Intermediate code generation

Intermediate codes are machine independent codes, but they are close to machine code instructions.

Syntax tree, Postfix Notation, these address codes can be used as intermediate language.

**Three-address code**

It is a sequence of statements of the general form \( X := Y \ op \ Z \)

\[ A = -B \times (C+D) \]

Three-address code is as follows:

\[ \begin{align*}
T_1 &= -B \\
T_2 &= C + D \\
T_3 &= T_1 \times T_2 \\
A &= T_3
\end{align*} \]

<table>
<thead>
<tr>
<th>Quadruple</th>
<th>OPERATOR</th>
<th>OPERAND1</th>
<th>OPERAND2</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>-</td>
<td>B</td>
<td></td>
<td>( T_1 )</td>
</tr>
<tr>
<td>(2)</td>
<td>+</td>
<td>C</td>
<td>D</td>
<td>( T_2 )</td>
</tr>
<tr>
<td>(3)</td>
<td>*</td>
<td>( T_1 )</td>
<td>( T_2 )</td>
<td>( T_3 )</td>
</tr>
<tr>
<td>(4)</td>
<td>=</td>
<td>( T_3 )</td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

**Trifles**

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>OPERAND1</th>
<th>OPERAND2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>-</td>
<td>B</td>
</tr>
<tr>
<td>(2)</td>
<td>+</td>
<td>C</td>
</tr>
<tr>
<td>(3)</td>
<td>*</td>
<td>( (1) )</td>
</tr>
<tr>
<td>(4)</td>
<td>=</td>
<td>A</td>
</tr>
</tbody>
</table>

**Student Trifles**

<table>
<thead>
<tr>
<th>STATEMENT</th>
<th>OPERATOR</th>
<th>OPERAND1</th>
<th>OPERAND2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(56)</td>
<td>(56)</td>
<td>-</td>
<td>B</td>
</tr>
<tr>
<td>(57)</td>
<td>(57)</td>
<td>+</td>
<td>C</td>
</tr>
<tr>
<td>(58)</td>
<td>(58)</td>
<td>*</td>
<td>( (56) )</td>
</tr>
<tr>
<td>(59)</td>
<td>(59)</td>
<td>=</td>
<td>A</td>
</tr>
</tbody>
</table>
Declarations -
In the declarative statements the data items along with their
data types are declared.

Computing the type and relative addresses of declared names -

<table>
<thead>
<tr>
<th>$S \rightarrow D$</th>
<th>${ \text{offset} := 0 }$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \rightarrow \text{id} : T$</td>
<td>${ \text{enter_tab(id.name, T.type, offset); }$ \text{ offset } := \text{offset} + \text{Width}}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{integer}$</td>
<td>${ \text{T.type} := \text{integer}; \text{ T.width} := 4 }$</td>
</tr>
<tr>
<td>$T \rightarrow \text{real}$</td>
<td>${ \text{T.type} := \text{real}; \text{ T.width} := 8 }$</td>
</tr>
<tr>
<td>$T \rightarrow \text{array}[\text{num}]\text{of}T$</td>
<td>${ \text{T.type} := \text{array( numval, T.type)}; \text{ T.width} := \text{numval } \times \text{T.width} }$</td>
</tr>
<tr>
<td>$T \rightarrow *T$</td>
<td>${ \text{T.type} := \text{pointer(T.type)}; \text{T.width} := 4 }$</td>
</tr>
</tbody>
</table>

Assignment Statements -
It mainly deals with the expressions. The expressions can be of
type integer, real, array and record. In this section we will

Sematic directed translation scheme for generating these address code -

<table>
<thead>
<tr>
<th>PRODUCTION RULE</th>
<th>SEMANTIC ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} := E$</td>
<td>${ p := \text{lookup(id.name)}; \text{ If } p \neq \text{null then}$ $\text{emit}(p \leftarrow E.p.place) \text{ else error } }$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>${ E.p.place := \text{new temp}; \text{emit}(E.p.place := \text{E}_1.p.place \text{ + E}_2.p.place) }$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 \times E_2$</td>
<td>${ E.p.place := \text{new temp}; \text{emit}(E.p.place := \text{E}_1.p.place \times E_2.p.place) }$</td>
</tr>
<tr>
<td>$E \rightarrow -E_1$</td>
<td>${ E.p.place := \text{new temp}; \text{emit}(E.p.place := \text{uminus E}_1.p.place) }$</td>
</tr>
<tr>
<td>$E \rightarrow (E)$</td>
<td>${ E.p.place := E_p.p.place }$</td>
</tr>
<tr>
<td>$E + \text{id}$</td>
<td>${ p := \text{lookup(id.name)}; \text{ If } p \neq \text{null then}$ $\text{E.p.place := p } \text{ else error } }$</td>
</tr>
</tbody>
</table>

Boolean Expressions - Type Conversion -
Type conversion take place when two different types are
used in a single expression.

