1. Role of intermediate code generation. OR What is intermediate code? Which are the advantages of it?

- In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate representation from which backend generates target code.
- The generation of an intermediate language leads to efficient code generation.

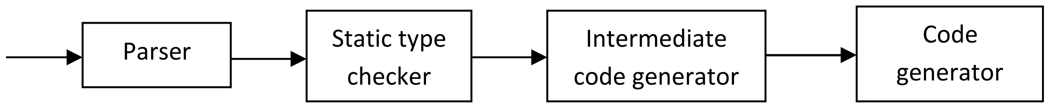

Fig.5.1 Position of intermediate code generator in compiler

- There are certain advantages of generating machine independent intermediate code,
  1. A compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
  2. A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

2. Explain different intermediate forms.

There are three types of intermediate representation,

1. Abstract syntax tree
2. Postfix notation
3. Three address code

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.
- A syntax tree and DAG for the assignment statement a = b*-c + b*-c is given in Fig. 5.2.

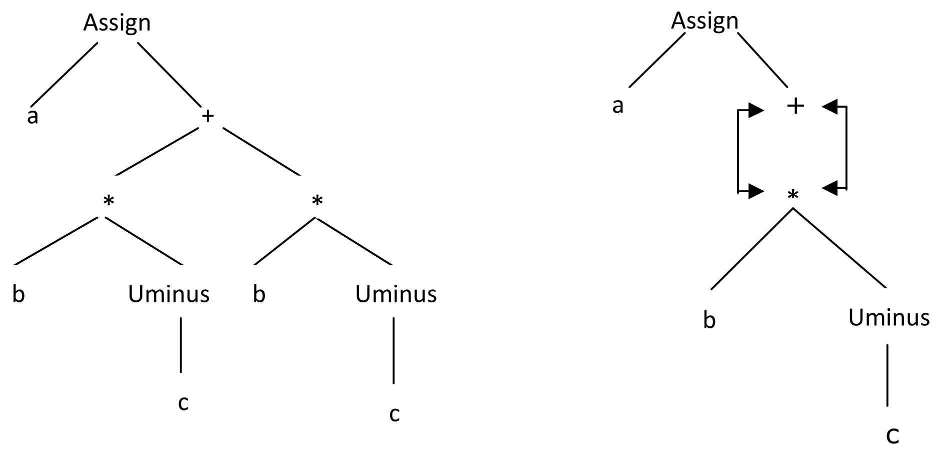

Fig.5.2 Syntax tree & DAG for a = b*-c + b*-c

Postfix notation

- Postfix notation is a linearization of a syntax tree.
- In postfix notation the operands occurs first and then operators are arranged.
- the postfix notation for the syntax tree in Fig. 5.2 is,
  a b c uminus * b c uminus * + assign.

Three address code

- Three address code is a sequence of statements of the general form,
  a:= b op c
- Where a, b or c are the operands that can be names or constants. And op stands for any operator.
- For the expression like a = b + c + d might be translated into a sequence,
  t_1=b+c
  t_2=t_1+d
  a= t_2
- Here t_1 and t_2 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. Implementations of three address code.

- There are three types of representation used for three address code,
  1. Quadruples
  2. Triples
  3. Indirect triples
- Consider the input statement x:= -a*b + -a*b.
- Three address code for above statement given in table 5.1,

t_1= - a
t_2 := t_1 * b
t_3= - a
t_4 := t_3 * b
t_5 := t_2 + t_4
x= t_5

Table 5.1 Three address code

Quadruple representation

- The quadruple is a structure with at the 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.
- Statement with unary operators like x= -y do not use arg2.

- Conditional and unconditional jumps put the target label in result.

Number	Op	Arg1	Arg2	result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		x

Table 5.2 Quadruple representation

Triples

- To avoid entering temporary names into the symbol table, we might refer a temporary value by the position of the statement that computes it.
- If we do so, three address statements can be represented by records with only three fields: op, arg1 and arg2.

