1) **What is intermediate code?**

Intermediate code is:
- The output of the parser and the input to the Code Generator.
- Relatively machine-independent: allows the compiler to be retargeted.
- Relatively easy to manipulate (optimize).

![Diagram of compiler components](image)

**Advantages of Using an Intermediate Language**
- Retargeting - Build a compiler for a new machine by attaching a new code generator to an existing front-end.
- Optimization - reuse intermediate code optimizers in compilers for different languages and different machines.

2) **Explain Different Intermediate forms**

There are three types of intermediate representation
1. Abstract syntax tree
2. Postfix notation
3. Three address code

**1. Abstract syntax tree**
- A syntax tree depicts the natural hierarchical structure of a source program.
- A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified.
- Syntax tree for the assignment statement \( x = -a*b + -a*b \) is given below

![Syntax tree](image)
2. **Postfix notation**
   - Postfix notation is a linearized representation of a syntax tree.
   - It a list of nodes of the tree in which a node appears immediately after its children
   - the postfix notation of above syntax tree is x a –b * a-b*+=

3. **Three address code**
   - The general form of three address code representation is:
     \[ a := b \text{ op } c \]
   - Where a, b or c are the operands that can be names or constants.
   - For the expression like \( a = b + c + d \) the three address code will be
     \[
     \begin{align*}
     t1 &= b + c \\
     t2 &= t1 + d \\
     \end{align*}
     \]
   - Here \( t1 \) and \( t2 \) are the temporary names generated by the compiler.
   - There are at most three addresses allowed (two for operands and one for result). Hence, this representation is called three-address code.

3) **Implementation of three address code**
   - There are three representations used for three address code such as quadruples, triples and indirect triples.
   - Consider the input statement \( x := -a*b + -a*b \)
   - Three address code for above statement given below:

   ```
   t1 = uminus a \\
   t2 := t1 * b \\
   t3 = - a \\
   t4 := t3 * b \\
   t5 := t2 + t4 \\
   x = t5
   ```

   **Quadruple representation**
   - The quadruple is a structure with at most four fields such as op, arg1, arg2.
   - The op field is used to represent the internal code for operator, the arg1 and arg2 represent the two operands. And result field is used to store the result of an expression.
Intermediate Code Generation

### Triangles
- The triple representation the use of temporary variables is avoided by referring the pointers in the symbol table.

<table>
<thead>
<tr>
<th>Number</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>minus</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>*</td>
<td>t1</td>
<td>b</td>
</tr>
<tr>
<td>(2)</td>
<td>minus</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>*</td>
<td>t3</td>
<td>b</td>
</tr>
<tr>
<td>(4)</td>
<td>+</td>
<td>t2</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>:=</td>
<td>t5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>minus</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>*</td>
<td>t1</td>
<td>b</td>
</tr>
<tr>
<td>(2)</td>
<td>minus</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>*</td>
<td>t3</td>
<td>b</td>
</tr>
<tr>
<td>(4)</td>
<td>+</td>
<td>t2</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>:=</td>
<td>t5</td>
<td></td>
</tr>
</tbody>
</table>

### Indirect Triples
- The indirect triple representation the listing of triples is been done. And listing pointers are used instead of using statements.

<table>
<thead>
<tr>
<th>Number</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(11)</td>
<td>minus</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(12)</td>
<td>*</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>(13)</td>
<td>minus</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(14)</td>
<td>*</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>(15)</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(16)</td>
<td>:=</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(11)</td>
<td>minus</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(12)</td>
<td>*</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>(13)</td>
<td>minus</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(14)</td>
<td>*</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>(15)</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(16)</td>
<td>:=</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4) Syntax directed translation mechanisms

- For obtaining the three address code the SDD translation scheme or semantic rules must be written for each source code statement.
- There are various programming constructs for which the semantic rules can be defined.
- Using these rules the corresponding intermediate code in the form of three address code can be generated. Various programming constructs are:
  1. Declarative statement
  2. Assignment statement
  3. Arrays
  4. Boolean expressions
  5. Control statement
  6. Switch case
  7. Procedure call

