Intermediate Code Generation

Slides by Cosmin E. Oancea
cosmin.oancea@diku.dk

(based on Jost Berthold's slides, and with further tweaks by Andrzej Filinski, andrzej@di.ku.dk)

Department of Computer Science (DIKU)
University of Copenhagen

May 2023 IPS Lecture Slides

Structure of a Compiler

Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree

↓

Typecheck

↓

Syntax tree

↓

Intermediate code generation ⟶ Intermediate code

↑

Machine code generation

↑

Symbolic machine code

↑

Register allocation

↑

Ditto with named registers

↑

Assembly and linking

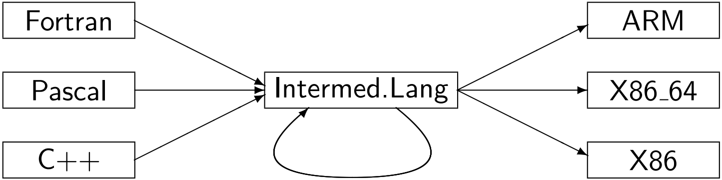
↑

Binary machine 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)?

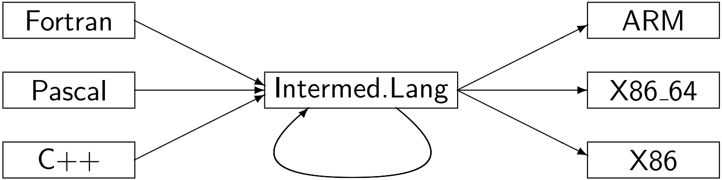
- Compilers for different platforms and languages can share parts.



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

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 architectures?
  Answer: $n \times m$ instead of $n + m$!
  - **Note:** IL and front-end(s) or back-end(s) may already exist!
  - E.g., LLVM, GCC [modern sense], JVM, .NET CLI, ...
- 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!
  - often a whole 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

$$
\begin{array}{lcl}
\mathit{Instrs} & \rightarrow & \mathit{Instr}\ ,\ \mathit{Instrs} \mid \mathit{Instr} \\
\mathit{Instr} & \rightarrow & \textbf{id} := \mathit{Atom} \mid \textbf{id} := \textbf{unop}\ \mathit{Atom} \\
 & & \mid \textbf{id} := \textbf{id}\ \textbf{binop}\ \mathit{Atom} \\
\mathit{Atom} & \rightarrow & \textbf{id} \mid \textbf{num}
\end{array}
$$

Give examples of (in)valid programs/instructions.

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 already calc'd).

$$
\begin{array}{lcl}
\mathit{Instrs} & \rightarrow & \mathit{Instr}\ ,\ \mathit{Instrs} \mid \mathit{Instr} \\
\mathit{Instr} & \rightarrow & \textbf{id} := \mathit{Atom} \mid \textbf{id} := \textbf{unop}\ \mathit{Atom} \\
 & & \mid \textbf{id} := \textbf{id}\ \textbf{binop}\ \mathit{Atom} \\
 & & \mid \textbf{id} := M[\mathit{Atom}] \mid M[\mathit{Atom}] := \textbf{id} \\
\mathit{Atom} & \rightarrow & \textbf{id} \mid \textbf{num}
\end{array}
$$

Give examples of (in)valid programs/instructions.

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 already calc'd).
- Jump labels, GOTO and conditional jump (IF).

$$
\begin{array}{lcl}
\mathit{Instrs} & \to & \mathit{Instr}\ ,\ \mathit{Instrs} \mid \mathit{Instr} \\
\mathit{Instr} & \to & \textbf{id} := \mathit{Atom} \mid \textbf{id} := \textbf{unop}\ \mathit{Atom} \\
 & & \mid \textbf{id} := \textbf{id}\ \textbf{binop}\ \mathit{Atom} \\
 & & \mid \textbf{id} := M[\mathit{Atom}] \mid M[\mathit{Atom}] := \textbf{id} \\
 & & \mid \texttt{LABEL}\ \textbf{label} \mid \texttt{GOTO}\ \textbf{label} \\
 & & \mid \texttt{IF}\ \textbf{id}\ \textbf{relop}\ \mathit{Atom} \\
 & & \qquad \texttt{THEN}\ \textbf{label}\ \texttt{ELSE}\ \textbf{label} \\
\mathit{Atom} & \to & \textbf{id} \mid \textbf{num}
\end{array}
$$

Give examples of (in)valid programs/instructions.