Eq - Generating these address code for an expression: $n := a + b \times c + d$ -

\[
\begin{align*}
  t_1 &= b \text{ int } c \\
  t_2 &= t_1 \text{ int } d \\
  t_3 &= \text{ int } \text{real } t_2 \\
  t_4 &= a \text{ real } t_3 \\
  n &= t_4
\end{align*}
\]
- Addressing array elements -
  Row-major order, \( A[i][j] = base + [(i - low) \times n2 + (j - low)] \times w \)
  Column-major order, \( A[i][j] = base + [i - low] \)

- Addressing array elements -
  Column-major order, \( A[i][j] = B + W \times [(i - low) + n2\times(j - low)] \)
  Row-major order, \( A[i][j] = B + W \times [n2\times(i - low) + (j - low)] \)

Translation scheme for addressing array elements -
1. \( S \rightarrow L : E \quad \text{if } L\.offset = \text{null then} \quad \text{// } L \text{ is simple } id \) +
   \[
   \text{emit } (L\.place : = 'E\.place');
   \]
   \[
   \text{else}
   \]
   \[
   \text{emit } (L\.offset' L\.place ['L\.offset'] : = 'E\.place');
   \]
2. \( E \rightarrow E_1 + E_2 \quad \text{if } E\.place : = \text{new temp}; \)
   \[
   \text{emit } (E\.place : = 'E_1\.place' + 'E_2\.place');
   \]
3. \( E \rightarrow (E_1) \quad \text{if } E\.place : = E_1\.place; \)
4. \( E \rightarrow L \quad \text{if } L\.offset = \text{null then} \quad \text{// } L \text{ is simple } id \) +
   \[
   \text{E\.place : = L\.place;}
   \]
   \[
   \text{else begin}
   \]
   \[
   \text{E\.place : = new temp;}
   \]
   \[
   \text{emit } (E\.place : = L\.place ['L\.offset']);
   \]
   \[
   \text{end;}
   \]
5. \( L \rightarrow E List \) +
   \[
   \text{L\.place : = new temp; L\.offset : = new temp;}
   \]
   \[
   \text{emit } (L\.place : = 'C EList\.array');
   \]
   \[
   \text{emit } (L\.offset : = 'EList\.place' \times 'width(EList\.array)');
   \]
6. \( L \rightarrow id \) +
   \[
   \text{L\.place : = id\.place; L\.offset : = null;}
   \]
7. \( E List \rightarrow E List, E \) +
   \[
   \text{t : = new temp; m : = EList\.ndim + 1;}
   \]
   \[
   \text{emit } (t : = 'EList\.place' \times 'limit(EList\.array, m)');
   \]
   \[
   \text{emit } (t' : = t + 'E\.place'); EList\.array = EList\.array;
   \]
   \[
   EList\.place : = t; EList\.ndim : = m;
   \]
8. \( E List \rightarrow id [E \) +
   \[
   \text{EList\.array : = id\.place; EList\.place : = E\.place; EList\.ndim = 1;}
   \]
Boolean Expressions:

It can be generated by the following grammar:

\[ E \rightarrow E \lor E \mid E \land E \mid \text{not } E \mid (E) \mid \text{id relop id} \mid \text{true} \mid \text{false} \]

relop \rightarrow \text{comparision operators} (\leq, \leq, =, \neq, >, \geq)

---

Methods of Translating Boolean Expressions:

