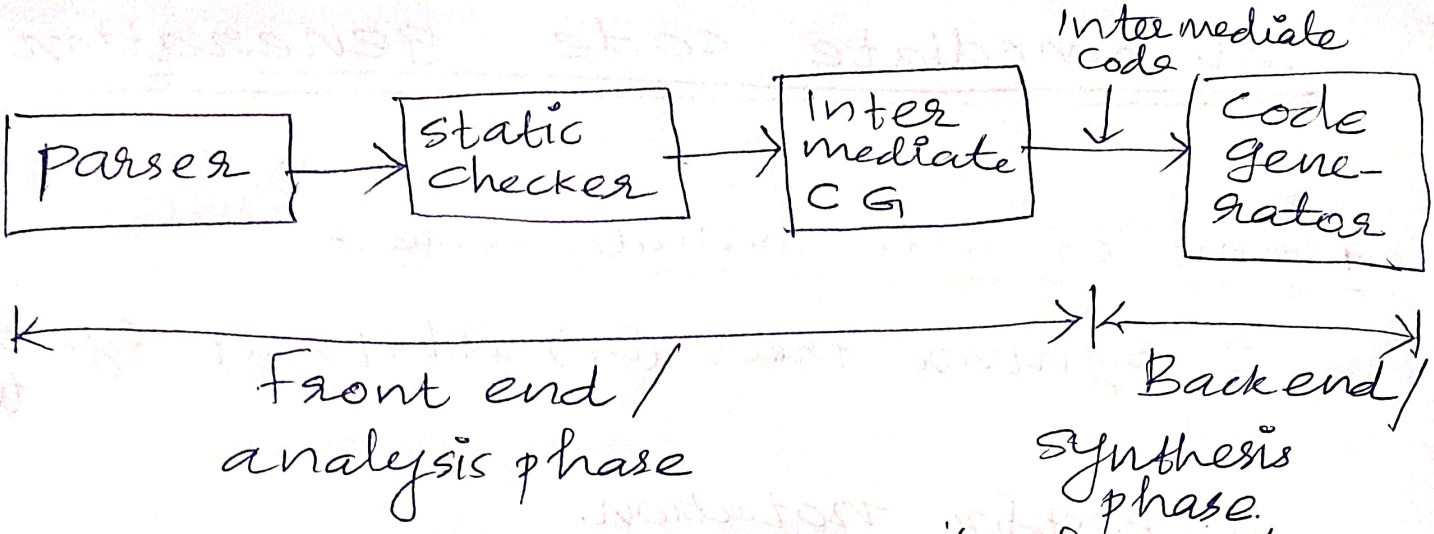
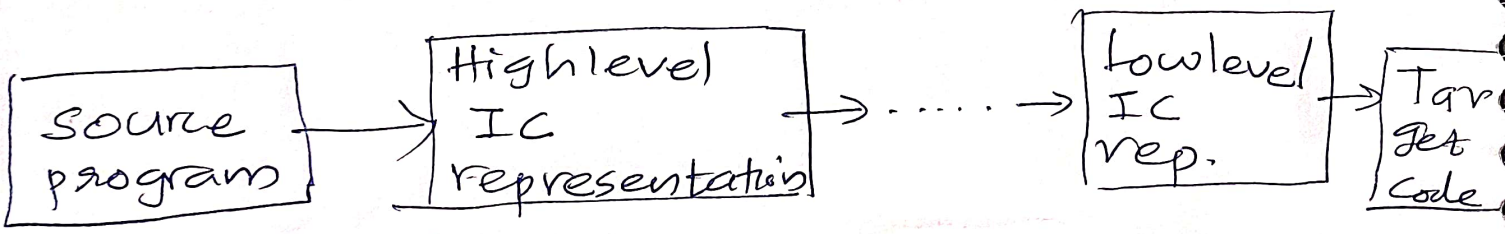


Fig: Logical structure of a compiler frontend.

- Static checking includes type checking, any syntactic checks that remain after parsing.



- Highlevel representations are close to the source language and low-level are close to the target code.
- Syntax trees are high level and are suited for type checking.

- Low-level is suited for m/c-dependent tasks like register allocation and instruction selection.

- intermediate codes are m/c independent codes, but they're close to m/c instructions.

- The given program ~~is con~~ in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.

- intermediate language can be,
  - syntax trees
  - postfix notation
  - Three address code

- some programming languages have well defined intermediate languages:
  - Java
  - prolog

Intermediate language types:

Graphical IRs:

- Abstract syntax trees
- DAGs
- Control Flow Graphs

Linear IRs:

- Stack based (postfix)
- Three_address code (quadraples)

DAG (Directed Acyclic Graphs)

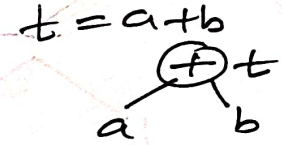
or DAG representation of a Basic Block.

DAG represents the structure of a basic block.

→ In a DAG, internal node represents operators.

→ leaf node represents identifiers, constants.

→ Internal node also represents result of Expression.



Applications of DAG

1.) Determining the common sub-expression.

