Compilers
Intermediate Code Generation

LEIC

FEUP-FCUP

2022

This lecture

- Intermediate Code
- Three address code
- Compilation to intermediate code
  - Expressions
  - Commands
  - Functions
  - Boolean operators
  - *Arrays*
- Implementation

Compiler

Program

↓

$\boxed{\text{Scanner}}$

↓

sequência de *tokens*

↓

$\boxed{\text{Parser}}$

↓

AST

↓

$\boxed{\text{Semantic Analysis}}$

↓

AST & symbol table

$\boxed{\textbf{Code generation}}$

↓

Intermediate Code

$\boxed{\text{Instruction Selection}}$

↓

Symbolic *assembly*

↓

$\boxed{\text{Register allocation}}$

↓

Assembly

↓

$\boxed{\textit{Assembler \& linker}}$

↓

Executable code

This Lecture

Intermediate Code

Three address code

Compilation to intermediate code

- Expressions
- Commands
- Functions
- Boolean operators
- *Arrays*

Implementation

Intermediate Code

- Tradeoff between the source language and the machine code
- Advantage: simplifies compilation and modularity
- Options:

  high-level simplifies the compilation of the source language (the price is a harder compilation to assemby)

  low-level simplifies the generation of assembly code (the price is a harder compilation of the source code)

Intermediate Code (cont.)

- Usually the choice of the intermediate code depends on the high-level programming language:
  - *Java Virtual Machine* (JVM) for Java
  - *Low Level Virtual Machine* (LLVM) for C/C++
- But the same intermediate code may be used for different languages:
  - Scala e Clojure compile to JVM
  - Rust, Swift e Julia compile to LLVM
- One may use more that one intermediate code
  (e.g. Haskell GHC uses 3 intermediate languages to compile Haskell)

This Lecture

Three address code

Three address code:

- An arbitrary number of temporary registers
- Operations with 2 ou 3 operands
- Without specific processor instructions
- Initially used to compile imperative languages (e.g. C or Pascal)

Example

Assignments:

```
x = 3*(4+5)
```

Example

Assignments:

```
x = 3*(4+5)
```

Intermediate code:

```
t1 = 3
t2 = 4
t3 = 5
t4 = t2+t3
t5 = t1*t4
x = t5
```

```
x =    *
      / \
     3   +
        / \
       4   5
```

Example

Assignments:

```
x = 3*(4+5)
```

Intermediate code:

```
t1 = 3
t2 = 4
t3 = 5
t4 = t2+t3
t5 = t1*t4
x = t5
```

```
x =     *
       / \
      3   +
         / \
        4   5
```

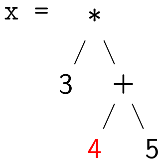
Example

Assignments:

```
x = 3*(4+5)
```

Intermediate code:

```
t1 = 3
t2 = 4
t3 = 5
t4 = t2+t3
t5 = t1*t4
x = t5
```



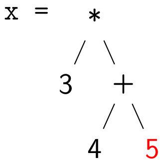
Example

Assignments:

```
x = 3*(4+5)
```

Intermediate code:

```
t1 = 3
t2 = 4
t3 = 5
t4 = t2+t3
t5 = t1*t4
x = t5
```



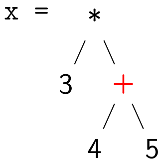
Example

Assignments:

```
x = 3*(4+5)
```

Intermediate code:

```
t1 = 3
t2 = 4
t3 = 5
t4 = t2+t3
t5 = t1*t4
x = t5
```



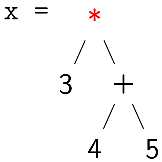
Example

Assignments:

```
x = 3*(4+5)
```

Intermediate code:

```
t1 = 3
t2 = 4
t3 = 5
t4 = t2+t3
t5 = t1*t4
x = t5
```



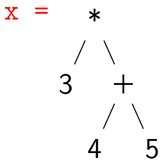
Example

Assignments:

```
x = 3*(4+5)
```

Intermediate code:

```
t1 = 3
t2 = 4
t3 = 5
t4 = t2+t3
t5 = t1*t4
x = t5
```



Example

Assignments:

```
x = 3*(4+5)
```

Intermediate code:

```
t1 = 3
t2 = 4
t3 = 5
t4 = t2+t3
t5 = t1*t4
x = t5
```

```
x =   *
     / \
    3   +
       / \
      4   5
```