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 already calc'd).
- Jump labels, `GOTO` and conditional jump (`IF`).
- Function calls and returns

$$
\begin{array}{lcl}
\mathit{Prg} & \rightarrow & \mathit{Fcts} \\
\mathit{Fcts} & \rightarrow & \mathit{Fct}\ \mathit{Fcts} \mid \mathit{Fct} \\
\mathit{Fct} & \rightarrow & \mathit{Hdr}\ \mathit{Bd} \\
\mathit{Hdr} & \rightarrow & \textbf{functionid}(\mathit{Args}) \\
\mathit{Bd} & \rightarrow & [\ \mathit{Instrs}\ ] \\
\mathit{Instrs} & \rightarrow & \mathit{Instr}\ ,\ \mathit{Instrs} \mid \mathit{Instr} \\
\mathit{Instr} & \rightarrow & \textbf{id} := \mathit{Atom} \mid \textbf{id} := \textbf{unop}\ \mathit{Atom} \\
 & & \mid \textbf{id} := \textbf{id}\ \textbf{binop}\ \mathit{Atom} \\
 & & \mid \textbf{id} := M[\mathit{Atom}] \mid M[\mathit{Atom}] := \textbf{id} \\
 & & \mid \texttt{LABEL}\ \textbf{label} \mid \texttt{GOTO}\ \textbf{label} \\
 & & \mid \texttt{IF}\ \textbf{id}\ \textbf{relop}\ \mathit{Atom} \\
 & & \qquad \texttt{THEN}\ \textbf{label}\ \texttt{ELSE}\ \textbf{label} \\
 & & \mid \textbf{id} := \texttt{CALL}\ \textbf{functionid}(\mathit{Args}) \\
 & & \mid \texttt{RETURN}\ \textbf{id} \\
\mathit{Atom} & \rightarrow & \textbf{id} \mid \textbf{num} \\
\mathit{Args} & \rightarrow & \textbf{id}\ ,\ \mathit{Args} \mid \textbf{id}
\end{array}
$$

Give examples of (in)valid programs/instructions.

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-do` and `repeat-until`).
- (Will see details as we go.)

Syntax-directed translation:

- In practice we work on the abstract syntax tree ABSYN (but use concrete-syntax-like notation for readability, like for interpretation and typechecking),
- 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)

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 Arithmetic Expressions

Expressions in Source Language

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

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \textbf{num} \mid \textbf{id} \\
 & & \mid \textbf{unop}\ \mathit{Exp} \\
 & & \mid \mathit{Exp}\ \textbf{binop}\ \mathit{Exp} \\
 & & \mid \textbf{id}(\mathit{Exps}) \\
\mathit{Exps} & \rightarrow & \mathit{Exp} \mid \mathit{Exp}\ ,\ \mathit{Exps}
\end{array}
$$

Translating Arithmetic Expressions

Expressions in Source Language

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

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \textbf{num} \mid \textbf{id} \\
 & & \mid \textbf{unop}\ \mathit{Exp} \\
 & & \mid \mathit{Exp}\ \textbf{binop}\ \mathit{Exp} \\
 & & \mid \textbf{id}(\mathit{Exps}) \\
\mathit{Exps} & \rightarrow & \mathit{Exp} \mid \mathit{Exp}\ ,\ \mathit{Exps}
\end{array}
$$

Translation function:

$$\mathit{Trans}_{\mathit{Exp}} : \mathit{Exp} * \mathit{VTable} * \mathit{FTable} * \mathit{Location} \rightarrow \mathit{Instrs}$$

- Returns a list of intermediate code instructions *Instrs* that ...
- ...upon execution, computes *Exp*'s result in IL variable *Location*.
- (Recursive) case analysis on *Exp*'s abstract syntax tree AbSyn.
  - "Syntax-directed translation"

Symbol Tables and Helper Functions

Translation function:

$$Trans_{Exp} : Exp * VTable * FTable * Location \rightarrow Instrs$$

Symbol Tables **for codegen phase**

vtable : maps each variable name in source language to its corresponding IL variable name (symbolic register).

ftable : source function names to IL 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 (side-effectful!)
- 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

$Trans_{Exp}$: $Exp * VTable * FTable * Location \rightarrow Instrs$

$Trans_{Exp}$ $(exp, vtable, ftable, place) =$ `case` exp `of`

num	$v = getvalue(\textbf{num})$ $[place := v]$
id	$x = lookup(vtable, getname(\textbf{id}))$ *// Can assume that source program has already been type-checked* $[place := x]$
unop Exp_1	$place_1 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $op = \texttt{trans\_op}(getname(\textbf{unop}))$ $code_1$ @ $[place := op\ place_1]$
Exp_1 **binop** Exp_2	$place_1 = newvar()$ $place_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, place_2)$ $op = \texttt{trans\_op}(getname(\textbf{binop}))$ $code_1$ @ $code_2$ @ $[place := place_1\ op\ place_2]$

Generating Code for a Function Call

$Trans_{Exp}\ (exp, vtable, ftable, place) = \texttt{case}\ exp\ \texttt{of}$

$\mathbf{id}(Exps)$	$(code_1, [a_1, \ldots, a_n]) = Trans_{Exps}(Exps, vtable, ftable)$ $fname = lookup(ftable, getname(\mathbf{id}))$ $code_1\ \texttt{@}\ [place := \texttt{CALL}\ fname(a_1, \ldots, a_n)]$

$Trans_{Exps}$ returns the code that evaluates the function's parameters, and the list of new-intermediate variables (that hold the results).

