Structure of a Compiler

Program text
  ↓
  Lexical analysis
  ↓
Symbol sequence
  ↓
  Syntax analysis
  ↓
Syntax tree
  ↓
  Typecheck
  ↓
Syntax tree
  ↓
Intermediate code generation

Binary machine code
  ↑
  Assembly and linking
  ↑
Ditto with named registers
  ↑
  Register allocation
  ↑
Symbolic machine code
  ↑
  Machine code generation
  ↑
Intermediate code
1 Why Intermediate Code?
- Intermediate Language
- To-Be-Translated Language

2 Syntax-Directed Translation
- Arithmetic Expressions
- Statements
- Boolean Expressions, Sequential Evaluation

3 Translating More Complex Structures
- More Control Structures
- Arrays and Other Structured Data
- Role of Declarations in the Translation
Why Intermediate Language (IL)?

- Compilers for different platforms and languages can share parts.

- Without IL: how many translators do I need to write to map $n$ languages to $m$ different hardware?

![Diagram showing Fortran, Pascal, and C++ connected to Intermed.Lang, which is then connected to ARM, X86_64, and X86.]
Why Intermediate Language (IL)?

- Compilers for different platforms and languages can share parts.

- Without IL: how many translators do I need to write to map n languages to m different hardware?
  Answer: $n \times m$ instead of $n + m$!

- Machine-independent optimizations are possible.

- Also enables interpretation ...
Intermediate Language (IL)

- **Machine Independent**: unlimited number of registers and memory space, no machine-specific instructions.

- **Mid-level(s)** between source and machine languages *(tradeoff)*: simpler constructs, easier to generate machine code.

- **What features/constructs should IL support?**
  - every translation loses information \(\Rightarrow\) use the information before losing it!
  - typically a chain of ILs moving from higher towards lower level.

- **How complex should IL’s instruction be?**
  - complex: good for interpretation (amortizes instruction-decoding overhead),
  - simple: can more easily generate optimal machine code.
Intermediate Language (IL)

**Here:** Low-level language, but keeping functions (procedures).
Small instructions:

- 3-address code: one operation per expression
Intermediate Language (IL)

Here: Low-level language, but keeping functions (procedures).
Small instructions:
- 3-address code: one operation per expression
- Memory read/write ($M$) (address is atom).
Intermediate Language (IL)

Here: Low-level language, but keeping functions (procedures).

Small instructions:

- **3-address code**: one operation per expression
- **Memory read/write** ($M$) (address is atom).
- **Jump labels**, **GOTO** and **conditional jump** (**IF**).
Intermediate Language (IL)

Here: Low-level language, but keeping functions (procedures).
Small instructions:

- 3-address code: one operation per expression
- Memory read/write (M) (address is atom).
- Jump labels, GOTO and conditional jump (IF).
- Function calls and returns

\[
\begin{align*}
Prg & \rightarrow Fcts \\
Fcts & \rightarrow Fct Fcts \mid Fct \\
Fct & \rightarrow Hdr Bd \\
Hdr & \rightarrow \text{functionid}(\text{Args}) \\
Bd & \rightarrow [ \text{Instrs} ] \\
Instrs & \rightarrow \text{Instr}, \text{Instrs} \mid \text{Instr} \\
Instr & \rightarrow \text{id} := \text{Atom} \mid \text{id} := \text{unop} \ \text{Atom} \\
& \quad \mid \text{id} := \text{id} \ \text{binop} \ \text{Atom} \\
& \quad \mid \text{id} := \text{M[Atom]} \ \mid \text{M[Atom]} := \text{id} \\
& \quad \mid \text{LABEL} \ \text{label} \ \mid \text{GOTO} \ \text{label} \\
& \quad \mid \text{IF} \ \text{id} \ \text{relop} \ \text{Atom} \\
& \quad \quad \text{THEN} \ \text{label} \ \text{ELSE} \ \text{label} \\
\text{Atom} & \rightarrow \text{id} \mid \text{num}
\end{align*}
\]
Intermediate Language (IL)

Here: Low-level language, but keeping functions (procedures).

Small instructions:
- 3-address code: one operation per expression
- Memory read/write (M) (address is atom).
- Jump labels, GOTO and conditional jump (IF).
- Function calls and returns

\[
egin{align*}
Prgr & \rightarrow Fcts \\
Fcts & \rightarrow Fct \ Fcts \mid Fct \\
Fct & \rightarrow Hdr \ Bd \\
Hdr & \rightarrow \text{functionid}(\text{Args}) \\
Bd & \rightarrow [ \ \text{Instrs} \ ] \\
\text{Instrs} & \rightarrow \text{Instr} \ , \ \text{Instrs} \mid \text{Instr} \\
\text{Instr} & \rightarrow \text{id} \ := \ \text{Atom} \mid \text{id} \ := \ \text{unop} \ \text{Atom} \\
& \quad \mid \text{id} \ := \ \text{id} \ \text{binop} \ \text{Atom} \\
& \quad \mid \text{id} \ := \ M[\text{Atom}] \mid M[\text{Atom}] := \ \text{id} \\
& \quad \mid \text{LABEL} \ \text{label} \mid \text{GOTO} \ \text{label} \\
& \quad \mid \text{IF} \ \text{id} \ \text{relop} \ \text{Atom} \ \\
& \quad \quad \text{THEN} \ \text{label} \ \text{ELSE} \ \text{label} \\
& \quad \mid \text{id} := \ \text{CALL} \ \text{functionid}(\text{Args}) \\
& \quad \mid \text{RETURN} \ \text{id} \\
\text{Atom} & \rightarrow \ \text{id} \mid \text{num} \\
\text{Args} & \rightarrow \ \text{id} \ , \ \text{Args} \mid \text{id}
\end{align*}
\]
The To-Be-Translated Language

We shall translate a simple procedural language:

- Arithmetic expressions and function calls, boolean expressions,
- conditional branching (if),
- two loops constructs (while and repeat until).

Syntax-directed translation:

- In practice we work on the abstract syntax tree \textsc{AbSyn} (but here we use a generic grammar notation),
- Implement each syntactic category via a translation function: Arithmetic expressions, Boolean expressions, Statements.
- Code for subtrees is generated independent of context, (i.e., context is a parameter to the translation function and/or a value returned by the translation function)
Why Intermediate Code?
- Intermediate Language
- To-Be-Translated Language

Syntax-Directed Translation
- Arithmetic Expressions
- Statements
- Boolean Expressions, Sequential Evaluation

Translating More Complex Structures
- More Control Structures
- Arrays and Other Structured Data
- Role of Declarations in the Translation
Translating Arithmetic Expressions

Expressions in Source Language

- Variables and number literals,
- unary and binary operations,
- function calls (with argument list).

\[
\begin{align*}
\text{Exp} & \rightarrow \text{num} \mid \text{id} \\
& \mid \text{unop Exp} \\
& \mid \text{Exp binop Exp} \\
& \mid \text{id}(\text{Exps}) \\
\text{Exps} & \rightarrow \text{Exp} \mid \text{Exp}, \text{Exps}
\end{align*}
\]
Translating Arithmetic Expressions

Expressions in Source Language

- Variables and number literals,
- unary and binary operations,
- function calls (with argument list).

