



# Semantic Analysis and Intermediate Representation

*Compilers course*

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year



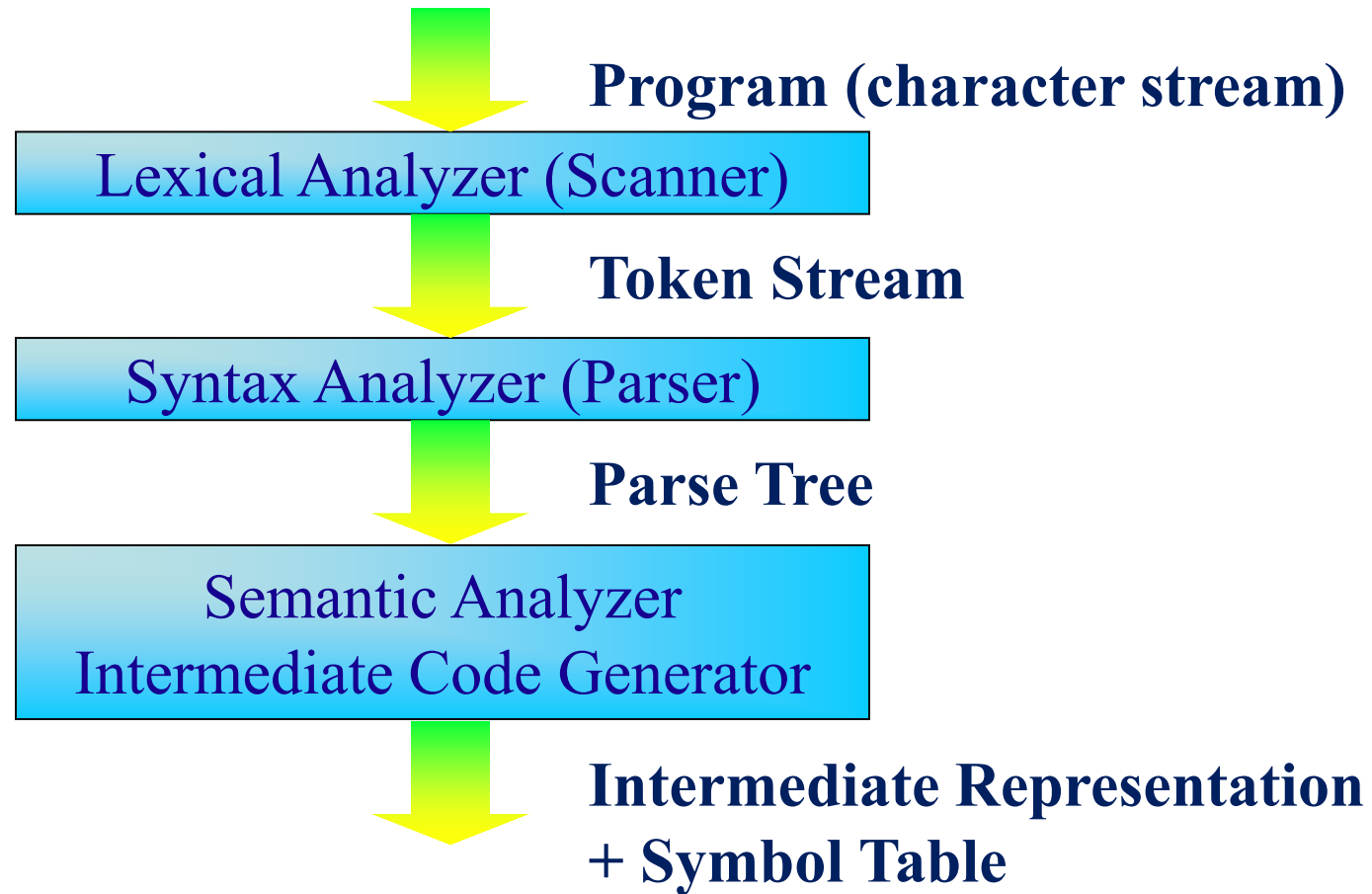
**João M. P. Cardoso**



Universidade do Porto  
**FEUP** Faculdade de Engenharia

Dep. de Engenharia Informática  
Faculdade de Engenharia (FEUP), Universidade do Porto,  
Porto, Portugal  
Email: [jmpc@acm.org](mailto:jmpc@acm.org)

# Compiler Stages



# What is the Semantic of a Program?

## ➤ Syntax

- How the program is structured
- Textual representation or structure

## ➤ Semantic

- What is the meaning of the program?

# Goals of the Semantic Analysis

- Verify if the program is according to the definitions of the programming language
- Report, whenever there are semantic errors, useful messages to the user
- There is not needed too much additional work if the analysis is integrated in the generation of the intermediate representation

# Errors Output by the Semantic Analysis

- Java (using the javac 1.7.0 compiler)

```
boolean sum(int A[], int N) {  
    int i, sum;  
    for(i=0; i<N; i++) {  
        sum1 = sum + A[i];  
    }  
    return sum;  
}  
...  
int s = sum(A);
```

6: error: cannot find symbol  
 sum1 = sum + A[i];  
 ^

symbol: variable sum1  
location: class semantic1

8: error: incompatible types  
 return sum;  
 ^

required: boolean  
found: int

12: error: method sum in class X cannot be applied to given types;  
 int s = sum(A);  
 ^

required: int[],int

found: int[]

reason: actual and formal argument lists differ in length

# Errors Output by the Semantic Analysis

- Java (using the javac 1.7.0 compiler)

```
boolean sum(int A[], int N) {  
    int i, sum;  
    for(i=0; i<N; i++) {  
        sum = sum + A[i];  
    }  
    return sum;  
}  
...  
int s = sum(A, N);
```

8: error: incompatible types  
 return sum;  
 ^  
 required: boolean  
 found: int

12: error: incompatible types  
 int s = sum(A, 100);  
 ^  
 required: int  
 found: boolean

# Errors Output by the Semantic Analysis

- Java (using the javac 1.7.0 compiler)

```
int sum(int A[], int N) {  
    int i, sum;  
    for(i=0; i<N; i++) {  
        sum = sum + A[i];  
    }  
    return sum;  
}
```

...

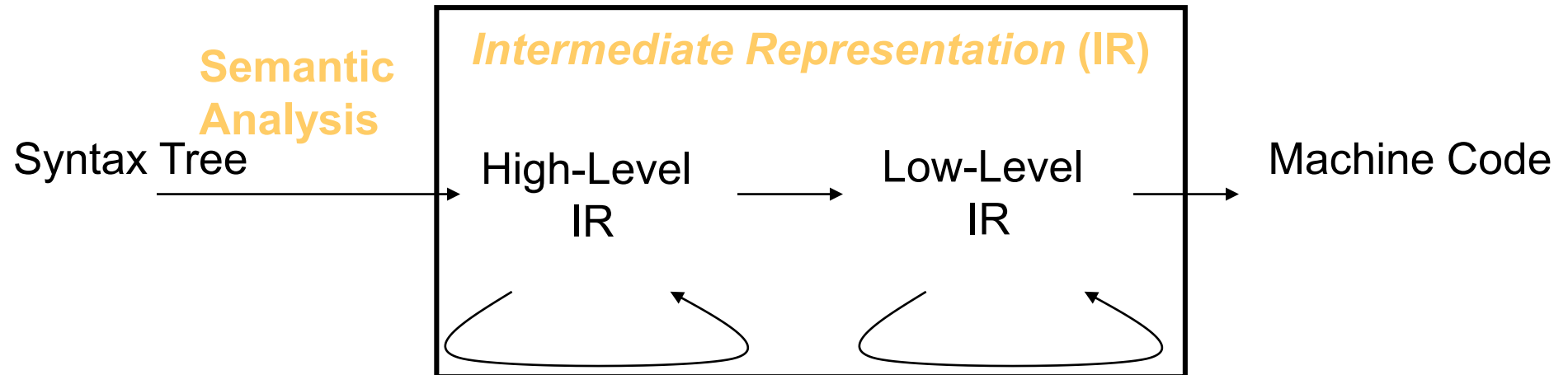
```
int s = sum(A, N);
```

6: error: variable sum might not have been initialized  
sum = sum + A[i];  
^

8: error: variable sum might not have been initialized  
return sum;  
^

# Goals of the Intermediate Representations (IRs)

- To allow analysis and transformations
  - Optimizations
- To structure translation of Machine Code
  - Sequence of steps





