Low-Level Intermediate Representation

Compilers course

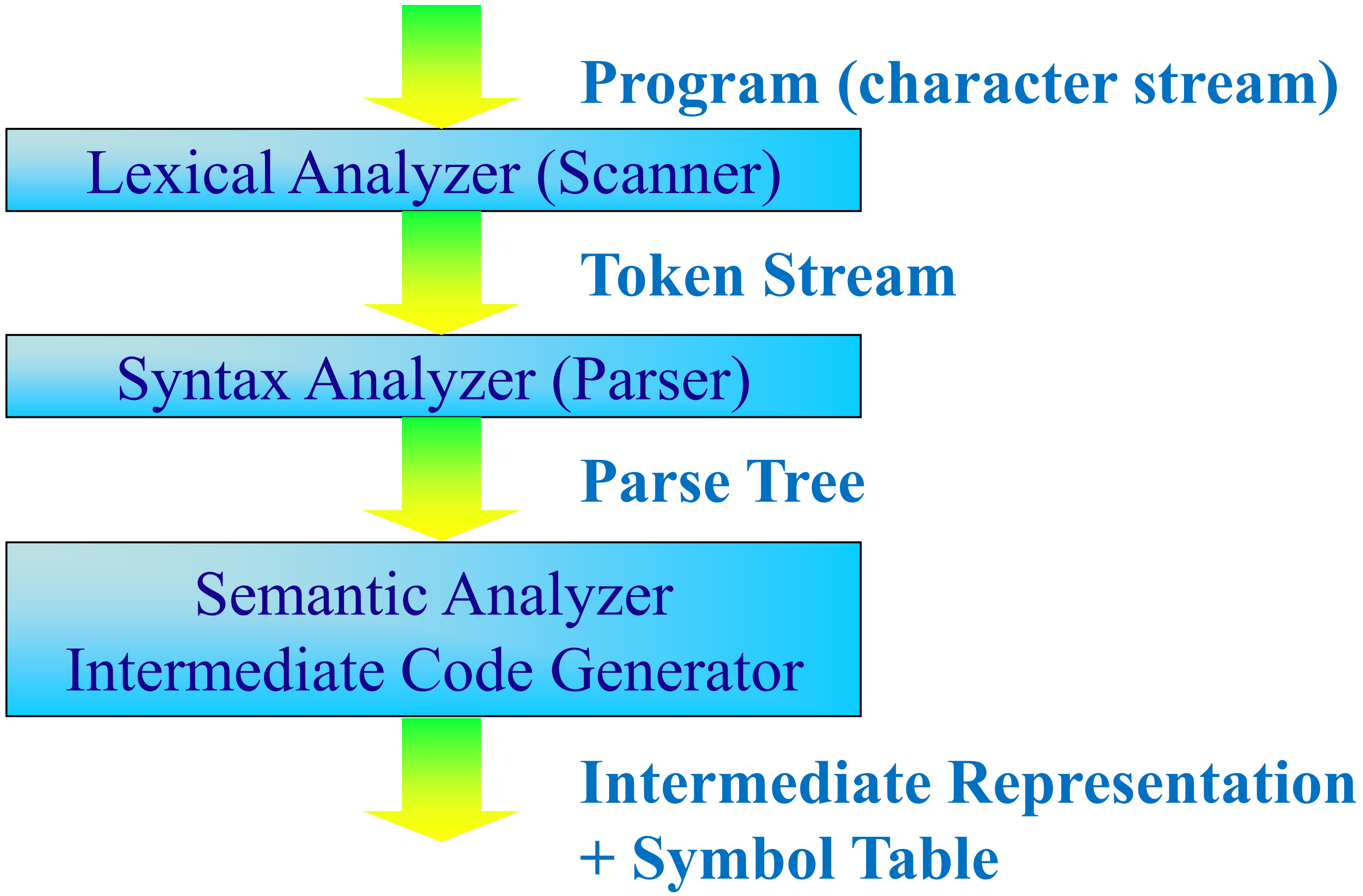
Masters in Informatics and Computing Engineering (MIEIC), 3rd Year

João M. P. Cardoso
Email: jmpc@fe.up.pt

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Compiler Stages


Program (character stream)
Lexical Analyzer (Scanner)
Token Stream
Syntax Analyzer (Parser)
Parse Tree
Semantic Analyzer
Intermediate Code Generator
Intermediate Representation
+ Symbol Table

Most slides based on the Martin Rinard's MIT course

Conversion to Low-Level IR

- Convert structured control flow into control flow based on jumps (non-structured)
  - Conditional and inconditional branches
- Convert structured memory model into flat memory model
  - Flat addressing for variables
  - Flat addressing for arrays
- Continues independent of the machine language, but:
  - Movement to very close to the machine, to a standard machine models (flat space address, jumps)