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	X	(4)

Table 5.3 Triple representation

Indirect Triples

- In the indirect triple representation the listing of triples has been done. And listing pointers are used instead of using statement.
- This implementation is called indirect triples.

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(11)	b
(2)	uminus	a	

	Statement
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)

(3)	*	(13)	b		(4)	(15)
(4)	+	(12)	(14)		(5)	(16)
(5)	:=	X	(15)			

Table 5.4 Indirect triple representation

4. Syntax directed translation mechanism.

- 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:

S→ D	{offset=0}
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}

Table 5.5 Syntax directed translation for Declarative statement

- 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 and T.width are the synthesized attributes.
- 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.

Assignment statement

- The assignment statement mainly deals with the expressions.
- The expressions can be of type integer, real, array and record.
- Consider the following grammar,
  $S \rightarrow id := E$
  $E \rightarrow E_1 + E_2$
  $E \rightarrow E_1 * E_2$
  $E \rightarrow -E_1$
  $E \rightarrow (E_1)$
  $E \rightarrow id$
  The translation scheme of above grammar is given in table 5.6.

Production Rule	**Semantic actions**

S→ id :=E	{ p=look_up(id.name);
	If p≠ nil then
	emit(p = E.place)
	else error;}

E→ E1 + E2	{ E.place=newtemp();
	emit (E.place=E_1.place '+' E_2.place)}

E→ E1 * E2	{ E.place=newtemp();
	emit (E.place=E_1.place '*' E_2.place)}

E→ -E1	{ E.place=newtemp();
	emit (E.place='uminus' E_1.place)}

E→ (E1)	{E.place=E1.place}

E→ id	{ p=look_up(id.name);
	If p≠ nil then
	emit (p = E.place)
	else
	error;}

Table 5.6. Translation scheme to produce three address code for assignments

- The look_up 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),

Production Rule	Semantic action for Attribute evaluation	Output

$E \rightarrow id$	E.place := a	
$E \rightarrow id$	E.place := b	
$E \rightarrow E_1 + E_2$	E.place := t_1	t_1 := a+b
$E \rightarrow id$	E.place := c	
$E \rightarrow id$	E.place := d	
$E \rightarrow E_1 + E_2$	E.place := t_2	t_2 := c+d
$E \rightarrow E_1 * E_2$	E.place := t_3	t_3 := (a+b)*(c+d)
$S \rightarrow id := E$		x := t_3

Table 5.7 Three address code for Assignment statement

Arrays

- Array is a contiguous storage of elements.
- Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is w, then the ith element of array A begins in location,
  base + (i – low) x w
- Where low is the lower bound on the subscript and base is the relative address of the storage allocated for the array. That is, base is the relative address of A[low].
- The expression can be partially evaluated at compile time if it is rewritten as,
  i x w + (base – low x w)
- The sub expression c = base – low x w can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A , so the relative address of A[i] is obtained by simply adding i x w to c.
- There are two representation of array,
  1. Row major representation.
  2. Column major representation.
- In the case of row-major form, the relative address of A[i1, i2] can be calculated by the formula:
  base + ((i1 – low1) x n2 + i2 – low2) x w
- where, low1 and low2 are the lower bounds on the values of i1 and i2 and n2 is the number of values that i2 can take. That is, if high2 is the upper bound on the value of i2, then n2 = high2 –low2 + 1.
- Assuming that i1 and i2 are the only values that are known at compile time, we can rewrite the above expression as
  ((i1 x n2) + i2) x w + (base – ((low1 x n2) + low2) x w)
- Generalized formula: The expression generalizes to the following expression for the relative address of A[i1,i2,…,ik]
  ((. . . ((i1n2 + i2) n3 + i3) . . .) nk + ik) x w + base – ((. . .((low1n2 + low2)n3 +

low3) . . .)nk + lowk) x w
for all j, nj = highj – lowj + 1

- The Translation Scheme for Addressing Array Elements :
  S → L : = E
  E→ E + E
  E→ (E)
  E→ L
  L→ Elist]
  L→ id
  Elist→Elist , E
  Elist→id [E
- The translation scheme for generating three address code is given by using appropriate semantic actions.