Declarative Statement

- In the declarative statements the data items along with their data types are declared.
- Example

<table>
<thead>
<tr>
<th>S -&gt; D</th>
<th>Offset = 0</th>
</tr>
</thead>
</table>
| D -> id: T | \{enter(id.name,T.type,offset);
| | Offset=offset+T.width \} |
| T -> integer | \{T.type:=integer; T.width:=4\} |
| T -> real | \{T.type:=real; T.width:=8\} |
| T -> array[num] of T_1 | \{T.type:=array(num.val,T_1.type);
| | T.width:=num.val X T_1.width \} |
| T -> *T_1 | \{T.type:=pointer(T.type)
| | T.width:=4\} |

- Initially, the value of offset is set to zero. The computation of offset can be done by using the formula \( offset = offset + width \).
- In the above translation scheme T.type, T.width are the synthesized attribute. The type indicates the data type of corresponding identifier and width is used to indicate the memory units associated with an identifier of corresponding type.
- The rule \( D -> id: T \) is a declarative statement for id declaration. The enter function used for creating the symbol table entry for identifier along with its type and offset.
- The width of array is obtained by multiplying the width of each element by number of elements in the array.
- The width of pointer type is supposed to be 4.
Assignment Statements

- The assignment statement mainly deals with the expressions. The expressions can be of type integer, real, array and record.
- Consider the following grammar
  
  \[
  S \rightarrow \text{id} := E \\
  E \rightarrow E_1 + E_2 \\
  E \rightarrow E_1 * E_2 \\
  E \rightarrow -E_1 \\
  E \rightarrow (E_1) \\
  E \rightarrow \text{id} 
  \]

- The translation scheme of above grammar is given below:

<table>
<thead>
<tr>
<th>Production Rule</th>
<th>Semantic actions</th>
</tr>
</thead>
</table>
| \[S \rightarrow \text{id} := E\] | \{ p=look_up(id.name);  
  If p≠ nil then  
  Emit(p = E.place)  
  Else  
  Error;  
  \} |
| \[E \rightarrow E_1 + E_2\] | \{ E.place=newtemp();  
  Emit (E.place=E1.place '+' E2.place)  
  \} |
| \[E \rightarrow E_1 * E_2\] | \{ E.place=newtemp();  
  Emit (E.place=E1.place '*' E2.place)  
  \} |
| \[E \rightarrow -E_1\] | \{ E.place=newtemp();  
  Emit (E.place=’uminus’ E1.place)  
  \} |
| \[E \rightarrow (E_1)\] | \{E.place=E1.place\} |
| \[E \rightarrow \text{id}\] | \{ p=look_up(id.name);  
  If p≠ nil then  
  Emit (p = E.place)  
  Else  
  Error;  
  \} |

- The p returns the entry for id.name in the symbol table if it exists there.
- The function Emit is for appending the three address code to the output file. Otherwise an error will be reported.
- newtemp() is the function for generating new temporary variables.
- E.place is used to hold the value of E. consider the assignment statement \(x := (a + b)\)
*(c+d)

<table>
<thead>
<tr>
<th>Production Rule</th>
<th>Semantic action for Attribute evaluation</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>E -&gt; id</td>
<td>E.place := a</td>
<td></td>
</tr>
<tr>
<td>E -&gt; id</td>
<td>E.place := b</td>
<td></td>
</tr>
<tr>
<td>E -&gt; E₁ + E₂</td>
<td>E.place := t₁  t₁ := a+b</td>
<td></td>
</tr>
<tr>
<td>E -&gt; id</td>
<td>E.place := c</td>
<td></td>
</tr>
<tr>
<td>E -&gt; id</td>
<td>E.place := d</td>
<td></td>
</tr>
<tr>
<td>E -&gt; E₁ + E₂</td>
<td>E.place := t₂  t₂ := c+d</td>
<td></td>
</tr>
<tr>
<td>E -&gt; E₁ * E₂</td>
<td>E.place := t₃  t₃ := (a+b)*(c+d)</td>
<td></td>
</tr>
<tr>
<td>S-&gt; id := E</td>
<td></td>
<td>x := t₃</td>
</tr>
</tbody>
</table>

