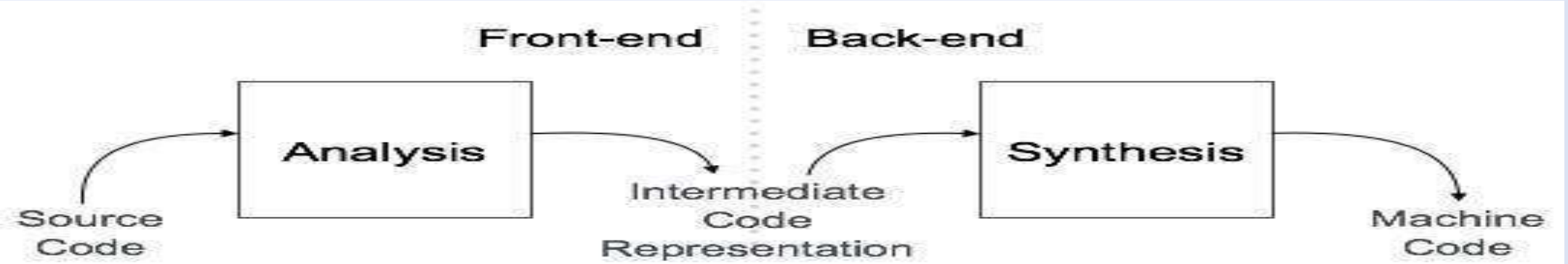
Unit-3
Intermediate Code Generation

Introduction to Intermediate Code Generation

Analysis-Synthesis Model



Analysis Phase

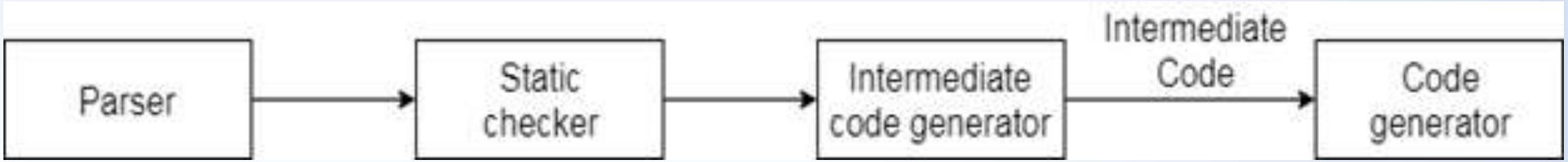
- Known as the front-end of the compiler, which reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors and generates an intermediate representation of the source program.
- This intermediate representation is fed to the Synthesis phase as input.

Synthesis Phase

Known as the back-end of the compiler, which generates the target program with the help of intermediate source code representation and symbol table.

Intermediate Code

Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.

