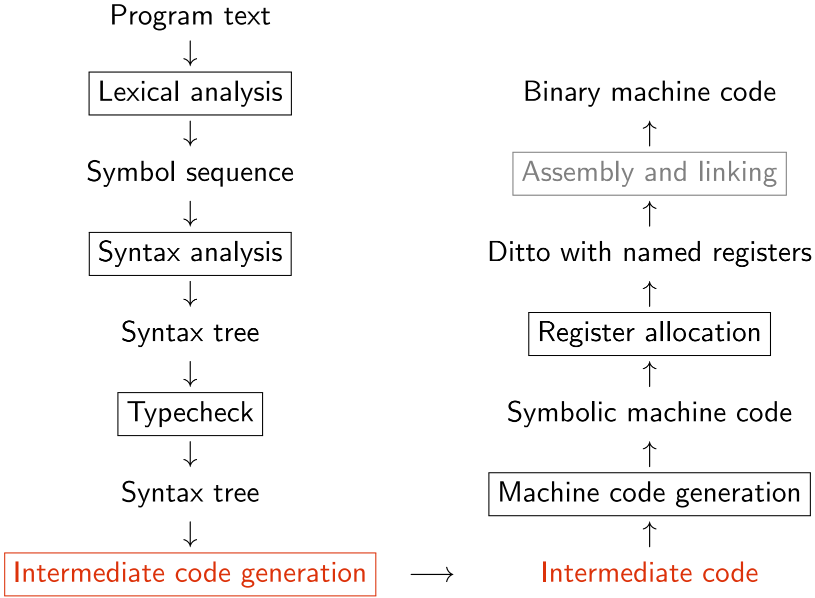
Intermediate Code Generation

Cosmin E. Oancea
cosmin.oancea@diku.dk

Modified by Marco Valtorta for CSCE 531 at UofSC
Based on Jost Berthold's slides and Torben Mogensen's book
Department of Computer Science
University of Copenhagen

February 2018 IPS Lecture Slides

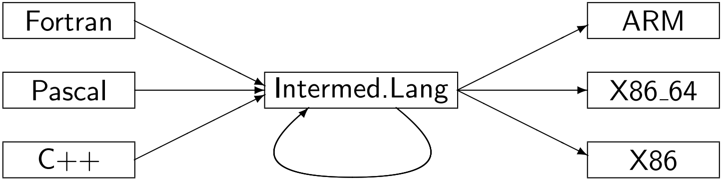
Structure of a Compiler



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)?

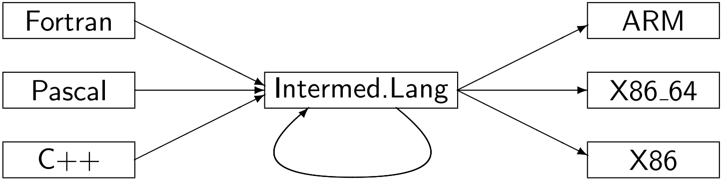
- Compilers for different platforms and languages can share parts.



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

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*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 ⇒ 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

Prg	→	*Fcts*
Fcts	→	*Fct Fcts* | *Fct*
Fct	→	*Hdr Bd*
Hdr	→	**functionid**(*Args*)
Bd	→	[*Instrs*]
Instrs	→	*Instr* , *Instrs* | *Instr*
Instr	→	**id** := *Atom* | **id** := **unop** *Atom*
		| **id** := **id binop** *Atom*
		| **id** := *M*[*Atom*] | *M*[*Atom*] := **id**
		| `LABEL` **label** | `GOTO` **label**
		| `IF` **id relop** *Atom* `THEN` **label** `ELSE` **label**
Atom	→	**id** | **num**

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

Prg	$\rightarrow$	*Fcts*
Fcts	$\rightarrow$	*Fct Fcts* | *Fct*
Fct	$\rightarrow$	*Hdr Bd*
Hdr	$\rightarrow$	**functionid**(*Args*)
Bd	$\rightarrow$	[*Instrs*]
Instrs	$\rightarrow$	*Instr* , *Instrs* | *Instr*
Instr	$\rightarrow$	**id** := *Atom* | **id** := **unop** *Atom*
		| **id** := **id binop** *Atom*
		| **id** := *M*[*Atom*] | *M*[*Atom*] := **id**
		| `LABEL` **label** | `GOTO` **label**
		| `IF` **id relop** *Atom* `THEN` **label** `ELSE` **label**
		| **id** := `CALL` **functionid**(*Args*)
		| `RETURN` **id**
Atom	$\rightarrow$	**id** | **num**
Args	$\rightarrow$	**id** , *Args* | **id**

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 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)

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:

$\mathit{Trans}_{\mathit{Exp}}$ `:: (Exp, VTable, FTable, Location) -> [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 AbSyn.