# High-Level Intermediate Representation

- It preserves the structured control flow
- Useful for optimizations at the loop level
  - Loop Unrolling, Loop Fusion, etc.
- It preserves the structure at class level
- Useful for optimizations for object-oriented languages

# Low-Level Intermediate Representation

- From an abstract data model to a flat region memory space
- Eliminates the structured control flow
- Useful for low-level compilation tasks
  - Register Allocation
  - Selection of Instructions
  - Scheduling

# Alternatives

- There are many possibilities
  - Tree of instructions and expressions
  - Control-Flow + Acyclic Data Graphs(DAGs)
  - Three address code (C3E)
  - And others...
- Representation selected based on the language and target
- The following classes illustrates a possible tree of instructions and expressions

# Compiler Tasks

- Determine format of the structures in the memory
  - Format of the arrays and objects in the memory
  - Format of the call stack in the memory
- Generate code
  - To read values (parameters, elements of the arrays, fields, etc.)
  - To evaluate expressions and compute new values
  - To write values
  - For control structures
- Enumerate functions and builds the symbol table
  - Invocation of a function accesses to the entry of the correspondent table of functions
- Generate code for the functions
  - Local variables and access to parameters
  - Invocations of functions

# Symbol Tables

- Key concept in compilation
  - While processing type declarations, declarations of variables and functions we are going to assign meaning to those identifiers using symbol tables
- Compilers use symbol tables to produce:
  - Layout of the structures in the memory
  - Function tables
  - Code to access fields, local variables, parameters, etc.

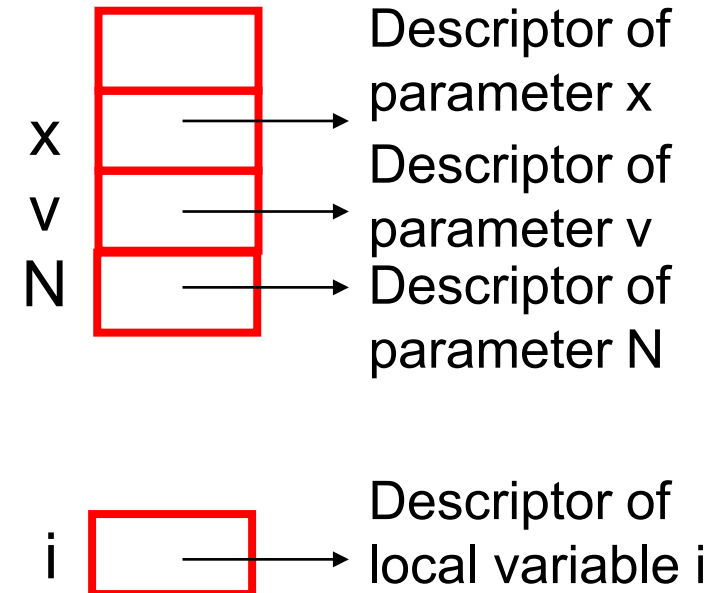
# Symbol Tables

- During the creation/translation of syntax trees
- During the transation of syntax trees to intermediate representation
  - Symbol tables map identifiers (strings) to descriptors (information about the identifier)
  - Basic operatio: Lookup
    - Given a string, find its descriptor
    - Typical implementation: hash table
- Example:
  - Given the name of a variable find its descriptor (local, parameter, global)

# Example

```
void add(int x, int[] v, int N)
{
    int i;
    i = 0;
    while (i < N) {
        v[i] = v[i]+x;
        i = i+1;
    }
}
```

Function add



# Example

```
void add(int x, int[] v, int N)
```

```
{
```

```
  int i;
```

```
  i = 0;
```

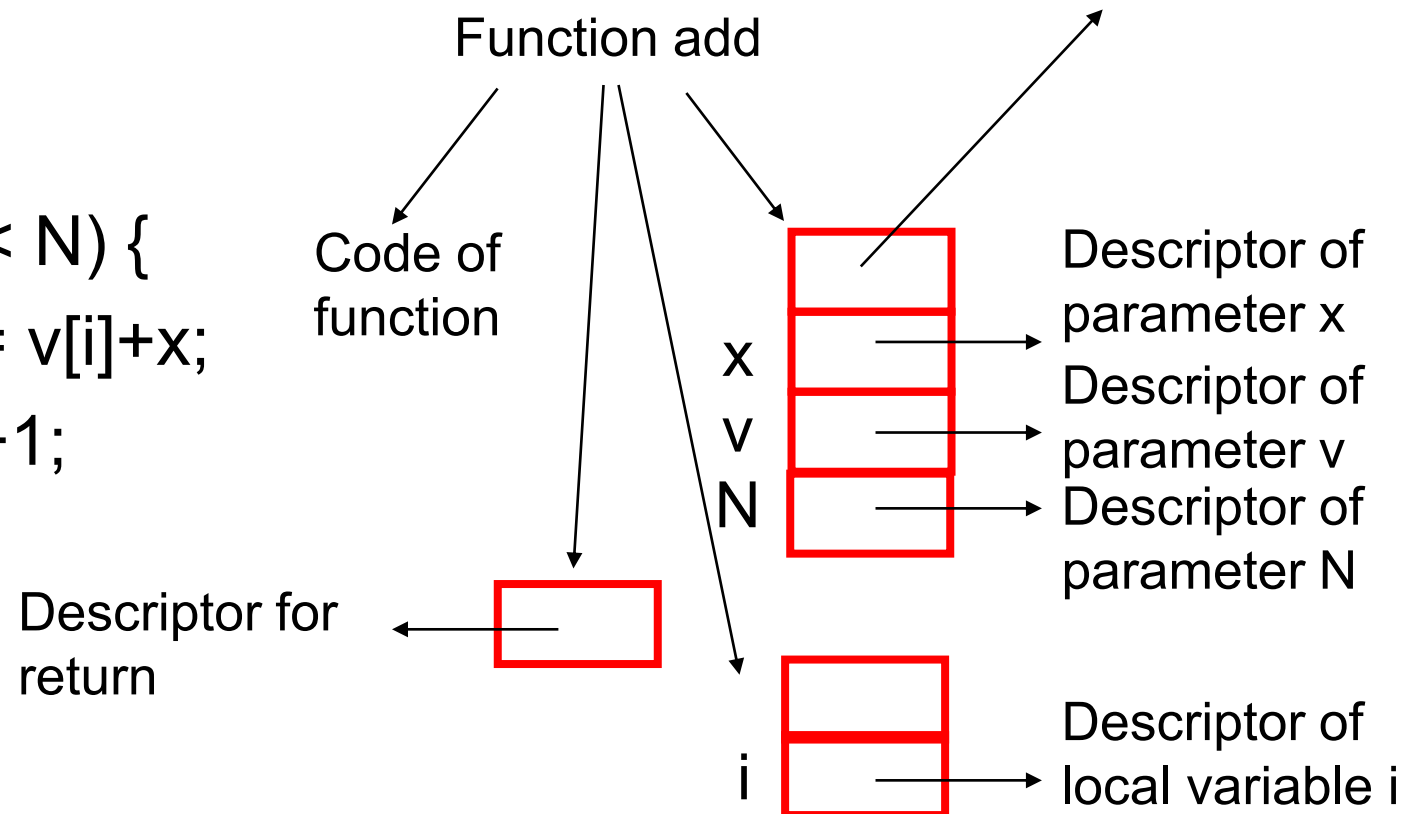
```
  while (i < N) {
```

```
    v[i] = v[i]+x;
```

```
    i = i+1;
```

```
  }
```