- Each variable corresponds to a node in the AST
- Each operation uses a maximum of three variables (three addresses)
- Code for assignments is generated traversing the AST

Intermediate code syntax

$$
\begin{aligned}
\mathit{Instr} \rightarrow\ & \textbf{temp} := \mathit{Atom} \\
\mid\ & \textbf{temp} := \textbf{temp}\ \textbf{binop}\ \mathit{Atom} \\
\mid\ & \texttt{LABEL}\ \textbf{label} \\
\mid\ & \texttt{JUMP}\ \textbf{label} \\
\mid\ & \texttt{COND}\ \textbf{temp}\ \textbf{relop}\ \mathit{Atom}\ \ \textbf{label}\ \textbf{label} \\
\mathit{Atom} \rightarrow\ & \textbf{temp} \mid \textbf{num}
\end{aligned}
$$

- variables (**temp**), assignments and constants (**num**)
- binary operations **binop**: $+$, $*$, etc.
- comparisons **relop**: $<$, $>$, $==$, etc.
- labels, simple and conditional jumps

Example (Euclides algorithm)

```
while (b != 0) {
  r = a%b;
  a = b;
  b = r;
}
```

```
LABEL loop
COND b != 0 next end
LABEL next
r := a % b
a := b
b := r
JUMP loop
LABEL end
```

This Lecture

Intermediate Code

Three address code

Compilation to intermediate code

- Expressions
- Commands
- Functions
- Boolean operators
- *Arrays*

Implementation

Source language

- Variables, arithmetic and boolean expressions
- Assignments, *if/else*, *while*
- Compilation is defined by cases
  - one recursive compilation function for each syntactic category (expressions, commands, etc.)
  - return a list of intermediate code instructions

This lecture

Compilation of arithmetic expressions

$$Exp \rightarrow \textbf{num} \;\mid\; \textbf{id} \;\mid\; Exp\ \textbf{binop}\ Exp$$

- The symbol table associates variables in the source code with temporaries in the intermediate code
- To generate new temporary variables define functions:

$$newTemp : () \rightarrow Temp$$

$$newLabel : () \rightarrow Label$$

- These are not pure functions: they return fresh variables or labels each time they are called
- The compilation function

$$transExp : (Exp, Table, Temp) \rightarrow [Instr]$$

has an extra argument which is the register where is the result (*inherited attribute*)

Compilation of arithmetic expressions (cont.)

$transExp\ (expr,\ table,\ dest)$	$=$ case $expr$ of
num	return $[dest := \textbf{num}]$
id	$temp = \text{lookup}(\textbf{id}, table)$ return $[dest := temp]$
e_1 **binop** e_2	$t_1 = newTemp()$ $t_2 = newTemp()$ $code_1 = transExp(e_1, table, t_1)$ $code_2 = transExp(e_2, table, t_2)$ return $code_1 +\!\!+ code_2 +\!\!+ [dest := t_1\ \textbf{binop}\ t_2]$

Example

Assume the symbol table $[x \mapsto t_1, \; y \mapsto t_2]$ used to compile the expression

`x + (3*y)`

with the result stored in `t0`.