Symbol Tables and Helper Functions

Translation function:

$Trans_{Exp}$ `:: (Exp, VTable, FTable, Location) -> [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

$Trans_{Exp}$: `(Exp, VTable, FTable, Location) -> [ICode]`

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

num	$v = getvalue(\textbf{num})$ $[place := v]$
id	$x = lookup(vtable, getname(\textbf{id}))$ $[place := x]$
unop Exp_1	$place_1 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ op = `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 = `trans_op`$(getname(\textbf{binop}))$ $code_1$ `@` $code_2$ `@` $[place := place_1\ op\ place_2]$

In this slide presentation, `@` (as in SML) is used instead of ++ (as in Haskell and in the Mogensen's book) for list concatenation.

Generating Code for a Function Call

$Trans_{Exp}$ (*exp*, *vtable*, *ftable*, *place*) = case *exp* of	
id(*Exps*)	$(code_1, [a_1, \ldots, a_n]) = Trans_{Exps}(Exps, vtable, ftable)$ $fname = lookup(ftable, getname(\mathbf{id}))$ $code_1$ @ $[place :=$ `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 store the result).

$Trans_{Exps}$ `: (Exps, VTable, FTable) -> ([ICode], [Location])`

$Trans_{Exps}(exps, vtable, ftable)$ = case *exps* of	
Exp	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_1, [place])$
Exp , *Exps*	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1$ @ $code_2$ $args_1 = place :: args$ $(code_3, args_1)$

Translation Example

Assume the following symbol tables:

- vtable $= [x \mapsto v0,\ y \mapsto v1,\ z \mapsto v2]$
- ftable $= [f \mapsto \_F\_1, \texttt{+} \mapsto \ \texttt{+}, \texttt{-} \mapsto \ \texttt{-}]$

Translation of `Exp` with place $= t0$:

- `Exp=x-3`

Translation Example

Assume the following symbol tables:

- vtable $= [x \mapsto v0,\ y \mapsto v1,\ z \mapsto v2]$
- ftable $= [f \mapsto \_F\_1, \texttt{+} \mapsto\ \texttt{+}, \texttt{-} \mapsto\ \texttt{-}]$

Translation of `Exp` with place $= t0$:

- `Exp=x-3`

```
t1 := v0
t2 := 3
t0 := t1 - t2
```

Translation Example

Assume the following symbol tables:

- vtable $= [x \mapsto v0,\ y \mapsto v1,\ z \mapsto v2]$
- ftable $= [f \mapsto \_F\_1, \texttt{+} \mapsto \texttt{+}, \texttt{-} \mapsto \texttt{-}]$

Translation of `Exp` with place $= t0$:

- `Exp=x-3`

```
t1 := v0
t2 := 3
t0 := t1 - t2
```

- `Exp=3+f(x-y,z)`

Translation Example

Assume the following symbol tables:

- vtable $= [x \mapsto v0,\ y \mapsto v1,\ z \mapsto v2]$
- ftable $= [f \mapsto \_F\_1, \texttt{+} \mapsto\ \texttt{+}, \texttt{-} \mapsto\ \texttt{-}]$

Translation of `Exp` with place $= t0$:

- `Exp=x-3`

```
t1 := v0
t2 := 3
t0 := t1 - t2
```

- `Exp=3+f(x-y,z)`

```
t1 := 3
   t4 := v0
   t5 := v1
  t3 := t4 - t5
  t6 := v2
t2 := CALL _F_1(t3, t6)
t0 := t1 + t2
```

Translating Statements

Statements in Source Language

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

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

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

Translating Statements

Statements in Source Language

- 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:

$\mathit{Trans}_{\mathit{Stat}}$ `:: (Stat, VTable, FTable) -> [ICode]`

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

Generating Code for Sequences, Assignments,...

$Trans_{Stat}$ `: (Stat, Vtable, Ftable) -> [ICode]`

$Trans_{Stat}(stat, vtable, ftable) =$ `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	$place = lookup(vtable, getname(\mathbf{id}))$ $Trans_{Exp}(Exp, vtable, ftable, place)$

... (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) -> [ICode]

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

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

$Trans_{Stat}(stat, vtable, ftable) = \texttt{case}\ stat\ \texttt{of}$	
$\texttt{if}\ Cond$ $\texttt{then}\ Stat_1$	$label_t = newlabel()$ $label_f = newlabel()$ $code_c = Trans_{Cond}(Cond, label_t, label_f, vtable, ftable)$ $code_s = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_c\ @\ [\texttt{LABEL}\ label_t]\ @\ code_s\ @\ [\texttt{LABEL}\ label_f]$
$\texttt{if}\ Cond$ $\texttt{then}\ Stat_1$ $\texttt{else}\ Stat_2$	$label_t = newlabel()$ $label_f = newlabel()$ $label_e = newlabel()$ $code_c = Trans_{Cond}(Cond, label_t, label_f, vtable, ftable)$ $code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_c\ @\ [\texttt{LABEL}\ label_t]\ @\ code_1\ @\ [\texttt{GOTO}\ label_e]$ $@\ [\texttt{LABEL}\ label_f]\ @\ code_2\ @\ [\texttt{LABEL}\ label_e]$

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.