1. The first method is to encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression.

Translation scheme using a numerical representation for boolean-

- \( E \rightarrow E_1 \lor E_2 \) \[ ?E\_place := \text{new temp} \; \text{emit}(\text{E\_place} := \text{E\_1.place} \lor \text{E\_2.place}) \]
- \( E \rightarrow E_1 \land E_2 \) \[ ?E\_place := \text{new temp} \; \text{emit}(\text{E\_place} := \text{E\_place and } \text{E\_place}) \]
- \( E \rightarrow \text{not } E \_1 \) \[ ?E\_place := \text{new temp} \; \text{emit}(\text{E\_place} := \text{not } \text{E\_place}) \]
- \( E \rightarrow (E) \) \[ \text{E\_place := E\_place} \]
- \( E \rightarrow \text{id relop id} \) \[ ?E\_place := \text{new temp} \; \text{emit}(\text{ifid\_place relop op id\_2 place} \text{go to nextstat} + 3) ; \text{emit}(\text{E\_place} := \text{'0'}) ; \text{emit}(\text{gob to nextstat + 2}) ; \text{emit}(\text{E\_place} := \text{'1'}) \]
- \( E \rightarrow \text{true} \) \[ \text{E\_place := new temp} \; \text{emit}(\text{E\_place} := \text{'1'}) \]
- \( E \rightarrow \text{false} \) \[ \text{E\_place := new temp} \; \text{emit}(\text{E\_place} := \text{'0'}) \]

\( a = b \Rightarrow c \)

100: if \( a < b \) go to 1003 (nextstat + 3)
101: \( t := 0 \)
102: goto 104 (nextstat + 2)
103: \( t := 1 \)
104:

(3) Short circuit code — We can also translate a boolean expression into three-address code without generating code for any of the boolean operations and without having the code recursively evaluate the entire expression.

(2) The second method of implementing boolean expressions is by flow of control that is, representing the value of a boolean expression by a position reached in a program.
\[ S \rightarrow \text{if } E \text{ then } S_1 \mid \text{if } E \text{ then } S_1 \text{ and } S_2 \mid \text{while } E \text{ do } S_1 \]

\[ \text{Code for if-then, if-then-else, and while-do statements is given below:} \]

\[ S \rightarrow \text{if } E \text{ then } S_1 \quad E \text{.true} \rightarrow \text{E.code} \quad E \text{.false} \rightarrow \text{E.code} \]
\[ S \rightarrow \text{if } E \text{ then } S_1 \text{ and } S_2 \quad S \text{.begin} \rightarrow \text{E.code} \quad E \text{.true} \rightarrow \text{S.code} \quad E \text{.false} \rightarrow \text{S.code} \]
\[ S \rightarrow \text{while } E \text{ do } S_1 \quad \text{E.false} \rightarrow \text{S.code} \quad \text{E.true} \rightarrow \text{S.code} \]

\[ \text{IF-THEN} \quad \text{IF-THEN-ELSE} \quad \text{WHILE-DO} \]

\[ \text{SDD for flow-of-control statements is given below:} \]

\[ S \rightarrow \text{if } E \text{ then } S_1 \quad E \text{.true} \rightarrow \text{newlabel} \quad E \text{.false} \rightarrow \text{newlabel} \quad S \text{.begin} \rightarrow \text{E.code} \quad S \text{.next} \rightarrow \text{E.code} \quad S \text{.begin} \rightarrow \text{E.code} \]
\[ S \rightarrow \text{if } E \text{ then } S_1 \text{ and } S_2 \quad S \text{.begin} \rightarrow \text{E.code} \quad S \text{.next} \rightarrow \text{E.code} \quad S \text{.begin} \rightarrow \text{E.code} \]

\[ \text{Control-flow Translation of boolean expressions:} \]

