8. Intermediate Code

- Intermediate code is closer to the target machine than the source language, and hence easier to generate code from.

- Unlike machine language, intermediate code is (more or less) machine independent. This makes it easier to retarget the compiler.

- It allows a variety of optimizations to be performed in a machine-independent way.

- Typically, intermediate code generation can be implemented via syntax-directed translation, and thus can be folded into parsing by augmenting the code for the parser.
Intermediate Languages: Design Issues

- The set of operators in the intermediate language must be rich enough to allow the source language operations to be implemented.

- A small set of operations in the intermediate language makes it easy to retarget the compiler to a new machine.

- Intermediate code operations that are closely tied to a particular machine or architecture may make it harder to port the compiler to other architectures.

- A small set of intermediate code operations may lead to long instruction sequences for some source language constructs. This may require more work during optimization.
High-Level Intermediate Representations

Examples: syntax trees, DAGs:

- *(abstract) syntax trees*: a compact form of a parse tree that represents the hierarchical structure of the program: nodes represent operators, the children of a node represent what it operates on.

- **DAGs**: similar to syntax trees, except that common subexpressions are represented by a single node.
Example:

\[
\text{if } (x > 0) \text{ then } x := 3 \times (y + 1) \text{ else } y := y + 1;
\]
8.1.1. Low-Level Intermediate Representations

Examples: Three Address Code

- This is a sequence of instructions of the form
  \[ x := y \ op \ z \]
  where \( x, y, \) and \( z \) are variable names, constants, or compiler generated variables ("temporaries").

- Only one operator is permitted on the RHS, so there are no "built-up" expressions. Instead, expressions are computed using temporaries. E.g. the source language construct
  \[ x := y + z*w \]
  might translate to
  \[ t1 := z * w \]
  \[ x := y + t1 \]
Different Kinds of Three-Address Statements

Assignment:
\[ x := y \ op \ z, \quad \text{op binary} \]
\[ x := \ op \ y, \quad \text{op unary} \]
\[ x := y \]

Jumps:
\[ \text{goto } L, \]
\[ \text{jumpt } t \ L, \]
\[ \text{jumpf } t \ L, \quad L \text{ a label} \]
\[ \text{if } x \ relop y \ \text{goto } L, \quad L \text{ a label} \]
Procedure Call/Return:

- param $x$, $x$ an actual parameter
- call $p$, $n$, $n = $ no. of params to $p$
- enter initialization (if any)
- exit cleanup actions (if any)
- return
- return $x$
- retrieve $x$
- save returned value in $x$
Indexed Assignment:
\[ x := y[i] \]
\[ x[i] := y \]

Address and Pointer Assignments:
\[ x := &y \]
\[ x := *y \]
\[ x := y \]

Miscellaneous:
label \( L \)
8.1.2. Implementing Three-Address Instructions

Each instruction is implemented as a structure called a *quadruple*:

- contains (upto) 4 fields: operation, (upto) two operands, and destination;

- for operands: use a bit to indicate whether it’s a constant or a pointer into the symbol table.

\[
x := y + z \quad \text{if } t1 \geq t2 \text{ goto } L
\]

\[
\begin{array}{|c|c|}
\hline
\text{Op} & \text{PLUS} \\
\hline
\text{Src1} & y \\
\hline
\text{Src2} & z \\
\hline
\text{Dest} & x \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{Op} & \text{JMP\_GE} \\
\hline
\text{Src1} & t1 \\
\hline
\text{Src2} & t2 \\
\hline
\text{Dest} & \text{instruction labelled L} \\
\hline
\end{array}
\]
8.2. Intermediate Code Generation

- Source language constructs are decomposed to simpler constructs at the intermediate code level.

- When generating code to evaluate expressions, temporary names must be made up for internal nodes in the syntax tree for the expression.

Example:

Source : if $x + 2 > 3 \cdot (y - 1) + 4$ then $z := 0$

Intermediate Code:

```
t1 := x+2
 t2 := y-1
 t3 := 3*t2
 t4 := t3+4
if t1 <= t4 goto L
 z := 0
label L
```
Intermediate Code Generation

Syntax-Directed Translation:

- Intermediate code represented as a list of instructions. Instruction sequences are concatenated using the operator ||.
  (In practice, we might choose to write the intermediate code instructions out into a file.)

- Attributes for Expressions E:
  - E.place: denotes the location that holds the value of E.
  - E.code: denotes the instruction sequence that evaluates E.
• **Attributes for Statements** $S$:
  
  – $S.begin$ : denotes the first instruction in the code for $S$.
  
  – $S.after$ : denotes the first instruction *after* the code for $S$.
  
