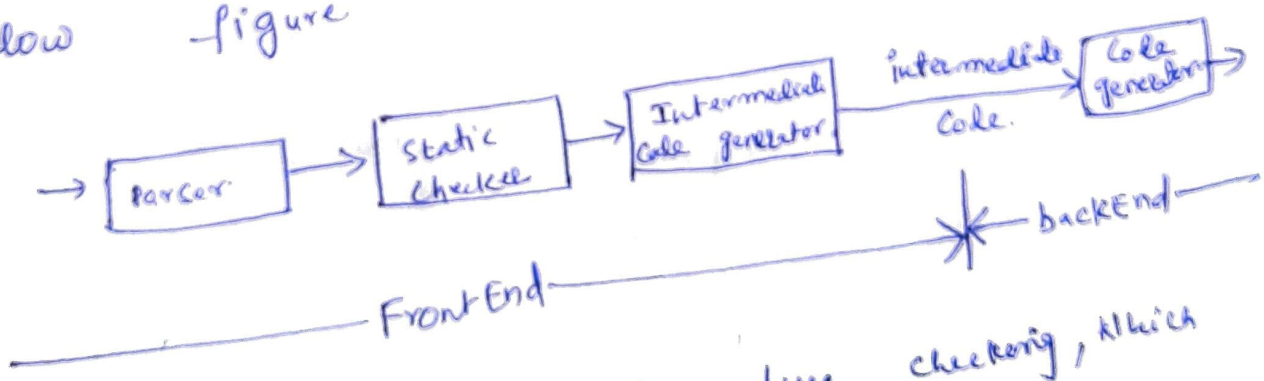
U-3.

Chapter-2

Intermediate Code Generation.

In analysis-synthesis model of a Compiler the front end analyzes a source program and creates a intermediate representation, from which back-end generates target code. Details of source language are used in front end, and details of target machine is to back end.

Compiler front end includes static checking, intermediate code generation. As shown in below figure



Static checking include type checking, which ensures Operators have compatible Operands. and it also includes Syntactic checking. For ex, static checking also ensures that a break statement is enclosed in either Swith, While, or in for loop, Otherwise an error is reported.

Variations of Syntax Tree.

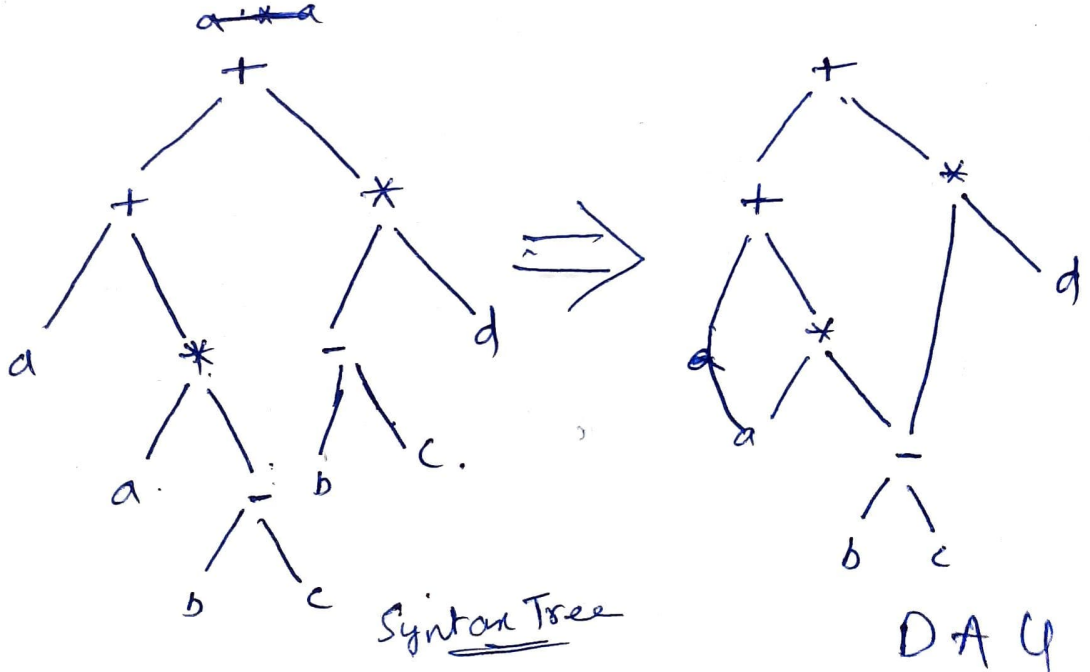
Directed Acyclic Graph for Expressions. (DAG)

A Directed Acyclic Graph for an expression identifies common subexpressions which occur more than once in an expression.

