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.
8.1.1. Low-Level Intermediate Representations

**Examples**: Three Address Code

- This is a sequence of instructions of the form
  \[ x := y \text{ op } z \]
  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
  \[ x := y + z*\text{w} \]
  might translate to
  \[ t1 := z * \text{w} \\
  x := y + t1 \]
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**:

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

param \( x \), \( x \) an actual parameter

call \( p, n \), \( n = \) no. of params to \( p \)

enter

exit initialization (if any)

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.

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

```
<table>
<thead>
<tr>
<th>Op</th>
<th>PLUS</th>
<th>t1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td></td>
<td>t2</td>
</tr>
<tr>
<td>Src2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dest</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Op</th>
<th>JMP_GE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>t1</td>
</tr>
<tr>
<td>Src2</td>
<td>t2</td>
</tr>
<tr>
<td>Dest</td>
<td></td>
</tr>
</tbody>
</table>
```

*instruction labelled L*
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 \( x + 2 > 3*(y - 1) + 4 \) then \( z := 0; \)

**Intermediate Code:**

\[
\begin{align*}
  t1 & := x+2 \\
  t2 & := y-1 \\
  t3 & := 3*t2 \\
  t4 & := t3+4 \\
  \text{if } t1 & <= t4 \text{ goto L} \\
  z & := 0 \\
  \text{label L}
\end{align*}
\]
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>
| $E \rightarrow \text{id}$ | $E\.place := \text{id\.place};$  
|                     | $E\.code := "\$ \$;$  
| $E \rightarrow (E_1)$ | $E\.place := E_1\.place;$  
|                     | $E\.code := E_1\.code;$  
| $E \rightarrow E_1 + E_2$ | $E\.place := \text{newtemp()};$  
|                     | $E\.code := E_1\.code \parallel$  
|                     | $E_2\.code \parallel$  
|                     | $\text{gen}(E\.place \text{'}:=\text{'}$  
|                     | $E_1\.place \text{'}+\text{'}$  
|                     | $E_2\.place)$  
| $E \rightarrow -E_1$ | $E\.place := \text{newtemp()};$  
|                     | $E\.code := E_1\.code \parallel$  
|                     | $\text{gen}(E\.place \text{'}:=\text{'}$  
|                     | $\text{'}-\text{'} E_1\.place)$  

CSE Department
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 `struct` types.

- The size of array element can be greater than 1 say $w$.

- 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

\[
\text{base} + (i - \text{lo}) \times w \\
= (\text{base} - \text{lo} \times w) + i \times w \\
= C_A + i \times w
\]

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 \((\text{TRUE} == 1, \text{FALSE} == 0)\):

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

\underline{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:

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

Structure of Generated Code

L1: Code for E
- if \( E == \text{FALSE} \) goto L3
- Code for S1
- goto L2
L3: Code for S2
L2: ...
8.4.1. Code Generation for Loops

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

Structure of Generated Code:

$L_1$:

- Code for $E$
- $\text{if } (E == \text{FALSE}) \text{ goto } L_2$
- Code for $S_1$
- $\text{goto } L_1$

$L_2: \ldots$

Semantic Rule:

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

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

- Semantic Rule:
  \[
  \{ \begin{align*}
  S.code &:= Lhs.code || Rhs.code || \\
  gen(Lhs.place ':=' Rhs.place)
  \end{align*} \}
  \]
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 : \text{ if } a < b \text{ goto } __
101 : \text{ goto 102}
102 : \text{ if } c < d \text{ goto 104}
103 : \text{ goto } __
104 : \text{ if } e < f \text{ goto } __
105 : \text{ goto } __

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

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

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

\[ E \rightarrow id_1 \text{ relop id}_2 \]
\[
\{ \\
E.\text{truelist} = \text{makelist(nextquad)}; \\
E.\text{falselist} = \text{makelist(nextquad + 1)}; \\
generate(\text{if } id_1.\text{addr relop id}_2.\text{addr goto}__) \\
generate(\text{goto}) \\
\} \]
E → \textit{true}
\{ 
    E.true\texttt{list} = \texttt{makelist} ( \texttt{nextquad} );
    \texttt{generate} ( \texttt{goto\_} )
\}\n
E → \textit{false}
\{ 
    E.false\texttt{list} = \texttt{makelist} ( \texttt{nextquad} );
    \texttt{generate} ( \texttt{goto\_} )
\}\n
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.

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

\[ N \rightarrow \epsilon \]
\[
\{
N.\text{nextlist} = \text{makelist}(\text{nextquad});
generate(\text{goto }_\text{ })
\}
\]

\[ M \rightarrow \epsilon \]
\[
\{M.\text{quad} = \text{nextquad}\}
\]

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

\[ S \rightarrow \text{while} \ M_1 \ E \ \text{do} \ M_2 \ S_1 \]
\[
\begin{align*}
&\{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{quad}); \\
&\quad \text{backpatch}(E.\text{truelist}, M_2.\text{quad}); \\
&\quad S.\text{nextlist} = E.\text{falselist}; \\
&\quad \text{generate}(\text{goto} M_1.\text{quad}); \\
&\} \\
\end{align*}
\]

\[ S \rightarrow \text{begin} \ L \ \text{end} \]
\[
\{ S.\text{nextlist} = L.\text{nextlist} \}
\]

\[ S \rightarrow A \]
\[
\{ S.\text{nextlist} = \text{nil} \}
\]

\[ L \rightarrow L_1 \ ; \ M \ S \]
\[
\begin{align*}
&\{ \text{backpatch}(L_1.\text{nextlist}, M.\text{quad}); \\
&\quad L.\text{nextlist} = S.\text{nextlist}; \\
&\} \\
\end{align*}
\]

\[ L \rightarrow S \]
\[
\{ L.\text{nextlist} = S.\text{nextlist} \}
\]

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.:

   ```
   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:

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

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

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

Problem: The first instruction in a function is

```c
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

goto L1

L2:

code for
body of foo

L1:

code for enter n
goto L2
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).