Translation function:

\[ Trans_{Exp} :: (Exp, VTable, FTable, Location) \rightarrow [ICode] \]

- Returns a list of intermediate code instructions [ICode] that . . .
- . . . upon execution, computes Exp’s result in variable Location.
- Case analysis on Exp’s abstract syntax tree AB SYN.
Symbol Tables and Helper Functions

Translation function:

\[ \text{Trans}_{\text{Exp}} :: (\text{Exp}, \text{VTable}, \text{FTable}, \text{Location}) \rightarrow \text{[ICode]} \]

Symbol Tables

- **vtable**: maps a variable name in source lang to its corresponding (translation) IL variable name.
- **ftable**: function names to function labels (for call)

Helper Functions

- **lookup**: retrieve entry from a symbol table
- **getvalue**: retrieve value of source language literal
- **getname**: retrieve name of source language variable/operation
- **newvar**: make new intermediate code variable
- **newlabel**: make new label (for jumps in intermediate code)
- **trans_op**: translates an operator name to the name in IL.
Generating Code for an Expression

\[\text{Trans}_{\text{Exp}} : (\text{Exp, VTable, FTable, Location}) \rightarrow [\text{ICode}]\]

\[\text{Trans}_{\text{Exp}}(\text{exp, vtable, ftable, place}) = \text{case exp of}\]

- **num**
  \[v = \text{getvalue(num)}\]
  \[[\text{place} := v]\]

- **id**
  \[x = \text{lookup(vtable, \text{getname(id)})}\]
  \[[\text{place} := x]\]

- **unop** \(\text{Exp}_1\)
  \[\text{place}_1 = \text{newvar}()\]
  \[\text{code}_1 = \text{Trans}_{\text{Exp}}(\text{Exp}_1, \text{vtable, ftable, place}_1)\]
  \[\text{op} = \text{trans_op}(	ext{getname(unop)})\]
  \[\text{code}_1 \quad ++ \quad [\text{place} := \text{op place}_1]\]

- **Exp\_1 binop Exp\_2**
  \[\text{place}_1 = \text{newvar}()\]
  \[\text{place}_2 = \text{newvar}()\]
  \[\text{code}_1 = \text{Trans}_{\text{Exp}}(\text{Exp}_1, \text{vtable, ftable, place}_1)\]
  \[\text{code}_2 = \text{Trans}_{\text{Exp}}(\text{Exp}_2, \text{vtable, ftable, place}_2)\]
  \[\text{op} = \text{trans_op}(	ext{getname(binop)})\]
  \[\text{code}_1 \quad ++ \quad \text{code}_2 \quad ++ \quad [\text{place} := \text{place}_1 \text{ op place}_2]\]
Generating Code for a Function Call

\[ \text{Trans}_{\text{Exp}} (\text{exp}, \text{vtable}, \text{ftable}, \text{place}) = \text{case } \text{exp} \text{ of} \]

\[ \text{id}(\text{Exps}) \quad (\text{code}_1, [a_1, \ldots, a_n]) = \text{Trans}_{\text{Exps}} (\text{Exps}, \text{vtable}, \text{ftable}) \]

\[ \text{fname} = \text{lookup} (\text{ftable}, \text{getname}(\text{id})) \]

\[ \text{code}_1 \; \text{++} \; [\text{place} := \text{CALL} \; \text{fname}(a_1, \ldots, a_n)] \]

\[ \text{Trans}_{\text{Exps}} \text{ returns the code that evaluates the function’s parameters,} \]
\[ \text{and the list of new-intermediate variables (that store the result).} \]

\[ \text{Trans}_{\text{Exps}} : (\text{Exps}, \text{VTable}, \text{FTable}) \rightarrow ([\text{ICode}], [\text{Location}]) \]

\[ \text{Trans}_{\text{Exps}} (\text{exp}, \text{vtable}, \text{ftable}) = \text{case } \text{exp} \text{ of} \]

\[ \text{Exp} \quad \text{place} = \text{newvar}() \]

\[ \text{code}_1 = \text{Trans}_{\text{Exp}} (\text{Exp}, \text{vtable}, \text{ftable}, \text{place}) \]

\[ (\text{code}_1, [\text{place}]) \]

\[ \text{Exp}, \text{Exps} \quad \text{place} = \text{newvar}() \]

\[ \text{code}_1 = \text{Trans}_{\text{Exp}} (\text{Exp}, \text{vtable}, \text{ftable}, \text{place}) \]

\[ (\text{code}_2, \text{args}) = \text{Trans}_{\text{Exps}} (\text{Exps}, \text{vtable}, \text{ftable}) \]

\[ \text{code}_3 = \text{code}_1 \; \text{++} \; \text{code}_2 \]

\[ \text{args}_1 = \text{place} :: \text{args} \]

\[ (\text{code}_3, \text{args}_1) \]
Translation Example

Assume the following symbol tables:

- **vtable** = \([x \mapsto v_0, \ y \mapsto v_1, \ z \mapsto v_2]\)
- **ftable** = \([f \mapsto _F\_1, + \mapsto +, - \mapsto -]\)

Translation of \(\text{Exp}\) with place = \(t_0\):

- \(\text{Exp} = x - 3\)
Translation Example

Assume the following symbol tables:

- **vtable** = \([x \mapsto v0, \ y \mapsto v1, \ z \mapsto v2]\)
- **ftable** = \([f \mapsto _F\_1, \ + \mapsto +, \ - \mapsto -]\)

Translation of Exp with place = \(t0\):

- **Exp=x-3**
  
  \(t1 := v0\)
  
  \(t2 := 3\)
  
  \(t0 := t1 - t2\)
Translation Example

Assume the following symbol tables:

- $v_{\text{table}} = [x \mapsto v_0, \; y \mapsto v_1, \; z \mapsto v_2]$
- $f_{\text{table}} = [f \mapsto F_1, \; + \mapsto +, \; - \mapsto -]$

Translation of $\text{Exp}$ with place $= t_0$:

- $\text{Exp} = x - 3$
  
  $t_1 := v_0$
  $t_2 := 3$
  $t_0 := t_1 - t_2$

- $\text{Exp} = 3 + f(x - y, z)$
Translation Example

Assume the following symbol tables:

- \( vtable = [x \mapsto v0, \ y \mapsto v1, \ z \mapsto v2] \)
- \( ftable = [f \mapsto _F_1, \ + \mapsto +, \ - \mapsto -] \)