Like in a Syntax Tree, a DAG has leaves corresponding to atomic values and interior nodes corresponding to operators. A node N in DAG, has more than one parent, where N represents common subexpression. In Syntax tree Common subexpression will be replicated as many times it appears in expression.

Ex:- Following expression represents Syntaxtree, and DAG for

a + a * (b - c) + (b - c) * d



The leaf for a has two parents, because a appears twice in expression. And subexpression (b-c), also has two parents because it also appears twice in a*(b-c) and (b-c)*d

Following is an SDD which Construct either Syntax Tree or DAG's. Functions and Node's create a fresh nodes each time when they called in Syntax Tree. In DAG, before Creating a new node These functions first Check whether an identical node exists, if previously created node exists it will be returned, otherwise it creats a new node.

Production	Semantic Rules
1) E → E1 + T	E.node = new Node ('+', E1.node, T.node)
2) E → E1 - T	E.node = new node ('-', E1.node, T.node)
3) E → T	E.node = T.node.
4) T → T1 * T2	T.node = new node ('*', T1.node, T2.node)
5) T → (E)	T.node = E.node.
6) T → id	T.node = new Leaf (id, id.entry)
7) T → num	T.node = new Leaf (num, num.val)

Following steps shows Construction of DAG in previous fig, provided Node and Leaf returns existing Node

$P1 =$ Leaf (id, entry-a)

$P2 =$ Leaf (id, entry-a) $= P1$

$P3 =$ Leaf (id, entry-b)

$P4 =$ Leaf (id, entry-c)

$P5 =$ Node ('−', P3, P4)

$P6 =$ Node ('*', P1, P5)

$P7 =$ Node ('+', P1, P6)

$P8 =$ ~~Node~~ Leaf (id, entry-b) $= P3$.

$P9 =$ Leaf (id, entry-c) $= P4$.

$P10 =$ Node ('−', P3, P4) $= P5$

$P11 =$ Leaf (id, entry-d)

$P12 =$ Node ('*', P5, P11)

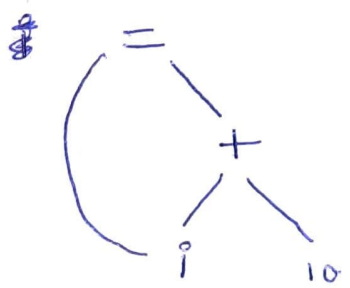
$P13 =$ Node ('+', P7, P12)

When ~~we~~ the call to Leaf (id, entry-a) is repeated at Step 2, The node created by previous call is returned. So $P2 = P1$. Similarly Nodes returned at 8 and 9 are same as Step 3 and 4 ($P8 = P3, P9 = P4$)

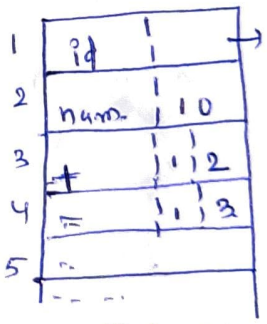
And $P10 = P5$.

The Value-Number Method For Constructing DAG's

The Nodes of a Syntax tree or DAG are stored in an array of records As shown in below figure. Each row represents one record ~~on~~ (node). In each record first field is an Operation code, indicating label of the node. Leafs has an additional field which holds the lexical values, and interior nodes have two additional fields indicating left and right children.



a) DAG



b) Array.

Nodes of Dag for i = i + 10 allocated in Array.

In the Array, nodes are referred by integer index of the record. This integer is known as Value-number for the node or for expression represented by the node. For ex, The node labeled + has Value number 3, and left and right children have value number 1 and 2 respectively

Value-number helps Construct Expression DAG's

Each Node is referred by its Value number

The signature of ~~Value~~ interior node be the triple $<op, l, r>$ where op is label, l is left child and r its right child.