Example

Assume the symbol table $[x \mapsto t_1,\ y \mapsto t_2]$ used to compile the expression

```
x + (3*y)
```

with the result stored in `t0`.

```
t3 := t1
t5 := 3
t6 := t2
t4 := t5 * t6
t0 := t3 + t4
```

$$\overbrace{\overbrace{x}^{t_3} + \overbrace{\underbrace{3}_{t_5} * \underbrace{y}_{t_6}}^{t_4}}^{t_0}$$

(we use temporaries $t_3, t_4, \ldots$)

This lecture

Commands

- Assignments
- Conditional commands (with and without *else*)
- *while* cycles
- Boolean expressions
- Blocks

$$
\begin{aligned}
Stm &\rightarrow \textbf{id} = Exp; \\
&\mid \textbf{if}(Cond)\ Stm \\
&\mid \textbf{if}(Cond)\ Stm\ \textbf{else}\ Stm \\
&\mid \textbf{while}(Cond)\ Stm \\
&\mid \{\ StmList\ \}
\end{aligned}
$$

$$Cond \rightarrow Exp\ \textbf{relop}\ Exp$$

$$StmList \rightarrow Stm\ StmList \mid \epsilon$$

Commands compilation:

$$transStm : (Stm, Table) \rightarrow [Instr]$$

Commands (cont.)

- Compilation function *transStm* is defined by cases
- Assignements and blocks

$$
\begin{array}{l}
\mathit{transStm}\ (\mathit{stm},\ \mathit{table}) = \text{case } \mathit{stm} \text{ of} \\
\hline
\begin{array}{c|l}
\end{array}
\end{array}
$$

$\mathbf{id} = \mathit{expr};$	$\mathit{dest} = \mathrm{lookup}(\mathbf{id},\ \mathit{table})$ $\text{return } \mathit{transExp}(\mathit{expr}, \mathit{table}, \mathit{dest})$
$\{\ \mathit{stm}_1 \ldots \mathit{stm}_n\ \}$	$\mathit{code}_1 = \mathit{transStm}(\mathit{stm}_1, \mathit{table})$ $\vdots$ $\mathit{code}_n = \mathit{transStm}(\mathit{stm}_n, \mathit{table})$ $\text{return } \mathit{code}_1 ++ \cdots ++ \mathit{code}_n$

Relational expressions

- Compilation of relational expressions
- Extra arguments: labels $label_t$ and $label_f$ to jump when the condition is *true/false* (*inherited attributes*)
- Used in the compilation of *if/else* and *while*

$$transCond : (Cond, Table, Label, Label) \to [Instr]$$
$$transCond\ (cond, tabl, label_t, label_f) = \text{case } cond \text{ of}$$

$expr_1$ **relop** $expr_2$	$t_1 = newTemp()$ $t_2 = newTemp()$ $code_1 = transExp(expr_1, tabl, t_1)$ $code_2 = transExp(expr_2, tabl, t_2)$ return $code_1 +\!\!+ code_2 +\!\!+ [\mathtt{COND}\ t_1\ \mathbf{relop}\ t_2\ label_t\ label_f]$

Simple *if*

$transStm\ (stm,\ table) = \text{case } stm \text{ of}$	
$\mathbf{if}(cond)\ stm_1$	$label_1 = newLabel()$
	$label_2 = newLabel()$
	$code_1 = transCond(cond, label_1, label_2, table)$
	$code_2 = transStm(stm_1, table)$
	$\text{return } code_1 +\!\!+ [\texttt{LABEL } label_1] +\!\!+ code_2 +\!\!+ [\texttt{LABEL } label_2]$

if/else

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{transStm}\ (\textit{stm},\ \textit{tabl}) = \text{case } \textit{stm} \text{ of}} \\
\hline
\textbf{if}(\textit{Cond})\ \textit{stm}_1 & \textit{label}_1 = \textit{newLabel}() \\
\textbf{else}\ \textit{stm}_2 & \textit{label}_2 = \textit{newLabel}() \\
 & \textit{label}_3 = \textit{newLabel}() \\
 & \textit{code}_1 = \textit{transCond}(\textit{cond}, \textit{label}_1, \textit{label}_2, \textit{table}) \\
 & \textit{code}_2 = \textit{transStm}(\textit{stm}_1, \textit{table}) \\
 & \textit{code}_3 = \textit{transStm}(\textit{stm}_2, \textit{table}) \\
 & \text{return } \textit{code}_1 +\!\!+ [\texttt{LABEL}\ \textit{label}_1] +\!\!+ \textit{code}_2 +\!\!+ [\texttt{JUMP}\ \textit{label}_3] \\
 & \qquad\quad +\!\!+ [\texttt{LABEL}\ \textit{label}_2] +\!\!+ \textit{code}_3 +\!\!+ [\texttt{LABEL}\ \textit{label}_3] \\
\hline
\end{array}
$$