\[ E \rightarrow E_1 \text{ or } E_2 \quad E \text{.true} \rightarrow E \text{.true} \quad E \text{.false} \rightarrow \text{newlabel} \quad E \text{.true} \rightarrow E \text{.true} \quad E \text{.false} \rightarrow E \text{.false} \]
\[ E \text{.code} \rightarrow E \text{.code} || \text{gen ('if' \ E \text{.true}' || E \text{.false}' || E \text{.code}') || E \text{.code}} \]
\[ E \rightarrow E_1 \text{ and } E_2 \quad E \text{.true} \rightarrow \text{newlabel} \quad E \text{.false} \rightarrow E \text{.false} \quad E \text{.true} \rightarrow E \text{.true} \quad E \text{.false} \rightarrow \text{newlabel} \]
\[ E \text{.code} \rightarrow E \text{.code} || \text{gen ('if' \ E \text{.true}' || E \text{.false}' || E \text{.code}') || E \text{.code}} \]
\[ E \rightarrow \text{not } E_1 \quad E \text{.true} \rightarrow \text{E.false} \quad E \text{.false} \rightarrow \text{E.true} \quad E \text{.code} \rightarrow E \text{.code} \]
\[ E \rightarrow (E_1) \quad E \text{.true} \rightarrow E \text{.true} \quad E \text{.false} \rightarrow E \text{.false} \quad E \text{.code} \rightarrow E \text{.code} \]
\[ E \rightarrow \text{id}_{1} \text{ relop id}_{2} \quad E \text{.code} \rightarrow \text{gen('if' \ id}_{1} \text{.place \ relop \ op \ id}_{2} \text{.place \ 'goto' \ E \text{.true}') ||} \]
\[ \text{gen ('goto' \ E \text{.false})} \]
\[ E \rightarrow \text{true} \quad E \text{.code} \rightarrow \text{gen('goto' \ E \text{.true})} \]
\[ E \rightarrow \text{false} \quad E \text{.code} \rightarrow \text{gen('goto' \ E \text{.false})} \]

\[ \text{SDD \ TO \ PRODUCE \ THREE-ADDRESS \ CODE \ FOR \ BOOLEANS} \]
Case Statement:

Switch Statement Syntax:

\[
\text{switch expression} \\
\text{begin} \\
\text{case value : statement} \\
\text{case value : statement} \\
\ldots \\
\text{case value : statement} \\
\text{default : statement} \\
\text{end,}
\]

Three-address code:

\[
\begin{align*}
\text{case } & V_1 \text{l}_1 \quad \text{case } V_2 \text{l}_2 \\
\ldots \\
\text{case } V_{n-1} \text{l}_{n-1} & \\
\text{case } t \text{l}_n & \\
\text{label next} & \\
\text{t = name holding the value} \\
\text{E = expression, } l_n = \text{default statement}
\end{align*}
\]

Translation of case statement:

Code to Evaluate E into t

goto test

_\text{l}_1 : \text{code for } s_1 \\
goto \text{next}

_\text{l}_2 : \text{code for } s_2 \\
goto \text{next}

\ldots \\

_\text{l}_{n-1} : \text{code for } s_{n-1} \\
goto \text{next}

_\text{l}_n : \text{code for } s_n \\
goto \text{next}

test: if t = V_1 goto l_1 \\
if t = V_2 goto l_2 \\
\ldots \\
if t = V_{n-1} goto l_{n-1} \\
goto l_n \\
\text{next:}
** Backpatching **

It is the activity of filling up unspecified information of labels using appropriate semantic actions in the code generation process.

Backpatching can be useful to generate code for control flow statements like for loop.

To manipulate lists of labels, we use three functions:

1. **make-list** - create a new list containing only an index into the array of quadruples. `make-list` returns a pointer to the list it has made.
2. **merge** - concatenates the lists pointed to by `p1` and `p2` and returns a pointer to the concatenated list.
3. **backpatch** - marks `i` as the target label for each of the statements on the list pointed to by `p`.

---

**Boolean Inference**

Translation scheme is as follows:

(1) $E \rightarrow E_1$ or $E_2$

\[
\text{backpatch (} E_1 \text{.false-list, } M \text{.quad})
\]

\[
E_1 \text{.true-list} := \text{merge (} E_1 \text{.true-list, } E_2 \text{.true-list)};
\]

\[
E_1 \text{.false-list} := E_2 \text{.false-list}?
\]

(2) $E \rightarrow E_1$ and $E_2$

\[
\text{backpatch (} E_1 \text{.true-list, } M \text{.quad})
\]

\[
E_1 \text{.false-list} := \text{merge (} E_1 \text{.false-list, } E_2 \text{.false-list)};
\]

(3) $E \rightarrow \text{not } E_1$

\[
\text{backpatch (} E_1 \text{.true-list} := E_1 \text{.false-list};
\]

\[
E_1 \text{.false-list} := \text{merge (} E_1 \text{.false-list, } E_2 \text{.false-list)};
\]

(4) $E \rightarrow (E_1)$

\[
E_1 \text{.true-list} := E_1 \text{.true-list};
\]

\[
E_1 \text{.false-list} := E_1 \text{.false-list}?
\]