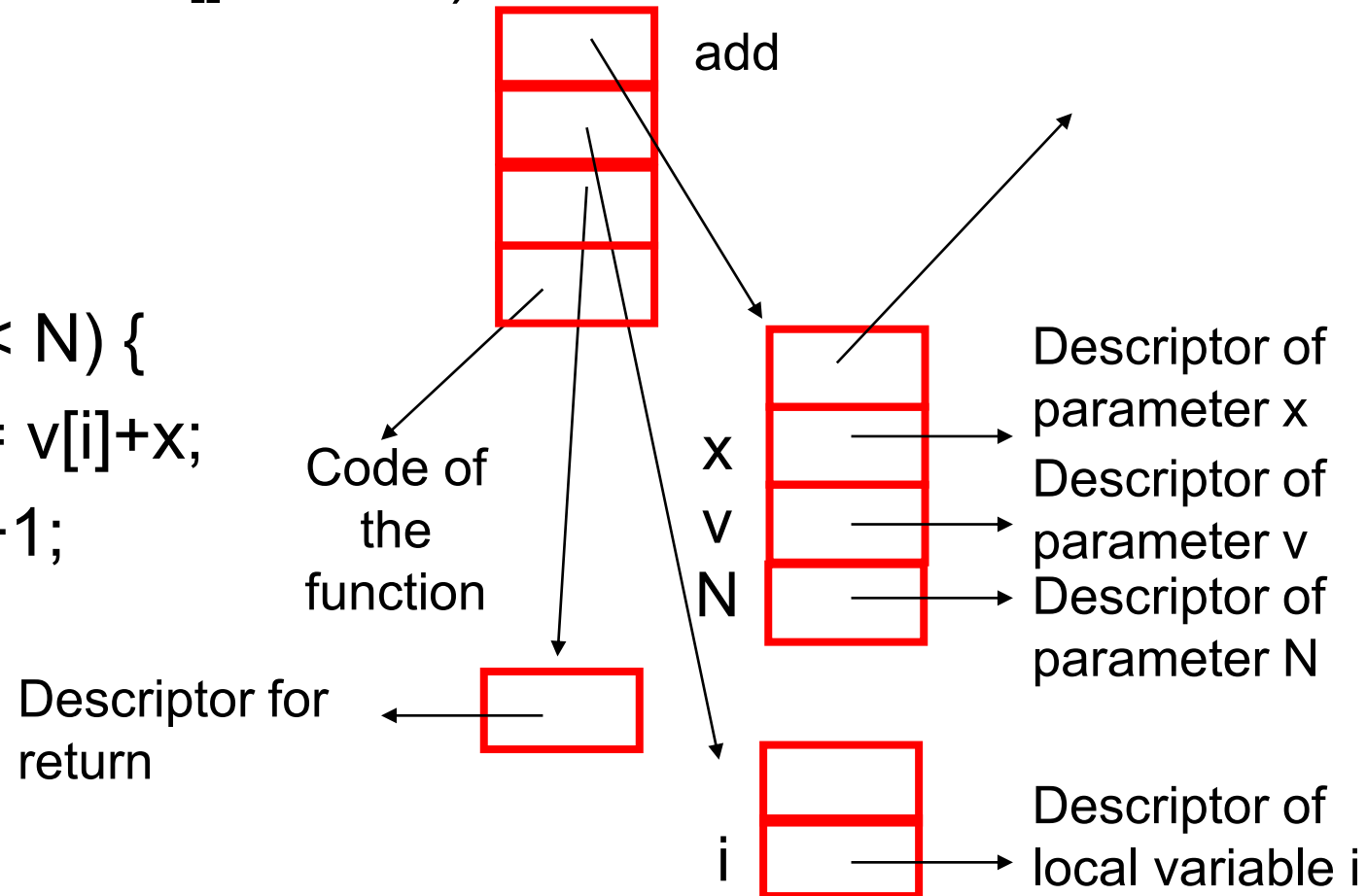
```
}
```





# Example

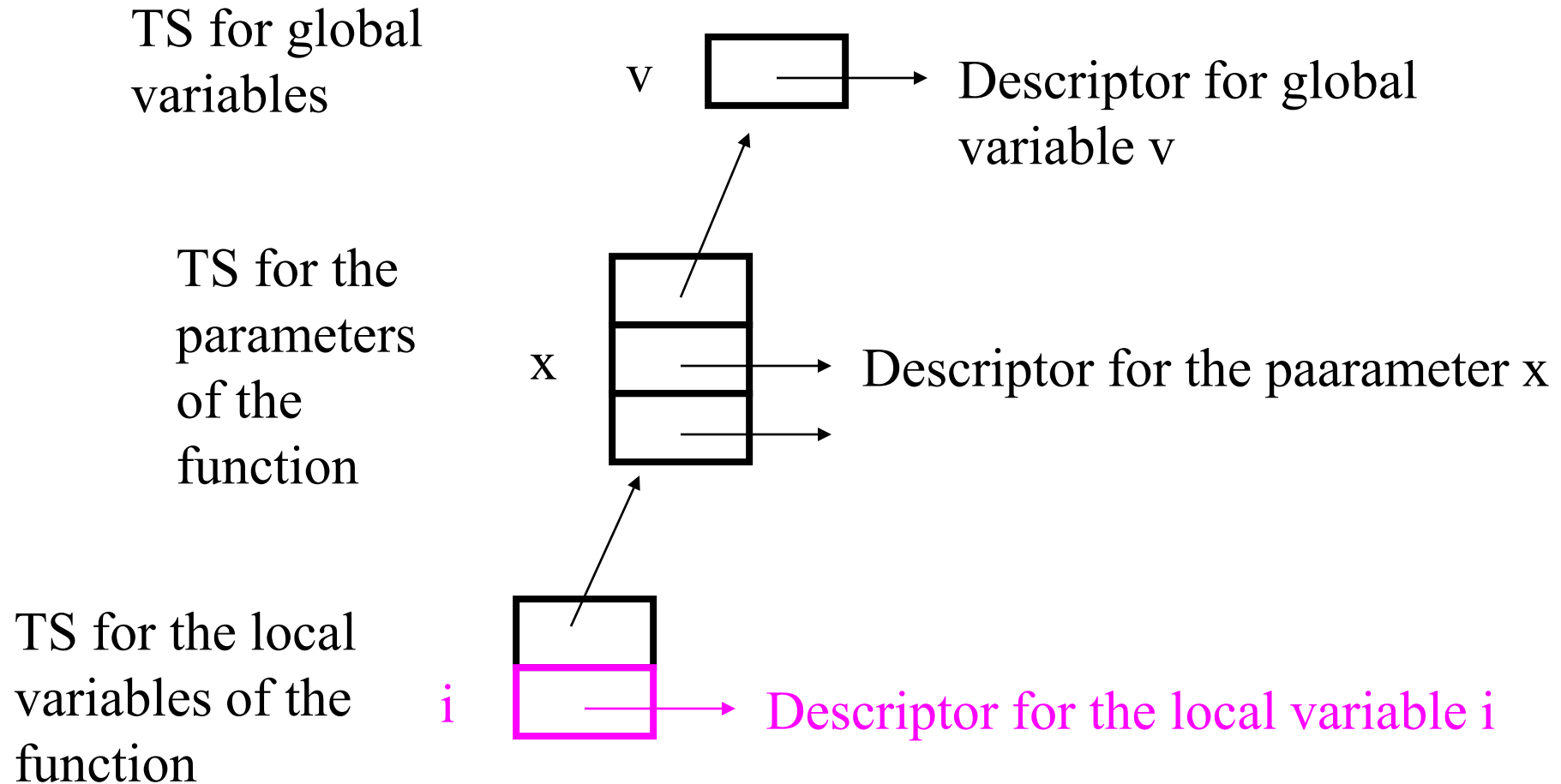
```
void add(int x, int[] v, int N)
{
  int i;
  i = 0;
  while (i < N) {
    v[i] = v[i]+x;
    i = i+1;
  }
}
```



