Intermediate Representation (IR):

- An abstract machine language
- Expresses operations of target machine
- Not specific to any particular machine
- Independent of source language

IR code generation not necessary:

- Semantic analysis phase can generate real assembly code directly.
- Hinders portability and modularity.
Intermediate Representation

Suppose we wish to build compilers for $n$ source languages and $m$ target machines.

**Case 1: no IR**

- Need separate compiler for each source language/target machine combination.
- A total of $n \times m$ compilers necessary.
- Front-end becomes cluttered with machine specific details, back-end becomes cluttered with source language specific details.

**Case 2: IR present**

- Need just $n$ front-ends, $m$ back ends.
Intermediate Representation

FIGURE 7.1. Compilers for five languages and four target machines: (left) without an IR, (right) with an IR. From *Modern Compiler Implementation in ML*, Cambridge University Press, ©1998 Andrew W. Appel
IR properties

• Must be convenient for semantic analysis phase to produce.
• Must be convenient to translate into real assembly code for all desired target machines.
  – RISC processors execute operations that are rather simple.
    * Examples: load, store, add, shift, branch
    * IR should represent abstract load, abstract store, abstract add, etc.
  – CISC processors execute more complex operations.
    * Examples: multiply-add, add to/from memory
    * Simple operations in IR may be “clumped” together during instruction selection to form complex operations.
IR expression trees

The IR may be represented in many forms:

- Liberty, IMPACT, and Elcor compilers use *pseudo-assembly*.
- gcc and the class project use *expression trees*.
- Intel’s Electron, and HP’s production compiler use both.

**Expression trees:**

- exp: constructs that compute some value, possibly with side effects.
- stm: constructs that perform side effects and control flow.

```plaintext
signature TREE = sig
datatype exp = CONST of int
  | NAME of Temp.label
  | TEMP of Temp.temp
  | BINOP of binop * exp * exp
  | MEM of exp
  | CALL of exp * exp list
  | ESEQ of stm * exp
```
IR expression trees

TREE continued:

and stm = MOVE of exp * exp
   | EXP of exp
   | JUMP of exp * Temp.label list
   | CJUMP of relop * exp * exp *
      Temp.label * Temp.label
   | SEQ of stm * stm
   | LABEL of Temp.label

and binop = PLUS | MINUS | MUL | DIV | AND | OR |
            LSHIFT | RSHIFT | ARSHIFT | XOR

and relop = EQ | NE | LT | GT | LE | GE | ULT | ULE | UGT | UGE

end
Expressions

Expressions compute some value, possibly with side effects.

CONST ($i$) integer constant $i$

NAME ($n$) symbolic constant $n$ corresponding to assembly language label (abstract name for memory address)

TEMP ($t$) temporary $t$, or abstract/virtual register $t$

BINOP ($op$, $e_1$, $e_2$) $e_1$ $op$ $e_2$, $e_1$ evaluated before $e_2$

- integer arithmetic operators: PLUS, MINUS, MUL, DIV
- integer bit-wise operators: AND, OR, XOR
- integer logical shift operators: LSHIFT, RSHIFT
- integer arithmetic shift operator: ARSHIFT
Expressions

\[ \text{MEM}(e) \] contents of \text{wordSize} bytes of memory starting at address \( e \)

- \text{wordSize} is defined in \text{Frame} module.
- if \text{MEM} is used as left operand of \text{MOVE} statement \( \Rightarrow \) store
- if \text{MEM} is used as right operand of \text{MOVE} statement \( \Rightarrow \) load

\[ \text{CALL}(f, l) \] application of function \( f \) to argument list \( l \)

- subexpression \( f \) is evaluated first
- arguments in list \( l \) are evaluated left to right

\[ \text{ESEQ}(s, e) \] the statement \( s \) evaluated for side-effects, \( e \) evaluated next for result
Statements

Statements have side effects and perform control flow.