Program Representations

- *Control Flow Graph* (CFG) where:
  - Nodes of the CFG are nodes of instructions
    - stl, sta, cbr, ldl, lda, ldp are nodes of instructions
    - +, *, <, ... are nodes of expressions
  - Edges in the CFG represent control flow
  - *Forks* in conditional branches
    - Represent two or more possible paths
  - *Merges* when the control can reach a point through multiple paths
  - An entry node and an exit node

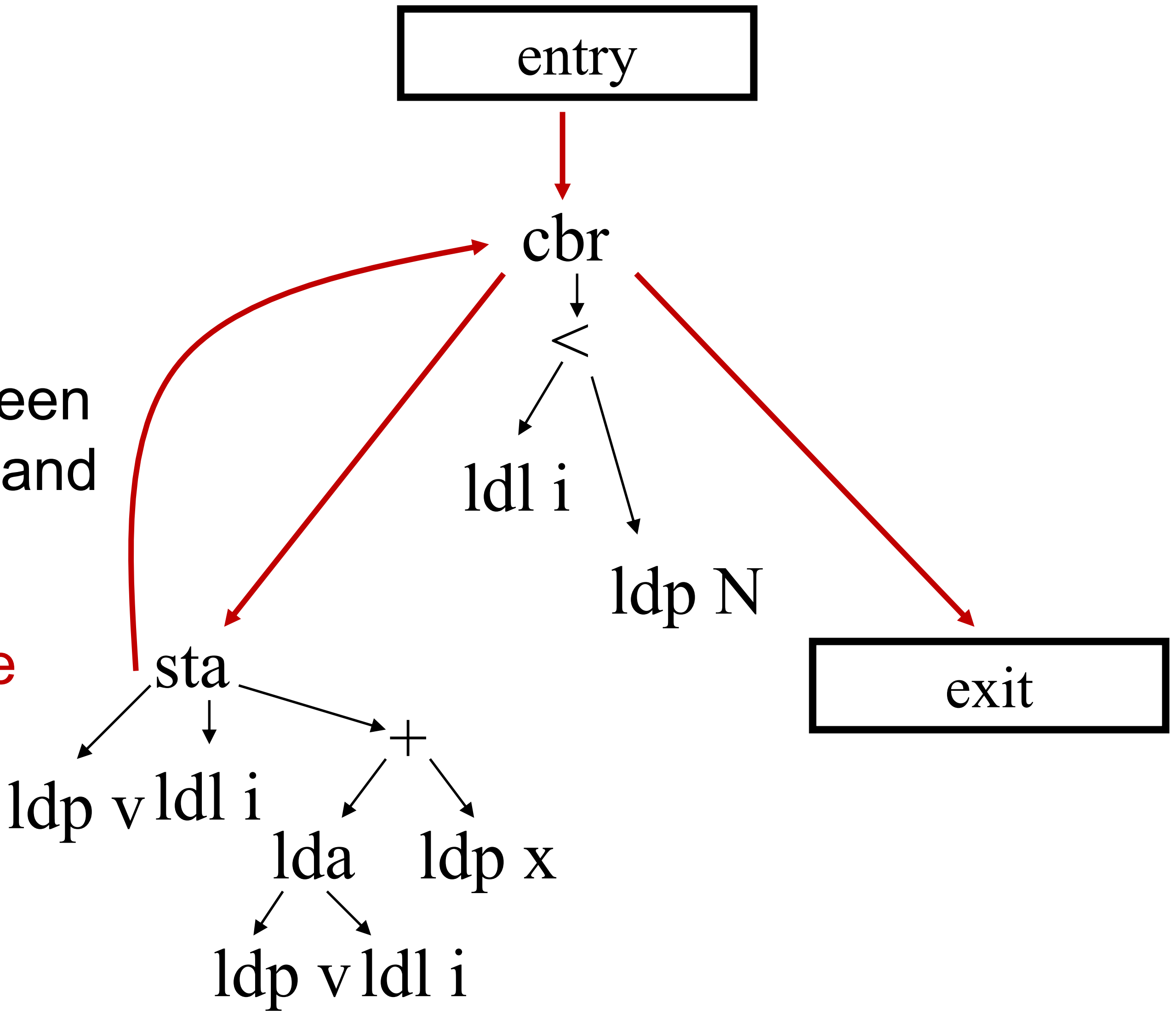
Note that there are other uses of CFGs, at higher-levels, with nodes representing basic blocks, etc.