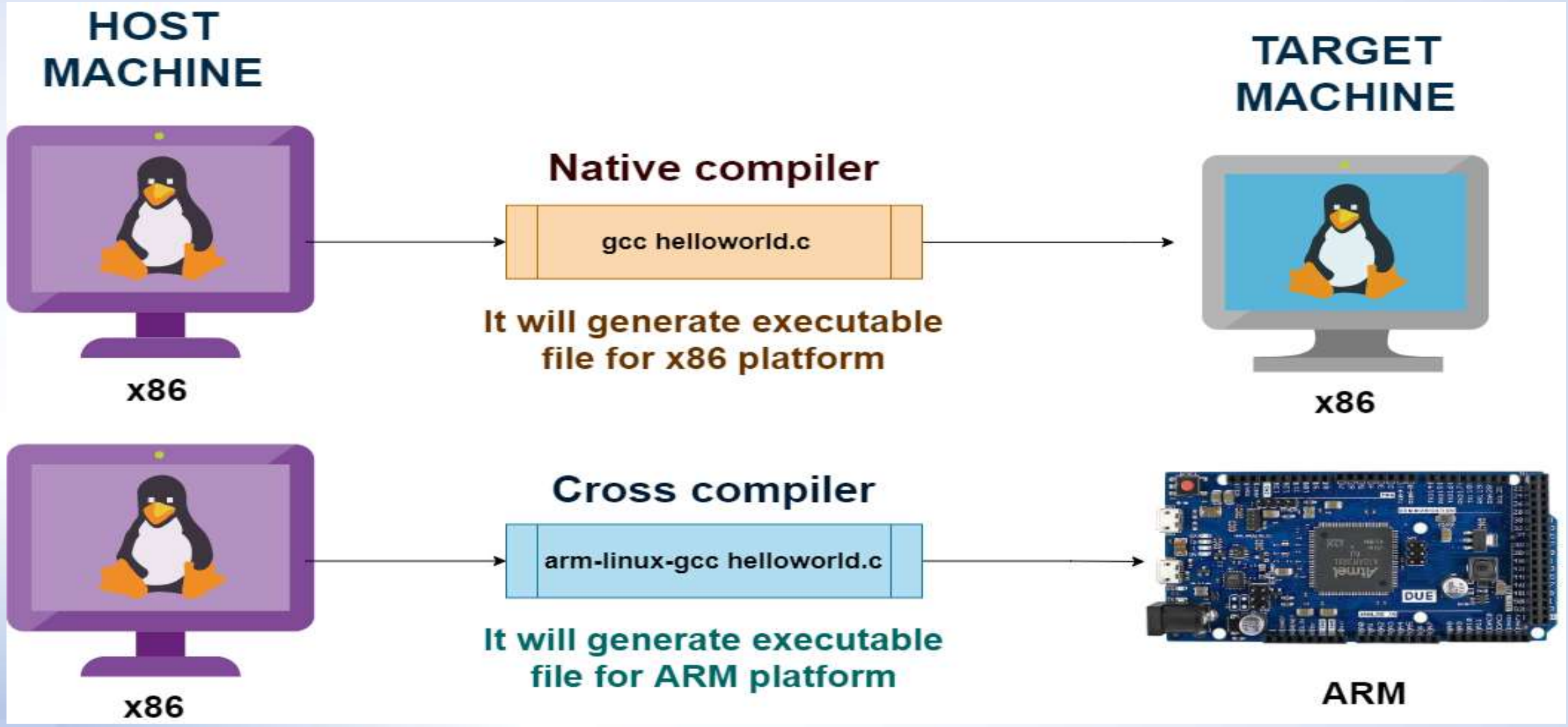


•If the compiler directly translates source code into the machine code without generating intermediate code then a **full native compiler** is required for each new machine.

•The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine.

•Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.

Native Compiler vs Cross Compiler



Intermediate Representation

Intermediate code can be represented in two ways:

1. High Level intermediate code:

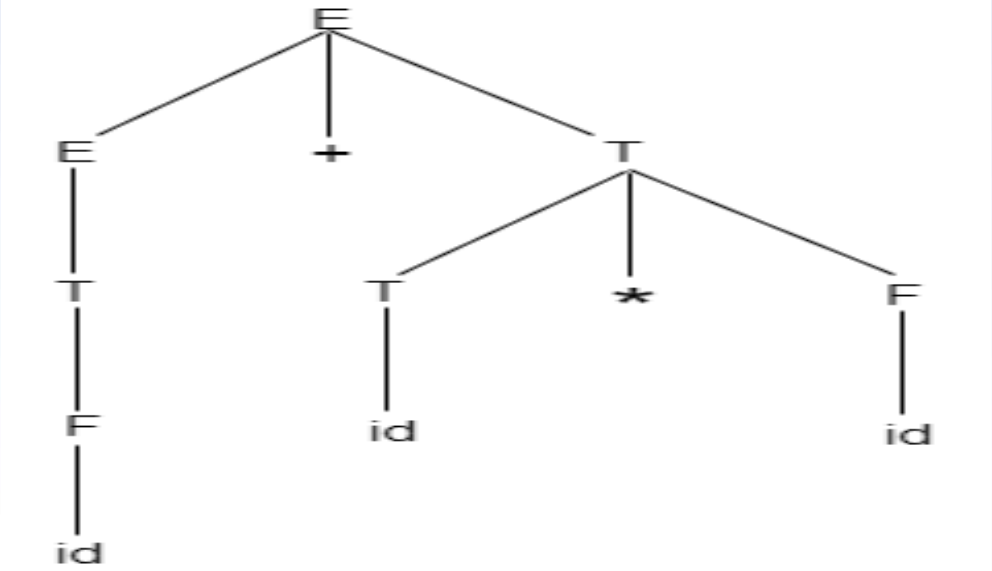
High level intermediate code can be represented as source code. To enhance performance of source code, we can easily apply code modification. But to optimize the target machine, it is less preferred.