(5) $E \rightarrow \text{id rel op id}$

\[
\text{make-list (nextquad)}
\]

\[
E_1 \text{.false-list} := \text{make-list (nextquad)};
\]

\[
\text{emit ('gotol' id.place rel.op id.place 'goto-')};
\]

(6) $E \rightarrow \text{true}$

\[
\text{make-list (nextquad)};
\]

\[
\text{emit ('goto-')}?
\]

(7) $E \rightarrow \text{false}$

\[
\text{make-list (nextquad)};
\]

\[
\text{emit ('goto-')} ?
\]

(8) $M \rightarrow E$

\[
M \text{.quad} := \text{nextquad}
\]
\[ Q = A < B \text{ OR } C < D \text{ AND } P < Q \]

100  \text{ if } A < B \text{ goto } 102
101  \text{ goto (102) } \rightarrow E \rightarrow E_1 \text{ or } E_2 \left( \text{ backpatch } (E_1, \text{ false list, } N_{\text{quad}}) \right)
102  \text{ if } C < D \text{ goto (104) } \rightarrow E \rightarrow E_1 \text{ and } E_2 \left( \text{ backpatch } (E_1, \text{ true list, } N_{\text{quad}}) \right)
103  \text{ goto } 
104  \text{ if } P < Q \text{ goto } 
105  \text{ goto } 

→ Flow of Control Statement: Translation scheme is given as: 

1. \( S \rightarrow \text{ if } E \text{ then } M_1 \text{ } S_1 \text{ else } M_2 \text{ } S_2 \)
   \[ \text{ backpatch } (E, \text{ true list, } M_1 \text{ } \text{quad}); \text{ backpatch } (E, \text{ false list, } M_2 \text{ } \text{quad}); \]
   \[ S \text{.entlist} := \text{merge} (S_1 \text{.entlist}, \text{merge} (N \text{.entlist}, S_2 \text{.entlist})) \]

2. \( S \rightarrow \text{ if } E \text{ then } N \text{ } S_1 \)
   \[ \text{ backpatch } (E, \text{ true list, } M_1 \text{ } \text{quad}); \]
   \[ S \text{.entlist} := \text{merge} (E, \text{ false list, } S_1 \text{.entlist}) \]

3. \( S \rightarrow \text{ while } E \text{ do } N \text{ } S_1 \)
   \[ \text{ backpatch } (S_1 \text{.entlist}, M_2 \text{ } \text{quad}); \]
   \[ \text{ backpatch } (E, \text{ true list, } M_1 \text{ } \text{quad}); \]
   \[ S \text{.entlist} := E \text{.false list} \]
   \[ \text{emit} ('\text{goto}', M_1 \text{.quad}) \]

4. \( N \rightarrow E \)
   \[ \text{N}.\text{entlist} := \text{make list} (\text{next quad}); \text{emit} ('\text{goto-}') \]

5. \( N \rightarrow E \)
   \[ N . \text{quad} := \text{next quad} \]

6. \( S \rightarrow \text{begin } L \text{ end} \)
   \[ S \text{.entlist} := L \text{.entlist} \]

7. \( S \rightarrow A \)
   \[ S \text{.entlist} := \text{nil} \]

8. \( L \rightarrow L_1; N \text{ } S \)
   \[ \text{ backpatch } (L_1 \text{.entlist}, M_2 \text{ } \text{quad}); \text{L}.\text{entlist} := S \text{.entlist} \]

9. \( L \rightarrow L \)
   \[ L \text{.entlist} := S \text{.entlist} \]

Procedure Calls:

Procedure or function is an important programming construct which is used to obtain the modularity in the mm program.

Consider the grammar for a simple procedure call:

\[ S \rightarrow \text{call id (Elist)} \]
\[ \text{Elist} \rightarrow \text{Elist, } E \mid E \]

[unformatted]
Translation scheme is given as -

1. \( S \rightarrow \text{call} \text{d} (E \text{list}) \) \{ for each item \( p \) on queue do emit (\'param\' \( p \)); emit (\'call\' id; \( place \)); \}

2. \( E \text{list} \rightarrow E \text{list}, E \) \{append \( E \text{place} \) to the end of queue \}

3. \( E \text{list} \rightarrow E \) \{initialize queue to contain only \( E \text{place} \) \}

**CODE GENERATION**

**Issues in the design of a code generator**

1. **Input to the code generator**
   - The code generation phase can therefore focus on the assumption that its input is free of errors (i.e., type checking, type conversion has been done, semantic error)

2. **Target program**
   - Advantages of machine language program is that it can be placed in a fixed location in memory and immediately executed.
   - Advantage of relocatable machine-language program is that it allows subprograms to be compiled separately.
   - Advantage of assembly language program is that it makes the process of code generation somewhat easier.