Example: CFG

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

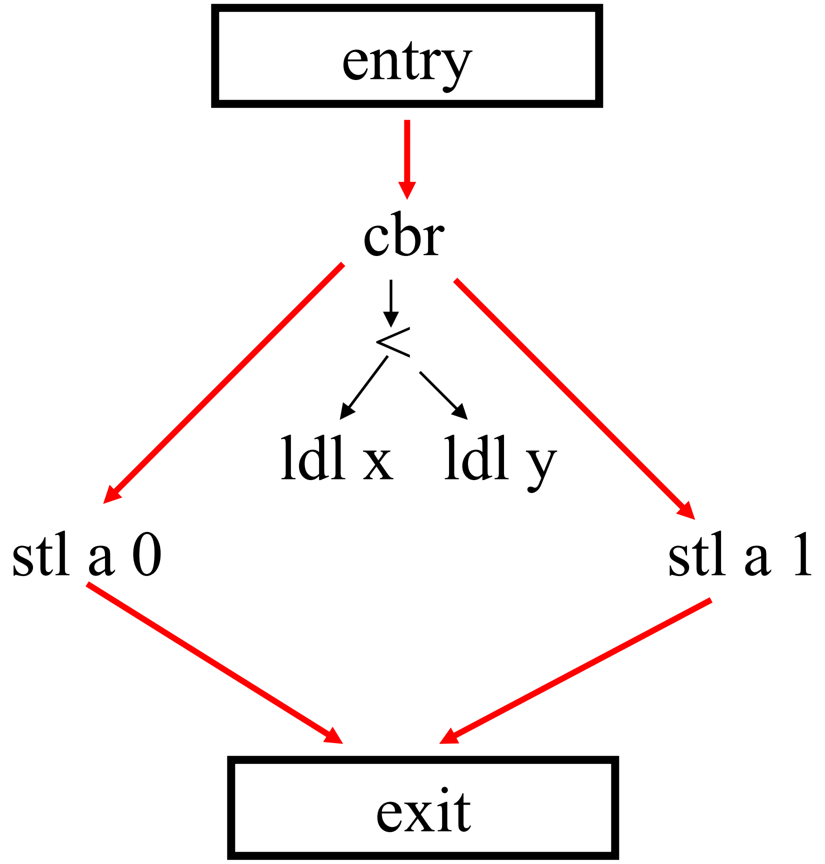
↓ Edges between instructions and expressions

↓ Edges of the control flow



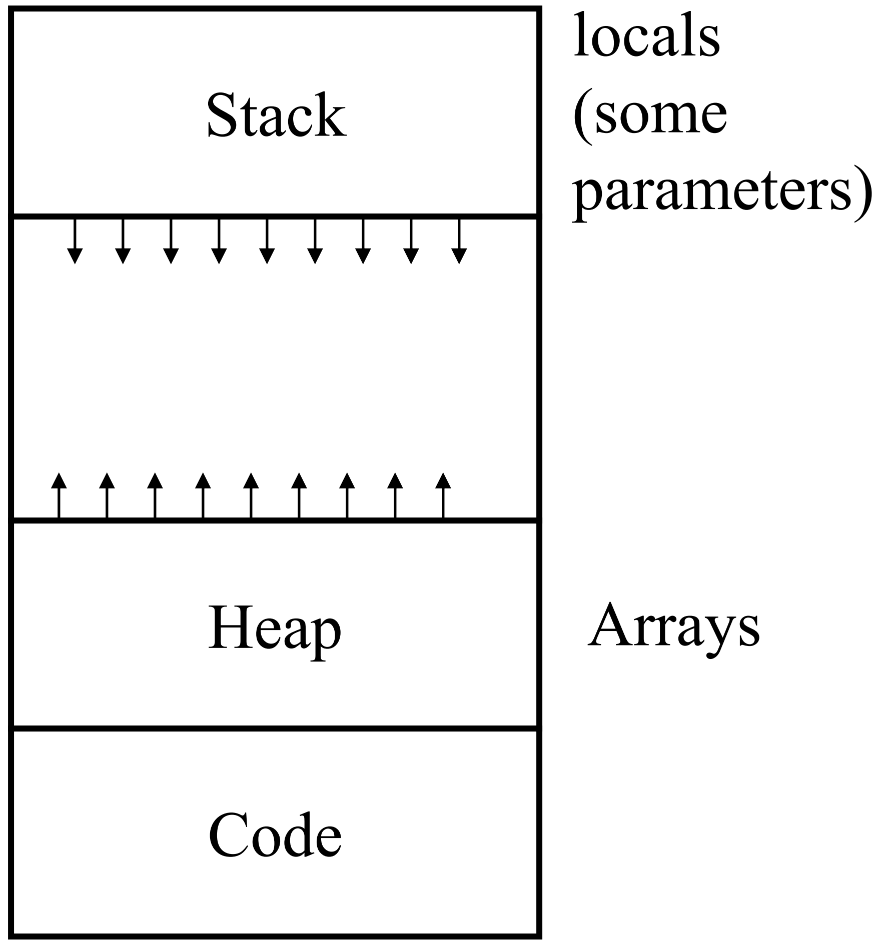
Example: CFG

```
if (x < y) {
    a = 0;
} else {
    a = 1;
}
```