$Trans_{Exps} : Exps * VTable * FTable \rightarrow Instrs * Args$

$Trans_{Exps}(exps, vtable, ftable) = \texttt{case}\ exps\ \texttt{of}$

Exp	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_1, [place])$
$Exp\ ,\ Exps$	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1\ \texttt{@}\ code_2$ $args_1 = place :: args$ $(code_3, args_1)$

Translation Example

Assume the following symbol tables:

- vtable $= [x \mapsto v_0,\ y \mapsto v_1,\ z \mapsto v_2]$
- ftable $= [f \mapsto \_F\_1]$

Translation of *Exp* with place $= t_0$:

- $Exp = \mathtt{x} - 3$

Translation Example

Assume the following symbol tables:

- vtable $= [x \mapsto v_0,\ y \mapsto v_1,\ z \mapsto v_2]$
- ftable $= [f \mapsto \_F\_1]$

Translation of *Exp* with place $= t_0$:

- $Exp = \mathtt{x} - 3$

$$\begin{array}{r}\mathtt{t}_1 := \mathtt{v}_0 \\ \mathtt{t}_2 := 3 \\ \mathtt{t}_0 := \mathtt{t}_1 - \mathtt{t}_2\end{array}$$

Translation Example

Assume the following symbol tables:

- vtable $= [x \mapsto v_0,\ y \mapsto v_1,\ z \mapsto v_2]$
- ftable $= [f \mapsto \_F\_1]$

Translation of *Exp* with place $= t_0$:

- $Exp = \mathtt{x} - 3$

$$\begin{aligned} \mathtt{t}_1 &:= \mathtt{v}_0 \\ \mathtt{t}_2 &:= 3 \\ \mathtt{t}_0 &:= \mathtt{t}_1 - \mathtt{t}_2 \end{aligned}$$

Note: generated IL code need not be "optimal"; may get improved by later optimization and/or machine-code generation phases.

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

  - **Hint:** When hand-translating, it's often convenient to write out the generated IL code from the bottom up.

Translation Example

Assume the following symbol tables:

- vtable $= [x \mapsto v_0,\ y \mapsto v_1,\ z \mapsto v_2]$
- ftable $= [f \mapsto \_F\_1]$

Translation of *Exp* with place $= t_0$:

- $Exp = \texttt{x} - 3$

$$\begin{aligned} \texttt{t}_1 &:= \texttt{v}_0 \\ \texttt{t}_2 &:= 3 \\ \texttt{t}_0 &:= \texttt{t}_1 - \texttt{t}_2 \end{aligned}$$

Note: generated IL code need not be "optimal"; may get improved by later optimization and/or machine-code generation phases.

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

$$\begin{aligned} \texttt{t}_1 &:= 3 \\ \texttt{t}_4 &:= \texttt{v}_0 \\ \texttt{t}_5 &:= \texttt{v}_1 \\ \texttt{t}_3 &:= \texttt{t}_4 - \texttt{t}_5 \\ \texttt{t}_6 &:= \texttt{v}_2 \\ \texttt{t}_2 &:= \texttt{CALL \_F\_1}(\texttt{t}_3, \texttt{t}_6) \\ \texttt{t}_0 &:= \texttt{t}_1 + \texttt{t}_2 \end{aligned}$$

  - **Hint:** When hand-translating, it's often convenient to write out the generated IL code from the bottom up.

Translating Statements

Statements in Source Lang.

- Sequence of statements
- Assignment
- Conditional Branching
- Loops: `while` and `repeat` (simple conditions for now)

$$
\begin{array}{lcl}
\mathit{Stat} & \rightarrow & \mathit{Stat}\ ;\ \mathit{Stat} \\
 & & \mid \mathbf{id} := \mathit{Exp} \\
 & & \mid \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \{\ \mathit{Stat}\ \} \\
 & & \mid \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \{\ \mathit{Stat}\ \}\ \texttt{else}\ \{\ \mathit{Stat}\ \} \\
 & & \mid \texttt{while}\ \mathit{Cond}\ \texttt{do}\ \{\ \mathit{Stat}\ \} \\
 & & \mid \texttt{repeat}\ \{\ \mathit{Stat}\ \}\ \texttt{until}\ \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \mathit{Exp}\ \mathbf{relop}\ \mathit{Exp}
\end{array}
$$

We assume relational operators translate directly (using `trans_op`).

Translating Statements

Statements in Source Lang.

- Sequence of statements
- Assignment
- Conditional Branching
- Loops: `while` and `repeat` (simple conditions for now)

$$
\begin{array}{lcl}
\mathit{Stat} & \rightarrow & \mathit{Stat}\ ;\ \mathit{Stat} \\
 & & \mid \mathbf{id} := \mathit{Exp} \\
 & & \mid \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \{\ \mathit{Stat}\ \} \\
 & & \mid \texttt{if}\ \mathit{Cond}\ \texttt{then}\ \{\ \mathit{Stat}\ \}\ \texttt{else}\ \{\ \mathit{Stat}\ \} \\
 & & \mid \texttt{while}\ \mathit{Cond}\ \texttt{do}\ \{\ \mathit{Stat}\ \} \\
 & & \mid \texttt{repeat}\ \{\ \mathit{Stat}\ \}\ \texttt{until}\ \mathit{Cond} \\
\mathit{Cond} & \rightarrow & \mathit{Exp}\ \mathbf{relop}\ \mathit{Exp}
\end{array}
$$

We assume relational operators translate directly (using `trans_op`).