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

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

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
    y := y - 1;
    z := z * x
```

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
    y := y - 1;
    z := z * x
```

```
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
    y := y - 1;
    z := z * x
```

```
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
    y := y - 1;
    z := z * x
```

```
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
    y := y - 1;
    z := z * x
```

```
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
```

More Complex Conditions, Boolean Expressions

Boolean Expressions as Conditions

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

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

More Complex Conditions, Boolean Expressions

Boolean Expressions as Conditions

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

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

We extend the translation functions $\mathit{Trans}_{\mathit{Exp}}$ and $\mathit{Trans}_{\mathit{Cond}}$:

- 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:

$Trans_{Cond}$ `: (Cond, Label, Label, Vtable, Ftable) -> [ICode]`
$Trans_{Cond}(cond, label_t, label_f, vtable, ftable) =$ `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) -> [ICode]`

$Trans_{Cond}(cond, label_t, label_f, vtable, ftable)$ = `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) -> [ICode]`

$Trans_{Exp}(exp, vtable, ftable, place)$ = `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 $label_2$, $place := t$]

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 $B_1 = \mathit{false}$, do not evaluate B_2 in $B_1 \&\& B_2$ (anyway *false*).
- If $B_1 = \mathit{true}$, 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 $B_1 = \mathit{false}$, do not evaluate B_2 in $B_1 \&\& B_2$ (anyway *false*).
- If $B_1 = \mathit{true}$, 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"

$Trans_{Cond}$ `: (Cond, Label, Label, Vtable, Ftable) -> [ICode]`

$Trans_{Cond}(cond, label_t, label_f, vtable, ftable) =$ `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` $label_{next}$`] @` $code_2$

Sequential Evaluation by "Jumping Code"

$Trans_{Cond}$: (Cond, Label, Label, Vtable, Ftable) -> [ICode]
$Trans_{Cond}(cond, label_t, label_f, vtable, ftable)$ = 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.

Sequential Evaluation by "Jumping Code"

$Trans_{Cond}$: `(Cond, Label, Label, Vtable, Ftable) -> [ICode]`

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

  $Cond \Rightarrow Exp \Rightarrow Exp$ **binop** Exp

  Translated as an arithmetic operation.

Sequential Evaluation by "Jumping Code"

$Trans_{Cond}$: `(Cond, Label, Label, Vtable, Ftable) -> [ICode]`

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

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

  Translated as an arithmetic operation. Evaluates both sides!

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: $\mathit{Stat} \rightarrow \textbf{label} : \mid \textbf{goto label}$

  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* → **label** :
  | **goto label**

  Necessary instructions are in the intermediate language.
  Needs to build symbol table of labels.
- Case/Switch: *Stat* → **case** *Exp* **of** [*Alts*]
  Alts → **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`.

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: $\mathit{Stat} \rightarrow$ **label** :
  | **goto label**
  Necessary instructions are in the intermediate language.
  Needs to build symbol table of labels.
- Case/Switch: $\mathit{Stat} \rightarrow$ **case** *Exp* **of** [*Alts*]
  $\mathit{Alts} \rightarrow$ **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: $\mathit{Stat} \rightarrow$ **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 $\mathit{Trans}_{\mathit{Stat}}$)

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: *Stat* → **label** :
  | **goto label**

  ... considered harmful (Dijkstra 1968)

  Necessary instructions are in the intermediate language.
  Needs to build symbol table of labels.
- Case/Switch: *Stat* → **case** *Exp* **of** [*Alts*]
  Alts → **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* → **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 $Trans_{Stat}$)

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)

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

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

$Trans_{Idx}(index, vtable, ftable) = \texttt{case}\ index\ \texttt{of}$

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

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:

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

...

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

- Write access in assignments:

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

...

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

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.

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

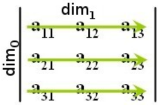
Multi-Dimensional Arrays

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.

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

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



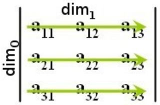
Multi-Dimensional Arrays

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.

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

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



- 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

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

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

Recursive index calculation, multiplies with dimension at each step.

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

$\textbf{id}[Exp]$	$(base, dims) = lookup(vtable, getname(\textbf{id}))$ $addr = newvar()$ $code = Trans_{Exp}(Exp, vtable, ftable, addr)$ $(code, addr, base, tail(dims))$
$Index[Exp]$	$(code_1, addr, base, dims) = Calc_{Idx}(Index, vtable, ftable)$ $d = head(dims)$ $t = newvar()$ $code_2 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_3 = code_1 \text{ @ } code_2 \text{ @ } [addr := addr * d, addr := addr + t]$ $(code_3, addr, base, tail(dims))$

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

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

Function $\mathit{Trans}_{\mathit{Decl}}$ `: (Decl, VTable) -> ([ICode], 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) -> ([ICode], 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]**	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\mathbf{id}), t_1)$ $([t_1 := HP,\ HP := HP + (4 * getvalue(\mathbf{num}))], vtable_1)$

. . . 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