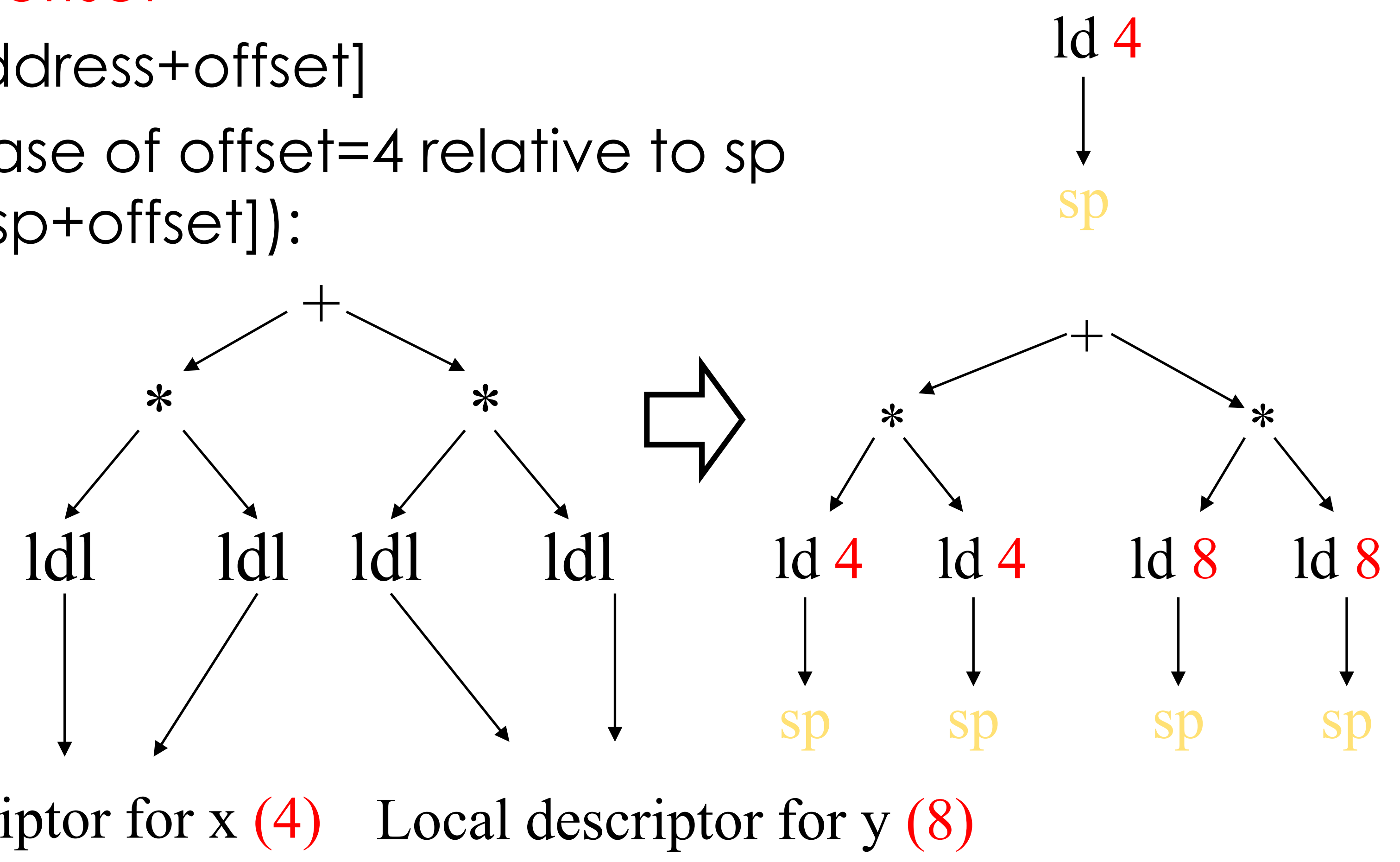
Memory Model of the Target Machine

- A flat memory
  - Composed by words
  - Byte addressable
- Nodes model Load and Store instructions
  - ld addr,offset – result is the value in the memory position: addr+offset
  - st addr, offset, valor – write value in the memory position: addr+offset
  - Substitute nodes lda and ldl by ld nodes
  - Substitute nodes sta and stl by st nodes