Translation function:
$\mathit{Trans}_{\mathit{Stat}} : \mathit{Stat} * \mathit{VTable} * \mathit{FTable} \rightarrow \mathit{Instrs}$

- As before: syntax-directed, case analysis on *Stat*
- Intermediate code instructions for statements

Generating Code for Sequences, Assignments,...

$Trans_{Stat}$: $Stat * VTable * FTable \rightarrow Instrs$

$Trans_{Stat}(stat, vtable, ftable) =$ `case` $stat$ `of`

$Stat_1$; $Stat_2$	$code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1$ `@` $code_2$
id $:=$ Exp	$place = lookup(vtable, getname(\mathbf{id}))$ $Trans_{Exp}(Exp, vtable, ftable, place)$ *// Book: introduces an extra temporary, in case of subtle aliasing between place and Exp*
...	(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

$$Trans_{Cond} : Cond * Label * Label * VTable * FTable \rightarrow Instrs$$

$$Trans_{Cond}(cond, label_t, label_f, vtable, ftable) = \texttt{case}\ cond\ \texttt{of}$$

Exp_1 **relop** Exp_2	$t_1 = newvar()$ $t_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = \texttt{trans\_op}(getname(\textbf{relop}))$ $code_1$ `@` $code_2$ `@` $[\texttt{IF}\ t_1\ op\ t_2\ \texttt{THEN}\ label_t\ \texttt{ELSE}\ label_f]$

- Uses the `IF` of the intermediate language
- Expressions need to be evaluated before
  (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

$Trans_{Stat}(stat, vtable, ftable) =$ case $stat$ of	
`if` $Cond$ `then` $Stat_1$	$label_t = newlabel()$ $label_f = newlabel()$ $code_c = Trans_{Cond}(Cond, label_t, label_f, vtable, ftable)$ $code_s = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_c$ @ [LABEL $label_t$] @ $code_s$ @ [LABEL $label_f$]
// *Machine-code generation for most architectures will usually eliminate the fall-through jump to* $label_t$.	
`if` $Cond$ `then` $Stat_1$ `else` $Stat_2$	$label_t = newlabel()$ $label_f = newlabel()$ $label_e = newlabel()$ $code_c = Trans_{Cond}(Cond, label_t, label_f, vtable, ftable)$ $code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_c$ @ [LABEL $label_t$] @ $code_1$ @ [GOTO $label_e$] @ [LABEL $label_f$] @ $code_2$ @ [LABEL $label_e$]

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.

$Trans_{Stat}(stat, vtable, ftable) = \texttt{case } stat \texttt{ of}$

`repeat` $Stat$ `until` $Cond$	$label_f = newlabel()$ $label_t = newlabel()$ $code_1 = Trans_{Stat}(Stat, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond, label_t, label_f, vtable, ftable)$ $[\texttt{LABEL } label_f]$ @ $code_1$ @ $code_2$ @ $[\texttt{LABEL } label_t]$
`while` $Cond$ `do` $Stat$	$label_s = newlabel()$ $label_t = newlabel()$ $label_f = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_t, label_f, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat, vtable, ftable)$ $[\texttt{LABEL } label_s]$ @ $code_1$ @ $[\texttt{LABEL } label_t]$ @ $code_2$ @ $[\texttt{GOTO } label_s]$ @ $[\texttt{LABEL } label_f]$

Translation Example

- Symbol table vtable: $[x \mapsto v_0,\ y \mapsto v_1,\ z \mapsto v_2]$
- Symbol table ftable: [`getInt` $\mapsto$ `libIO_getInt`]

```
x := 3;
y := getInt();
z := 1;
while y > 0 do
   {y := y - 1;
    z := z * x}
```

Note: using the simplified translation of assignments from earlier slide, $\neq$ Book.

Translation Example

- Symbol table vtable: $[x \mapsto v_0,\ y \mapsto v_1,\ z \mapsto v_2]$
- Symbol table ftable: [`getInt` $\mapsto$ `libIO_getInt`]

```
x := 3;
y := getInt();
z := 1;
while y > 0 do
   {y := y - 1;
    z := z * x}
```

Note: using the simplified translation of assignments from earlier slide, $\neq$ Book.

```
v_0 := 3
v_1 := CALL libIO_getInt()
v_2 := 1
```

Translation Example

- Symbol table vtable: $[x \mapsto v_0,\ y \mapsto v_1,\ z \mapsto v_2]$
- Symbol table ftable: [getInt ↦ libIO_getInt]

```
x := 3;
y := getInt();
z := 1;
while y > 0 do
   {y := y - 1;
    z := z * x}
```

Note: using the simplified translation of assignments from earlier slide, ≠ Book.

```
v_0 := 3
v_1 := CALL libIO_getInt()
v_2 := 1
 LABEL l_s
  t_1 := v_1
  t_2 := 0
  IF t_1 > t_2 THEN l_t else l_f
  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: [getInt ↦ libIO_getInt]

```
x := 3;
y := getInt();
z := 1;
while y > 0 do
   {y := y - 1;
    z := z * x}
```

Note: using the simplified translation of assignments from earlier slide, ≠ Book.