  – $S.code$ : denotes the instruction sequence that represents $S$. 
• **Auxiliary Functions**:
  
  - `newtemp()` : returns a new temporary each time it is called.
  
    * returns a pointer to the ST entry of a temp.
  
    * may take a parameter specifying the type of the temp (useful if reusing temps).
  
  - `newlabel()` : returns a new label name each time it is called.

• **Notation** : we write

  \[ \text{gen}(x \ := \ y \ + \ z) \]

  to represent the instruction \( x := y + z \).
## Intermediate Code Generation: Simple Expressions

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULE</th>
</tr>
</thead>
</table>
| $E \rightarrow \text{id}$ | $E\.place := \text{id}.place$;  
$E\.code := "$"$; |
| $E \rightarrow (\ E_1 \ )$ | $E\.place := E_1\.place$;  
$E\.code := E_1\.code$; |
| $E \rightarrow E_1 + E_2$ | $E\.place := \text{newtemp}()$;  
$E\.code := E_1\.code ||$  
$E_2\.code ||$  
$\text{gen}(E\.place \:\Rightarrow E_1\.place \:\Rightarrow E_2\.place)$ |
| $E \rightarrow -E_1$ | $E\.place := \text{newtemp}()$;  
$E\.code := E_1\.code ||$  
$\text{gen}(E\.place \:\Rightarrow \:\Rightarrow -$  
$E_1\.place)$ |
8.3.2. Accessing Array Elements I

- Array elements can be accessed quickly if the elements are stored in a block of consecutive locations.

- Assume:
  - we want the $i^{th}$ element of an array $A$ whose subscript ranges from $lo$ to $hi$;
  - the address of the first element of the array is $base$.

- We can avoid address computations in the intermediate code if we have indexed “addressing modes” at the intermediate code level.

In this case, $A[i]$ is the $(i - lo)^{th}$ element of the array located at $base$ (starting at element 0). So a reference $A[i]$ translates to the code

\[
t1 := i - lo \\
t2 := A[t1]
\]
8.3.2. Accessing Array Elements II

- Address computations can't be avoided in general, because of pointer and record types.

- The simple approach using indexed expressions may recompute base addresses repeatedly, leading to inefficient code.

- Assume:
  - we want the $i^{th}$ element of an array $A$ whose subscript ranges from $lo$ to $hi$;
  - the address of the first element of the array is $base$;
  - each element of $A$ has width $w$. 
Then, the address of $x[i]$ is

$$base + (i - lo) \times w$$

$$= (base - lo \times w) + i \times w$$

$$= C_A + i \times w$$

where $C_A$ depends on the array $A$ and is known at compile time.

*Note*: $C_A$ is a memory address if $A$ is a global, and is a stack displacement if $A$ is a local.

- The idea extends to multidimensional arrays in the obvious way: need to know whether the elements are stored in row-major or column-major order.
8.4. Logical Expressions

\[ BExp \rightarrow E_1 \text{ relop } E_2 \]

8.4.3. Naive but Simple Approach :

Intermediate Code (TRUE == 1, FALSE == 0) :

\[
\begin{align*}
\text{t1} & \leftarrow \text{value of } E_1 \\
\text{t2} & \leftarrow \text{value of } E_2 \\
\text{t3} & := \text{TRUE} \\
\text{if } \text{t1 relop t2} \text{ goto L} \\
\text{t3} & := \text{FALSE} \\
\text{label L}
\end{align*}
\]

\textbf{Disadvantage} : Lots of (usually unnecessary) memory traffic.
8.4.1. Code Generation for Conditionals

**Production**: \( S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2 \)

**Semantic Rule**: 

\[
\begin{align*}
S.\text{begin} & := \text{newlabel}(); \\
S.\text{after} & := \text{newlabel}(); \\
S.\text{code} & := \text{gen('label' } S.\text{begin}) \ || \\
& \quad E.\text{code} \ || \\
& \quad \text{gen('if 'E.\text{place}' = '0' 'goto' } S_2.\text{begin}) \ || \\
& \quad S_1.\text{code} \ || \\
& \quad \text{gen('goto' } S.\text{after}) \ || \\
& \quad S_2.\text{code} \ || \\
& \quad \text{gen('label' } S.\text{after}) \\
\end{align*}
\]
8.4.1. Code Generation for Loops

Production : $S \rightarrow \text{while } E \text{ do } S_1$

Structure of Generated Code :

$L_1 :$

\[
\begin{align*}
\text{eval} &\text{uate } E \\
\text{if } (E == \text{FALSE}) &\text{ goto } L_2 \\
\text{Code for } S_1 \\
\text{goto } &L_1 \\
L_2 : &\ldots
\end{align*}
\]