Translation of \( Exp \) with place = \( t0 \):

- \( Exp=x-3 \)
  
  \[
  \begin{align*}
  t1 & := v0 \\
  t2 & := 3 \\
  t0 & := t1 - t2
  \end{align*}
  \]

- \( Exp=3+f(x-y,z) \)
  
  \[
  \begin{align*}
  t1 & := 3 \\
  t4 & := v0 \\
  t5 & := v1 \\
  t3 & := t4 - t5 \\
  t6 & := v2 \\
  t2 & := \text{CALL } _F_1(t3, t6) \\
  t0 & := t1 + t2
  \end{align*}
  \]
1 Why Intermediate Code?
   - Intermediate Language
   - To-Be-Translated Language

2 Syntax-Directed Translation
   - Arithmetic Expressions
   - Statements
     - Boolean Expressions, Sequential Evaluation

3 Translating More Complex Structures
   - More Control Structures
   - Arrays and Other Structured Data
   - Role of Declarations in the Translation
Translating Statements

Statements in Source Language

- Sequence of statements
  \[ Stat \rightarrow Stat ; Stat \]

- Assignment
  \[ id := Exp \]

- Conditional Branching
  \[ if Cond then \{ Stat \} \]
  \[ if Cond then \{ Stat \} else \{ Stat \} \]

- Loops: while and repeat
  \[ while Cond do \{ Stat \} \]
  \[ repeat \{ Stat \} until Cond \]

We assume relational operators translate directly (using \text{trans}_{op}).
Translating Statements

Statements in Source Language

- Sequence of statements
  \[ Stat \rightarrow Stat ; Stat \]
- Assignment
  \[ id := Exp \]
- Conditional Branching
  \[ \text{if } Cond \text{ then } \{ Stat \} \]
  \[ \text{if } Cond \text{ then } \{ Stat \} \text{ else } \{ Stat \} \]
- Loops: while and repeat
  \[ \text{while } Cond \text{ do } \{ Stat \} \]
  \[ \text{repeat } \{ Stat \} \text{ until } Cond \]

We assume relational operators translate directly (using trans_op).

Translation function:

\[ Trans_{Stat} :: (Stat, VTable, FTable) \rightarrow [ICode] \]

- As before: syntax-directed, case analysis on Stat
- Intermediate code instructions for statements
Generating Code for Sequences, Assignments, . . .

\[ \text{Trans}^{\text{Stat}} : (\text{Stat}, \text{Vtable}, \text{Ftable}) \rightarrow [\text{ICode}] \]

\[ \text{Trans}^{\text{Stat}}(\text{stat}, \text{vtable}, \text{ftable}) = \text{case stat of} \]

\[ \text{Stat}_1 ; \text{Stat}_2 \quad \text{code}_1 = \text{Trans}^{\text{Stat}}(\text{Stat}_1, \text{vtable}, \text{ftable}) \]
\[ \text{code}_2 = \text{Trans}^{\text{Stat}}(\text{Stat}_2, \text{vtable}, \text{ftable}) \]
\[ \text{code}_1 \; +\!+ \; \text{code}_2 \]

\[ \text{id} := \text{Exp} \quad \text{place} = \text{lookup}(\text{vtable}, \text{getname}(\text{id})) \]
\[ \text{Trans}^{\text{Exp}}(\text{Exp}, \text{vtable}, \text{ftable}, \text{place}) \]

. . . (rest coming soon)

- Sequence of statements, sequence of code.
- Symbol tables are inherited attributes.
Generating Code for Conditional Jumps: Helper

- Helper function for loops and branches
- Evaluates `Cond`, i.e., a boolean expression, then jumps to one of two labels, depending on result

\[
\text{\textit{Trans}}_{\text{Cond}} : (\text{Cond}, \text{Label}, \text{Label}, \text{Vtable}, \text{Ftable}) \rightarrow [\text{ICode}]
\]

\[
\text{\textit{Trans}}_{\text{Cond}}(\text{cond}, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) = \text{case cond of}
\]

\[
\begin{align*}
\text{Exp}_1 & \quad \text{relop} \quad \text{Exp}_2 \\
\text{t}_1 &= \text{newvar}() \\
\text{t}_2 &= \text{newvar}() \\
\text{code}_1 &= \text{Trans}_\text{Exp}(\text{Exp}_1, \text{vtable}, \text{ftable}, \text{t}_1) \\
\text{code}_2 &= \text{Trans}_\text{Exp}(\text{Exp}_2, \text{vtable}, \text{ftable}, \text{t}_2) \\
\text{op} &= \text{trans}\_\text{op}(\text{getname}(\text{relop})) \\
\text{code}_1 \quad \text{++} \quad \text{code}_2 \quad \text{++} \quad \text{[IF t}_1\text{ op t}_2\text{ THEN label}_t \text{ ELSE label}_f]\end{align*}
\]

- Uses the `IF` of the intermediate language
- Expressions need to be evaluated before
  (\textit{restricted IF: only variables and atoms can be used})
Generating Code for If-Statements

- Generate new labels for branches and following code
- Translate If statement to a conditional jump
Generating Code for If-Statements

- Generate **new labels** for branches and following code
- **Translate** `if` statement to a **conditional jump**

\[
\text{Trans}_{\text{Stat}}(\text{stat}, \text{vtable}, \text{ftable}) = \text{case stat of}
\]
\[
\text{if } \text{Cond} \quad \text{then } \text{Stat}_1 \quad \text{label}_t = \text{newlabel()}
\]
\[
\text{code}_c = \text{Trans}_{\text{Cond}}(\text{Cond}, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable})
\]
\[
\text{code}_s = \text{Trans}_{\text{Stat}}(\text{Stat}_1, \text{vtable}, \text{ftable})
\]
\[
\text{code}_c \text{ ++ [LABEL label}_t\text{]} \text{ ++ code}_s \text{ ++ [LABEL label}_f\text{]}
\]
\[
\text{if } \text{Cond} \quad \text{then } \text{Stat}_1 \quad \text{label}_f = \text{newlabel()}
\]
\[
\text{if } \text{Cond} \quad \text{then } \text{Stat}_1 \quad \text{label}_e = \text{newlabel()}
\]
\[
\text{else } \text{Stat}_2 \quad \text{label}_e = \text{newlabel()}
\]
\[
\text{code}_c = \text{Trans}_{\text{Cond}}(\text{Cond}, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable})
\]
\[
\text{code}_1 = \text{Trans}_{\text{Stat}}(\text{Stat}_1, \text{vtable}, \text{ftable})
\]
\[
\text{code}_2 = \text{Trans}_{\text{Stat}}(\text{Stat}_2, \text{vtable}, \text{ftable})
\]
\[
\text{code}_c \text{ ++ [LABEL label}_t\text{]} \text{ ++ code}_1 \text{ ++ [GOTO label}_e\text{]} \text{ ++ [LABEL label}_f\text{]} \text{ ++ code}_2 \text{ ++ [LABEL label}_e\text{]}
\]
Generating Code for Loops