```
v_0 := 3
v_1 := CALL libIO_getInt()
v_2 := 1
 LABEL l_s
  t_1 := v_1
  t_2 := 0
  IF t_1 > t_2 THEN l_t else l_f
  LABEL l_t
   t_3 := v_1
   t_4 := 1
   v_1 := t_3 - t_4




  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: [getInt ↦ libIO_getInt]

```
x := 3;
y := getInt();
z := 1;
while y > 0 do
   {y := y - 1;
    z := z * x}
```

Note: using the simplified translation of assignments from earlier slide, ≠ Book.

```
v_0 := 3
v_1 := CALL libIO_getInt()
v_2 := 1
 LABEL l_s
  t_1 := v_1
  t_2 := 0
  IF t_1 > t_2 THEN l_t else l_f
  LABEL l_t
   t_3 := v_1
   t_4 := 1
   v_1 := t_3 - t_4
   t_5 := v_2
   t_6 := v_0
   v_2 := t_5 * t_6
  GOTO l_s
 LABEL l_f
```

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 (esp. integer variables) used as Boolean
- Logical operators (not, and, or)
- Boolean expressions used in arithmetics

$$
\begin{array}{lcl}
\mathit{Cond} & \rightarrow & \mathit{Exp}\ \textbf{relop}\ \mathit{Exp} \\
 & & \mid \mathit{Exp} \\
 & & \mid \texttt{not}\ \mathit{Cond} \\
 & & \mid \mathit{Cond}\ \texttt{and}\ \mathit{Cond} \\
 & & \mid \mathit{Cond}\ \texttt{or}\ \mathit{Cond} \\
\mathit{Exp} & \rightarrow & \cdots \mid \mathit{Cond}
\end{array}
$$

More Complex Conditions, Boolean Expressions

Boolean Expressions as Conditions

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

$$\begin{array}{lcl} \mathit{Cond} & \rightarrow & \mathit{Exp}\ \textbf{relop}\ \mathit{Exp} \\ & & \mid \mathit{Exp} \\ & & \mid \texttt{not}\ \mathit{Cond} \\ & & \mid \mathit{Cond}\ \texttt{and}\ \mathit{Cond} \\ & & \mid \mathit{Cond}\ \texttt{or}\ \mathit{Cond} \\ \mathit{Exp} & \rightarrow & \cdots \mid \mathit{Cond} \end{array}$$

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

- Interpret numeric values as Boolean expressions: 0 treated as `false`, all other values as `true`.
- Conversely, truth values as arithmetic expressions: `false` is 0, `true` is 1

Numbers and Boolean Values, Negation

Expressions as Boolean values, negation:

$Trans_{Cond}$: $Cond * Label * Label * VTable * FTable \rightarrow Instrs$

$Trans_{Cond}(cond, label_t, label_f, vtable, ftable) =$ `case` $cond$ `of`

...

Exp	$t = newvar()$ $code = Trans_{Exp}(Exp, vtable, ftable, t)$ $code$ @ $[$IF $t \neq 0$ THEN $label_t$ ELSE $label_f]$
`not` $Cond$	$Trans_{Cond}(Cond, label_f, label_t, vtable, ftable)$

...

Numbers and Boolean Values, Negation

Expressions as Boolean values, negation:

$Trans_{Cond}$: $Cond * Label * Label * VTable * FTable \to Instrs$

$Trans_{Cond}(cond, label_t, label_f, vtable, ftable) =$ `case` $cond$ `of`

...

Exp	$t = newvar()$ $code = Trans_{Exp}(Exp, vtable, ftable, t)$ $code$ @ [`IF` $t \neq 0$ `THEN` $label_t$ `ELSE` $label_f$]
`not` $Cond$	$Trans_{Cond}(Cond, label_f, label_t, vtable, ftable)$

...

Conversion of Boolean values to numbers (by jumps):

$Trans_{Exp}$: $Exp * VTable * FTable * Location \to Instr$

$Trans_{Exp}(exp, vtable, ftable, place) =$ `case` exp `of`

...