Semantic Rule :

\[
\{ \text{S.begin := newlabel();} \\
\text{S.after := newlabel();} \\
\text{S.code := gen('label' S.begin) }|| \\
\text{E.code }|| \\
\text{gen('if' E.place' = "0" goto' S.after) }|| \\
\text{S_1.code }|| \\
\text{gen('goto' S.begin) }|| \\
\text{gen('label' S.after)} \}
\]
Intermediate Code Generation: Assignment

- Grammar productions:
  \[ S \rightarrow Lhs \ := \ Rhs \]

- Semantic Rule:
  \[
  \begin{cases}
  S.code := Lhs.code \parallel \\
  Rhs.code \parallel \\
  gen(Lhs.place \ := \ ' := ' Rhs.place) \\
  \end{cases}
  \]
Relational Expressions: Better Approach

- Often, relational expressions occur in the context of boolean conditions of control statements.

- Instead of creating temporaries which are set to true or false, based upon the outcome of evaluating a boolean condition, generate direct branches to true and false targets.

- Short circuit evaluation of boolean expressions can also be handled effectively by this approach.
Relational Expressions: Example

\[ E = a < b \text{ or } c < d \text{ and } e < f \]

100 : if \( a < b \) goto __
101 : goto 102
102 : if \( c < d \) goto 104
103 : goto __
104 : if \( e < f \) goto __
105 : goto __

\[ E.\text{truelist} = \{100, 104\} \]
\[ E.\text{falselist} = \{103, 105\} \]
\[ E \rightarrow E_1 \text{ or } M \ E_2 \]
\[
\begin{align*}
&\{ \\
&\quad \text{backpatch}(E_1.\text{falselist}, M.\text{quad}) ; \\
&\quad E.\text{truelist} = \text{merge}(E_1.\text{truelist}, E_2.\text{truelist}) ; \\
&\quad E.\text{falselist} = E_2.\text{falselist} ; \\
&\} \\
M \rightarrow \epsilon \\
&\{ M.\text{quad} = \text{nextquad} \}
\end{align*}
\]
\[ E \rightarrow E_1 \text{ and } M \ E_2 \]
\[
\begin{align*}
&\{ \\
&\quad \text{backpatch}(E_1.\text{truelist}, M.\text{quad}) ; \\
&\quad E.\text{truelist} = E_2.\text{truelist} ; \\
&\quad E.\text{falselist} = \text{merge}(E_1.\text{falselist}, E_2.\text{falselist}) ; \\
&\} \\
\]
Relational Expressions: cont’d.

\[ E \rightarrow \text{not } E_1 \]
\[
\{ \\
E\.\text{truelist} = E_1\.\text{falselist}; \\
E\.\text{falselist} = E_1\.\text{truelist}; \\
\}
\]

\[ E \rightarrow (E_1) \]
\[
\{ \\
E\.\text{truelist} = E_1\.\text{truelist}; \\
E\.\text{falselist} = E_1\.\text{falselist}; \\
\}
\]

\[ E \rightarrow id_1 \text{ relop } id_2 \]
\[
\{ \\
E\.\text{truelist} = \text{makelist}(\text{nextquad}); \\
E\.\text{falselist} = \text{makelist}(\text{nextquad} + 1); \\
\text{generate(} if \ id_1\.\text{addr relop } id_2\.\text{addr goto}\_\_\_\) \\
\text{generate(goto}\_\_\_\) \\
\}
\]
E → true
   
   \{ 
   E.truelist = makelist ( nextquad );
   \hspace{1em} generate ( goto___ )
   \}

E → false

\{ 
E.falselist = makelist ( nextquad );
\hspace{1em} generate ( goto___ )
\}
Code Generation for Loops and Conditionals

- Straightforward approach can introduce branch instructions whose targets are unconditional jumps.

```
while a<b do
    if x<y then  S  endif
endwhile
```

100:  if a < b go to 102
101:  go to 106
102:  if x < y go to 104
103:  go to 105  **100**
104:  S.code
105:  go to 100
106:

- We can avoid this by maintaining an additional attribute for statements called the *nextlist*. This attribute tracks branches in the statements whose target should be set to code that follows them in the execution sequence.
Loops and Conditionals: cont’d.

\[ S \rightarrow if\ E\ then\ M_1\ S_1\ N\ else\ M_2\ S_2 \]
\[
\begin{array}{l}
\{ \\
backpatch(E.truelist,M_1.quad); \\
backpatch(E.falselist,M_2.quad); \\
S.nextlist = merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist))
\}
\end{array}
\]

\[ N \rightarrow \epsilon \]
\[
\begin{array}{l}
\{ \\
N.nextlist = makelist(nextquad); \\
generate(goto\_)
\}
\end{array}
\]

\[ M \rightarrow \epsilon \]
\[
\begin{array}{l}
\{ M.quad = nextquad \}
\end{array}
\]

\[ S \rightarrow if\ E\ then\ M\ S_1 \]
\[
\begin{array}{l}
\{ \\
backpatch(E.truelist,M.quad); \\
S.nextlist = merge(E.falselist,S_1.nextlist)
\}
\end{array}
\]
Loops and Conditionals: cont’d.