Production Rule	Semantic Rule

S → L : = E	{ if L.offset = null then
	emit (L.place ': =' E.place) ;
	else
	emit (L.place ' [' L.offset ']' ': =' E.place) }

E→ E + E	{ E.place : = newtemp;
	emit (E.place ': =' E1.place ' +' E2.place) }

E→ (E)	{ E.place : = E1.place }

E→ L	{ if L.offset = null then
	E.place : = L.place
	else begin
	E.place : = newtemp;
	emit (E.place ': =' L.place ' [' L.offset ']')
	end }

L→ Elist]	{ L.place : = newtemp;
	L.offset : = newtemp;
	emit (L.place ': =' c(Elist.array));
	emit(L.offset':='Elist.place'*'width
	(Elist.array)) }

L→ id	{ L.place := id.place;
	L.offset := null }

Elist→Elist , E	{ t := newtemp;
	dim : = Elist1.ndim + 1;
	emit(t':='Elist1.place'*'limit(Elist1.array,dim));
	emit (t ': =' t '+' E.place);
	Elist.array : = Elist1.array;
	Elist.place : = t;
	Elist.ndim : = dim }

Elist→id [E	{ Elist.array : = id.place;

	Elist.place : = E.place;
	Elist.ndim : = 1 }

Table 5.8 Syntax directed translation scheme to generate three address code for Array

- Annotated parse tree for x=A[i, j] is given in figure 5.3.