Algorithm :- Value-number method for constructing Nodes of a DAG.

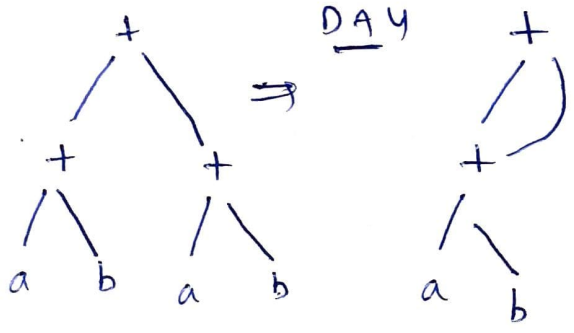
input :- Label op, node l, and node r

Output :- The Value-number of a node in the array with signature (op, l, r).

Method :- Search the Array ~~with~~ for a node M, with label op, left child l, and right child r, if such node exists return value number of N, if not create a new node N with label op, left child l, right child r, and return its Value number.

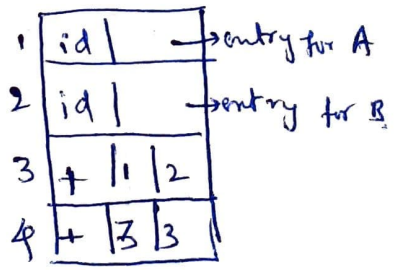
Ex:→ Construct DAG, and identify Value numbers for Subexpressions $a+b+(a+b)$

1. Parse Tree (Syntax Tree)



DAG

Value-Numbers



Three Address Code.

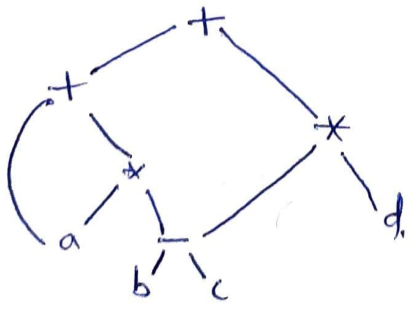
In Three Address Code, there is atmost one Operator on right side of expression. A source language expression like $x + y * z$ can be translated into a series of Three address instructions

$$t_1 = y * z.$$

$$t_2 = x + t_1$$

t1 and t2 are compiler generated temporary names. The temporary names allows the three address code to be rearranged easily

Three address code can be used to represent a Syntax tree or DAG, where explicit names corresponds to interior nodes of the graph. Following figure shows a DAG and its Three Address code



$$t1 = b - c.$$

$$t_2 = a * t_1$$

$$t_3 = a + t_2$$

$$t_4 = t1 * d$$

$$t_5 = t_3 + t_4.$$

Addresses and instructions.

Three Address code is built from two concepts addresses and instructions. Three address code can be implemented using records with fields for addresses

An address can be one of the following.

→ A name → For convenience, source-program names can appear as addresses in Three address code. While implementation source name is replaced by symbol-table entry where all info about name is kept.

→ A constant → Compiler will deal with many types of constants and variables.

→ A compiler generated temporary. → Compiler creates a distinct name each time a temporary is needed

Common Three Address instructions:-

1. Assignment instructions. of the form $x = y \text{ op } z$, where op is binary arithmetic or logical operation and x, y, z are addresses.

2. Assignment of the form $x = op\ y$, where op is unary operation like unary minus, logical negation, conversion operators

3. Copy Instruction of the form $x = y$, where y value is assigned to x.

4. Unconditional Jump goto L., The Three address inst^n with label L is next executed.

5. Condition Jumps if x goto L and

   if False x goto L

6. Conditional Jumps like if x relop y goto L.

   where relop is a relational operator (<, ==, >=, etc)

7. procedure calls and return statements using following inst^n.

   → Param x for parameters.

   → call p n

   → y = call p n

   → return y

   p → stands for procedure, n stands for number of Actual parameters.

8. indexed copy instructions of the form

   $x[i] = y$ and $x = y[i]$. where x will be set value at i'n memory location starting from y. and

in $x[i] = y$, The value of y is stored into memory location i beyond x

9. Address and pointer Assignments of the form

$x = \&y$, $x = *y$ and $*x = y$

in $x = \&y$ ⇒ This sets address of y to x.

in $x = *y$ ⇒ The value at address in y is assigned in x.

in $*x = y$, The value of y will be Stored in the Address of x.