Example:

- ld address offset
  - MEM[address+offset]
  - In the case of offset=4 relative to sp (MEM[$sp+offset]):
- x*x+y*y



Parameters

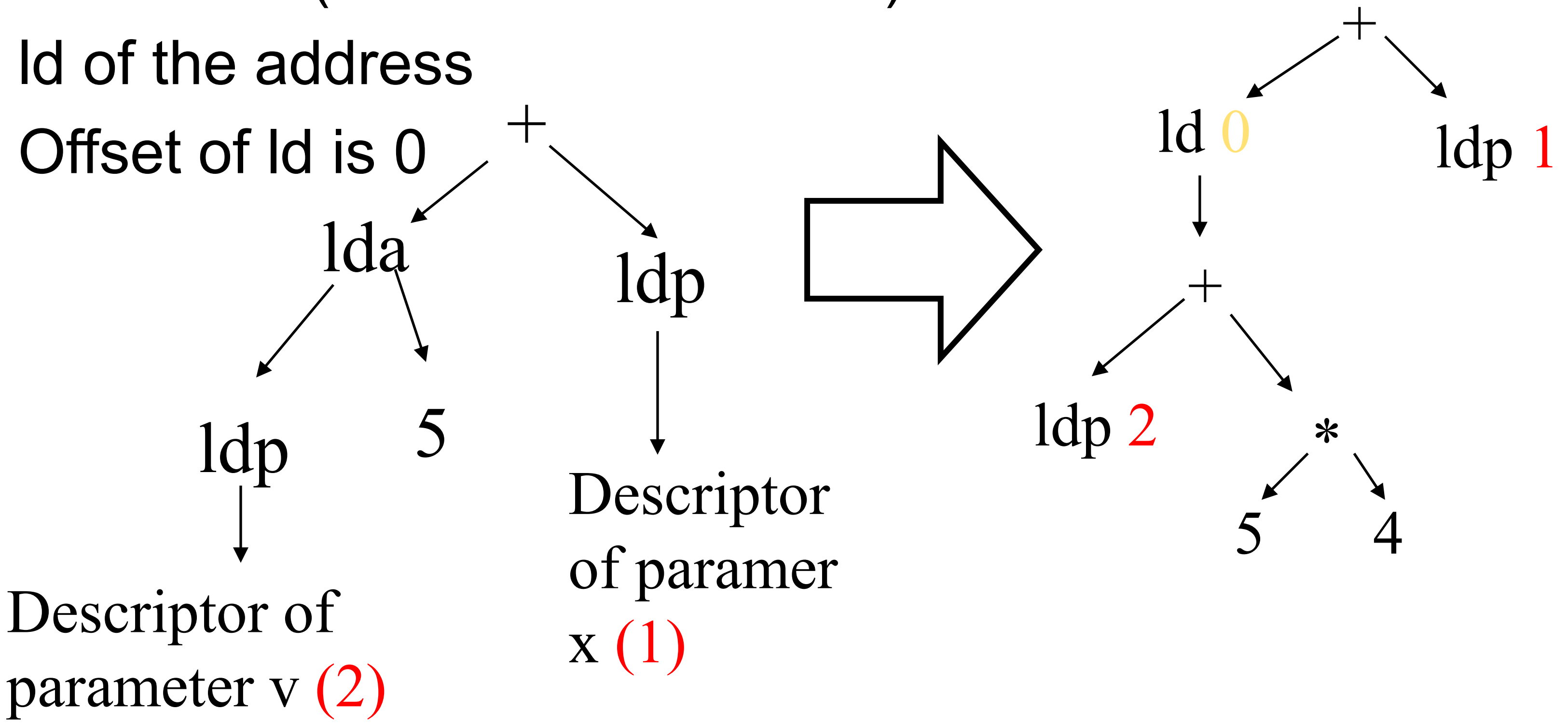
- Many processors have conventions in the calls
  - First parameter in register 5, second parameter in register 6, ...
  - See $a0, $a1, … in MIPS
- Conventions vary with the machine
- Let’s assume that each parameter is a word
- Let’s address the parameters by the number
  - ldp <number of parameter>

Access to Array Elements

- Assume that the variable points to the first element of tha array
- Array elements stored in contiguous memory positions
- What is the address of v[5]?
  - v is an array of integers: assume integers of 4 bytes
  - (addres of v) + (5*4)
- Address calculation
  - Base of Array + (index * element size)

Example: v[5]+x

- Conversion of lda nodes to ld nodes
- Calculate address
  - Base + (index * element size)
- ld of the address
- Offset of ld is 0



Local Variables

- Assume they are stored in the call stack
  - Address calculated using the offset from the stack pointer
- Remember:
  - Stack grows down and thus the offsets are positive numbers
  - Special symbol sp contains pointer to the stack