- repeat-until loop is the easy case: Execute body, check condition, jump back if false.
- while loop needs check before body, one extra label needed.
Generating Code for Loops

- **repeat-until** loop is the easy case: Execute body, check condition, jump back if false.
- **while** loop needs check before body, one extra label needed.

\[
\text{Trans}_{\text{Stat}}(\text{stat}, \text{vtable}, \text{ftable}) = \text{case } \text{stat} \text{ of}
\]

<table>
<thead>
<tr>
<th>Case</th>
<th>Code 1</th>
<th>Code 2</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>repeat</strong></td>
<td>(\text{label}_f = \text{newlabel}())</td>
<td>(\text{code}<em>1 = \text{Trans}</em>{\text{Stat}}(\text{Stat}, \text{vtable}, \text{ftable}))</td>
<td>([\text{LABEL } \text{label}_f] ++)\</td>
</tr>
<tr>
<td><strong>until</strong></td>
<td>(\text{label}_t = \text{newlabel}())</td>
<td>(\text{code}<em>2 = \text{Trans}</em>{\text{Cond}}(\text{Cond}, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}))</td>
<td>(\text{code}_1 ++)\</td>
</tr>
<tr>
<td><strong>do</strong></td>
<td>(\text{label}_s = \text{newlabel}())</td>
<td>(\text{code}<em>1 = \text{Trans}</em>{\text{Cond}}(\text{Cond}, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}))</td>
<td>(\text{code}_2 ++)\</td>
</tr>
<tr>
<td></td>
<td>(\text{label}_t = \text{newlabel}())</td>
<td>(\text{code}<em>2 = \text{Trans}</em>{\text{Stat}}(\text{Stat}, \text{vtable}, \text{ftable}))</td>
<td>(\text{GOTO } \text{label}_s)</td>
</tr>
<tr>
<td></td>
<td>(\text{label}_f = \text{newlabel}())</td>
<td></td>
<td>(\text{GOTO } \text{label}_f)</td>
</tr>
</tbody>
</table>
Translation Example

- Symbol table vtable: \([x \mapsto v_0, \ y \mapsto v_1, \ z \mapsto v_2]\)
- Symbol table ftable: \([\text{getInt} \mapsto \text{libIO\_getInt}]\)

\[
\begin{align*}
x & := 3; \\
y & := \text{getInt}(); \\
z & := 1; \\
\text{while} \ y > 0 & \\
\quad & y := y - 1; \\
\quad & z := z \ast x
\end{align*}
\]
Translation Example

- Symbol table vtable: \([x \mapsto v_0, y \mapsto v_1, z \mapsto v_2]\]
- Symbol table ftable: \([\text{getInt} \mapsto \text{libIO_getInt}]\)

\[
\begin{align*}
x & := 3; \\
y & := \text{getInt}(); \\
z & := 1; \\
\text{while } y > 0 & \\
\quad & y := y - 1; \\
\quad & z := z \times x
\end{align*}
\]

\[
\begin{align*}
v_0 & := 3 \\
v_1 & := \text{CALL libIO_getInt()} \\
v_2 & := 1
\end{align*}
\]
Translation Example

- Symbol table vtable: \([x \mapsto v_0, y \mapsto v_1, z \mapsto v_2]\)
- Symbol table ftable: \([\text{getInt} \mapsto \text{libIO_getInt}]\)

\[
x := 3;
y := \text{getInt}();
z := 1;
\text{while } y > 0
\begin{align*}
  &y := y - 1; \\
  &z := z \times x
\end{align*}
\]

\[
v_0 := 3 \\
v_1 := \text{CALL libIO_getInt}() \\
v_2 := 1 \\
\text{LABEL } l_s \\
  t_1 := v_1 \\
  t_2 := 0 \\
  \text{IF } t_1 > t_2 \text{ THEN } l_t \text{ else } l_f \\
\text{LABEL } l_t
\]

GOTO l_s
LABEL l_f
Translation Example

- Symbol table vtable: \([x \mapsto v_0, y \mapsto v_1, z \mapsto v_2]\)
- Symbol table ftable: \([\text{getInt} \mapsto \text{libIO_getInt}]\)

\[
x := 3;
y := \text{getInt}();
z := 1;
\text{while } y > 0
\begin{align*}
&y := y - 1; \\
z := z \times x
\end{align*}
\]

\[
v_0 := 3 \\
v_1 := \text{CALL libIO_getInt}() \\
v_2 := 1 \\
\text{LABEL } l_s \\
t_1 := v_1 \\
t_2 := 0 \\
\text{IF } t_1 > t_2 \text{ THEN } l_t \text{ else } l_f \\
\text{LABEL } l_t \\
t_3 := v_1 \\
t_4 := 1 \\
v_1 := t_3 - t_4 \\
\text{GOTO } l_s \\
\text{LABEL } l_f
\]
Translation Example

- Symbol table **vtable**: \([x \mapsto v_0, \ y \mapsto v_1, \ z \mapsto v_2]\)
- Symbol table **ftable**: \([\text{getInt} \mapsto \text{libIO}\_\text{getInt}]\)

\[
\begin{align*}
x & := 3; \\
y & := \text{getInt}(); \\
z & := 1; \\
\text{while} \ y > 0 & \\
\quad & \text{y := y - 1;} \\
\quad & \text{z := z * x}
\end{align*}
\]

\[
\begin{align*}
v_0 & := 3 \\
v_1 & := \text{CALL} \ \text{libIO}\_\text{getInt}() \\
v_2 & := 1 \\
\text{LABEL} \ l_s & \\
\quad & \text{t}_1 := v_1 \\
\quad & \text{t}_2 := 0 \\
\quad & \text{IF} \ \text{t}_1 > \text{t}_2 \ \text{THEN} \ \text{l_t} \ \text{else} \ \text{l_f} \\
\text{LABEL} \ l_t & \\
\quad & \text{t}_3 := v_1 \\
\quad & \text{t}_4 := 1 \\
\quad & v_1 := \text{t}_3 - \text{t}_4 \\
\quad & \text{t}_5 := v_2 \\
\quad & \text{t}_6 := v_0 \\
\quad & v_2 := \text{t}_5 * \text{t}_6 \\
\quad & \text{GOTO} \ l_s \\
\text{LABEL} \ l_f
\end{align*}
\]
1 Why Intermediate Code?
   - Intermediate Language
   - To-Be-Translated Language

2 Syntax-Directed Translation
   - Arithmetic Expressions
   - Statements
   - Boolean Expressions, Sequential Evaluation

