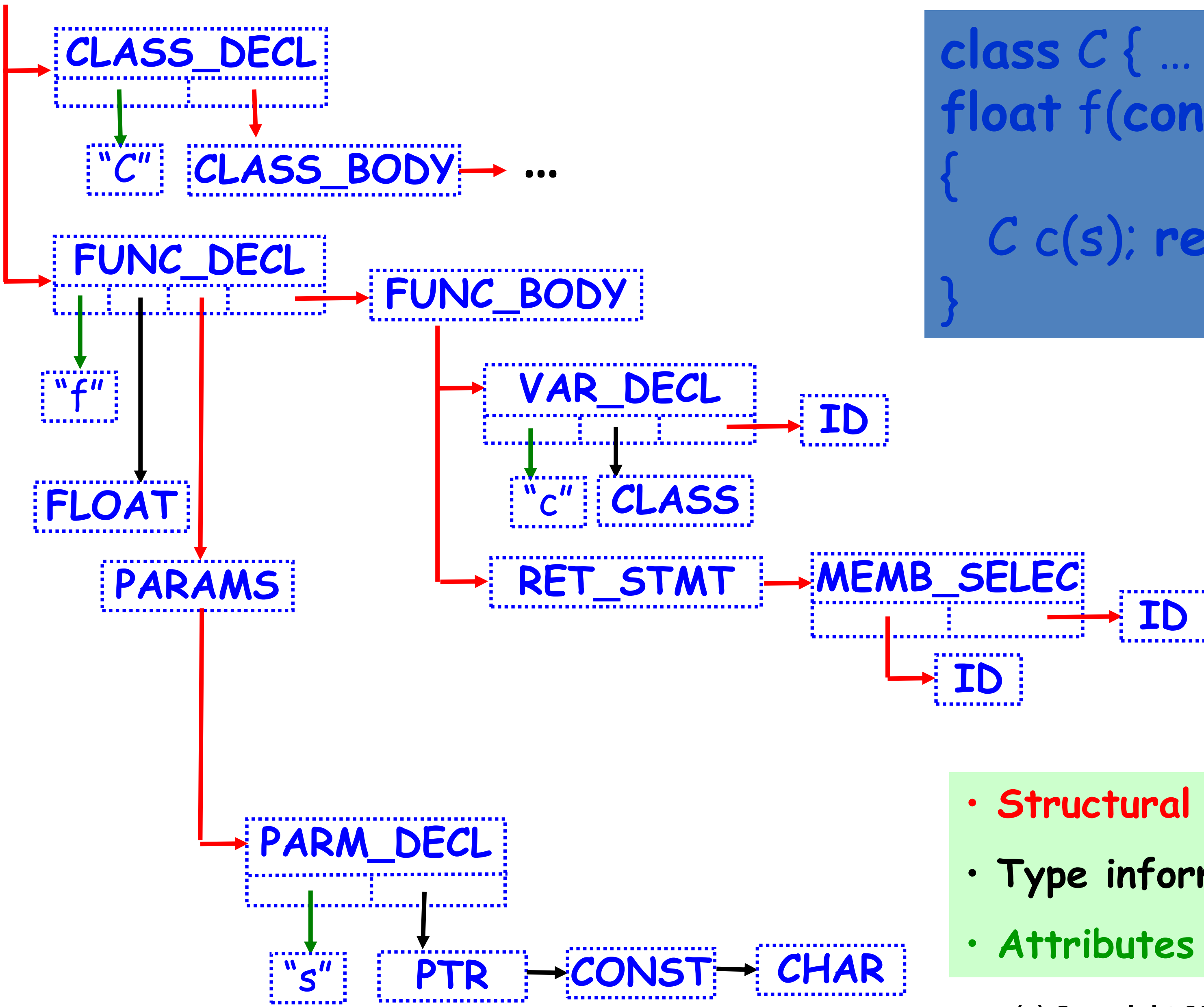
- Each program tree node contains **both** structure (its parts) **and full set or operators** on the given node (and its sub-trees).

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();
        ...