# Hierarchy in Symbol Tables

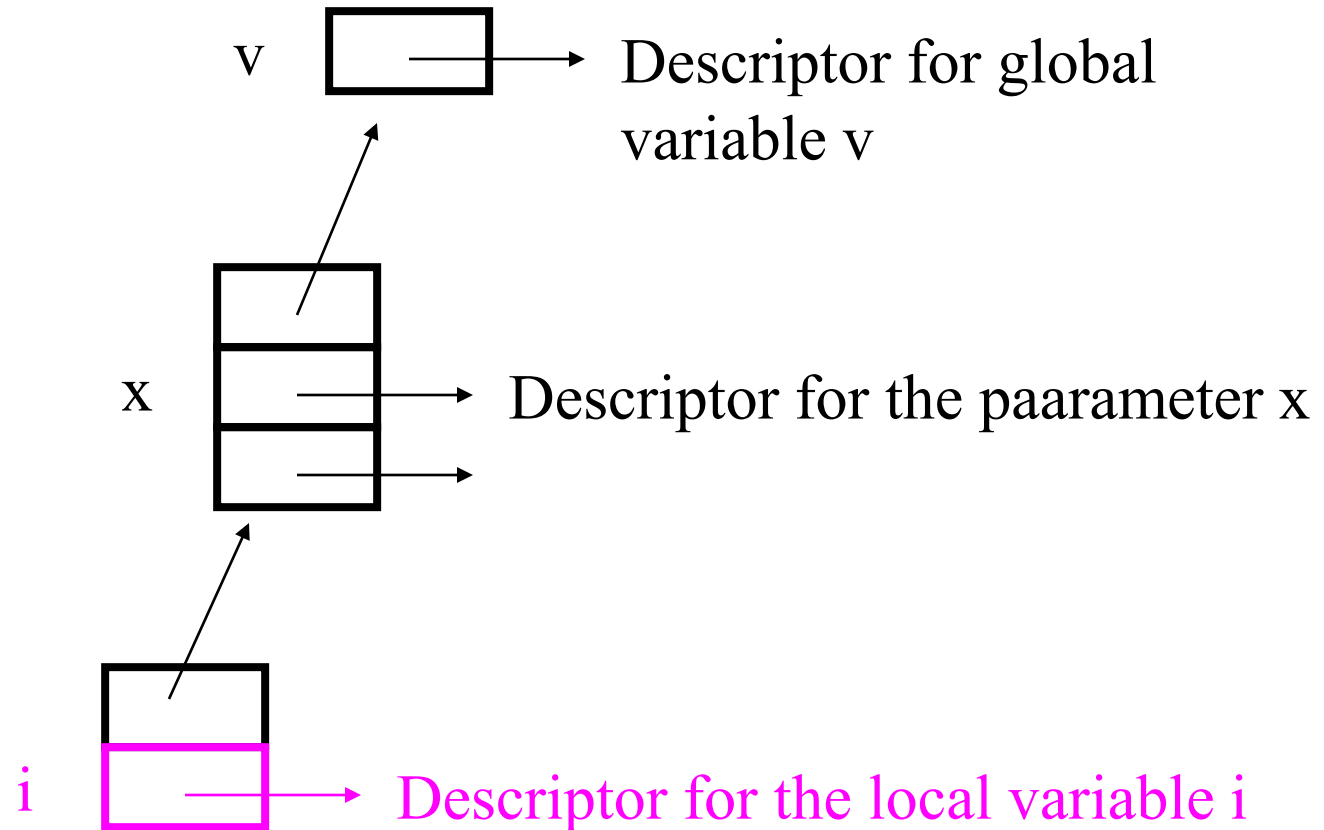
- Scope
  - The same name for a variable can have different meanings in different code locations
  - It is necessary a symbol tables for each scope
- The hierarchy derives from the nested scopes
- Hierarchy in the symbol tables reflects that hierarchy
- Lookup bottom-up traverses the hierarchy until it finds the descriptor

# Lookup i in an Example



# Lookup i in an Example

- $v[i] = v[i] + x;$
- First it searches in the TS of the local variables
- If don't find it then goes up and searches in the next hierarchy level