3. **Memory Management**
   - Using the symbol table information about memory requirements, code generator determines the addresses in the target code. Similarly, if the target code contains the labels then these labels can be converted into equivalent memory addresses.

4. **Instruction Selection**
   - The nature of the instruction set of the target machine determines the difficulty of instruction selection. Important factors are uniformity, completeness, instruction speed and machine idiom.

5. **Register Allocation**
   - Instructions involving register operands are usually shorter and faster than those involving operands in memory. The use of registers is often subdivided into two subproblems:
   - During register allocation, we select the set of variables that will reside in registers at a point in the program.

**companion**
(ii) During a subsequent register assignment phase, we pick the specific registers that a variable will reside in.

Certain machine registers register pairs for some operands and results.

(6) Choice of Evaluation Order:

The evaluation order is an important factor in generating an efficient target code. We can avoid the decision of choice by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

(7) Approaches to Code Generation:

Designing a code generator so it can be easily implemented, tested, maintained and produce correct code is an indispensable goal.

3. Basic Block and Flow Graphs:

Basic Blocks:

It is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt of formality of branching except at the end.

A name in a basic block is said to be live at a given point if its value is used after that point or the program and if not used after that point in the program it said to be dead.

Pinning into Basic Blocks Algorithm:

INPUT → A sequence of three-address statements

OUTPUT → list of basic blocks with each three-address statement in exactly one block.

METHOD:

(i) We first determine the set of leaders, the first statement of basic blocks.
The rules we use are the following:

1) The first statement is a leader.
2) Any statement that is the target of a conditional or unconditional goto is also a leader.
3) Any statement that immediately follows a goto or conditional goto statement is a leader.

(ii) For each leader, its basic block consists of the leader, and all statements up to but not included the next leader or the end of the program.
Transformation on basic blocks -

Two basic blocks are said to be equivalent if they compute the same set of expressions. There are two important classes of local transformations that can be applied to basic blocks:

1. Structure-Preserving Transformations - They are as follows:
   i. Common subexpression elimination
   ii. Dead code elimination
   iii. Renaming of temporary files
   iv. Interchange of two independent adjacent statements
2. Algebraic transformations

Flow Graphs -

A graph representation of three-address statements is called a flow graph. It is useful for understanding code-generation algorithms.

Nodes of the flow graph - Basic flowblocks.

Block whose header is the first statement → Initial block.

There is a directed edge from block B₁ to block B₂ if B₂ immediately follows B₁ in the given sequence or there is any conditional or unconditional jump. We can say B₁ is a predecessor of B₂ or B₂ is a successor of B₁.

Loops -

It is a collection of nodes in the flow graph such that:

i. All nodes are strongly connected that means always there is a path from any node to any other node within that loop.

ii. The collection of nodes has unique entry that means there is only one path from a node outside the loop to the node inside of loop.

The loop that contains no other loop is called inner loop.

Register allocation and assignment -

The most commonly used strategy to register allocation and assignment is to assign specific values to specific registers.

Advantage - Simplified design of the code generator.
Disadvantages - Design of compiler becomes complicated because of restricttive use of registers.

Various strategies used in register allocation and assignment are:

(1) Global Register allocation -

Allocation of variables to specific registers that is consistent across the block boundaries is called global register allocation. Adopted strategies are:

(i) Store the most frequently used variables in fixed registers throughout the loop.
(ii) Assign some fixed number of registers to hold the most active values in each inner loop.
(iii) Register not already allocated may be used to hold values local to one block.
(iv) By using register declaration use by C or BPLSS programmers.