2) Determining which names are used inside the block and computed outside the block.

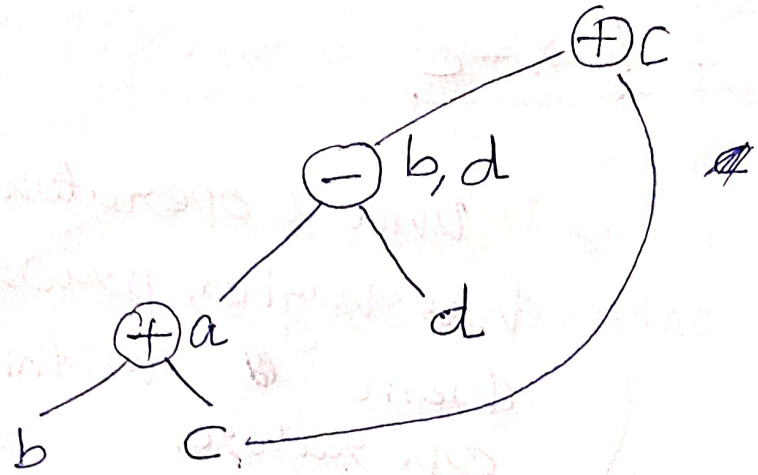
3) Determining which statements of the block could have their computed value outside the block.

4. simplify list of quadraples
by eliminating common sub.
expression.

Construction of DAG

(2) Construct DAG for the block

1) $a = b + c$
2) $b = a - d$
3) $c = b + c$
4) $d = a - d$



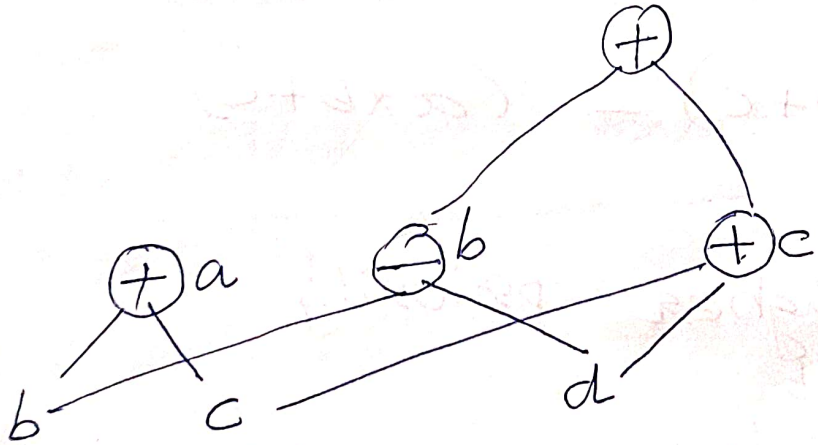
(3)

$a = b + c$

$b = b - d$

$c = c + d$

$e = b + c$



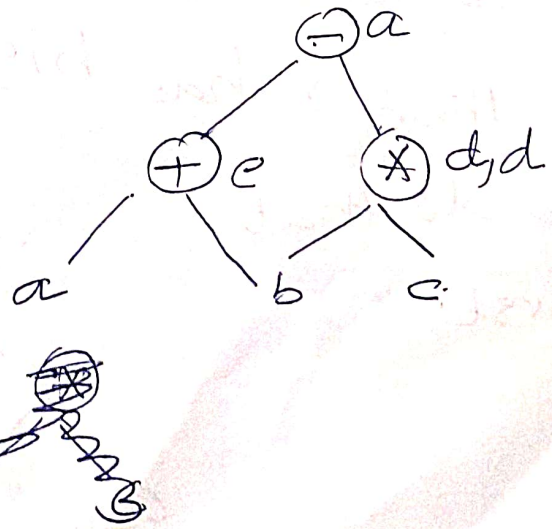
(4)

$d = b * c$

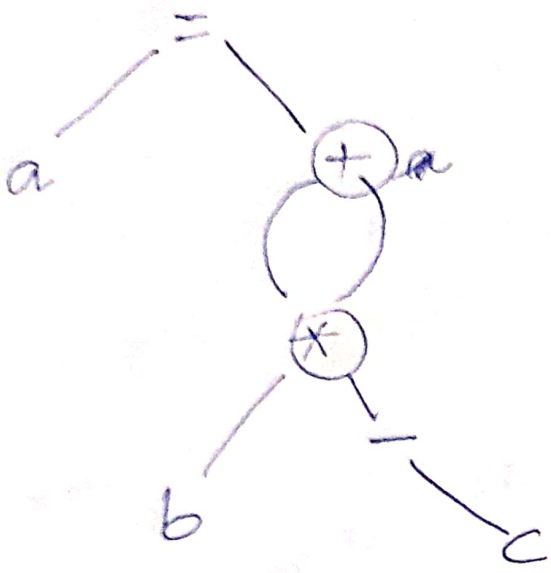
$e = a + b$

$b = b * c$

$a = e - d$



⑤ $a = \underline{b * -c} + \underline{b * -c}$



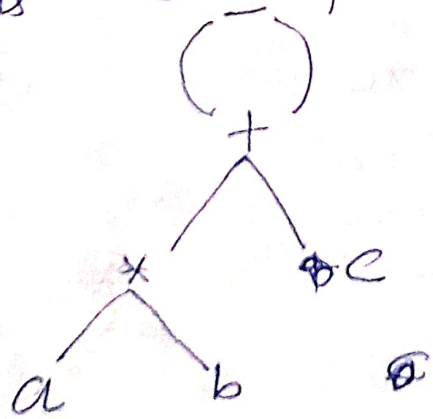
unary operator '-' has higher priority than arithmatic operators