**Boolean Expression**

- Normally there are two types of Boolean expressions used:
  1. For computing the logical values.
  2. In conditional expressions using if-then-else or while-do.
- Consider the Boolean expression generated by following grammar:
  
  ```
  E -> E OR E
  E-> E AND E
  E-> NOT E
  E-> (E)
  E-> id relop id
  E-> TRUE
  E-> FALSE
  ```

  The relop is denoted by <, >, <=, >=. The OR and AND are left associate.

  The highest precedence is to NOT then AND and lastly OR.

```plaintext
E-> E₁ OR E₂
{
    E .place:=newtemp()
    Emit (E.place :=E₁.place "OR' E₂.place)
}

E-> E₁ AND E₂
{
    E .place:=newtemp()
    Emit (E.place :=E₁.place "AND' E₂.place)
}
```
<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E -&gt; NOT E1</td>
<td></td>
</tr>
</tbody>
</table>
E.place := newtemp() 
Emit (E.place := "NOT" E1.place) |
| E -> (E1) | 
E.place := E1.place |
| E -> id1 relop id2 | 
E.place := newtemp() 
Emit ('if id1.place relop.op id2.place 'goto' next_state +3') 
Emit (E.place := '0') 
Emit ('goto' next_state +2) 
Emit (E.place := '1') |
| E -> TRUE | 
E.place := newtemp(); 
Emit (E.place := '"1") |
| E -> FALSE | 
E.place := newtemp(); 
Emit (E.place := '"0") |

- The function Emit generates the three address code and newtemp() is for generation of temporary variables.
- For the semantic action for the rule E -> id1 relop id2 contains next_state which gives the index of next three address statement in the output sequence.
- Let us take an example and generate the three address code using above translation scheme: 
  \[ p > q \text{ AND } r < s \text{ OR } u > v \]
  
  ```
  100: if p > q goto 103 
  101: t1:=0 
  102: goto 104 
  103: t1:=1 
  104: if r < s goto 107 
  105: t2:=0 
  106: goto 108 
  107: t2=1 
  108: if u>v goto 111 
  109: t3=0 
  110: goto 112 
  111: t3=1 
  112: t4=t1 AND t2 
  113: t5=t4 OR t3
  ```
Flow of control statement

✓ The control statements are if-then-else and while-do.
✓ The grammar for such statements is given below

✓ S --> if E then S1 | if E then S1 else S2 | while E do S1
✓ Translation scheme is given below

S->if E then S1

- {E.true=new_label()}
- E.False=new_label()
- S1.next=S.next
- S2.next=S.next
- S.code= E.code ||gen_code(E.true':')
- ||S1.code}

S->if E then S1 else S2

- {E.true=new_label()}
- E.False=new_label()
- S1.next=S.next
- S2.next=S.next
- S.code=E.code ||gen_code(E.true':')||S1.code
- ||gen_code('goto',s.next)
- ||$gen_code(E.false':') || S2.code}

S->while E do S1

- {S.begin=new_label()}
- E.True=new_label()
- E.False=S.next
- S1.next=S.begin
- S.code=gen_code(S.begin':')||E.code
- ||gen_code(E.true':')
- ||S1.code||gen_code('goto',S.begin)}

✓ Consider the statement: if a<b then a=a+5 else a=a+7
✓ Three address code for above statement using semantic rule is

100: if a<b goto L1
101: goto 103
102: L1: a=a+5
103: a=a+7

(*Array & case statement refer from text book)