$Cond$	$label_1 = newlabel()$ $label_2 = newlabel()$ $t = newvar()$ $code = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ [$t := 0$] @ $code$ @ [`LABEL` $label_1$, $t := 1$] @ [`LABEL` $label_2$, $place := t$] *// Book: stores result directly into final destination, not via t: unsafe with simplifed trans. of assignments.*

Sequential Evaluation of Conditions

Short-circuiting conjunction and disjunction may look like ordinary (infix) functions, but they are not!

```
Microsoft (R) F# Interactive version 12.4.0.0 for F# 7.0

> (+);;
val it : (int -> int -> int) = <fun:it@1>
> (&&);;
val it : (bool -> bool -> bool) = <fun:it@2-1>

> (+) 3 4;;
val it : int = 7
> (&&) false (failwith "oops");;
val it : bool = false

> let myplus = (+) in myplus 3 4;;
val it : int = 7
> let myand = (&&) in myand false (failwith "oops");;
System.Exception: oops
...
```

Need to treat them specially in compiler.

Sequential Evaluation by “Jumping Code”

$$Trans_{Cond} \; : \; Cond * Label * Label * VTable * FTable \rightarrow Instrs$$

$$Trans_{Cond}(cond, label_t, label_f, vtable, ftable) = \texttt{case}\ cond\ \texttt{of}$$

...

$Cond_1$ `and` $Cond_2$	$label_{next} = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_{next}, label_f, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$ @ [LABEL $label_{next}$] @ $code_2$
$Cond_1$ `or` $Cond_2$	$label_{next} = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_t, label_{next}, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$ @ [LABEL $label_{next}$] @ $code_2$

Sequential Evaluation by "Jumping Code"

$Trans_{Cond} \;:\; Cond * Label * Label * VTable * FTable \rightarrow Instrs$

$Trans_{Cond}(cond, label_t, label_f, vtable, ftable) = \texttt{case}\ cond\ \texttt{of}$

...

$Cond_1$ `and` $Cond_2$	$label_{next} = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_{next}, label_f, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$ `@` `[LABEL` $label_{next}$`]` `@` $code_2$
$Cond_1$ `or` $Cond_2$	$label_{next} = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_t, label_{next}, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$ `@` `[LABEL` $label_{next}$`]` `@` $code_2$

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

Sequential Evaluation by "Jumping Code"

$Trans_{Cond}$: $Cond * Label * Label * VTable * FTable \rightarrow Instrs$

$Trans_{Cond}(cond, label_t, label_f, vtable, ftable) =$ `case` $cond$ `of`

...

$Cond_1$ `and` $Cond_2$	$label_{next} = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_{next}, label_f, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$ `@` `[LABEL` $label_{next}$`]` `@` $code_2$
$Cond_1$ `or` $Cond_2$	$label_{next} = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_t, label_{next}, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$ `@` `[LABEL` $label_{next}$`]` `@` $code_2$

- Note: No logical operations in intermediate language! Logics of `and` and `or` encoded by jumps.
- Alternative: Logical operators in intermediate language

  $$Cond \Rightarrow Exp \Rightarrow Exp\ \mathbf{binop}\ Exp$$

  Translated like an arithmetic operation (bitwise and/or). Right results for **false** ~ 0, **true** ~ 1, but...

Sequential Evaluation by "Jumping Code"

$Trans_{Cond} \; : \; Cond * Label * Label * VTable * FTable \rightarrow Instrs$

$Trans_{Cond}(cond, label_t, label_f, vtable, ftable) = \texttt{case}\ cond\ \texttt{of}$

. . .

$Cond_1$ `and` $Cond_2$	$label_{next} = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_{next}, label_f, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$ `@` [`LABEL` $label_{next}$] `@` $code_2$
$Cond_1$ `or` $Cond_2$	$label_{next} = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_t, label_{next}, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1$ `@` [`LABEL` $label_{next}$] `@` $code_2$

- Note: No logical operations in intermediate language! Logics of `and` and `or` encoded by jumps.
- Alternative: Logical operators in intermediate language

  $$Cond \Rightarrow Exp \Rightarrow Exp\ \textbf{binop}\ Exp$$

  Translated like an arithmetic operation (bitwise and/or). Right results for **false** ~ 0, **true** ~ 1, but... always 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

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: $\mathit{Stat} \rightarrow \textbf{label} :$
  $\qquad\qquad\qquad\quad \mid \texttt{goto}\ \textbf{label}$
  - Necessary instructions are already in the intermediate language.
  - Need to build symbol table for mapping source-to-IL 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: $\mathit{Stat} \rightarrow \textbf{label}: \mid \texttt{goto}\ \textbf{label}$
  - Necessary instructions are already in the intermediate language.
  - Need to build symbol table for mapping source-to-IL labels.
- Case/Switch:

$$\begin{array}{lcl} \mathit{Stat} & \rightarrow & \texttt{case}\ \mathit{Exp}\ \texttt{of}\ \{\ \mathit{Alts}\ \} \\ \mathit{Alts} & \rightarrow & \textbf{num} : \mathit{Stat} \mid \textbf{num} : \mathit{Stat},\ \mathit{Alts} \end{array}$$