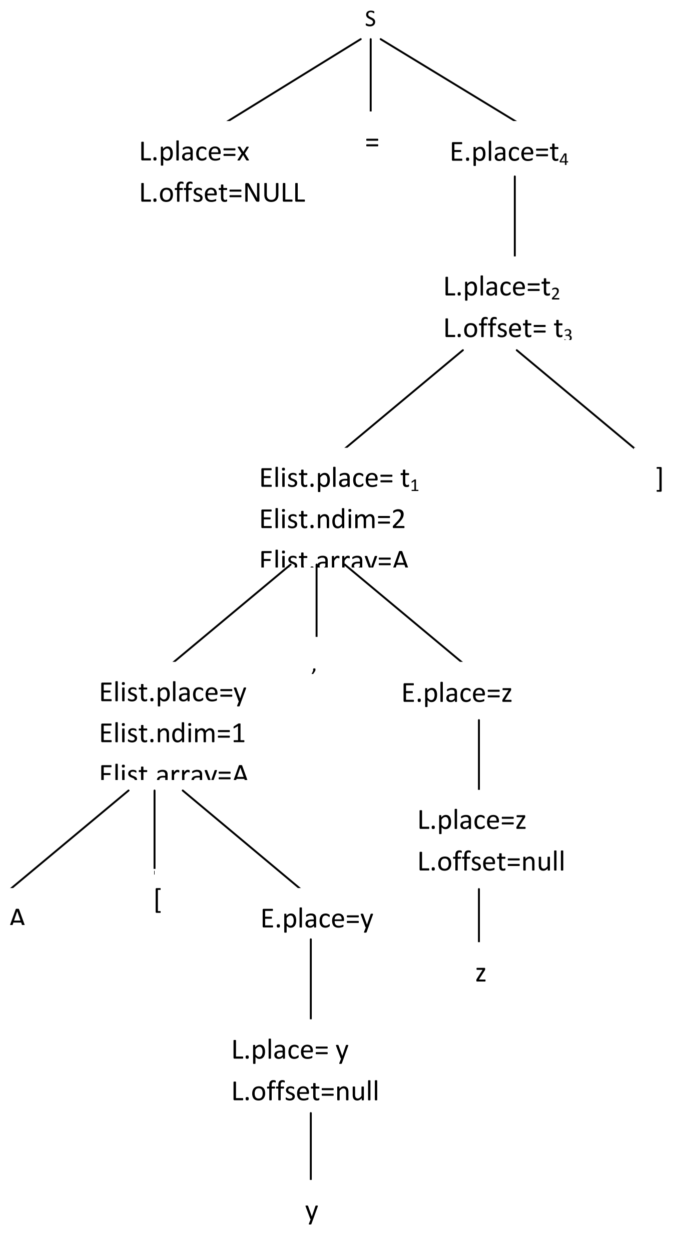

Fig 5.3. Annotated parse tree for x:=A[y, z]

Boolean expressions

- 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.

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)}
E→**NOT** E1	{E .place:=newtemp() Emit (E.place ':="NOT' E1.place)}
E→(E1)	{E.place := E1.place }

E→id$_1$ **relop** id$_2$	{E. place := newtemp()
	Emit ('if id.place relop.op id$_2$.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')}

Table 5.9 Syntax directed translation scheme to generate three address code for Boolean expression

- 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 AND r < s 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

Control statement

- The control statements are if-then-else and while-do.
- The grammar and translation scheme for such statements is given in table 5.10.
- S → if E then S1| if E then S1 else S2 | while E do S1

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)}

Table 5.10 Syntax directed translation scheme to generate three address code for Control statement

- 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

Switch case

- Consider the following switch statement;
  switch E

```
begin
        case $V_1$: $S_1$
        case $V_2$: $S_2$
                ....
        case $V_{n-1}$: $S_{n-1}$
        default: $S_n$
end
```

- Syntax directed translation scheme to translate this case statement into intermediate code is given in table 5.11.

	Code to evaluate E into t
	goto test
L_1:	Code for S_1
	goto next
L_2	Code for S_2
	goto next
	…….
L_{n-1}	Code for S_{n-1}
	goto next
L_n	Code for S_n
	goto next
test:	If t=V_1 goto L_1
	If t=V_1 goto L_1
	If t=V_1 goto L_1
	goto L_n
next:	

Table 5.11 Syntax directed translation scheme to generate three address code for switch case

- When we see the keyword switch, we generate two new labels test and next and a new temporary t.
- After processing E, we generate the jump goto test.
- We process each statement case V_i : S_i by emitting the newly created label Li, followed by code for Si, followed by the jump goto next.
- When the keyword end terminating the body of switch is found, we are ready to generate the code for n-way branch.
- Reading the pointer value pairs on the case stack from the bottom to top , we can generate a sequence of three address code of the form,

```
case $V_1$ $L_1$
case $V_2$ $L_2$
……
case $V_{n-1}$ $L_{n-1}$
case t $L_n$
label next
```

- Where t is the name holding the value of selector expression E, and L_n is the label for default statement.
- The case V_i L_i three address statement is a synonym for if t= V_i goto L_i.

Procedure call

- Procedure or function is an important programming construct which is used to obtain the modularity in the user program.
- Consider a grammar for a simple procedure call,
  S→call id (L)
  L→L, E
  L→E
- Here S denotes the statement and L denotes the list of parameters.
- And E denotes the expression.
- The translation scheme can be as given below,

Production rule	**Semantic Action**

S→call id (L)	{ for each item p in queue do
	append('param' p);
	append('call' id.place);}

L→L,E	{ insert E.place in the queue }
L→E	{ initialize the queue and insert E.place in the queue }

Table 5.12 Syntax directed translation scheme to generate three address code for procedure call

- The data structure queue is used to hold the various parameters of the procedure.
- The keyword param is used to denote list of parameters passed to the procedure.
- The call to the procedure is given by 'call id' where id denotes the name of procedure.
- E.place gives the value of parameter which is inserted in the queue.
- For L→E the queue gets empty and a single pointer to the symbol table is obtained. This pointer denotes the value of E.