3 Translating More Complex Structures
   - More Control Structures
   - Arrays and Other Structured Data
   - Role of Declarations in the Translation
More Complex Conditions, Boolean Expressions

Boolean Expressions as Conditions

- Arithmetic expressions used as Boolean
- Logical operators (not, and, or)
- Boolean expressions used in arithmetics

Cond → Exp relop Exp
| Exp
| not Cond
| Cond and Cond
| Cond or Cond

Exp → … | Cond
More Complex Conditions, Boolean Expressions

Boolean Expressions as Conditions

- Arithmetic expressions used as Boolean
- Logical operators (not, and, or)
- Boolean expressions used in arithmetics

We extend the translation functions $Trans_{Exp}$ and $Trans_{Cond}$:

- Interpret numeric values as Boolean expressions: 0 is `false`, all other values `true`.
- Likewise: truth values as arithmetic expressions
Numbers and Boolean Values, Negation

Expressions as Boolean values, negation:

\[ Trans_{Cond}: (\text{Cond}, \text{Label}, \text{Label}, \text{Vtable}, \text{Ftable}) \rightarrow [\text{ICode}] \]
\[ Trans_{Cond}(\text{cond}, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) = \text{case } \text{cond} \text{ of} \]
\[ \ldots \]
\[ \text{Exp} \quad t = \text{newvar}() \]
\[ \quad \text{code} = Trans_{Exp}(\text{Exp}, \text{vtable}, \text{ftable}, t) \]
\[ \quad \text{code} \text{ ++ } [\text{IF } t \neq 0 \text{ THEN } \text{label}_t \text{ ELSE } \text{label}_f] \]
\[ \text{notCond} \quad Trans_{Cond}(\text{Cond}, \text{label}_f, \text{label}_t, \text{vtable}, \text{ftable}) \]
\[ \ldots \]
Numbers and Boolean Values, Negation

Expressions as Boolean values, negation:

\[ Trans_{Cond} : (\text{Cond}, \text{Label}, \text{Label}, \text{Vtable}, \text{Ftable}) \rightarrow [\text{ICode}] \]

\[ Trans_{Cond}(\text{cond}, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) = \text{case cond of} \]

\[ \ldots \]

\[ \text{Exp} \quad t = \text{newvar()} \]
\[ \quad \text{code} = Trans_{Exp}(\text{Exp}, \text{vtable}, \text{ftable}, t) \]
\[ \quad \text{code} \ ++ \ [\text{IF } t \neq 0 \ \text{THEN} \ \text{label}_t \ \text{ELSE} \ \text{label}_f] \]

\[ \text{not} Cond \quad Trans_{Cond}(\text{Cond}, \text{label}_f, \text{label}_t, \text{vtable}, \text{ftable}) \]

\[ \ldots \]

Conversion of Boolean values to numbers (by jumps):

\[ Trans_{Exp} : (\text{Exp}, \text{Vtable}, \text{Ftable}) \rightarrow [\text{ICode}] \]

\[ Trans_{Exp}(\text{exp}, \text{vtable}, \text{ftable}, \text{place}) = \text{case exp of} \]

\[ \ldots \]

\[ \text{Cond} \quad \text{label}_1 = \text{newlabel()} \]
\[ \quad \text{label}_2 = \text{newlabel()} \]
\[ \quad t = \text{newvar()} \]
\[ \quad \text{code} = Trans_{Cond}(\text{Cond}, \text{label}_1, \text{label}_2, \text{vtable}, \text{ftable}) \]
\[ \quad [t := 0] \ ++ \ \text{code} \ ++ \ [\text{LABEL } \text{label}_1, \ t := 1] \ ++ \ [\text{LABEL } \text{label}_2, \ \text{place} := t] \]
Sequential Evaluation of Conditions

Moscow ML version 2.01 (January 2004)
Enter ‘quit();’ to quit.
- fun f l = if (hd l = 1) then "one" else "not one";
> val f = fn : int list -> string
- f [];
  ! Uncaught exception:
  ! Empty
Sequential Evaluation of Conditions

Moscow ML version 2.01 (January 2004)
Enter ‘quit();’ to quit.
- fun f l = if (hd l = 1) then "one" else "not one";
> val f = fn : int list -> string
- f [];
  ! Uncaught exception:
  ! Empty

In most languages, logical operators are evaluated sequentially.

- If \( B_1 = false \), do not evaluate \( B_2 \) in \( B_1 && B_2 \) (anyway \( false \)).
- If \( B_1 = true \), do not evaluate \( B_2 \) in \( B_1 || B_2 \) (anyway \( true \)).
Sequential Evaluation of Conditions

Moscow ML version 2.01 (January 2004)
Enter ‘quit();’ to quit.
- fun f l = if (hd l = 1) then "one" else "not one";
> val f = fn : int list -> string
- f [];
  ! Uncaught exception:
  ! Empty

In most languages, logical operators are evaluated sequentially.

- If $B_1 = false$, do not evaluate $B_2$ in $B_1 \&\& B_2$ (anyway $false$).
- If $B_1 = true$, do not evaluate $B_2$ in $B_1 || B_2$ (anyway $true$).

- fun g l = if not (null l) andalso (hd l = 1) then "one" else "not one";
> val g = fn : int list -> string
- g [];
> val it = "not one" : string
# Sequential Evaluation by “Jumping Code”

\[\text{Trans}_\text{Cond} : (\text{Cond}, \text{Label}, \text{Label}, \text{Vtable}, \text{Ftable}) \rightarrow [\text{ICode}]\]

\[\text{Trans}_\text{Cond}(\text{cond}, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) = \text{case cond of} \]

\[\begin{align*}
\text{Cond}_1 & \quad \text{label}_{\text{next}} = \text{newlabel}() \\
\text{and} & \quad \text{code}_1 = \text{Trans}_\text{Cond}(\text{Cond}_1, \text{label}_{\text{next}}, \text{label}_f, \text{vtable}, \text{ftable}) \\
\text{Cond}_2 & \quad \text{code}_2 = \text{Trans}_\text{Cond}(\text{Cond}_2, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) \\
& \quad \text{code}_1 \; \text{++} \; [\text{LABEL} \; \text{label}_{\text{next}}] \; \text{++} \; \text{code}_2
\end{align*}\]

\[\begin{align*}
\text{Cond}_1 & \quad \text{label}_{\text{next}} = \text{newlabel}() \\
\text{or} & \quad \text{code}_1 = \text{Trans}_\text{Cond}(\text{Cond}_1, \text{label}_t, \text{label}_{\text{next}}, \text{vtable}, \text{ftable}) \\
\text{Cond}_2 & \quad \text{code}_2 = \text{Trans}_\text{Cond}(\text{Cond}_2, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) \\
& \quad \text{code}_1 \; \text{++} \; [\text{LABEL} \; \text{label}_{\text{next}}] \; \text{++} \; \text{code}_2
\end{align*}\]

Note: No logical operations in intermediate language!