    }
}
```

(c) Copyright 2018, E.Zouev

AAST example (a fragment)



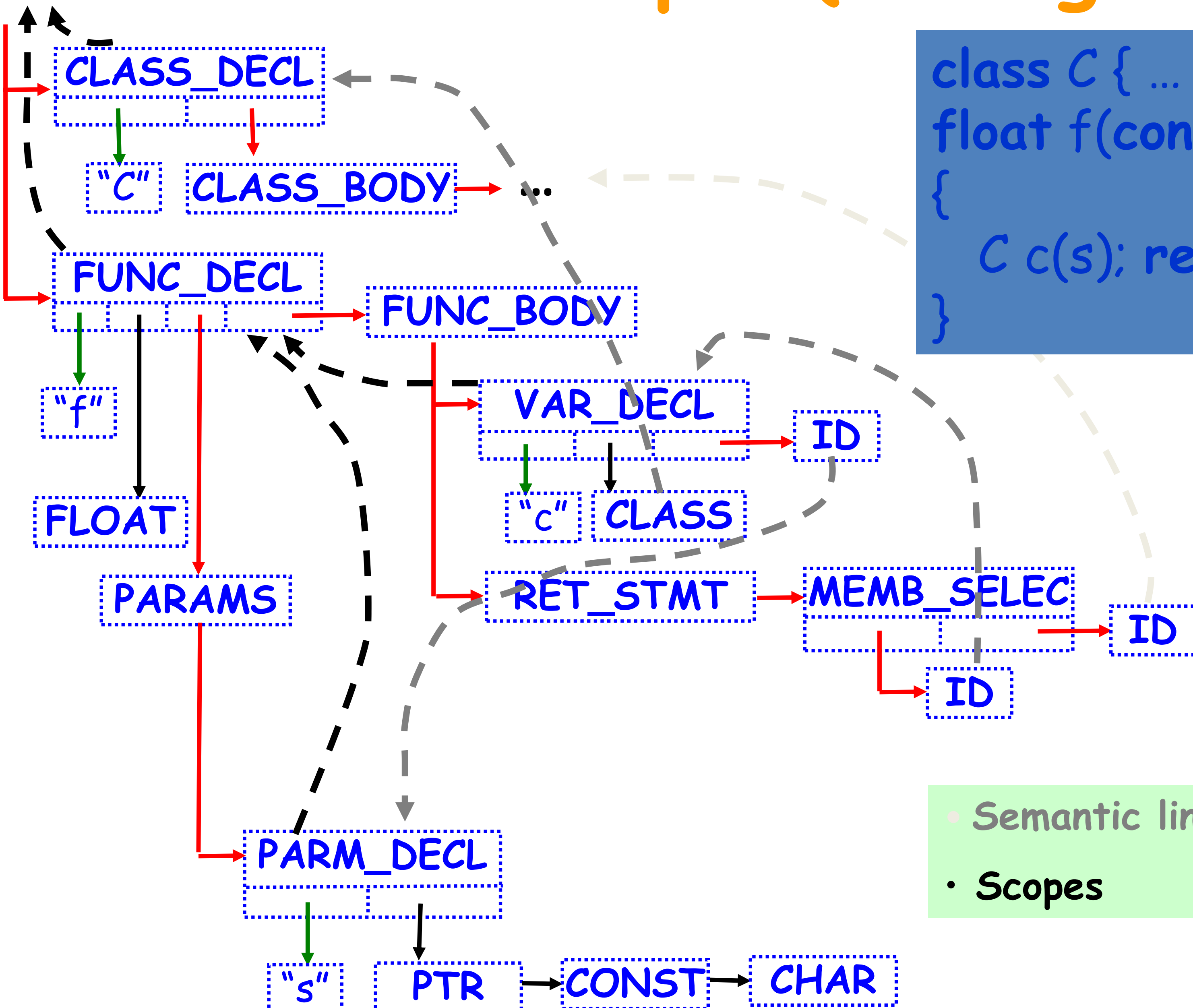

```
class C { … };
float f(const char* s)
{
   C c(s); return c.m;
}
```

- Structural links
- Type information
- Attributes

(c) Copyright 2018, E.Zouev

AAST example (a fragment)

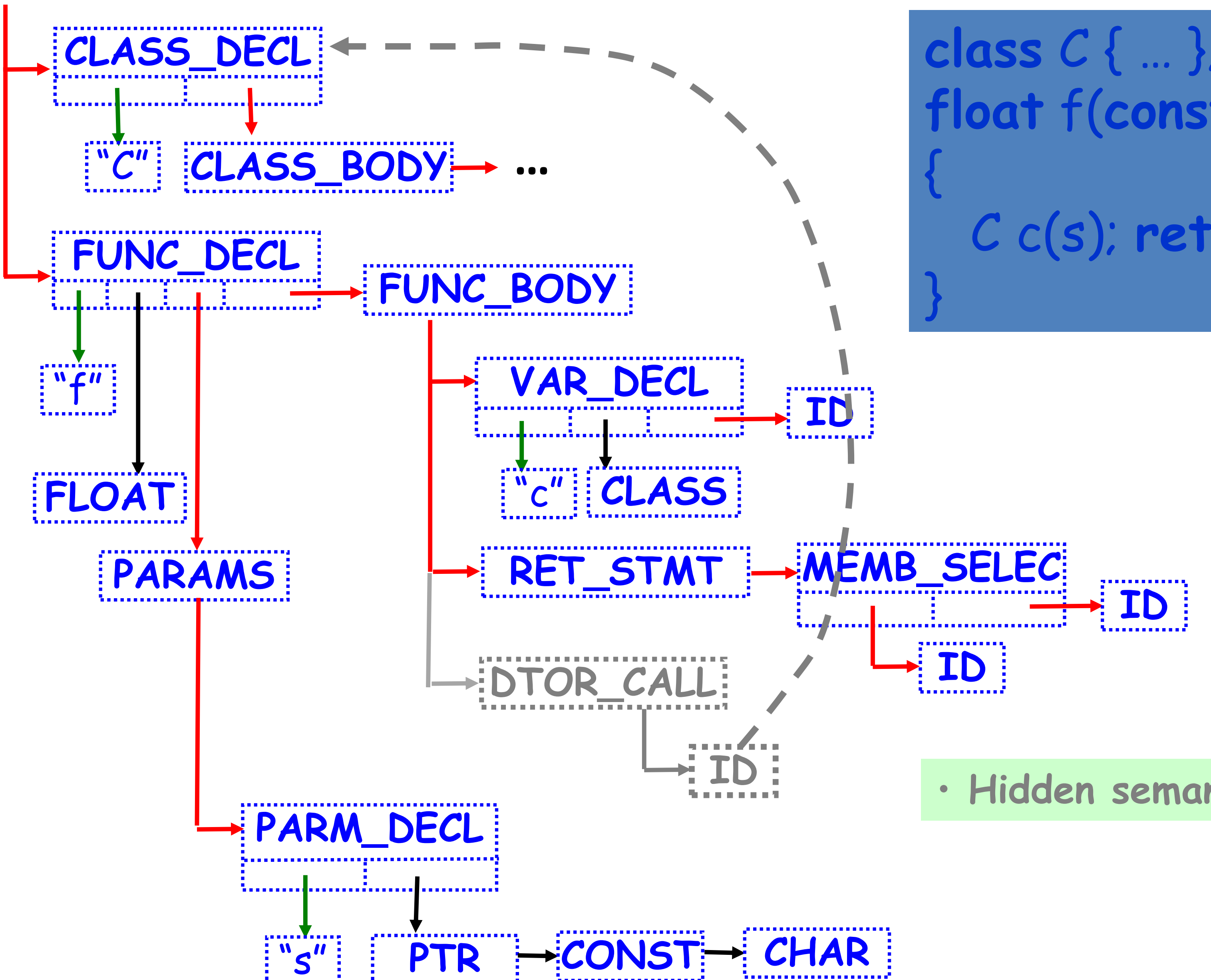
```
class C { … };
float f(const char* s)
{
   C c(s); return c.m;
}
```



- Semantic links
- **Scopes**

(c) Copyright 2018, E.Zouev

AAST example (a fragment)




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

- Hidden semantics

(c) Copyright 2018, E.Zouev

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

(c) Copyright 2018, E.Zouev

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

(c) Copyright 2018, E.Zouev