(2) Usage Count -

It is the count for the use of some variable x in some register used in any inner block. The approximate formula for usage count for the loop L in some basic block B can be given as, \[ \sum_{x \in L} [\text{use}(x, B) + 2 \times \text{live}(x, B)] \]

\[ \text{use}(x, B) \rightarrow \text{number of times } x \text{ used in block } B \]

\[ \text{live}(x, B) \rightarrow 1, \text{if } x \text{ is live on exit from } B \text{ otherwise } 0 \]

(3) Register Assignment for Outer Loop - \( L_1 \rightarrow \text{outer loop}, a \rightarrow \text{variable}, L_2 \rightarrow \text{inner loop} \)

Following criteria should be adopted for register assignment for outer loop:

(i) If \( a \) is allocated in loop \( L_2 \) then it should not be allocated in \( L_1 \).

(ii) If \( a \) is allocated in \( L_1 \) and it is not allocated in \( L_2 \) then store \( a \) on a
    enhance to \( L_2 \) and load \( a \) while leaving \( L_2 \).

(iii) If \( a \) is allocated to \( L_2 \) and not in \( L_1 \) then load \( a \) on entrance of \( L_2 \) and store it on exits from \( L_2 \).

(4) Graph Coloring for Register Assignment -

If all registers are occupied then which register should be freed for computation is solved by graph coloring technique which works in two phases.

(a) In the first pass the specific machine instruction is selected for register allocation. For each variable, a symbol register is allocated.

(2) In the second pass the register inference graph is prepared. In the register
    inference graph each node is a symbolic register and an edge connects two nodes.
where one is live at a point when other is defined.

Then the graph coloring technique is applied for this register inference graph using k-colour in which no two symbolic registers can interfere with each other with assigned physical registers.

**DAG representation of Basic Blocks - DAG → Directed Acyclic Graph**

A DAG is constructed from three address statement which is used to apply the transformations on Basic block.

A DAG is constructed for the following type of labels on nodes -
1. Left nodes are labelled by identifiers or variable names or constants.
2. Right nodes are operator values.

**Algorithm for construction of DAG -**

We assume the three address statement and could of following type -
- case (i) $x = y \text{ op } z$
- case (ii) $x = \text{ op } y$
- case (iii) $x = y$

**Step 1** - If $y$ is undefined then create node($y$). Similarly, if $z$ is undefined create a node($z$).

**Step 2** - For the case (i) create a node($\text{op}$) whose left child is node($y$) and node($z$) will be the right child. The check for any common sub-expressions. For the case (ii) determine whether it is a node labeled $\text{op}$, such node will have a child node ($y$).

In case (iii) node $y$ will be node($y$).

**Step 3** - Delete $x$ from list of identifiers from node($x$). Append $x$ to the list of attached identifier for node $x$ found in 2.

**Application of DAG -**
1. Determine the common sub-expressions.
2. Determine which statements of the block could have their computer value outside the block.
3. Determine which names are used inside the block and computed outside the block.
4. Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form $x = y$ until and until it is must.
Array pointer and procedure calls -

The rules to be enforced are the following:

1. Any evaluation of or assignment to an array element of array 'a' must follow the previous assignment to an element of that array if there is one.
2. Any assignment to an element of array 'a' must follow any previous calculation of 'a'.
3. Any use of any identifier must follow the previous procedure call or indirect assignment through a pointer if there is one.
4. Any procedure call or indirect assignment through a pointer must follow all previous evaluation of any identifier.

5. Peephole Optimization -

A simple but effective technique for locally improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called peephole) and replacing them instructions by a shorter or faster sequence, whereas possible.

Characteristics of peephole optimization are -

1. Redundant instruction elimination -
   Redundant loads and stores and unreachable code is eliminated.
2. Flow of control optimization -
   Unnecessary jumps or jumps can be eliminated.
3. Algebraic simplification
4. Reduction in strength -
   Certain machine instructions are cheaper than the others. We can replace instructions by equivalent, cheaper instructions. E.g. $x^2$ is cheaper that $x \times x$.
5. Use of machine idioms -
   We can replace target instructions by equivalent machine instructions in order to improve the efficiency. E.g. Some machines have auto-increment or auto-decrement addressing modes that are used to perform add or subtract operations.

my companion
Generating code from DAG -

It is much simpler than the linear sequence of three address code which
unseen the efficiency of the code. Various algorithms are -