Ex:- Consider the following statement.

```
do  i = i + 1;
while (a[i] < v);
```

This code can be translated into Three address code as follows

```
L : t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a[t2]
    if t3 < v goto L
```

i*8 denotes index of next element.

Following is an alternative which uses position numbers starting from 100.

100: $t1 = i + 1$

101: $i = t1$

102: $t2 = i * 8$

103: $t3 = a[t2]$

104: if $t3 < V$ goto 100.

~~Quadruples~~.

Quadruples :- Quadruples is method to represent three address code. In Compiler instructions can be implemented as objects or as records, with fields for operator and operands. Three such representations are Quadruples, Triples and indirect triples.

A Quadruple or simply a Quad has four fields, op, arg1, arg2 and result. Op field contains internal code for operator. For eg, The Three address instruction $x = y + z$ has following record.

op	arg1	arg2	result.
+	y	z	x.

But for some instructions like

1. Unary Operators like $x = \text{minus } y$, has only arg1, and op is minus. $x = y$ do not have arg2 and = is op.

2. Operators like param are neither arg2 nor result.

3. Conditional and unconditional Jumps puts the target Label in result.

Ex:-1). Three Address code for Assignment

$a = b * -c + b * -c$. is given below.

Special Operator minus, represents unary minus Operator in $-c$. This unary-minus Three Adress Statement has only two Addresses.

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c.$

$t_4 = b * t_3.$

$t_5 = t_2 + t_4$

$a = t_5.$

	op	arg1	arg2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

Ex 2:- For the expression

$X = -a * b + -a * b$, give Three address code in quadruple format.

$X = -a * b + -a * b.$

Three address code is as follows

$t_1 = \text{minus } a$

$t_2 = t_1 * b.$

$t_3 = \text{minus } a$

$t_4 = t_3 * b.$

$t_5 = t_2 + t_4$

$X = t_5$

	op	arg1	arg2	result
0	minus	a		t_1
1	*	t_1	b	t_2
2	minus	a		t_3
3	*	t_3	b	t_4
4	+	t_2	t_4	t_5
5	=	t_5		X

Eg 3:- Translate $a + -(b + c)$ into Quadraple.

Sol:- Three Address code.

$t_1 = b + c.$

$t_2 = \text{minus } t_1$

$t_3 = a + t_1$

	op	arg1	arg2	result
0	+	b	c	t_1
1	minus	t_1		t_2
2	+	a	t_1	t_3

Triples.

A triple has only three fields op, arg1 and arg2

The result field is used for temporary names.

In triples the result of x op y by its position.

rather than explicit temporary name. The position in agruments is used in parenthesis.

The Triple representation for

$a = -b * -c + b * -c$ is given as follows.

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$.

$t_5 = t_2 + t_4$.

$a = t_5$

	op	arg1	arg2
0	minus	c.	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

In triple representation by placing a in arg1 field and (4) in arg2 field.

Ex 2 → Triple for $a + -(b + c)$

$t_1 = b + c$

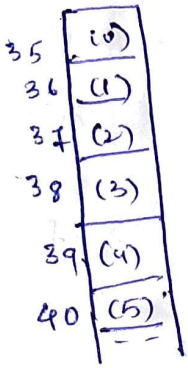
$t_2 = \text{minus } t_1$

$t_3 = a + t_1$

	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

Indirect Triples.

In indirect triples, pointers to keeping triples are kept in instructions rather than triples themselves As below. for $a = b * -c + b * -c$.



	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

With indirect triples consisting of pointers to triple, The Optimization Compiler can move an instruction by reordering the instruction list, Without affecting the triples

Ex 2 → Construct indirect triple for following

$(a+b) * (a+b) - (a+b) * d.$

$t_1 = a + b.$

$t_2 = t_1 * t_1$

$t_3 = t_2 - t_1$

$t_4 = t_3 * d$

	op	arg1	arg2
0	+	a	b
1	*	(0)	(0)
2	-	(1)	(0)
3	*	(2)	d

triple.