Logics of and or encoded by jumps.

Alternative: Logical operators in intermediate language

\[\text{Cond} \Rightarrow \text{Exp} \Rightarrow \text{Exp} \; \text{binop} \; \text{Exp}\]

Translated as an arithmetic operation. Evaluates both sides!
Sequential Evaluation by “Jumping Code”

\[ Trans_{Cond} : (\text{Cond}, \text{Label}, \text{Label}, \text{Vtable}, \text{Ftable}) \rightarrow [\text{ICode}] \]

\[ Trans_{Cond}(\text{cond}, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) = \text{case } \text{cond} \text{ of} \]

\[ \begin{align*}
\text{Cond}_1 & \quad \text{label}_{next} = \text{newlabel}() \\
\text{and} & \quad \text{code}_1 = Trans_{Cond}(\text{Cond}_1, \text{label}_{next}, \text{label}_f, \text{vtable}, \text{ftable}) \\
\text{and} & \quad \text{code}_2 = Trans_{Cond}(\text{Cond}_2, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) \\
\text{and} & \quad \text{code}_1 \quad ++ \quad [\text{LABEL } \text{label}_{next}] \quad ++ \quad \text{code}_2
\end{align*} \]

\[ \begin{align*}
\text{Cond}_1 & \quad \text{label}_{next} = \text{newlabel}() \\
\text{or} & \quad \text{code}_1 = Trans_{Cond}(\text{Cond}_1, \text{label}_t, \text{label}_{next}, \text{vtable}, \text{ftable}) \\
\text{or} & \quad \text{code}_2 = Trans_{Cond}(\text{Cond}_2, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) \\
\text{or} & \quad \text{code}_1 \quad ++ \quad [\text{LABEL } \text{label}_{next}] \quad ++ \quad \text{code}_2
\end{align*} \]

- Note: No logical operations in intermediate language!
  Logics of \textbf{and} and \textbf{or} encoded by jumps.
Sequential Evaluation by “Jumping Code”

\[
Trans_{Cond} : (\text{Cond, Label, Label, Vtable, Ftable}) \rightarrow [\text{ICode}]
\]

\[
Trans_{Cond}(\text{cond}, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) = \text{case cond of}
\]

\[
\begin{array}{ll}
\text{Cond}_1 & \text{label}_{\text{next}} = \text{newlabel}() \\
\text{and} & \text{code}_1 = Trans_{Cond}(\text{Cond}_1, \text{label}_{\text{next}}, \text{label}_f, \text{vtable}, \text{ftable}) \\
\text{Cond}_2 & \text{code}_2 = Trans_{Cond}(\text{Cond}_2, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) \\
\end{array}
\]

\[
\begin{array}{ll}
\text{Cond}_1 & \text{label}_{\text{next}} = \text{newlabel}() \\
\text{or} & \text{code}_1 = Trans_{Cond}(\text{Cond}_1, \text{label}_t, \text{label}_{\text{next}}, \text{vtable}, \text{ftable}) \\
\text{Cond}_2 & \text{code}_2 = Trans_{Cond}(\text{Cond}_2, \text{label}_t, \text{label}_f, \text{vtable}, \text{ftable}) \\
\end{array}
\]

\[
\text{code}_1 \; \text{++} \; [\text{LABEL} \; \text{label}_{\text{next}}] \; \text{++} \; \text{code}_2
\]

- **Note:** No logical operations in intermediate language!
  Logics of \textbf{and} and \textbf{or} encoded by jumps.

- **Alternative:** Logical operators in intermediate language
  \[\text{Cond} \Rightarrow \text{Exp} \Rightarrow \text{Exp} \; \text{binop} \; \text{Exp}\]
  Translated as an arithmetic operation.
Sequential Evaluation by “Jumping Code”

TransCond : (Cond, Label, Label, Vtable, Ftable) \rightarrow [ICode]

TransCond(cond, label_t, label_f, vtable, ftable) = case cond of

...  

Cond_1  

label_{next} = newlabel()  

code_1 = TransCond(Cond_1, label_{next}, label_f, vtable, ftable)  

code_1 ++ [LABEL label_{next}] ++ code_2  

and  

Cond_2  

code_2 = TransCond(Cond_2, label_t, label_f, vtable, ftable)  

code_2 = TransCond(Cond_2, label_t, label_f, vtable, ftable)  

code_2 = TransCond(Cond_2, label_t, label_f, vtable, ftable)  

code_1 ++ [LABEL label_{next}] ++ code_2  

Cond_1  

label_{next} = newlabel()  

code_1 = TransCond(Cond_1, label_t, label_{next}, vtable, ftable)  

code_1 = TransCond(Cond_1, label_t, label_{next}, vtable, ftable)  

code_1 = TransCond(Cond_1, label_t, label_{next}, vtable, ftable)  

code_1 ++ [LABEL label_{next}] ++ code_2  

or  

Cond_2  

...  

Note: No logical operations in intermediate language!
Logics of and and or encoded by jumps.

Alternative: Logical operators in intermediate language

Cond \Rightarrow Exp \Rightarrow Exp \text{ binop } Exp

Translated as an arithmetic operation. Evaluates both sides!
1 Why Intermediate Code?
- Intermediate Language
- To-Be-Translated Language

2 Syntax-Directed Translation
- Arithmetic Expressions
- Statements
- Boolean Expressions, Sequential Evaluation

3 Translating More Complex Structures
- More Control Structures
- Arrays and Other Structured Data
- Role of Declarations in the Translation
More Control Structures

- Control structures determine control flow: which instruction to execute next
- A **while**-loop is enough
Control structures determine control flow: which instruction to execute next

A **while**-loop is enough ... but ... languages usually offer more.

**Explicit jumps:** 

\[
\text{Stat} \rightarrow \text{label} : \\
| \text{goto label}
\]

Necessary instructions are in the intermediate language. Needs to build symbol table of labels.
More Control Structures

- Control structures determine control flow: which instruction to execute next
- A **while**-loop is enough ... but ... languages usually offer more.
- **Explicit jumps:** \[ Stat \rightarrow \text{label} : \]
  \[ \mid \text{goto label} \]
  Necessary instructions are in the intermediate language. Needs to build symbol table of labels.
- **Case/Switch:** \[ Stat \rightarrow \text{case Exp of [ Alts ]} \]
  \[ \text{Alts} \rightarrow \text{num : Stat} \mid \text{num : Stat, Alts} \]
  When exited after each case: chain of if-then-else
  When “falling through” (e.g., in C): if-then-else and goto.
More Control Structures

- Control structures determine control flow: which instruction to execute next.
- A **while**-loop is enough . . . but . . . languages usually offer more.
- **Explicit jumps:** $Stat \rightarrow label :$
  
  $| goto \ label$

  Necessary instructions are in the intermediate language.
  Needs to build symbol table of labels.