Actions in Function Calls (remember)

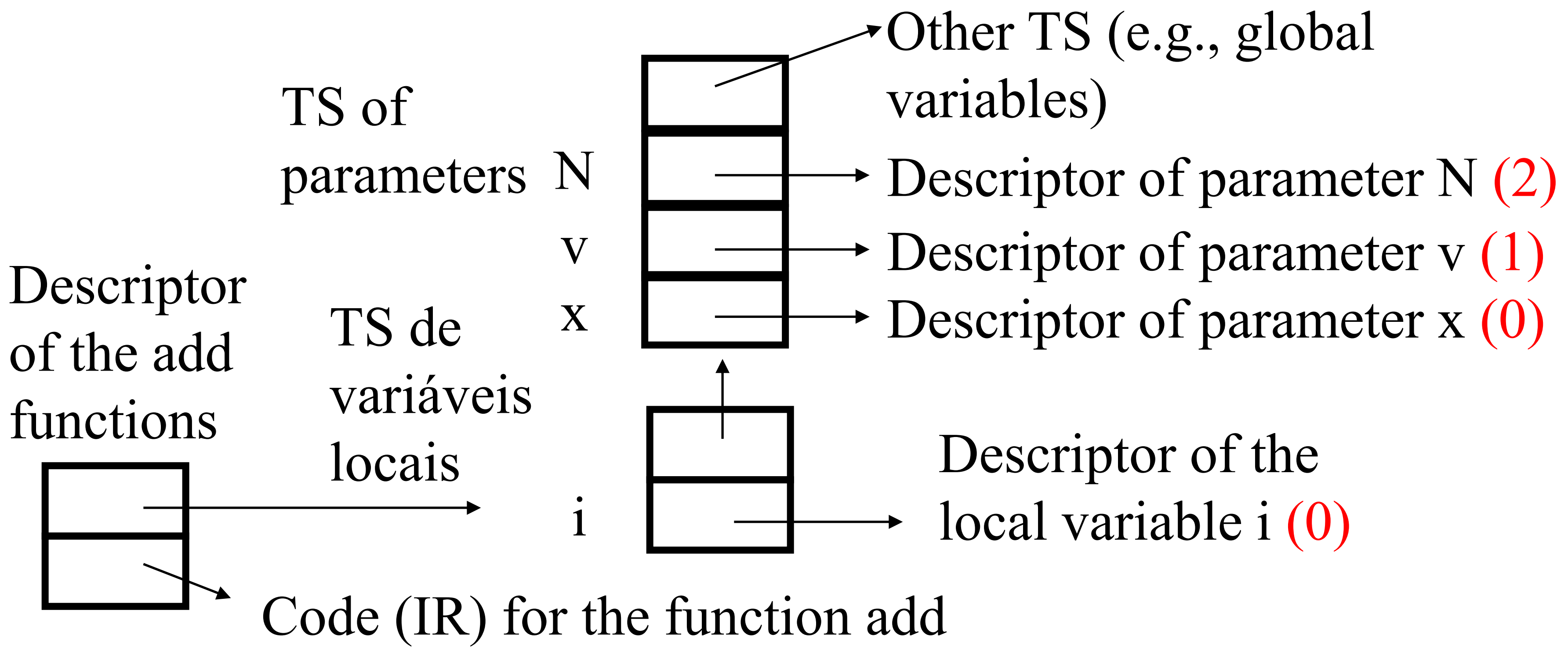
- Caller
  - Define parameters according to the convention on calls
  - Define return address using the calling convention
  - Branch to callee function
- Callee
  - Allocate stack frame = move (down) the stack pointer (sp)
  - Define return value according to the calling convention
  - Free stack frame = move (up) the stak pointer (sp)
  - Return to caller function

Management of the Stack (remember)

- Determine stack frame size
  - Allocate when the execution enters in the function
  - Free before returning from the function
  - Store all local variables
  - Additional space for parameters (when these surpass the number of registers assigned by convention for the arguments of the function)
- Define offsets for the local variables and parameters stored in the stack
  - Stored in the symbol tables (descriptors) of the locals and of the parameters
  - Continues to use ldp nodes to access to parameters