10	(0)
11	(1)
12	(2)
13	(3)

indirect triple.

Static Single Assignment Form. (SSA)

→ SSA form is an intermediate representation that facilitates code optimization.

→ SSA is different from Three address code in two aspects.

i. All Assignments is SSA, to variables with distinct names, hence it is static assignment.

following figure shows differences b/w Three Address and SSA.

$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$ ⇒	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_2$
a) Three Address code.	b) SSA.

Note that in SSA, the subscripts distinguish each definition of variables p and q.

The same variable may be defines twice in control flow path programs like

```
if (flag)
  x = 1;
else
  x = 1;
y = x * a;
```

If we use two names for x, x_1 in true part and x_2 in false part, Then which name we should use in $y = x * a$. SSA uses a notational called as ϕ- function, to combine two definition of x, as follows.

```
if (flag)
    x1 = -1 ;
else
    x2 = 1 ;
x3 = φ(x1, x2);
y = x3 * a
```

Here $\phi(x_1, x_2)$ returns value x_1, if control passes through true part, and x_2, if control passes through false part.

Types and Declaration.

Type checking uses rules to reason about behaviores of a program at runtime. Type checking ensures that type of operands match the type expected by operator, For exp, && operator in java expects the operands to be boolcans.

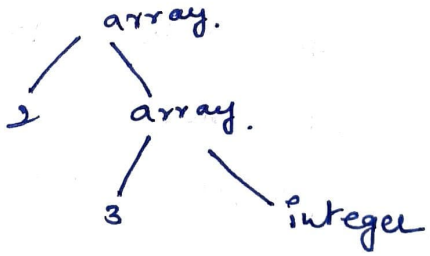
Translations Applications.

From the type of a name, a compiler can determine the storage needed for that name at run time. It also needs to calculate the address.

Type Expressions.

Types have structure, which is represented using type expressions, which is either a basic type or formed by applying called as a type constructor The set of basic types and constructors depend on the language to be checked.

Ex:- Array type int [2][3] can be read as "array of 2 arrays 3 integers each", and written by type expression array(2, array(3, integer)). And this can be represented by following tree



The Operator array takes two parameters a number and a type.

Type expression includes

1. A basic type is a type expressions. Basic types includes boolean, char, integer, float and void
2. A type name is type expressions.
3. A type expression is formed by array constructer
4. A record is a data structure with named fields. Type expression can be formed by record type constructer to field names and their types.
5. Type expression can be formed by using type constructer $\rightarrow$ for function types. We write $s \rightarrow t$ for "function from type s to type t'.
6. If s and t are type expressions then their Cartecian product $s \times t$ is a type expression.
7. Type expressions can contain type variables, whose values are type expressions.

Declarations

The following grammar

$D \rightarrow T$ id ; $D \mid \epsilon$

$T \rightarrow B\,C \mid$ record '{' D '}'

$B \rightarrow$ int | float.

$C \rightarrow \epsilon \mid$ [num] C.

deals with array types and Basic Types.

→ Non Terminal D generates sequence of declarations.

→ Non Terminal T generates Basic, Array or record types.

→ Non Terminal C (Component) generates strings of zero or more integers surrounded by brackets

→ An Array Type consists of Basic type B, followed by Array Component. Specified by C

→ Record type is a sequence of declarations for the fields surrounded by curly braces.

Storage Layout for Local Names

From the type of a name, we can determine the amount of storage that will be needed at run time. This info is used to allocate the Addresses. The type and the relative address are saved in Symbol tabel entry for the name.

The width of a Type is the number storage units needed for objects for that type. Basic types like Charecter, integer, float requires some number of bytes. Storage for array is allocated in one contigeous block of bytes.

Sequence of Declarations.

Languages like C and Java allows declaration to be processed as a group. But in Java declarations can be distributed within Java procedure, But these can be processed before procedure is excuted. We use a offset address to keep track of next available addresses.

Following figure shows Semence of declarations of the form T id, where T generates a type.

Before first declaration is concidered, offset is 0. As each new name x is seen, x is entered into Symbol table, and current value of offset which is incremented by width of type of x.

P →
D {offset = 0;}

D → T id : {top.put(id.lexeme, T.type, offset), offset = offset + T.width}

D1

D → ∈.