⑥ $a = (a * b + c) - (a * b + c)$

Bracket has higher priority so do that first.

then * has higher priority than +.

so do that first.



Three - Address Code

In a three-address code, there is atmost one operator on the right side of an instruction. Each instruction should contain atmost 3 instructions;

→ Quadruples
→ Triples
→ Indirect Triple.

Consider a source language expression

$x + y * z$;

this might be translated into the sequence of three-address instructions:

$t_1 = y * z$
$t_2 = x + t_1$

where t_1 and t_2 are compiler generated temporary names.

Three-address code is built from two concepts: addresses and instructions.

An address can be one of the following:

1) Name

Allow source-program names to appear as addresses in 3AC. A source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.

2) constant: a compiler must deal with many different types of constants.

3) A compiler-generated temporary. (t_1)

Temporaries are needful to create a distict name each time.

Common 3AC instructions.

1) Assignment instructions of the form $x = y \text{ op } z$, x, y, z – addresses
op – operators

2) $x = \text{op } y$, op – unary operation

3) Copy instructions of the form $x = y$.

4) An unconditional jumb goto L.
L is the next to be executed.

5) Conditional jumb of the form
if x goto L

6) Conditional jumps such as
if x relop y goto L.

7) procedure calls and returns
using param x for parameters.
call p, n and y = call p, n for
procedure ~~return value~~. and
function.

8) Indexed copy instructions.

9) Address and pointer assignments
of the form $x = \&y$, $x = *y$ and $*x = y$

① Quadruples

A quadruple has 4 fields

op, arg1, arg2, and result

op — contains an internal code for the operator.

Eg: $x = y + z$

op	arg1	arg2	result
+	y	z	x

Exceptions:-

i) Instructions like unary operators as $x = -y$ or $x = y$, do not use arg_2.

ii) operators like 'param' use neither arg_2 nor result.

iii) conditional and unconditional jumps put the target label in result.

Example

$a = b * -c + b * -c$

$t_1 = -c$ // minus has higher priority (← unary op.)

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

or

$t_3 = b * t_1$

$t_4 = t_2 + t_3$

⇨ 3AC

quadruple representation

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

Here, we're using too many temporary variables. Inorder store these temp. variables, we need more memory. Inorder to overcome this situation, 'triple' approach can use.

(2) Triple

It requires 3 fields,

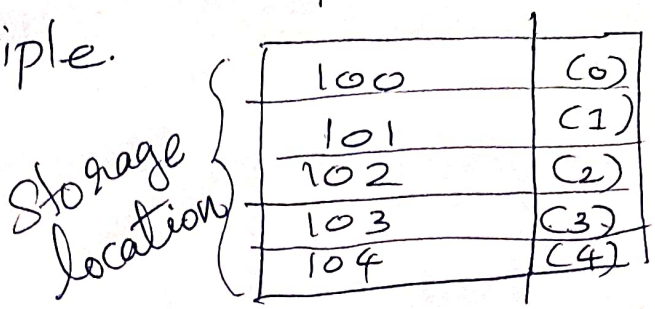
op, arg1 and arg2 (shows the address of previous one)

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

(3) Indirect Triple

pointer to triple.

pointer table.



+ triple table also.

Indirect triples consists of a listing of pointers to triples, rather than a listing of triples themselves.

Example: we can use an array 'instruction' to list pointers to triples in the desired order.

Then triple can be represented as pointer table ~~and~~ along with triple table.

Example (2)

$A = -B * (C + D)$

i) convert the given expr. to 3AC.

$t_1 = -B$
$t_2 = C + D$
$t_3 = t_1 * t_2$
$A = t_3$

ii) represented in quadraples.

	op	arg1	arg2	result
(0)	−	B	-	t_1
(1)	+	C	D	t_2
(2)	*	t_1	t_2	t_3
(3)	=	t_3	-	A

iii) Triples

	op	arg1	arg2
(0)	−	B	
(1)	+	C	D
(2)	*	(0)	(1)
(3)	=	A	

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

3AC

$t_1 = a+b$

$t_2 = c+d$

$t_3 = t_1 * t_2$

$t_4 = t_1 + c$

$t_5 = t_3 - t_4$

Quadraples

	op	arg1	arg2	result
(0)	+	a	b	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	+	t_1	c	t_4
(4)	-	t_3	t_4	t_5

Triples

	op	arg1	arg 2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	+	(0)	c
(4)	−	(2)	(3)

Indirect triple

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)