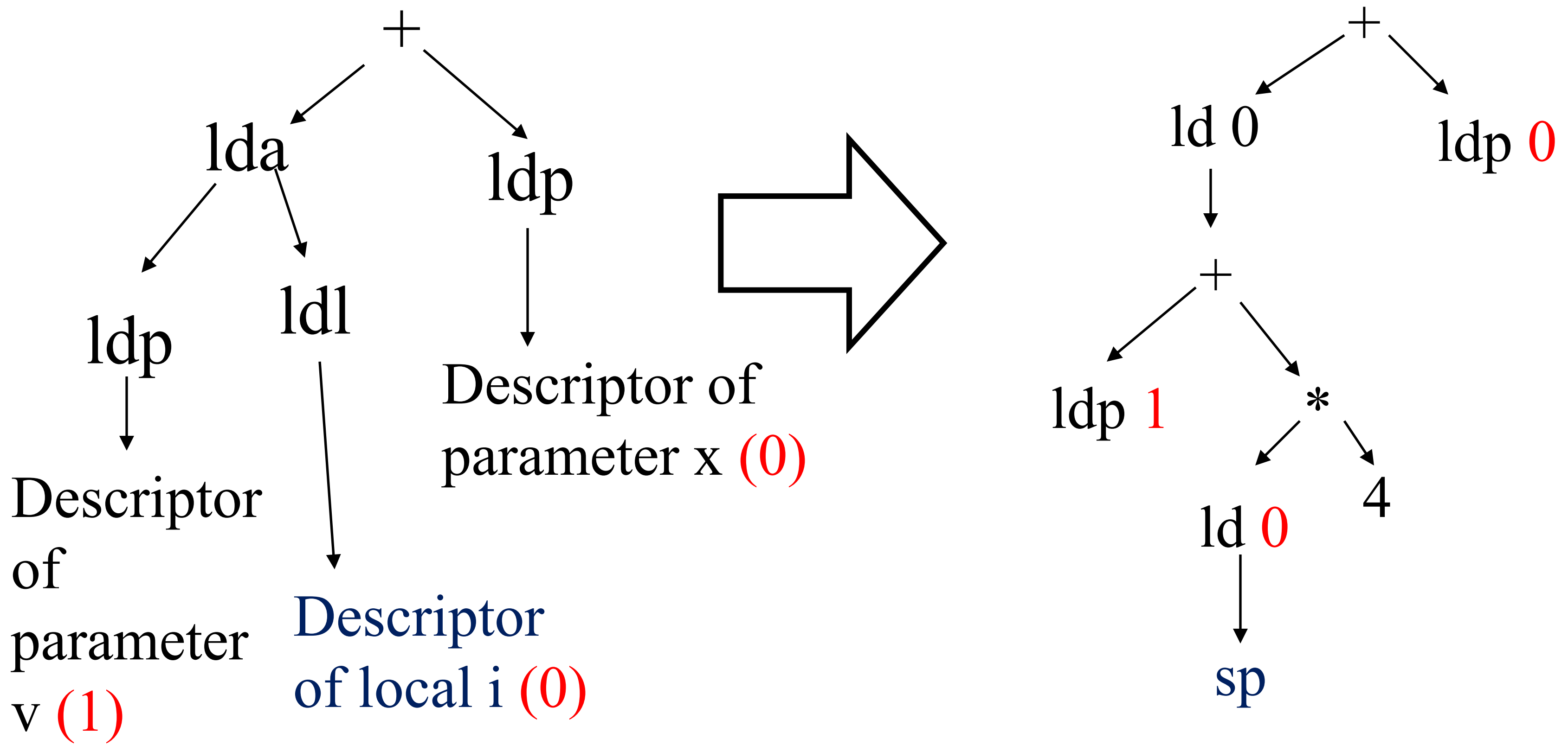
Elimination of ldl Nodes

- Use of *offsets* in the symbol table of locals and sp
- Replace *ldl* nodes by *ld* nodes
- Example of offsets for locals and parameters



TS of parameters
N
v
x
Other TS (e.g., global variables)
Descriptor of parameter N (2)
Descriptor of parameter v (1)
Descriptor of parameter x (0)
Descriptor of the add functions
TS de variáveis locais
i
Descriptor of the local variable i (0)
Code (IR) for the function add

Example: v[i]+x


+
lda
ldp
ldp
ldl
Descriptor of parameter v (1)
Descriptor of local i (0)
Descriptor of parameter x (0)
+
ld 0
ldp 0
+
ldp 1
*
ld 0
4
sp

Enter and Exit Nodes: add function

```
void add(int x, int[] v, int N) {
    int i;
    ...
}
```

$$\begin{tikzcd}[row sep=small] \text{enter } 4 \arrow[d] \\ \text{....} \arrow[d] \\ \text{exit } 4 \end{tikzcd}$$

- Stack size for add function?
  - **4 bytes (space for i)**
    - Assuming 4 bytes for int
    - Assuming function parameters in registers to pass arguments
- Enter and exit nodes are anotated with the size of the stack needed for the function