Example

Assume the following symbol table: $[x \mapsto t_0, y \mapsto t_1, z \mapsto t_2]$

```
if (x < y)
   z = y;
else
   z = x;
```

```
t3 := t0
t4 := t1
COND t3 < t4 label1 label2
LABEL label1
t2 := t1
JUMP label3
LABEL label2
t2 := t0
LABEL label3
```

while

$transStm\ (stm,\ table) = \text{case } stm \text{ of}$	
$\textbf{while}(cond)\ stm_1$	$label_1 = newLabel()$ $label_2 = newLabel()$ $label_3 = newLabel()$ $code_1 = transCond(cond,\ table,\ label_2, label_3)$ $code_2 = transStm(stm_1, table)$ $\text{return } [\texttt{LABEL}\ label_1] +\!\!+ code_1$ $\quad +\!\!+ [\texttt{LABEL}\ label_2] +\!\!+ code_2$ $\quad +\!\!+ [\texttt{JUMP}\ label_1, \texttt{LABEL}\ label_3]$

Example

Assume the following symbol table: $[n \mapsto t_0, r \mapsto t_1]$

```
{
  n = 5;
  r = 1;
  while (n>0) {
     r = r*n;
     n = n-1;
  }
}
```

```
t2 := 5
t0 := t2
t3 := 1
t1 := t3
LABEL label1
t4 := t0
t5 := 0
COND t4 > t5 label2 label3
LABEL label2
t6 := t1
t7 := t0
t1 := t6 * t7
t8 := t0
t9 := 1
t0 := t8 - t9
JUMP label1
LABEL label3
```

This lecture

Functions

- Function call and definitions are primitive operations in the intermediate code
- Arguments must be temporary variables
- The implementation of the functional parameter passage mechanism, using registers or the stack, is going to be explained later in the *assembly* code generation stage
- To simplify presentation we use only one type (**int**)

Syntax

Source code

$$
\begin{aligned}
\mathit{Program} &\to \mathit{Function} \ldots \mathit{Function} \\
\mathit{Function} &\to \textbf{int id}(\mathit{ArgList})\ \{\mathit{StmList}\} \\
\mathit{ArgList} &\to \textbf{int id}, \ldots, \textbf{int id} \\
\mathit{Stm} &\to \ \cdots \\
&\mid\ \textbf{return}\ \mathit{Exp}; \\
\mathit{Exp} &\to \ \cdots \\
&\mid\ \textbf{id}(\mathit{ExpList}) \\
\mathit{ExpList} &\to \mathit{Exp}, \ldots, \mathit{Exp}
\end{aligned}
$$

Intermediate code

$$
\begin{aligned}
\mathit{Program} &\to \mathit{Function} \ldots \mathit{Function} \\
\mathit{Function} &\to \textbf{id}(\mathit{TempList})\ [\ \mathit{InstrList}\ ] \\
\mathit{TempList} &\to \textbf{temp}, \ldots, \textbf{temp} \\
\mathit{InstrList} &\to \mathit{Instr}; \ldots; \mathit{Instr} \\
\mathit{Instr} &\to \ \cdots \\
&\mid\ \texttt{PARAM}\ \textbf{temp} \\
&\mid\ \textbf{temp} := \texttt{CALL}\ \textbf{id}(\mathit{TempList}) \\
&\mid\ \texttt{RETURN}\ \textbf{temp}
\end{aligned}
$$