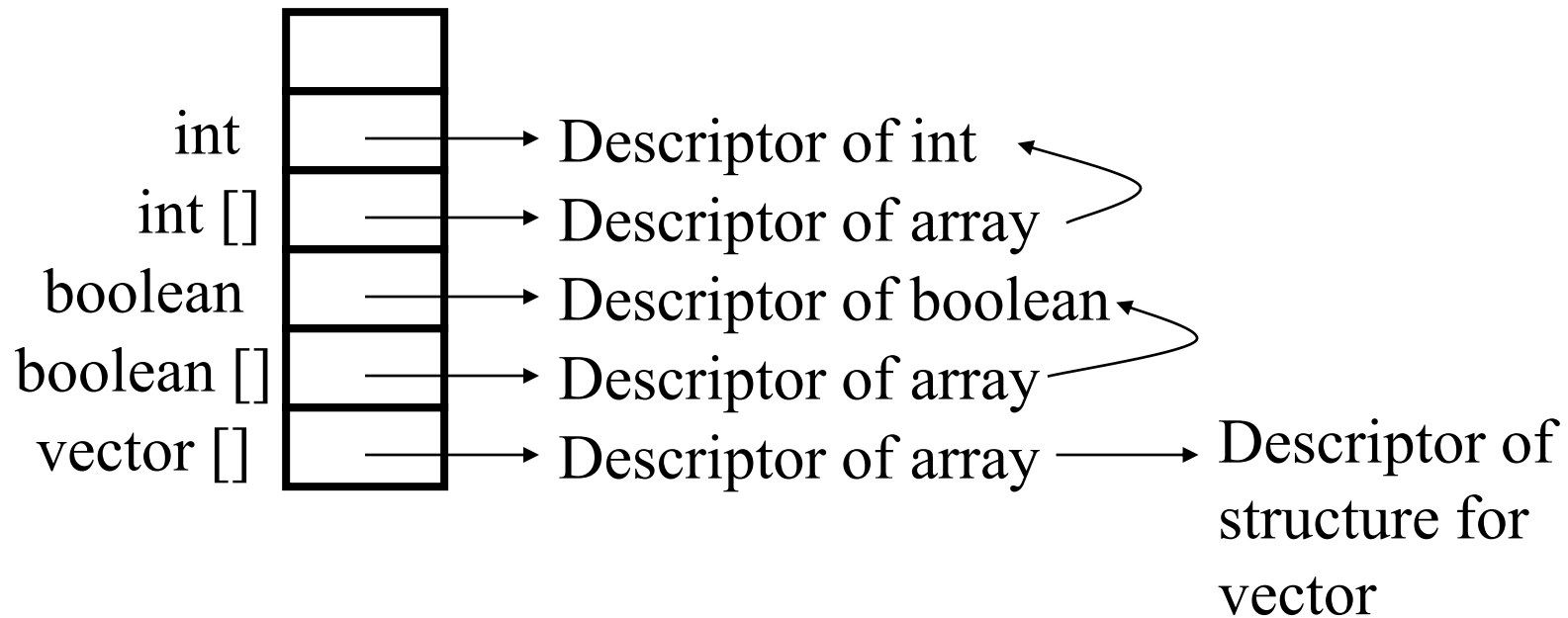
# Descriptors

- What they contain?
- Information used to perform semantic analysis and to generate code
  - Local descriptors: name, type, offset in the stack
  - Descriptors of functions
    - Signature (type of return, parameters)
    - Reference to the local symbol table
    - Reference to the code (IR) of the function

# Parameters, Local, and Descriptors of Types

- Parameters and Locals refer to type descriptors
  - Descriptor of base type: int, boolean, etc.
  - Descriptor of the array type: contains reference to the descriptor of the type for the array elements
  - Descriptor of structure, etc.

# Example: Symbol Table for Types

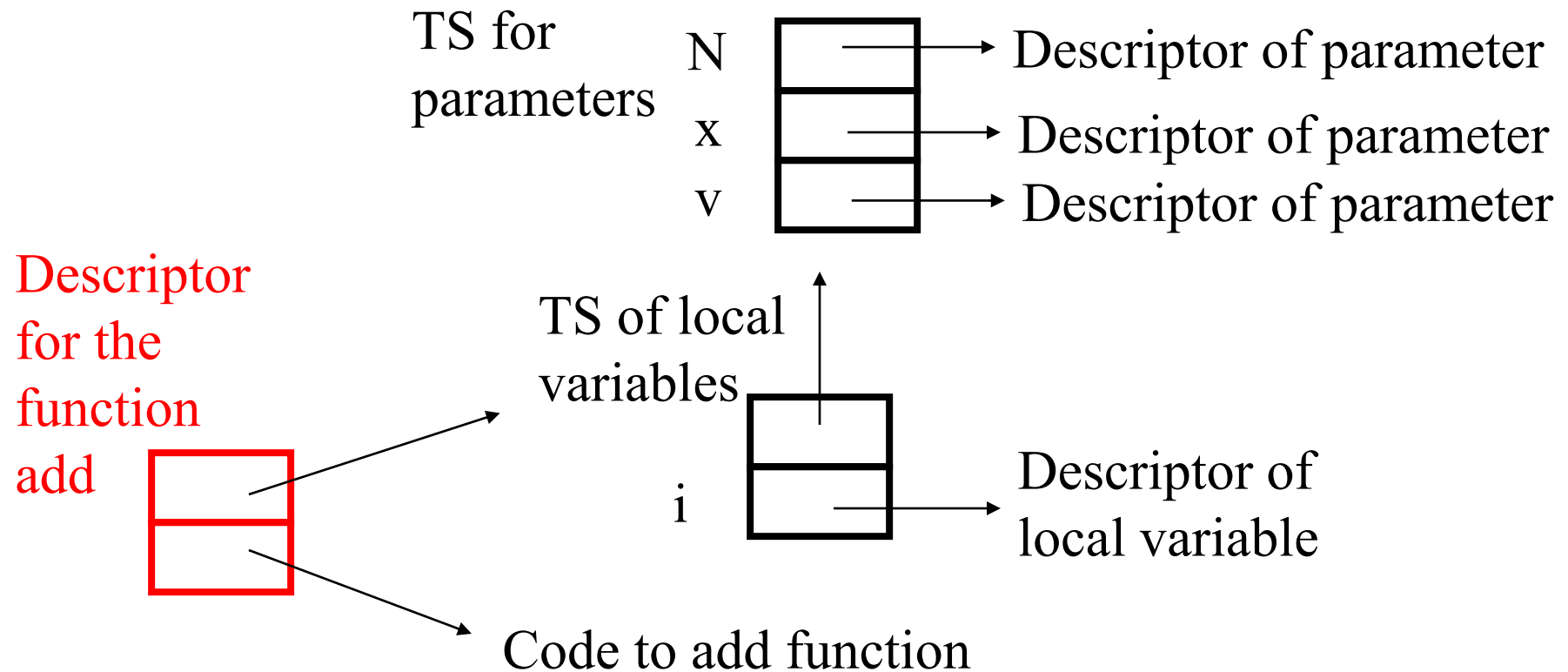


# Descriptor of Functions

- Contain reference for the code (IR) of the function
- Contain reference to the local symbol table (local variables of the function)
  - Note that the existence of more than one local scope implies the existence of a subhierarchy of local symbol tables
- In the hierarchy of the symbol tables the symbol table for the parameters is parent of the symbol table for the local variables



# Descriptor of the Function add



# What is a Syntax Tree?

- Tree that stores results of the syntactic analysis
- External nodes are terminals/tokens
- Internal nodes are non-terminals

# Abstract Trees vs. Syntax Trees

- Remember modifications to grammars
  - Left factorization, elimination of ambiguity, precedence of operators...
- Modifications result in trees that do not reflect an interpretation of the program intuitive and clear
- It can be more convenient to work with ASTs
  - ASTs can be seen as the syntax tree representing the grammar without the modifications

# Alternative Constructions for Intermediate Representations

- Construct the concrete syntax tree, translate it to AST, then translate AST to another intermediate representation
- Construct AST, then translate AST to another intermediate representation
- Include the construction of the intermediate representation during the syntax analysis
  - Eliminated the construction of the syntax tree – improves compiler performance
  - Less code to write

# Symbol Table

- Given a syntax tree (abstract or concrete)
  - Traverse recursively the tree
  - Construct the symbol table while traversing the tree

# Nested Scopes

- Various forms of nesting
  - Symbol Table of the functions nested in the symbol table of the globals
  - Symbol Table of the locals nested in symbol table function
- Nesting solves ambiguity in possible conflicts
  - Same name used for a global and a local variable
  - Name refers a local variable in a function

# Scopes in Nested Code

- Symbol tables can have arbitrary depth when considering nested code:

```
boolean x;  
int foo(int x) {  
    double x = 5.0;  
    { float x = 10.0;  
        { int x = 1; ... x ...}  
        ... X ...  
    }  
    ... X ...  
}
```

Note: Conflicts in names with nesting can refer program errors. Compilers usually report warning messages in the presence of this kind of conflicts.