(1) Rearranging order -

By changing the order in which computations are done we can obtain
digit code with minimum cost. \( C_0 = (a+b) + (c-d) \)

\[
\begin{align*}
\text{MOV} & \quad a, R0 & \quad t_1 = a + b \quad t_2 = c - d \quad \text{MOV} & \quad c, R0 \\
\text{ADD} & \quad b, R0 & \quad t_2 = c - d \quad t_3 = e + t_2 \quad \text{SUB} & \quad d, R0 \\
\text{MOV} & \quad c, R1 & \quad t_3 = e + t_2 \quad t_4 = a + b \quad \text{MOV} & \quad e, R1 \\
\text{SUB} & \quad d, R1 & \quad t_4 = t_4 + t_3 \quad t_5 = t_1 + t_3 \quad \text{ADD} & \quad R_0, R_2 \\
\text{MOV} & \quad R_0, R_4 \quad \text{DAG for } (a+b) + (c-d) \\
\text{MOV} & \quad e, R0 \quad \text{ADD} & \quad b, R0 \\
\text{ADD} & \quad R_0, R_1 \quad \text{ADD} & \quad R_1, R_0 \\
\text{MOV} & \quad t_4, R0 \quad \text{MOV} & \quad R_0, t_4 \\
\text{ADD} & \quad R_0, R_0 \quad \text{MOV} & \quad t_4, R_0 \\
\text{MOV} & \quad R_0, t_4
\end{align*}
\]

(2) Heuristic Ordering -

Node listing algorithm is given as -

while unlisted internal node remain do begin

select an unlisted node \( n \), all of whose parents have been listed;

list \( n \); while the leftmost child \( m \) of \( n \) has no unlisted parents and is not a leaf do

/* since \( n \) was just listed, \( m \) is not yet listed */

begin

list \( m \);

\( n \) := \( m \);

end

end

(3) Labelling algorithm - Generate optimal code in which minimum registers are required.

\[
\begin{align*}
\text{label}(n) = \begin{cases} 
\text{max}(l_1, l_2) & \text{if } l_1 \neq l_2 \\
\text{bottom-up order} & \text{if } l_1 = l_2
\end{cases}
\end{align*}
\]

where companion

\[
\begin{align*}
l_1 + 1 \quad \text{if } l_1 \neq l_2 \\
1 \quad \text{if } l_1 = l_2
\end{align*}
\]
procedure gencode(n);
begin
  /* case 0 */
  if n is a leaf representing operand name and n is the leftmost child of its parent then
    print 'MOV' || name || ' ', || top(rstack)
  else if n is an interior node with operator op, left child n₁, and right child n₂ then
    /* case 1 */
    if label(n₂) = 0 then begin
      let name be the operand represented by n₂;
      gencode(n₂);
      print op || name || ' ', || top(rstack)
    end
    /* case 2 */
    else if 1 ≤ label(n₁) < label(n₂) and label(n₁) < r then begin
      swap(rstack);
      gencode(n₁);
      R := pop(rstack); /* n₁ was evaluated into register R */
      gencode(n₂);
      print op || R || ' ', || top(rstack);
      push(rstack, R);
      swap(rstack)
    end
    /* case 3 */
    else if 1 ≤ label(n₂) ≤ label(n₁) and label(n₂) < r then begin
      gencode(n₂);
      R := pop(rstack); /* n₂ was evaluated into register R */
      gencode(n₁);
      print op || top(rstack) || ' ', || R;
      push(rstack, R)
    end
    /* case 4 */
    else both labels ≥ r, the total number of registers ≥
    else begin
      gencode(n₁);
      T := pop(rstack);
      print 'MOV' || top(rstack) || ' ', || T;
      gencode(n₂);
      push(rstack, T);
      print op || T || ' ', || top(rstack)
    end
  end

Let us generate a code from this labelling algorithm.

Consider the labelled tree as follows.

```
Fig. 7.19 Labelled tree with generated code for ((a+(b+c))-(a+(d*e)))
```

```
MOV b, R0
MOV c, R0
ADD R1, R0
MOV R0, T0
MOV d, R1
MUL e, R1
MOV c, R0
ADD R0, R1
SUB R1, T0
```

Label computation

1. If n is a leaf then
2. If n is the leftmost child of its parent then
3. label(n) := 1
4. else label(n) := 0
   else begin /* n is a interior node */
5. let n₁, n₂, ..., nᵢ be the children of n ordered by label,
   so label(n₁) ≥ label(n₂) ≥ ... ≥ label(nᵢ); label(n) := max_{i≤n}(label(nᵢ)) + 1
6. end