MOVE (TEMP (t), e) evaluate e and move result into temporary t.

MOVE (MEM (e1), e2) evaluate e1, yielding address a; evaluate e2, store result in wordSize bytes of memory stating at address a

EXP (e) evaluate expression e, discard result.

JUMP (e, labs) jump to address e

- e may be literal label (NAME (l)), or address calculated by expression
- labs specifies all locations that e can evaluate to (used for dataflow analysis)
- jump to literal label l: JUMP (NAME (l), [l])

CJUMP (op, e1, e2, t, f) evaluate e1, then e2; compare results using op; if true, jump to t, else jump to f

- EQ, NE: signed/unsigned integer equality and non-equality
- LT, GT, LE, GE: signed integer inequality
- ULT, UGT, ULE, UGE: unsigned integer inequality
Statements

SEQ($s_1$, $s_2$) statement $s_1$ followed by $s_2$

LABEL($l$) label definition - constant value of $l$ defined to be current machine code address

- similar to label definition in assembly language
- use NAME($l$) to specify jump target, calls, etc.

- The statements and expressions in TREE can specify function bodies.
- Function entry and exit sequences are machine specific and will be added later.
Translation of Abstract Syntax

- if Absyn.exp computes value ⇒ Tree.exp
- if Absyn.exp does not compute value ⇒ Tree.stm
- if Absyn.exp has boolean value ⇒ Tree.stm and Temp.labels

datatype exp = Ex of Tree.exp  
| Nx of Tree.stm  
| Cx of Temp.label * Temp.label -> Tree.stm

- Ex “expression” represented as a Tree.exp
- Nx “no result” represented as a Tree.stm
- Cx “conditional” represented as a function. Given a false-destination label and a true-destination label, it will produce a Tree.stm which evaluates some conditionals and jumps to one of the destinations.
Translation of Abstract Syntax - Conditionals

Conditional:

\[ x > y : \]
\[ \text{Cx}(fn \ (t, f) => \text{CJUMP}(GT, x, y, t, f)) \]

\[ a > b | c < d : \]
\[ \text{Cx}(fn \ (t, f) => \text{SEQ}(\text{CJUMP}(GT, a, b, t, z), \]
\[ \text{SEQ}(\text{LABEL} \ z, \text{CJUMP}(LT, c, d, t, f)))) \]

May need to convert conditional to value:

\[ a := x > y : \]

\[ \text{Cx} \text{ corresponding to } \text{“}x > y\text{”} \text{ must be converted into } \text{Tree.exp} \ e. \]
\[ \text{MOVE(TEMP(a), e)} \]

Need three conversion functions:

\[ \text{val unEx: exp -> Tree.exp} \]
\[ \text{val unNx: exp -> Tree.stm} \]
\[ \text{val unCx: exp -> (Temp.label * Temp.label -> Tree.stm)} \]
Translation of Abstract Syntax - Conditionals

The three conversion functions:

\[
\begin{align*}
\text{val } & \text{ unEx: exp } \to \text{ Tree.exp} \\
\text{val } & \text{ unNx: exp } \to \text{ Tree.stm} \\
\text{val } & \text{ unCx: exp } \to (\text{Temp.label } \times \text{Temp.label } \to \text{Tree.stm})
\end{align*}
\]

\[a := x > y:\]
\[
\text{MOVE(TEMP(a), unEx(Cx(t,f) => ...)}
\]

unEx makes a Tree.exp even though \(e\) was Cx.
Translation of Abstract Syntax

Implementation of function UnEx:

structure T = Tree

fun unEx(Ex(e)) = e
  | unEx(Nx(s)) = T.ESEQ(s, T.CONST(0))
  | unEx(Cx(genstm)) =
      let val r = Temp.newtemp()
          val t = Temp.newlabel()
          val f = Temp.newlabel()
      in T.ESEQ(seq[T.MOVE(T.TEMP(r), T.CONST(1)),
                     genstm(t, f),
                     T.LABEL(f),
                     T.MOVE(T.TEMP(r), T.CONST(0)),
                     T.LABEL(t)],
                     T.TEMP(r))
      end
Translation of Abstract Syntax