- **Case/Switch:** $Stat \rightarrow case \ Exp \ of \ [ \ Alts \ ]$
  
  $Alts \rightarrow num : Stat \ | \ num : Stat, \ Alts$

  When exited after each case: chain of if-then-else
  When “falling through” (e.g., in C): if-then-else and goto.

- **Break and Continue:** $Stat \rightarrow break \ | \ continue$
  
  (break: jump behind loop, continue: jump to end of loop body).
  Needs two jump target labels used only inside loop bodies
  (parameters to translation function $Trans_{Stat}$)
More Control Structures

- Control structures determine control flow: which instruction to execute next
- A while-loop is enough ... but ... languages usually offer more.
- Explicit jumps: \( \text{Stat} \to \text{label:} \quad \text{goto label} \)

Necessary instructions are in the intermediate language. Needs to build symbol table of labels.

- Case/Switch: \( \text{Stat} \to \text{case Exp of} \ [ \text{Alts} ] \)
  \( \text{Alts} \to \text{num: Stat} | \text{num: Stat, Alts} \)

When exited after each case: chain of if-then-else
When “falling through” (e.g., in C): if-then-else and goto.

- Break and Continue: \( \text{Stat} \to \text{break} | \text{continue} \)
  (break: jump behind loop, continue: jump to end of loop body).

Needs two jump target labels used only inside loop bodies (parameters to translation function \( \text{TransStat} \))
1 Why Intermediate Code?
- Intermediate Language
- To-Be-Translated Language

2 Syntax-Directed Translation
- Arithmetic Expressions
- Statements
- Boolean Expressions, Sequential Evaluation

3 Translating More Complex Structures
- More Control Structures
- Arrays and Other Structured Data
- Role of Declarations in the Translation
Translating Arrays (of \texttt{int} elements)

Extending the Source Language

- Array elements used as an expression
- Assignment to an array element
- Array elements accessed by an index (expression)

\[
\begin{align*}
\text{Exp} & \rightarrow \ldots | \text{Idx} \\
\text{Stat} & \rightarrow \ldots | \text{Idx} := \text{Exp} \\
\text{Idx} & \rightarrow \text{id} [ \text{Exp} ]
\end{align*}
\]
Translating Arrays (of \texttt{int} elements)

Extending the Source Language

- Array elements used as an expression
- Assignment to an array element
- Array elements accessed by an index (expression)

\[\begin{align*}
\text{Exp} & \rightarrow \ldots | \text{Idx} \\
\text{Stat} & \rightarrow \ldots | \text{Idx} := \text{Exp} \\
\text{Idx} & \rightarrow \text{id}[\text{Exp}]\end{align*}\]

Again we extend \textit{Trans}_\text{Exp} and \textit{Trans}_\text{Stat}.

- Arrays stored in pre-allocated memory area, generated code will use memory access instructions.
- Static (compile-time) or dynamic (run-time) allocation possible.
Generating Code for Address Calculation

- `vtable` contains the base address of the array.
- Elements are `int` here, so 4 bytes per element for address.

\[
Trans_{Idx}(index, vtable, ftable) = \text{case } index \text{ of}
\]
\[
\text{id}[Exp] \quad \begin{align*}
\text{base} &= \text{lookup}(vtable, \text{getname(id)}) \\
\text{addr} &= \text{newvar}() \\
\text{code}_1 &= Trans_{Exp}(Exp, vtable, ftable, \text{addr}) \\
\text{code}_2 &= \text{code}_1 + \left[ \text{addr} := \text{addr} \times 4, \text{addr} := \text{addr} + \text{base} \right] \\
(\text{code}_2, \text{addr})
\end{align*}
\]

Returns:
- Code to calculate the absolute address . . .
- of the array element in memory (corresponding to `index`), . . .
- . . . and a new variable (`addr`) where it will be stored.
Generating Code for Array Access

Address-calculation code: in expression and statement translation.

- **Read access** inside expressions:

  \[
  Trans_{Exp}(\text{exp}, \text{vtable}, \text{ftable}, \text{place}) = \text{case exp of}
  \]
  
  \[
  \ldots
  \]
  
  \[
  \text{Idx} \ (\text{code}_1, \text{address}) = Trans_{Idx}(\text{Idx}, \text{vtable}, \text{ftable})
  \]
  
  \[
  \text{code}_1 \ ++ \ [\text{place} := M[\text{address}]]
  \]

- **Write access** in assignments:

  \[
  Trans_{Stat}(\text{stat}, \text{vtable}, \text{ftable}) = \text{case stat of}
  \]
  
  \[
  \ldots
  \]
  
  \[
  \text{Idx} := \text{Exp} \ (\text{code}_1, \text{address}) = Trans_{Idx}(\text{Index}, \text{vtable}, \text{ftable})
  \]
  \[
  t = \text{newvar}()
  \]
  \[
  \text{code}_2 = Trans_{Exp}(\text{Exp}, \text{vtable}, \text{ftable}, t)
  \]
  \[
  \text{code}_1 \ ++ \ \text{code}_2 \ ++ \ [M[\text{address}] := t] \]
Multi-Dimensional Arrays

Arrays in Multiple Dimensions

- Only a small change to previous grammar: \( Idx \) can now be recursive.
- Needs to be mapped to an address in one dimension.

\[
\begin{align*}
\text{Exp} & \rightarrow \ldots | Idx \\
\text{Stat} & \rightarrow \ldots | Idx ::= Exp \\
Idx & \rightarrow \text{id}[Exp] | Idx[Exp]
\end{align*}
\]

Arrays stored in row-major or column-major order.
Standard: row-major, index of \( a[k][l] \) is \( k \cdot \text{dim}_1 + l \).
Address calculation need to know sizes in each dimension.
Symbol table: base address and list of array-dimension sizes.
Need to change \( \text{Trans} \), i.e., add recursive index calculation.
Multi-Dimensional Arrays

Arrays in Multiple Dimensions

- Only a small change to previous grammar: \( Idx \) can now be recursive.
- Needs to be mapped to an address in one dimension.

- Arrays stored in row-major or column-major order.

  Standard: row-major, index of \( a[k][l] \) is \( k \cdot dim_1 + l \)
  
  (Index of \( b[k][l][m] \) is \( k \cdot dim_1 \cdot dim_2 + l \cdot dim_2 + m \))
Multi-Dimensional Arrays

Arrays in Multiple Dimensions

- Only a small change to previous grammar: \( idx \) can now be recursive.
- Needs to be mapped to an address in one dimension.

\[
\begin{align*}
\text{Exp} & \rightarrow \ldots \mid idx \\
\text{Stat} & \rightarrow \ldots \mid idx := \text{Exp} \\
idx & \rightarrow \text{id}[\text{Exp}] \mid idx[\text{Exp}]
\end{align*}
\]

- Arrays stored in row-major or column-major order.
  - Standard: row-major, index of \( a[k][l] \) is \( k \cdot \text{dim}_1 + l \)
  - (Index of \( b[k][l][m] \) is \( k \cdot \text{dim}_1 \cdot \text{dim}_2 + l \cdot \text{dim}_2 + m \))

- Address calculation need to know sizes in each dimension.
  - Symbol table: base address and list of array-dimension sizes.