# High-Level Code Representation

## ➤ Basic idea

- Moving towards the target language (e.g., assembly)
- Preserve control structure
  - Format of objects
  - Structured control flow
  - Distinction between parameters, local variables, fields, etc.
- High-level of abstraction of the assembly language
  - load and store nodes
  - Access to abstract local storage, parameters and fields, and not memory positions directly



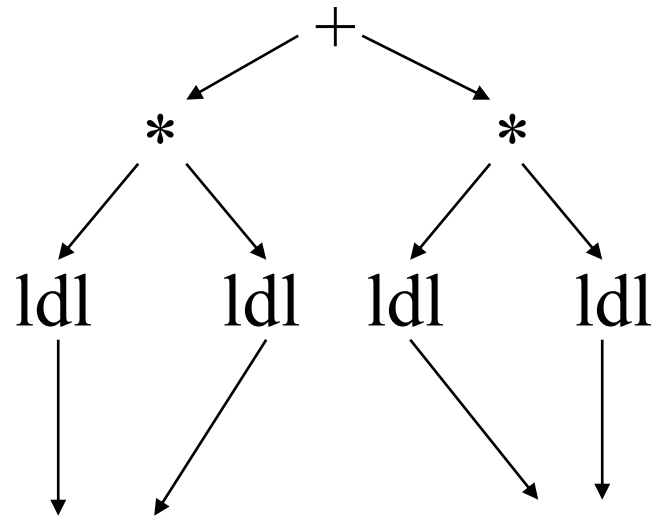
# Representation of Expressions

- Expression trees represent the expressions
  - Internal nodes – operations such as +, -
  - Leafs – Load nodes represent access to variables
- Load nodes
  - **ldl** to access local variables – local descriptors
  - **ldp** to access parameters – parameter descriptors
  - **lda** to access array elements
    - Expression tree for the value
    - Expression tree for the index
  - For loads of class attributes, of fields of structs...