The Semantic Action within the production

$D \rightarrow T\ id\ ;\ D_1$ creates a symbol table entry by executing

top.put (id.lexeme, T.type, offset)

Here top denotes current top symbol table. The method top.put creates a symbol table entry for id.lexeme, with T.type *, and offset address.

Fields in Records and classes.

Records can be Added to the Grammar by adding following production

$$T \rightarrow \text{record}\ '\{'\ D\ '\}'$$

The fields in record are specified by D

→ The field names in a record must be distinct.

→ The offset or relative address for a field name is relative to the area of the record.

Ex:- The use of a name x for a field within record does not conflict with other same name outside the record. Following shows x declarations Three times,

```
float x
record { float x; float y; } p
record { int flag; float x; float y; } q;
```

A subsequent assignment

$X = p.x + q.x$, sets variable x to sum of two fields named x in records p and q. The relative addresses of x in p differs from relative address of x in q.

Type checking.

To do type checking a compiler needs to assign a type expression to each component of the source program. Type system is a collection of logical rules which must be used while type checking.

Type checking is used to catch errors in program. Generally Type checking is done dynamically. A sound type system eliminates the need of dynamic checking for errors, because it can determine statically that these errors cannot occur when the target program runs. A program is known as strongly typed, if compiler generates the program which runs without errors.

Rules for Type checking.

* Type checking is of two forms

1. Sythesis.
2. inperence.

Type Synthesis builds up the type of an expression from the type of Sub Expressions. The type of "E1 + E2" is defined interms of E1 and E2

Type inperence determines the type of a language construct depending up on the way it is used. if x is a list, and if it is empty null(x), tells that x is empty list, even We don't know the type of x.

Type Conversions.

Conceder an expression x+i. where x is of float type, i is of type integer. Since both Operands are of two different types and + needs both Operands of same type then the addition occurs. Integers are converted to floats when necessary using unary operator (float).

type's can , and r.

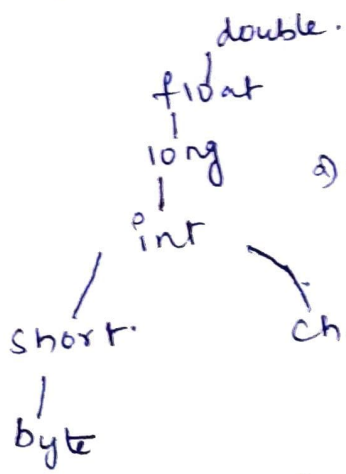
For ex, The following expression 2 * 3.14 can be written as

$t_1 = float(2)$

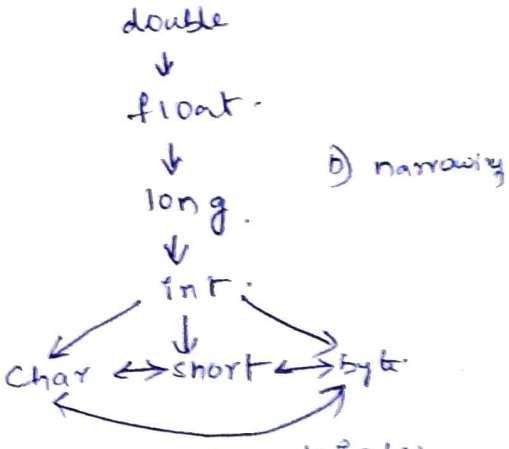
$t_2 = t_1 * 3.14$.

Type conversions rules vary from language to language. The rules for java are Widening which preserve the info, and narrowing which loses the information.

Widening rules can be given by following hirarchy.



a) Widening.



b) narrowing

Any lower type can be Widened to higher type. For ex char can be converted to int or float but can not be Widened to short.

In narrowing a type 's' can be narrowed to a type 't', if there is path between s and t.

Note that char, short, byte are pair Wise Convertible to each other.

Conversion from one type to another is said to be implicit if it is done automatically this is known as Coersion. In Explicit Conversion programmar needs to write something to cause Conversion. Explicit type conversion is known as casting.

Overloading of Functions and Operators.

Overloaded symbols has different meanings depending up on context. Overloading can be resolved depending up on occerance.

Ex:- The + Operator in Java denotes either String Concatenation or addition depending up on type of Operands.