Example

```
int max(int x, int y)
{
  if (x<y) return y;
  else return x;
}

int max3(int x, int y, int z)
{
  return max(x,max(y,z));
}
```

```
max(t0,t1) [
COND t0 < t1 ltrue lfalse
LABEL ltrue
RETURN t1
LABEL lfalse
RETURN t0
]

max3(t0, t1, t2) [
PARAM t1
PARAM t2
t3 := CALL max 2
PARAM t0
PARAM t3
t4 := CALL max 2
RETURN t4
]
```

Functions

- Variables t_0, t_1, etc. are locals to each function
- To compile **return** e use *transExp* to generate code for expression e
- To compile $f(e_1, \ldots, e_n)$ we need to compile each argument e_i
- We use an auxiliary function *transExps* which also returns the list of parameters

Functions (cont.)

$transExp\ (exp,\ table,\ dest)$	$=$ case exp of
$\mathbf{id}(exps)$	$(code, params) = transExps(exps, table)$ return $code +\!\!+ buildParams(params) +\!\!+ [dest := \texttt{CALL}\ \mathbf{id}]$

$transExps\ (exps,\ table)$	$=$ case $exps$ of
$e_1, \ldots, e_n$	$t_1 = newTemp()$ $\vdots$ $t_n = newTemp()$ $code_1 = transExp(e_1, table, t_1)$ $\vdots$ $code_n = transExp(e_n, table, t_n)$ return $(code_1 +\!\!+ \cdots +\!\!+ code_n, [t_1, \ldots, t_n])$

This lecture

Boolean operators

- Boolean operators: conjunction (`&&`), disjunction (`||`) and negation (`!`)
- *short-circuit evaluation*:
  - do not evaluate the second argument if the first determines the output
- Note that

  ```
  if (k!=0 && n%k==0) { ... }
  ```

  is equivalent to

  ```
  if (k!=0) { if (n%k==0) { ... } }
  ```

- We enable the use of boolean expressions as general expressions and the opposite, e.g.

  ```
  c = a==0 || b==0;
  if(c) { ... }
  ```

Boolean operators (cont.)

$$
\begin{aligned}
\mathit{Exp} \rightarrow\ & \mathbf{num} \\
\mid\ & \mathbf{id} \\
\mid\ & \mathit{Exp}\ \mathbf{binop}\ \mathit{Exp} \\
\mid\ & \mathbf{id}(\mathit{Exps}) \\
\mid\ & \texttt{true} \\
\mid\ & \texttt{false} \\
\mid\ & \mathit{Cond}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Cond} \rightarrow\ & \mathit{Exp}\ \mathbf{relop}\ \mathit{Exp} \\
\mid\ & \texttt{!}\,\mathit{Cond} \\
\mid\ & \mathit{Cond}\ \texttt{\&\&}\ \mathit{Cond} \\
\mid\ & \mathit{Cond}\ \texttt{||}\ \mathit{Cond} \\
\mid\ & \texttt{true} \\
\mid\ & \texttt{false} \\
\mid\ & \mathit{Exp}
\end{aligned}
$$

Boolean expressions

- False and True are represented by 0 e 1
- Compile `true` e `false` to a direct assignment

Boolean expressions (cont.)

$transExp\,(exp,\ table,\ dest)$	$=$ case $expr$ of
$\vdots$	
`true`	return $[dest := 1]$
`false`	return $[dest := 0]$
$cond$	$label_1 = newLabel()$ $label_2 = newLabel()$ $code_1 = transCond(cond,\ table,\ label_1,\ label_2)$ return $code_1 +\!\!+ [\texttt{LABEL}\ label_1, dest := 1, \texttt{JUMP}\ label_3]$ $+\!\!+\ [\texttt{LABEL}\ label_2, dest := 0, \texttt{LABEL}\ label_3]$

Boolean expressions

The compilation function

$$transCond\ (cond,\ table,\ \textcolor{blue}{label_t},\ \textcolor{red}{label_f})$$

generates code that jumps to $\textcolor{blue}{label_t}$ and $\textcolor{red}{label_f}$ if the expression is é true or false, respectively.