# Example

x and y are local variables

$x * x + y * y$



Local descriptor for **x**  
In the local symbol table

Local descriptor for **y**  
In the local symbol table

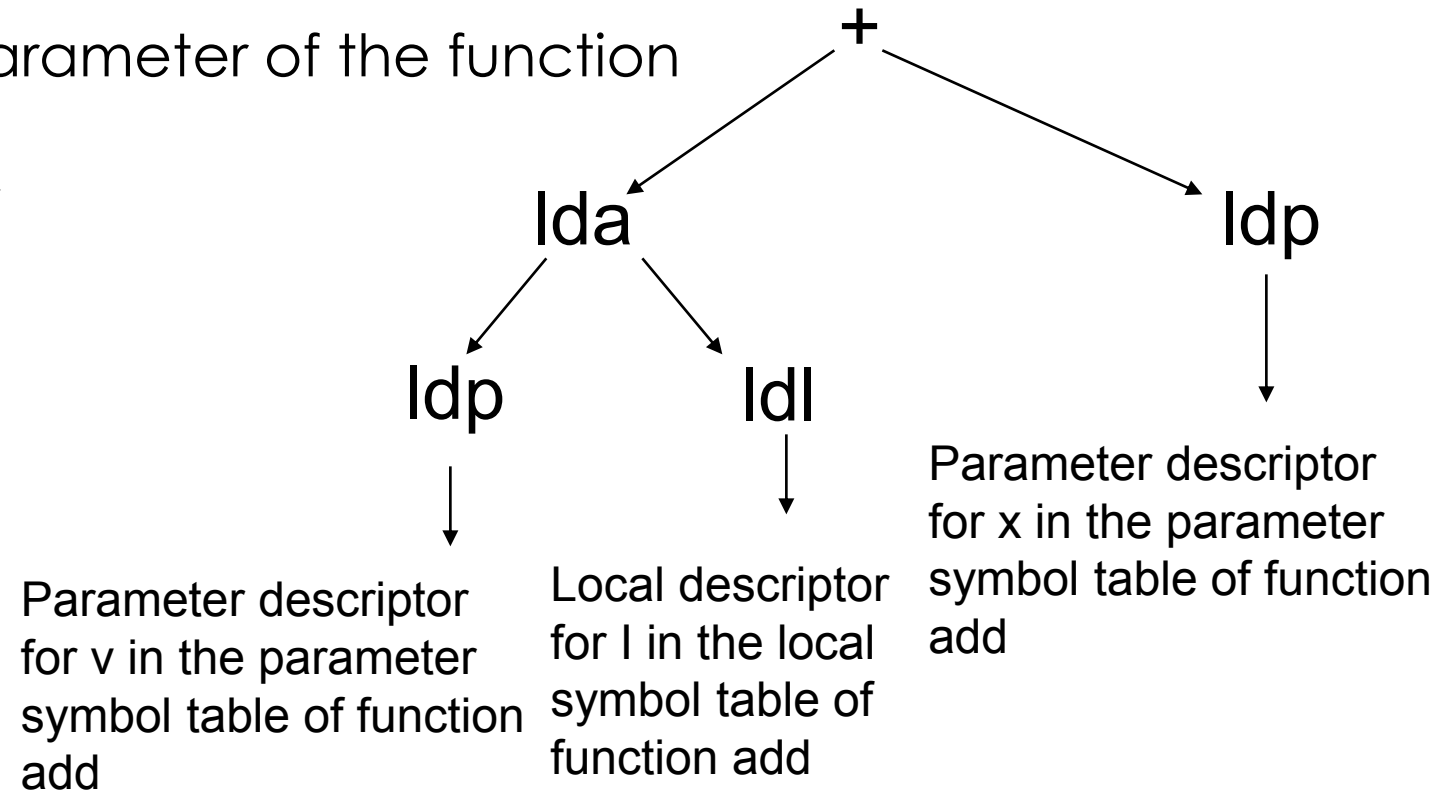
# Example

v is an array passed as parameter in function add

i is a local variable

x is a parameter of the function

$v[i] + x$

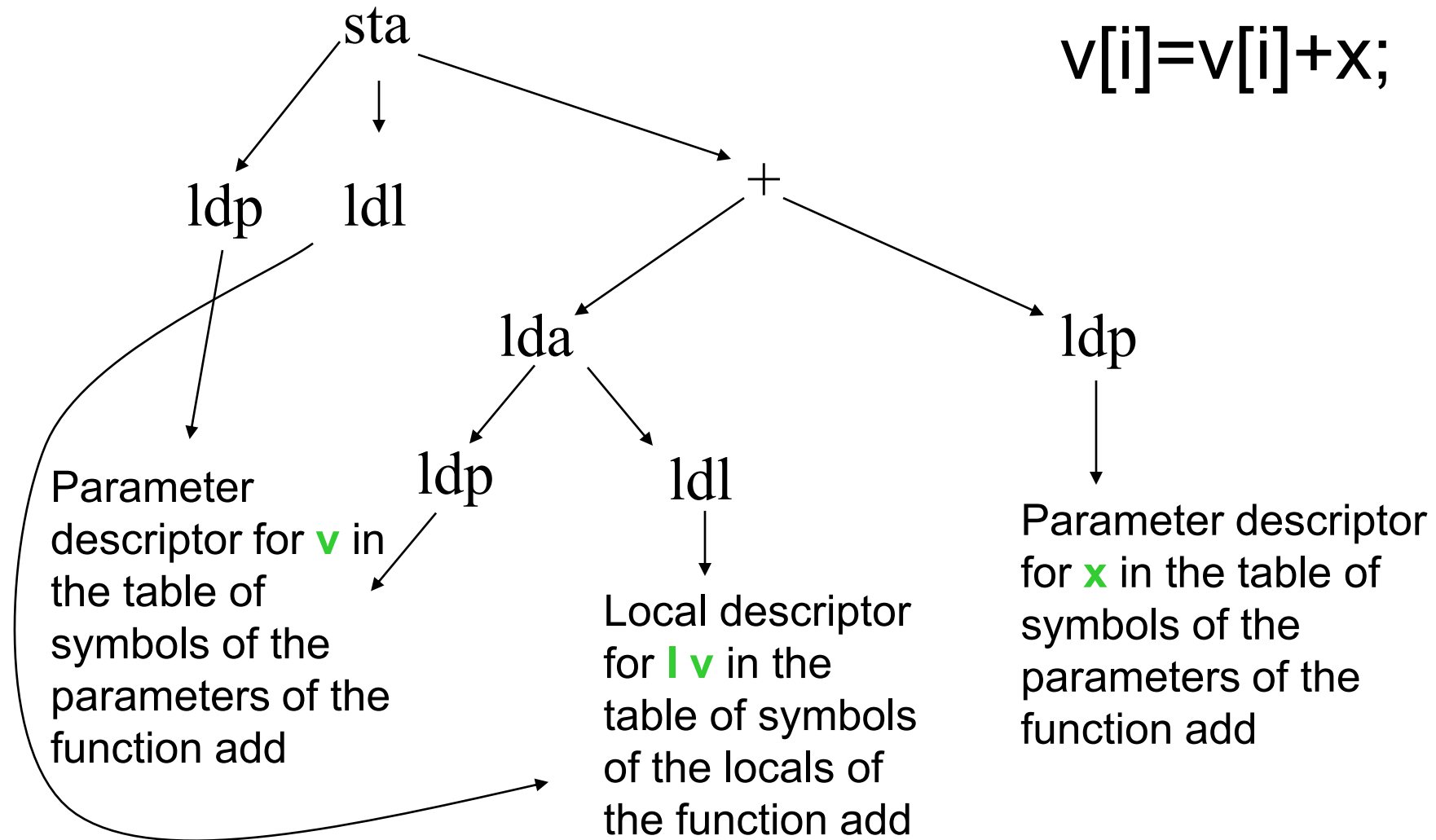


# Representing Assignment Statements

## ➤ Store Nodes

- **stl** for *stores* of local variables
  - Local descriptor
  - Expression tree for the value to store
- **sta** for *stores* in array elements
  - Expression tree for the array
  - Expression tree for the index
  - Expression tree for the value to store
- For stores in class attributes, in fields of structs...

# Example



# Orientation

- Intermediate representations
  - Moving in the direction of the target language (e.g., machine language)
  - Support for compiler analysis and transformations
- High-Level IR (*intermediate representation*)
  - Preserves the structure of objects, arrays, control flow,...
  - Symbol Tables
  - Descriptors

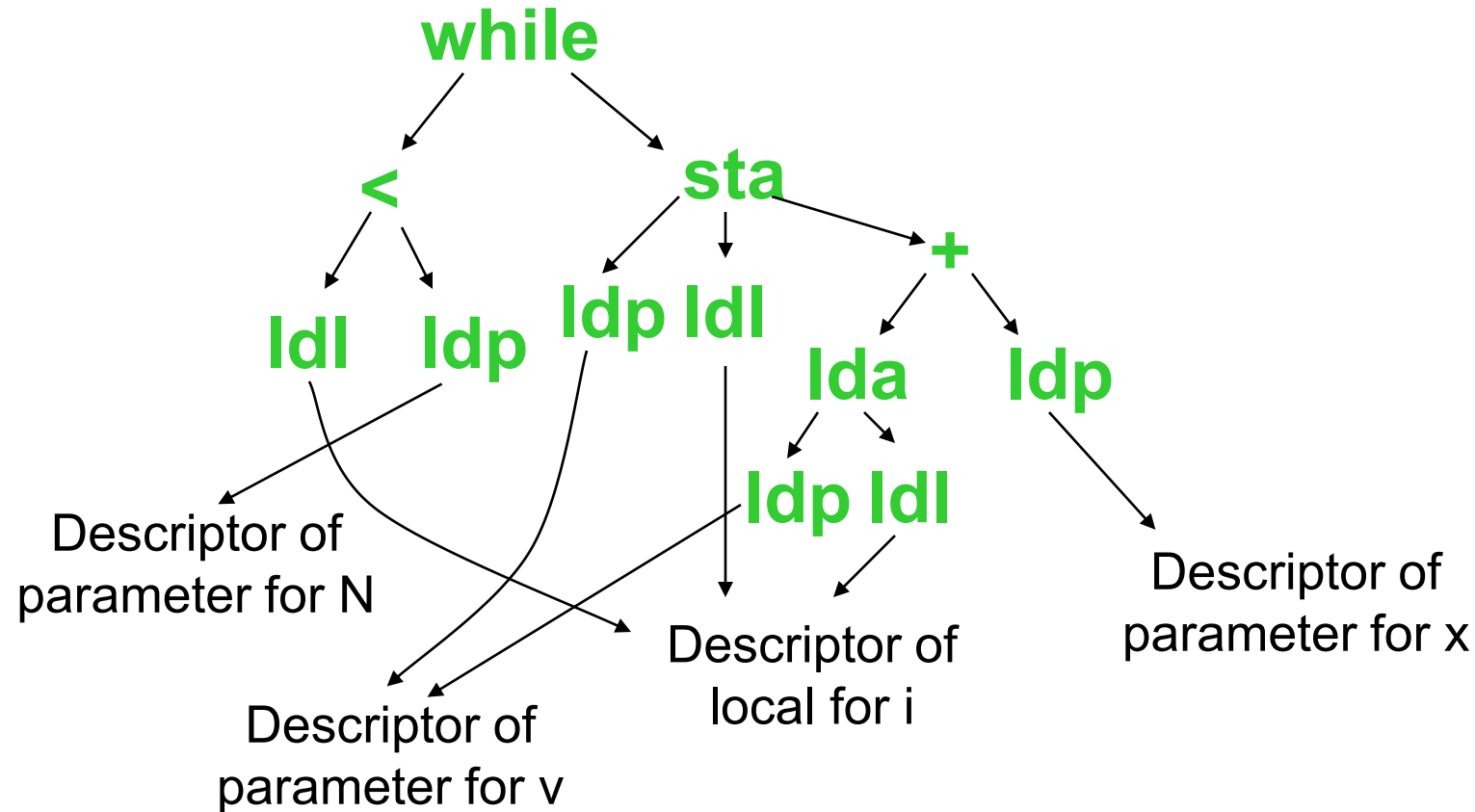
# Representing Control Flow

## ➤ Nodes of statements

- **if** node
  - Expression tree for **condition**
  - Node for the body of the **then** and node for the body of the **else**
- **while** node
  - Expression tree for **condition**
  - Node for the **body**
- **return** node
  - Expression tree for the return **value/expression**
- **One can easily think about what is needed for:**
  - **For** node
  - **Do while** node
  - **Switch** node
  - Etc.