\[ S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1 \{
\text{backpatch}(S_1.\text{nextlist}, M_1.\text{quad});
\text{backpatch}(E.\text{truelist}, M_2.\text{quad});
S.\text{nextlist} = E.\text{falselist};
generate(gotoM_1.\text{quad});
\}
S \rightarrow \text{begin } L \text{ end }
\{S.\text{nextlist} = L.\text{nextlist}\}
S \rightarrow A
\{S.\text{nextlist} = \text{nil}\}
L \rightarrow L_1 ; M \ S
\{\text{backpatch}(L_1.\text{nextlist}, M.\text{quad});
L.\text{nextlist} = S.\text{nextlist};
\}
L \rightarrow S
\{L.\text{nextlist} = S.\text{nextlist}\}

Optimization takes place here
while a<b do
    while x<y do
        S
    endwhile
endwhile

100: if a < b go to 102
101: go to 107
102: if x < y go to 104
103: go to 106 100
104: S.code
105: go to 102
106: go to 100
107: ....
Intermediate Code Generation: case Statements

Implementation issue: Need to generate code so that we can (efficiently) choose one of a set of different alternatives, depending on the value of an expression.

Implementation choices:
1. linear search
2. binary search
3. jump table
Implementation considerations:

1. **Execution Cost**: linear or binary search may be cheaper if the no. of cases is small.
   
   For a large no. of cases, a jump table may be cheaper.

2. **Space cost**: a jump table may take too much space if the case values are not clustered closely together, e.g.:

   ```java
   switch (x) {
       case 1 : ...
       case 1000 : ...
       case 1000000 : ...
   }
   ```
8.5. Code Generation for Function Calls

**Calling Sequence: Caller:**

- Evaluate actual parameters; place actuals where the callee wants them.
  
  *Instruction*: `param t`

- Save machine state (current stack and/or frame pointers, return address) and transfer control to callee.
  
  *Instruction*: `call p, n` \((n = \text{no. of actuals})\)

**Calling Sequence: Callee:**

- Save registers (if necessary); update stack and frame pointers to accommodate \(m\) bytes of local storage.
  
  *Instruction*: `enter m`
Return Sequence: Callee:

- Place return value $x$ (if any) where the caller wants it; adjust stack/frame pointers (maybe); jump to return address.

  Instruction: return $x$ or return.

Return Sequence: Caller:

- Save the value returned by the callee (if any) into $x$.

  Instruction: retrieve $x$. 
Intermediate Code for Function Calls: An Example

**Source Code:**

\[ x = f(0, y+1) - 1; \]

**Intermediate Code Generated:**

\[ t1 := y+1 \]
\[ \text{param } t1 \quad /* \text{ arg } 2 */ \]
\[ \text{param } 0 \quad /* \text{ arg } 1 */ \]
\[ \text{call } f, 2 \]
\[ \text{retrieve } t2 \quad /* t2 := f(0, t1) */ \]
\[ t3 := t2 - 1 \]
\[ x := t3 \]

Suppose function \( f \) needs 24 bytes of space for its locals and temporaries. Its code has the form

\[ \text{enter } 24 \]
\[ \ldots \]
\[ \text{return } t17 \]
\[ /* \text{ suppose return value is in } t17 */ \]
Code Generation for Functions: Storage Allocation

**Problem**: The first instruction in a function is

```
enter n  /* n = space for locals, temps */
```

but $n$ is not known until the whole function has been processed.

**Solution 1**: generate final code into a list, “back-patch” the appropriate instructions after processing the function body.

**Advantage**: Can also do machine-dependent optimizations (e.g., instruction scheduling).

**Disadvantage**: slower, requires more memory.
Solution 2: Generate code of the form

code
for function
foo

\[
\begin{align*}
&\text{L1: code for enter } n \\
&\text{goto L2}\\
&\text{L2: code for body of } \text{foo}\\
&\text{goto L1}
\end{align*}
\]
Reusing Temporaries

Storage requirements can be reduced considerably if we reuse temporaries:

- Maintain a free list of temporaries:
  - When a temporary is no longer necessary, it is returned to the free list.
  - The function `newtemp()` is modified to first search the free list, and to allocate a new temporary only if there is nothing in the free list.

- To handle objects of different sizes, we can maintain a free list for each type (or size).