Overloading can be done in functions also,

Concidee functions

```
void err()
{...
}
```

```
void err(String s)
{
...
}
```

Compiler Choses a function depending up on the Number and type of arguments.

Control Flow

The if-else statement and while-statements uses Boolean expression. Boolean expressions are often used to

1. Alter the flow of control. → Boolean expr are used as conditional expressions. For ex in if (E) S, E is a boolean expr, which might be true if we want to execute S.

2. Compute logical values.

→ Boolean expr can represent true or false values. Such Bodean expr can be evaluated using Three address instructions with logical Operators.

Boolean Expressions.

Boolean expressions are composed of boolean Operators like AND (&&), OR (||), NOT (!). Relational expressions are of the form E1 rel E2, where E1 and E2 are arithmetic Expressions.

Boolean expressions are generated by following grammar

$$B \rightarrow B \| B \mid B \&\& B \mid !B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{False}$$

We use rel.op to represent any of the six Comparision operators from $<, <=, ==, !=, >$ or $>=$.

|| and && are left-associative, || has lowest precedence then &&, then !.

→ Given Expression is $B_1 \| B_2$, if B_1 is true we can conclude that entire Expr is true without Evaluating B_2, Similarly in $B_1 \&\& B_2$, if B_1 is false, Then entire expr is false irrespective of B_2.

Short-Circuit Code.

Short-circuit code also known as jumping code, The Boolean expressions &&, || and ! are translated into Jumps.

For ex, in following statement

```
if (x<100 || x>200 && x!=y)
    x=0;
```

This code can be translated as follows.

```
if  X<100   goto L2.
ifFalse  X>200   goto L1
ifFalse  X != y   goto L1
L2 : X = 0
L1 :
```

The Boolean expr is true if control reaches to Label L2. if Expr is false, control goes immediately to L1, skipping L2, and X=0. This is known as short circuit evaluation or jumping.

Flow-of-Control Statements.

Following Grammar generates Three address code for boolean expressions.

S → if (B) S1

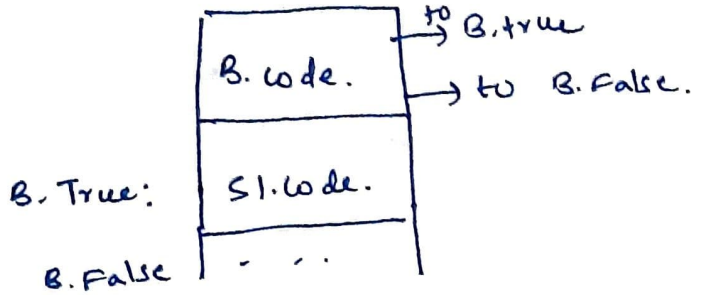
S → if (B) S1 else S2

S → while (B) S1.

Non Terminal B represent Boolean Expression and Non Terminal S represent statement. Both B and S has Synthesized Attribute .code; which

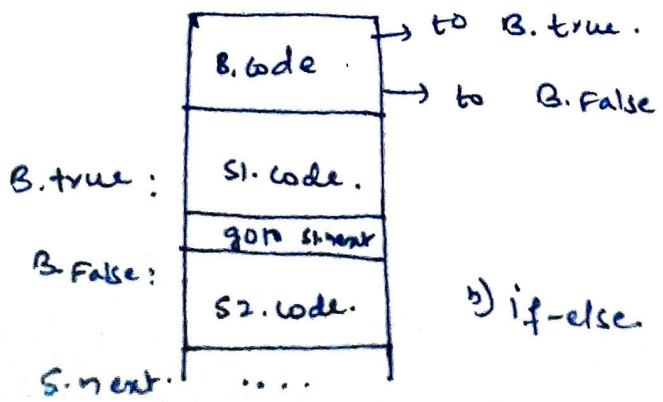
gives Translation into Three address code

The translation of if (B) S1, consists of B.code followed by S.code, as shown below. B.code has jumps based on values of B. if B is true, control flows to first instruction of S1.code, if B is false control flows to instruction immediately to instruction following S1.code