  - When exited after each case (e.g., Pascal): chain of `if-then-else`
  - When "falling through" (e.g., C, Java): `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: $\mathit{Stat} \rightarrow \textbf{label} :$ $\mid$ `goto` $\textbf{label}$
  - Necessary instructions are already in the intermediate language.
  - Need to build symbol table for mapping source-to-IL labels.
- Case/Switch:

$$\begin{array}{lcl} \mathit{Stat} & \rightarrow & \texttt{case}\ \mathit{Exp}\ \texttt{of}\ \{\ \mathit{Alts}\ \} \\ \mathit{Alts} & \rightarrow & \textbf{num} : \mathit{Stat} \mid \textbf{num} : \mathit{Stat},\ \mathit{Alts} \end{array}$$

  - When exited after each case (e.g., Pascal): chain of `if-then-else`
  - When "falling through" (e.g., C, Java): `if-then-else` and `goto`.
- Break and Continue: $\mathit{Stat} \rightarrow$ `break` $\mid$ `continue`
  - (`break`: jump to after loop, `continue`: jump to end of loop body).
  - Needs two jump target labels used only inside loop bodies (parameters to translation function $\mathit{Trans}_{\mathit{Stat}}$)

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: $\mathit{Stat} \rightarrow \textbf{label} :$
  $\mid$ `goto` **label** ... considered harmful (Dijkstra 1968)
  - Necessary instructions are already in the intermediate language.
  - Need to build symbol table for mapping source-to-IL labels.
- Case/Switch:
  $\mathit{Stat} \rightarrow$ `case` Exp `of` $\{\ \mathit{Alts}\ \}$
  $\mathit{Alts} \rightarrow \textbf{num} : \mathit{Stat} \mid \textbf{num} : \mathit{Stat},\ \mathit{Alts}$
  - When exited after each case (e.g., Pascal): chain of `if-then-else`
  - When "falling through" (e.g., C, Java): `if-then-else` and `goto`.
- Break and Continue: $\mathit{Stat} \rightarrow$ `break` $\mid$ `continue`
  - (`break`: jump to after loop, `continue`: jump to end of loop body).
  - Needs two jump target labels used only inside loop bodies (parameters to translation function $\mathit{Trans}_{\mathit{Stat}}$)

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 `int` elements)

Extending the Source Language

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

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \ldots \mid \mathit{Idx} \\
\mathit{Stat} & \rightarrow & \ldots \mid \mathit{Idx} \;:=\; \mathit{Exp} \\
\mathit{Idx} & \rightarrow & \mathbf{id}\ [\ \mathit{Exp}\ ]
\end{array}
$$

Translating Arrays (of `int` elements)

Extending the Source Language

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

$$
\begin{array}{lcl}
Exp & \rightarrow & \ldots \mid Idx \\
Stat & \rightarrow & \ldots \mid Idx \; := \; Exp \\
Idx & \rightarrow & \textbf{id} \; [\; Exp \;]
\end{array}
$$

Again we extend $Trans_{Exp}$ and $Trans_{Stat}$.

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

$Trans_{Idx} : Idx * VTable * FTable \rightarrow Instrs * Location$
$Trans_{Idx}(index, vtable, ftable) = \texttt{case } index \texttt{ of}$

id[*Exp*]	$base = lookup(vtable, getname(\mathbf{id}))$ $addr = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, addr)$ $code_2 = code_1 \ @ \ [addr := addr * 4, addr := addr + base]$ $(code_2, addr)$

// Unless C-like language, should also generate code to check that index value is within array bounds!

Returns:

- Code to calculate the absolute address ...
- of the array element in memory (corresponding to `index`), ...
- ... and a new variable (*addr*) containing that address.

Generating Code for Array Access

Address-calculation code: in expression and statement translation.

- Read access inside expressions:

$Trans_{Exp}(exp, vtable, ftable, place) = \texttt{case}\ exp\ \texttt{of}$

$\quad \ldots$

Idx	$(code_1, address) = Trans_{Idx}(Idx, vtable, ftable)$ $code_1\ \texttt{@}\ [place := M[address]]$

- Write access in assignments:

$Trans_{Stat}(stat, vtable, ftable) = \texttt{case}\ stat\ \texttt{of}$

$\quad \ldots$

$Idx := Exp$	$(code_1, address) = Trans_{Idx}(Index, vtable, ftable)$ $t = newvar()$ $code_2 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_1\ \texttt{@}\ code_2\ \texttt{@}\ [M[address] := t]$

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.