# Example

```
while (i < N)  
    v[i] = v[i] + x;
```

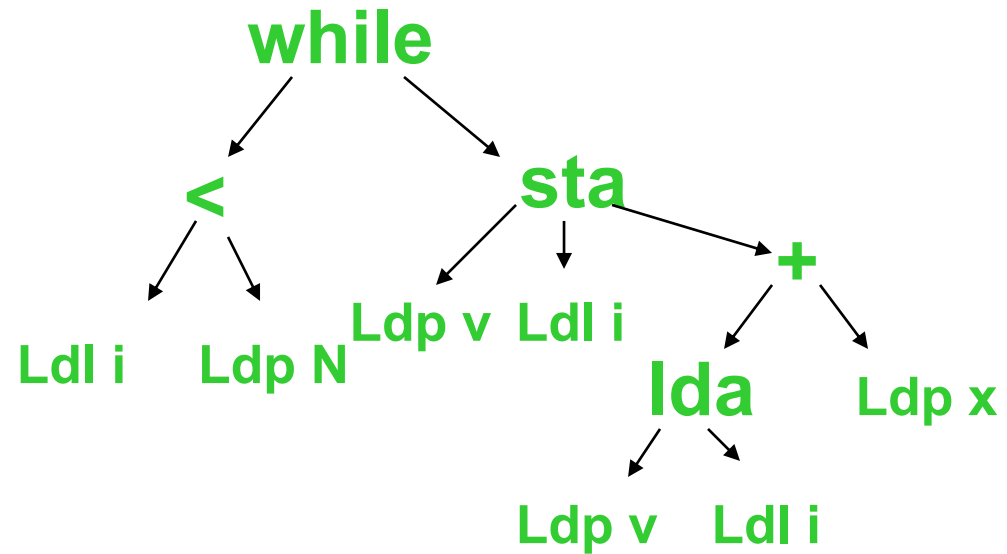




# Example

## ○ Abbreviated notation

while (i < N)  
v[i] = v[i] + x;



# From Syntax Trees to IR

- Traverse recursively the syntax tree
- Build representation *Bottom-Up*
  - Check identifier of the variable in the symbol tables
  - Construct load nodes to access the variables
  - Construct expressions from the load nodes and the operation nodes
  - Construct store nodes for the assignments
  - Include while, if, return for the control constructs

# Summary

## High-Level Intermediate Representation

- Goal: to represent the program in an intuitive mode to support further compilation stages
- Representation of the data in the program
  - Symbol tables
  - Hierarchic organization
- Representation of the computations
  - Expression trees
  - Various types of load and store nodes
  - Structured control flow

# Semantic Analysis: Errors

- We assume the inexistence of problems during the construction of the IR
- However, it is necessary to do many verifications while constructing the IR
- Named by **Semantic Analysis**
- Semantic Analysis is usually done at the abstract syntax tree level
  - In order errors be informative/clear it is necessary to that the tree nodes are annotated with positions in the program

# Objective of the Semantic Analysis

- To ensure that the program obeys to a set of sanity checks, such as:
  - All the variables used have been declared
  - Types are used in a correct way
  - Calls to functions have the correct number of arguments, the correct types of the arguments, and the correct type for the return
- Verification while building the IR

# Descriptors for Identifiers

- When the descriptor of a local variable, a parameter, etc. is built we have:
  - Name of the type
  - Name of the variable
- What is verified?
  - Verify if the name of the type identifies a valid type
    - *lookup* name in the symbol table for the types
      - If it was not found then **semantic error**

# Local Table of Symbols

- When we build a local symbol table we have a list of local descriptors
- We shall verify what?
  - Duplicated names of variables
- When to do the verification?
  - When the descriptor is inserted in the local symbol table
- Similar to table of symbols of parameters, globals, etc.

# Verification for loads, stores, etc.

- What does the compiler have?
  - Name of variable
- What does it do?
  - *Lookup* name of variable:
    - Verifies if it is in the symbol table of locals, reference to a local descriptor
    - Verifies if it is in the symbol table of parameters, reference to a parameter descriptor
    - Verifies if it is in the symbol table of globals, reference to a global descriptor
    - If a descriptor was not found then **semantic error (the variable was not declared)**



# Verification for Load Instructions for Arrays

- What does the compiler have?
  - Name of the variable
  - Expression of indexing the array
- What does it do?
  - *Lookup* name of the variable
    - If it is not found then **semantic error**
  - Verifies type of expression
    - If it is not an integer then **semantic error**

# Addition Operation

- What does the compiler have?
  - 2 expressions
- What can be wrong?
  - Expressions have the wrong type
  - E.g., they must be both integers
- It is why the compiler verifies the type of the expressions
  - Load instructions store the type of the variable accessed
  - Operations store the type of the produced expression
  - So, it is only necessary verify the types
    - If fails then **semantic error**

# Inference of types for addition operations

- Some languages let add floats, ints, doubles
- What are the problems?
  - Type of the result of the operation
  - Conversion of the operands of the operation
- Standard rules are usually applied:
  - If addition of an **int** with a **float**
    - Convert the **int** to **float**, add the two **floats**, and the result is a **float**
  - If addition of a **float** with a **double**
    - convert **float** to **double**, add the two **doubles**, result is a **double**

# Rules for Addition

- Basic principle: hierarchy of types for numbers (int, then float, then double)
- All the “forced” conversions are done in bottom-up mode in the hierarchy
  - E.g., int to float; float to double;
- Result has the type of the operand with type in the highest level of the hierarchy:
  - $\text{int} + \text{float} \rightarrow \text{float}$ ,
  - $\text{int} + \text{double} \rightarrow \text{double}$ ,
  - $\text{float} + \text{double} \rightarrow \text{double}$

# Type Inference

- Inference of types without explicit declaration of types
- Addition is a restrict case of type inference
- Very important topic in the context of some programming languages (e.g., dynamic languages such as JavaScript, MATLAB)

# Store Instruction

- What does have the compiler?
  - Name of the variable
  - expressions
- What does it do?
  - *Lookup* of the name of the variable
    - if it is not found: **semantic error**
  - Verifies if the type of the variable is compatible with the type of the expression
    - If not: **semantic error**

# Store Instruction for Arrays

- What does have the compiler?
  - name of the variable, expression for indexing
  - expression
- What does it do?
  - *Lookup* with name of variable
    - if it is not found: **semantic error**
  - Verifies if the type of the indexing expression is integer
    - If not: **semantic error**
  - Verifies if the type of the elements of the array is compatible with the type of the expression
    - If not: **semantic error**

# Function Calls

- What does have the compiler?
  - Name of the function, arguments
- Verifications:
  - Name of the function is identified in the table of the functions of the program
  - Type of arguments match with the type of parameters in the declaration of the function



# Summary of Semantic Verifications

- Do the semantic verifications during the construction of the Intermediate Representation (IR)
- Many verifications are to certify that we build a correct IR (i.e., an IR that represents the same functionality of the input program)
- Other verifications are simple sanity checks
- Each programming language has a list of verifications
- Semantic analysis can report many potential errors

# Summary

- Translation of syntax trees to high-level IR
  - Preserves the structured control flow
  - Representation efficient for high level analysis and high-level optimizations (e.g., target-independent transformations)