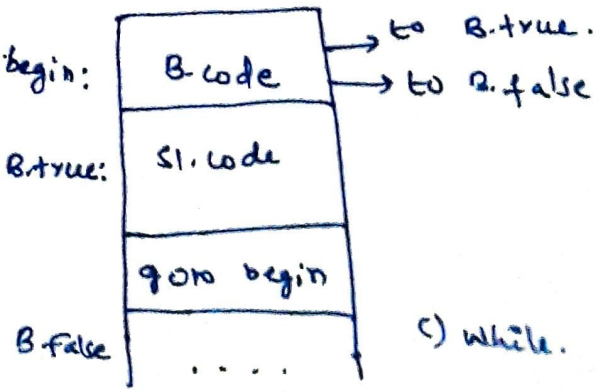


(a) if.

With Every Boolean expression Two labels are attached B.True, and B.False. Control will flows to these labels depending up on B's values. The Statement S has inherited attribute S.next denoting label for instruction immediately after the code for S.



b) if-else.



c) while.

The Three Address code for following Boolean Expression can be given as below.

```
if (x<100 || x>200 && x!=y)
    X= 0;
```

⟹

```
if   x<100   goto  L2
goto  L3

L3: if   x>200   goto  L4.
    goto  L1

L4: if   x!=y   goto  L2
    goto L1.

L2 :  X=0.

L1 :
```

Switch Statements.

Switch is a multiway selection statement. It contains a selector expression E, which is to be evaluated, and followed by n constant values $V_1, V_2, \ldots V_n$, including a default value., which always matches the expression if no other value does. It's Syntax can be given as follows.

```
Switch (E)
{
Case V1 : S1
Case V2 : S2
   :
Case Vn-1 = Sn-1
default: Sn
}
```

Translation of Switch Statement.

The switch code is translated as follows.

1. Evaluate the Expression E
2. Find the value V_j in the list that has value as Expression E.
3. Evaluate the statement S_j, associated with value found. If no match found Evaluate

statement in default.

In 'switch' if no of cases are small say 10 at most, then we can use conditional Jumps, which tests an individual value and transfers to the code for the corresponding statement.

Another method is to use a table pair, each pair consisting of a value and label for the corresponding statement's code.

Syntax-Directed Translation of Switch-Statements.

Following is a Translation of switch statement. The tests all appear at the end so that a simple generator can recognize multiway branch and generate efficient code for it. When we see keyword switch, we generate two new labels test, next and a temporary t. We generate code to Evaluate E into t. After processing E, we generate jump goto test.

```
    code to Evaluate E into t.
    goto test
L1: code for S1
    goto next
L2: code for S2
    goto next
    .
    .
    .
Ln-1: code for Sn-1
    goto next
Ln: code for Sn
    goto next.
test: if t = V1 goto L1
      if t = V2 goto L2
      .
      .
      .
      if t = Vn-1 goto Ln-1
      goto Ln
next:
```

Intermediate code for procedures. (Functions)

Foll
for

In Three Address code a function call is converted into evaluation of parameters in preparation of call, and call itself.

For ex, Consider below instruction in which a function with array of integers, and an assignment as below

$$n = f(a[i])$$

This can be translated to following Three Address code

1) $t_1 = i * 4$.
2) $t_2 = a[t_1]$
3) Param t_2
4) t_3 = Call 'f,'
5) $n = t_3$.

→ First two lines ~~returns~~ Computes the value of $a[i]$ and stores into temporary t_2.

→ Line 3 makes t_2 as actual parameter, which will be actual parameter for call of f at line 4., This lines return the value of to temporary t_3, and line 5 assigns the result of $a[i]$ to n.

Following an SDD to define functions and function calls.

$D \rightarrow$ define. T id $(F)\{S\}$.

$F \rightarrow \epsilon \mid T$ id , F.

$S \rightarrow$ return E ;

$E \rightarrow$ id (A)

$A \rightarrow \epsilon \mid E, A$.

Non Terminals D and T generates declarations and Types respectively.

→ Function definition generated by D, consists of keyword define, a return Type (T), Function name (id), formal parameters (F) in parenthesis, and function body consisting of statements (s) in curly braces.

→ Formal parameter F generates zero or more formal parameters which consists of Type and identifier

→ Non Terminal S and E generate statements and Expressions respectively.

→ E adds function call with Actual parameters generated by A. An Actual parameter is an Expression.