- Recall type and value environments $\mathit{tenv}$, $\mathit{venv}$.
- The function $\mathit{transVar}$ return a record $\{\mathit{exp}, \mathit{ty}\}$ of $\mathit{Translate.exp}$ and $\mathit{Types.ty}$.
- $\mathit{exp}$ is no longer a place-holder.
Simple Variables

- **Case 1:** variable $v$ declared in current procedure’s frame

  \[
  \text{InFrame}(k): \quad \text{MEM(BINOP(PLUS, TEMP(FP), CONST(k)))}
  \]

  $k$: offset in own frame

  FP is declared in FRAME module.

- **Case 2:** variable $v$ declared in temporary register

  \[
  \text{InReg}(t_{103}): \quad \text{TEMP}(t_{103})
  \]
Simple Variables

- **Case 3:** variable $v$ not declared in current procedure’s frame, need to generate IR code to follow static links

```
InFrame(k_n):
    MEM(BINOP(PLUS, CONST(k_n)),
        MEM(BINOP(PLUS, CONST(k_n-1),
                        ...
                        MEM(BINOP(PLUS, CONST(k_2),
                                    MEM(BINOP(PLUS, CONST(k_1), TEMP(FP))))))))
```

$k_1, k_2, \ldots, k_{n-1}$: static link offsets

$k_n$: offset of $v$ in own frame
Simple Variables

To construct simple variable IR tree, need:

- $l_f$: level of function $f$ in which $v$ used
- $l_g$: level of function $g$ in which $v$ declared
- MEM nodes added to tree with static link offsets $(k_1, \ldots, k_{n-1})$
- When $l_g$ reached, offset $k_n$ used.
Array Accesses

Given array variable $a$,

$$
&(a[0]) = a \\
&(a[1]) = a + w, \text{ where } w \text{ is the word-size of machine} \\
&(a[2]) = a + (2 \times w) \\
\ldots
$$

Let $e$ be the IR tree for $a$:

$$a[i]: \\
\quad \text{MEM(BINOP(PLUS, } e, \text{ BINOP(MUL, i, CONST(w))))}$$

Compiler must emit code to check whether $i$ is out of bounds.
Record Accesses

type rectype = {f1:int, f2:int, f3:int}

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>offset:</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

var a:rectype := rectype{f1=4, f2=5, f3=6}

Let e be IR tree for a:

\[ a.f3: \]
\[ \text{MEM(BINOP(PLUS, e, BINOP(MUL, CONST(3), CONST(w))))} \]

Compiler must emit code to check whether a is nil.
Conditional Statements

if \( e_1 \) then \( e_2 \) else \( e_3 \)

- Treat \( e_1 \) as \( \text{Cx} \) expression \( \Rightarrow \) apply \( \text{unCx} \).
- Treat \( e_2, e_3 \) as \( \text{Ex} \) expressions \( \Rightarrow \) apply \( \text{unEx} \).

\[
\begin{align*}
\text{Ex} & (\text{ESEQ} (\text{SEQ} (\text{unCx}(e_1)(t, f), \\
\quad \text{SEQ}(\text{LABEL}(t), \\
\quad \quad \text{SEQ}(\text{MOVE}(\text{TEMP}(r), \text{unEx}(e_2)), \\
\quad \quad \text{SEQ}(\text{JUMP}(\text{NAME}(\text{join})), \\
\quad \quad \text{SEQ}(\text{LABEL}(f), \\
\quad \quad \quad \text{SEQ}(\text{MOVE}(\text{TEMP}(r), \text{unEx}(e_3)), \\
\quad \quad \quad \quad \text{LABEL}(\text{join}))))))) \\
\text{TEMP}(r)))
\end{align*}
\]