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \ldots \mid \mathit{Idx} \\
\mathit{Stat} & \rightarrow & \ldots \mid \mathit{Idx} \;:=\; \mathit{Exp} \\
\mathit{Idx} & \rightarrow & \mathbf{id}[\mathit{Exp}] \;\mid \mathit{Idx}[\mathit{Exp}]
\end{array}
$$

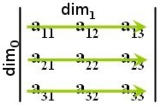
Multi-Dimensional Arrays

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.

$$\begin{array}{lcl} Exp & \rightarrow & \ldots \mid Idx \\ Stat & \rightarrow & \ldots \mid Idx \; := \; Exp \\ Idx & \rightarrow & \mathbf{id}[Exp] \;\mid Idx[Exp] \end{array}$$

- 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 $k \cdot dim_1 \cdot dim_2 + l \cdot dim_2 + m$)



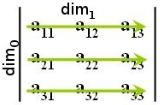
Multi-Dimensional Arrays

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.

$$\begin{array}{lcl} Exp & \rightarrow & \ldots \mid Idx \\ Stat & \rightarrow & \ldots \mid Idx \;:=\; Exp \\ Idx & \rightarrow & \textbf{id}[Exp] \mid Idx[Exp] \end{array}$$

- 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 $k \cdot dim_1 \cdot dim_2 + l \cdot dim_2 + m$)



- Address calculation need to know sizes in each dimension.
  Symbol table: base address and list of array-dimension sizes.
- Need to change $Trans_{Idx}$, i.e., add recursive index calculation.

Address Calculation in Multiple Dimensions

$Trans_{Idx}(index, vtable, ftable) =$

$(code_1, t, base, []) = Calc_{Idx}(index, vtable, ftable)$
$code_2 = code_1 \texttt{ @ } [t := t * 4, t := t + base]$
$(code_2, t)$

Address Calculation in Multiple Dimensions

$$
\begin{array}{l}
Trans_{Idx}(index, vtable, ftable) = \\
\hline
\quad (code_1, t, base, []) = Calc_{Idx}(index, vtable, ftable) \\
\quad code_2 = code_1 \ \texttt{@} \ [t := t * 4, t := t + base] \\
\quad (code_2, t) \\
\hline
\end{array}
$$

Recursive index calculation, multiplies with dimension at each step.

$$
\begin{array}{ll}
\multicolumn{2}{l}{Calc_{Idx}(index, vtable, ftable) = \texttt{case } index \texttt{ of}} \\
\hline
\mathbf{id}[Exp] & (base, dims) = lookup(vtable, getname(\mathbf{id})) \\
 & addr = newvar() \\
 & code = Trans_{Exp}(Exp, vtable, ftable, addr) \\
 & (code, addr, base, tail(dims)) \\
\hline
Index[Exp] & (code_1, addr, base, dims) = Calc_{Idx}(Index, vtable, ftable) \\
 & d = head(dims) \\
 & t = newvar() \\
 & code_2 = Trans_{Exp}(Exp, vtable, ftable, t) \\
 & code_3 = code_1 \ \texttt{@} \ code_2 \ \texttt{@} \ [addr := addr * d, addr := addr + t] \\
 & (code_3, addr, base, tail(dims)) \\
\hline
\end{array}
$$

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

$$\begin{array}{lcl} Stat & \rightarrow & Decl;\ Stat \\ Decl & \rightarrow & \texttt{int}\ \mathbf{id} \\ & & \mid \texttt{int}\ \mathbf{id[num]} \end{array}$$

Function $Trans_{Decl} : Decl * VTable \rightarrow Instrs * VTable$

- translates declarations to code and new symbol table.

Translating Declarations to Scope and Allocation

Code with local scope (extended symbol table):

$Trans_{Stat}(stat, vtable, ftable) =$ `case` $stat$ `of`	
$Decl\ ;\ Stat_1$	$(code_1, vtable_1) = Trans_{Decl}(Decl, vtable)$ $code_2 = Trans_{Stat}(Stat_1, vtable_1, ftable)$ $code_1$ `@` $code_2$

Translating Declarations to Scope and Allocation

Code with local scope (extended symbol table):

$Trans_{Stat}(stat, vtable, ftable) =$ `case` $stat$ `of`

$Decl\ ;\ Stat_1$	$(code_1, vtable_1) = Trans_{Decl}(Decl, vtable)$ $code_2 = Trans_{Stat}(Stat_1, vtable_1, ftable)$ $code_1$ `@` $code_2$

Building the symbol table and allocating:

$Trans_{Decl} : Decl * VTable \rightarrow Instrs * VTable$
$Trans_{Decl}(decl, vtable) =$ `case` $decl$ `of`

`int` **id**	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\mathbf{id}), t_1)$ $([],\ vtable_1)$
`int` **id[num]**	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\mathbf{id}), t_1)$ $size = 4 * getvalue(\mathbf{num})$ // compile-time calculation, cf. Book $([t_1 := HP,\ HP := HP + size],\ vtable_1)$

. . . where `HP` is the IL variable containing 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 Unicode chars (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 Unicode chars (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.

Coming up

- LAB today: Fasto's `RiscV` and `CodeGen` modules
- Wednesday (with Robert):
  - Machine-code generation from IL (with code-pattern selection)
- (Thursday: info meeting about 3rd-year BSc electives)
  - Aud. 1 (and 6, 8), HCØ, 16:15–18:00
  - See separate annoucement
- Monday next week:
  - Optimizations in the Fasto compiler
- (Tuesday, May 23, 15–18: "Open house" at ITX exam house:)
  - **Register** and attend if you've never tried ITX at South Campus
  - See link on Absalon or in your study messages.