We now extend this compilation to the boolean operators:

Boolean expressions (cont.)

$transCond\ (cond, table, label_t, label_f) =$	case $cond$ of
e_1 **relop** e_2	$\vdots$ *(como anteriormente)*
`true`	return [JUMP $label_t$]
`false`	return [JUMP $label_f$]
`!`$cond_1$	$transCond(cond_1, table, label_f, label_t)$

- Constants `true` e `false` are compiled to jumps
- Compilation of negation switches the labels

Boolean expressions (cont.)

$transCond\ (cond,\ table,\ label_t,\ label_f) = \text{case } cond \text{ of}$	
$cond_1$ `&&` $cond_2$	$label_2 = newLabel()$ $code_1 = transCond(cond_1, table, label_2, label_f)$ $code_2 = transCond(cond_2, table, label_t, label_f)$ $\text{return } code_1 +\!\!+ [\texttt{LABEL}\ label_2] +\!\!+ code_2$
$cond_1$ `||` $cond_2$	$label_2 = newLabel()$ $code_1 = transCond(cond_1, table, label_t, label_2)$ $code_2 = transCond(cond_2, table, label_t, label_f)$ $\text{return } code_1 +\!\!+ [\texttt{LABEL}\ label_2] +\!\!+ code_2$

Compilation of conjunctions and disjunctions uses a fresh label

Boolean expressions (cont.)

$transCond\ (cond, table, label_t, label_f)$	$=$ case $cond$ of
exp	$t = newTemp()$ $code_1 = transExp(exp, table, t)$ return $code_1$ $++$[`COND` t`!=`0 $label_t$ $label_f$]

Other cases: compile the expression and compare the result with zero

Example

Suppose $[a \mapsto t_0, b \mapsto t_1]$.

```
if (a != 0 && b > a) {
    b = b - a;
}
```

```
t2 := t0
t3 := 0
COND t2 != t3 label2 label4
LABEL label2
t4 := t1
t5 := t0
COND t4 > t5 label3 label4
LABEL label3
t6 := t1
t7 := t0
t1 := t6 - t7
LABEL label4
```

Remarks

- There are not boolean operators in the intermediate code (they are implemented as conditional jumps
- Alternative: compile boolean operators similarly to arithmetic operators (but then evaluate both arguments)
- *Cond* and *Exp* overlap a lot (ambiguous grammar)
- Alternative: one unique non terminal in the grammar *Exp* and two mutially recursive compilation functions

$$
\begin{aligned}
\mathit{transExp} &: (\mathit{Exp}, \mathit{Table}, \mathit{Temp}) \to [\mathit{Instr}] \\
\mathit{transCond} &: (\mathit{Exp}, \mathit{Table}, \mathit{Label}, \mathit{Label}) \to [\mathit{Instr}]
\end{aligned}
$$

where we use *transExp* for expressions and *transCond* for the conditions in *if* and *while*

This lecture

Intermediate Code

Three address code

Compilation to intermediate code

- Expressions
- Commands
- Functions
- Boolean operators
- *Arrays*

Implementation

Arrays

- *arrays* (expressions)

$$Exp \rightarrow \cdots \quad | \quad \mathbf{id}[Exp]$$

- Assignment to an *array*

$$Stm \rightarrow \cdots \quad | \quad \mathbf{id}[Exp] := Exp$$

- New instructions in the interemdiate code to read and write in memory

$$\begin{aligned} Instr \rightarrow \cdots & \\ | \quad & \mathbf{temp} := \mathtt{M}[Atom] \\ | \quad & \mathtt{M}[Atom] := \mathbf{temp} \end{aligned}$$

($Atom$ is a variable or a constant)

Arrays (cont.)