Example

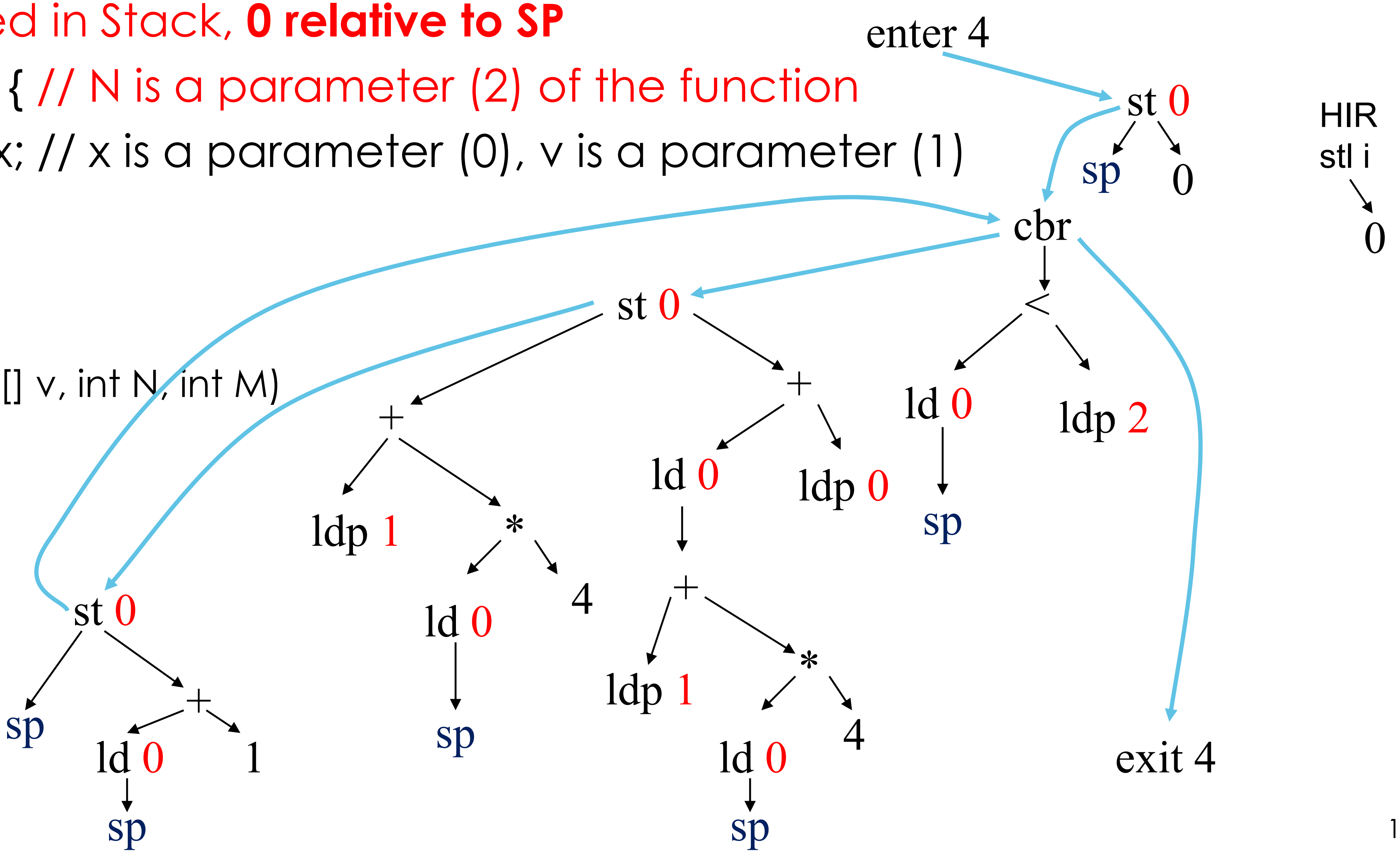
```
i=0; // i stored in Stack, 0 relative to SP
while (i < N) { // N is a parameter (2) of the function
   v[i] = v[i]+x; // x is a parameter (0), v is a parameter (1)
   i++;
}
```

Void f(int x, int[] v, int N, int M)

X: $a0

V: $a1

N: $a2



Summary of the Low-Level IR

- Array accesses translated to *ld* or *st* nodes
  - Address is the base address of the array + index * element size
- Local accesses translated to *ld* or *st* nodes
  - Address in *sp*, offset is local offset
- Access to parameters is translated to:
  - Instructions *lpd* – specify number of the parameter
- Nodes Enter and Exit of a function identify stack size used by the function

Summary

- Translation of high-level IR to low-level IR
  - Flat address space
  - Elimination of the structured control flow, substituition by conditional and unconditional branches
- Moving toward the target machine