- Need to change \( \text{Trans}_{idx} \), i.e., add recursive index calculation.
Address Calculation in Multiple Dimensions

\[
\text{Trans}_{ldx}(\text{index}, \text{vtable}, \text{ftable}) = \\
\hspace{1cm} (\text{code}_1, t, \text{base}, []) = \text{Calc}_{ldx}(\text{index}, \text{vtable}, \text{ftable}) \\
\hspace{1cm} \text{code}_2 = \text{code}_1 \text{ ++ } [t := t \ast 4, t := t + \text{base}] \\
\hspace{1cm} (\text{code}_2, t)
\]
Address Calculation in Multiple Dimensions

\( Trans_{Idx}(index, vtable, ftable) = \)
\[
(code_1, t, base, []) = Calc_{Idx}(index, vtable, ftable) \\
\]
\( code_2 = code_1 + \)  \[ t := t \times 4, t := t + base \] 
\( (code_2, t) \)

Recursive index calculation, multiplies with dimension at each step.

\( Calc_{Idx}(index, vtable, ftable) = \text{case } index \text{ of} \)

\( \text{id}[Exp] \quad (\text{base, dims}) = \text{lookup(vtable, getname(id))} \)
\( addr = \text{newvar()} \)
\( code = Trans_{Exp}(Exp, vtable, ftable, addr) \)
\( (code, addr, base, tail(dims)) \)

\( \text{Index}[Exp] \quad (code_1, addr, base, dims) = Calc_{Idx}(Index, vtable, ftable) \)
\( d = \text{head(dims)} \)
\( t = \text{newvar()} \)
\( code_2 = Trans_{Exp}(Exp, vtable, ftable, t) \)
\( code_3 = code_1 + code_2 + [addr := addr \times d, addr := addr + t] \)
\( (code_3, addr, base, tail(dims)) \)
1 Why Intermediate Code?
   - Intermediate Language
   - To-Be-Translated Language

2 Syntax-Directed Translation
   - Arithmetic Expressions
   - Statements
   - Boolean Expressions, Sequential Evaluation

3 Translating More Complex Structures
   - More Control Structures
   - Arrays and Other Structured Data
   - Role of Declarations in the Translation
Declarations in the Translation

Declarations are necessary

- to allocate space for arrays,
- to compute addresses for multi-dimensional arrays,
- ... and when the language allows local declarations (scope).
Declarations in the Translation

Declarations are necessary
- to allocate space for arrays,
- to compute addresses for multi-dimensional arrays,
- ...and when the language allows local declarations (scope).

Declarations and scope
- Statements following a declarations can see declared data.
- Declaration of variables and arrays
- Here: Constant size, one dimension

Function $Trans_{Decl}$ : (Decl, VTable) $\rightarrow$ ([ICode], VTable)
- translates declarations to code and new symbol table.
Translating Declarations to Scope and Allocation

Code with local scope (extended symbol table):

\[
Trans_{\text{Stat}}(\text{stat}, \text{vtable}, \text{ftable}) = \text{case stat of}
\]

\[
\begin{align*}
\text{Decl ; Stat}_1 & \quad (\text{code}_1, \text{vtable}_1) = Trans_{\text{Decl}}(\text{Decl}, \text{vtable}) \\
\text{code}_2 & = Trans_{\text{Stat}}(\text{Stat}_1, \text{vtable}_1, \text{ftable}) \\
\text{code}_1 & \text{ ++ code}_2
\end{align*}
\]

where HP is the heap pointer, indicating the first free space in a managed heap at runtime; used for dynamic allocation.
Translating Declarations to Scope and Allocation

Code with local scope (extended symbol table):

\[
Trans_{Stat}(stat, vtable, ftable) = \text{case } stat \text{ of } \\
\begin{align*}
Decl &; Stat_1 \quad (code_1, vtable_1) = Trans_{Decl}(Decl, vtable) \\
    & \quad code_2 = Trans_{Stat}(Stat_1, vtable_1, ftable) \\
    & \quad code_1 \; \text{++} \; code_2
\end{align*}
\]

Building the symbol table and allocating:

\[
Trans_{Decl} : (Decl, VTable) -> ([ICode], VTable )
\]

\[
Trans_{Decl}(decl, vtable) = \text{case } decl \text{ of } \\
\begin{align*}
\text{int } id & \quad t_1 = \text{newvar()} \\
    & \quad vtable_1 = \text{bind}(vtable, \text{getName(id)}, t_1) \\
    & \quad ([], vtable_1) \\
\text{int } id[num] & \quad t_1 = \text{newvar()} \\
    & \quad vtable_1 = \text{bind}(vtable, \text{getName(id)}, t_1) \\
    & \quad ([t_1 := HP, HP := HP + (4 \times \text{getValue(num)})], vtable_1)
\end{align*}
\]

... where HP is the heap pointer, indicating the first free space in a managed heap at runtime; used for dynamic allocation.
Other Structures that Require Special Treatment

- **Floating-Point values:**
  - Often stored in different registers
  - Always require different machine operations
  - Symbol table needs type information when creating variables in intermediate code.
Other Structures that Require Special Treatment

- **Floating-Point values:**
  Often stored in different registers
  Always require different machine operations
  Symbol table needs type information when creating variables in intermediate code.

- **Strings**
  Sometimes just arrays of (1-byte) *char* type, but *variable length*.
  In modern languages/implementations, elements can be *char* or *unicode* (UTF-8 and UTF-16 *variable size!*)
  Usually handled by library functions.
Other Structures that Require Special Treatment

- Floating-Point values:
  Often stored in different registers
  Always require different machine operations
  Symbol table needs type information when creating variables in intermediate code.

- Strings
  Sometimes just arrays of (1-byte) *char* type, but *variable length*.
  In modern languages/implementations, elements can be *char* or *unicode* (UTF-8 and UTF-16 *variable size!*)
  Usually handled by library functions.

- Records and Unions
  Linear in memory. Field *types and sizes* can be different.
  Field selector known at compile time: compute offset from base.
Structure of a Compiler

Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree

↓

Typecheck

↓

Syntax tree

↓

Intermediate code generation

Binary machine code

↑

Assembly and linking

↑

Ditto with named registers

↑

Register allocation

↑

Symbolic machine code

↑

Machine code generation

↑

Intermediate code