- The symbol table assciates the name *array* to its base address
- Auxiliary function to calculate the index address (elements of 4 *bytes*)
- Output: code and a temporary variable holding the address

$\mathit{transIndex}\ (\mathit{exp},\ \mathit{table}) =$	case exp of
$\mathbf{id}[\mathit{exp}_1]$	$\mathit{base} = \mathrm{lookup}(\mathbf{id}, \mathit{table})$ $\mathit{addr} = \mathit{newTemp}()$ $\mathit{code}_1 = \mathit{transExp}(\mathit{exp}_1, \mathit{table}, \mathit{addr})$ return $(\mathit{code}_1 +\!\!+ [\mathit{addr} := \mathit{addr} * 4, \mathit{addr} := \mathit{addr} + \mathit{base}], \mathit{addr})$

Arrays (cont.)

$$
\begin{array}{c|l}
\hline
transExp\ (exp,\ table,\ dest) & = \text{case } exp \text{ of} \\
\hline
\mathbf{id}[e_1] & (code_1, addr) = transIndex(\mathbf{id}[e_1], table) \\
 & \text{return } code_1 \mathbin{++} [dest := \mathtt{M}[addr]] \\
\hline
\end{array}
$$

$$
\begin{array}{c|l}
\hline
transStm\ (stm,\ table) & = \text{case } stm \text{ of} \\
\hline
\mathbf{id}[e_1] := e_2 & (code_1, addr) = transIndex(\mathbf{id}[e_1], table) \\
 & t = newTemp() \\
 & code_2 = transExp(e_2, table, t) \\
 & \text{return } code_1 \mathbin{++} code_2 \mathbin{++} [\mathtt{M}[addr] := t] \\
\hline
\end{array}
$$

Arrays declarations

- We need to allocate memory for the array
- Memory allocation may be static (global) or dynamic (in the *stack* or in the *heap*)
- Stack allocation: we will study this later...
- Static allocation: declare space in a *data* segment

```
// C
int my_array[10];

# Assembly MIPS
.data
my_array:     .space 40 # 4 bytes * 10 elements
```

This Lecture

Representing intermediate code (in C)

```
#include <stdint.h>
typedef struct { Opcode opcode;
                 Addr arg1, arg2, arg3, ...; // arguments
               } Instr;
typedef enum { MOVE, MOVEI, ... } Opcode; // instructions
typedef intptr_t Addr;  // address (integer or pointer)
```

- A structure:
- Different code for each kind of instruction (`MOVE`, `MOVEI`, etc.)
- Most instructions uses 2 or 3 arguments (exception: `COND`)
- Fill with 0 or NULL the unused fields

Variables and labels

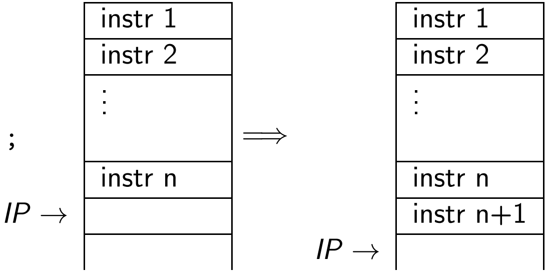
- One may represent variables and labels by integers
- Use a global variable (counter) to generate new variables and labels:

```
int temp_count = 0, label_count = 0;

int newTemp() {
    return temp_count ++;
}

int newLabel () {
    return label_count ++;
}
```

Code generation

- Instead of lists we may generate a global array of instructions
- Auxiliary functions to add a new instruction to the position identified by the *instruction pointer (IP)*
- *emit* code in the middle of recursive calls to avoid list concatenations

```
void emit2(opc,arg1,arg2);
void emit3(opc,arg1,arg2,arg3);
...
```



Code generation (cont.)

```
void transExp(Exp exp, dest) {
 switch(...) {
 case BINOP:
   t1 = newTemp();
   t2 = newTemp();
   transExp(exp->binop.left, t1);   // left expression
   transExp(exp->binop.right, t2);  // right expression
   emit3(opcode(exp->binop.op), dest, t1, t2);  // final instruction
   break;
 }
}
```