2. Low Level intermediate code:

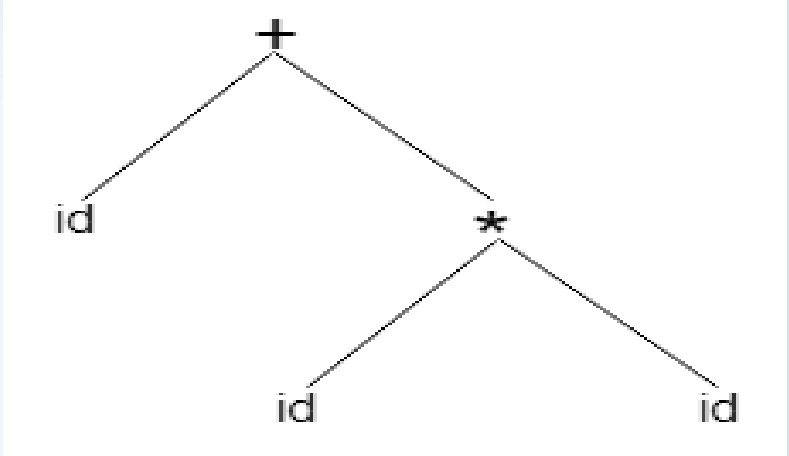
Low level intermediate code is close to the target machine, which makes it suitable for register and memory allocation etc. it is used for machine-dependent optimizations.

Parse tree and Syntax tree

Parse Tree



Syntax Tree



- In the parse tree, most of the leaf nodes are single child to their parent nodes.
- In the syntax tree, we can eliminate this extra information.
- Syntax tree is a variant of parse tree. In the syntax tree, interior nodes are operators and leaves are operands.
- Syntax tree is usually used when represent a program in a tree structure.

Variants of Syntax Trees

- Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct.
- A **Directed Acyclic Graph** (hereafter called a DAG) for an expression identifies the common subexpressions of the expression.
- DAG's can be constructed by using the same techniques that construct syntax trees.

Directed Acyclic Graph

A **DAG** for basic block is a directed acyclic graph with the following labels on nodes:

- The leaves of graph are labeled by unique identifier and that identifier can be variable names or constants.
- Interior nodes of the graph is labeled by an operator symbol.
- Nodes are also given a sequence of identifiers for labels to store the computed value.

    - DAGs are a type of data structure. It is used to implement transformations on basic blocks.
    - DAG provides a good way to determine the **common sub-expression**.
    - It gives a picture representation of how the value computed by the statement is used in subsequent statements.

Construction of DAGs

Rule-01:

- In a DAG, Interior nodes always represent the operators.
- Exterior nodes (leaf nodes) always represent the names, identifiers or constants.

Rule-02:

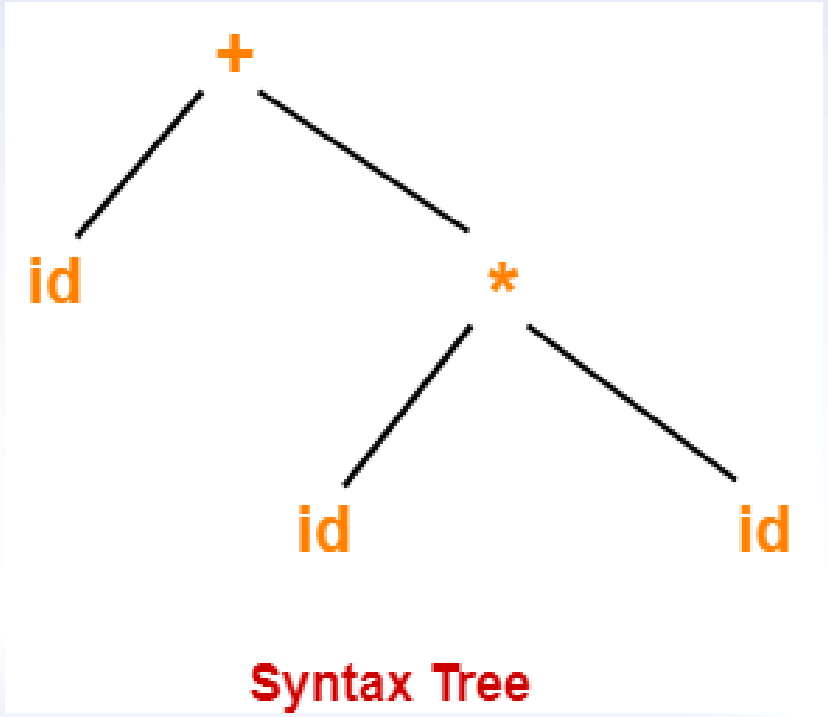
While constructing a DAG,

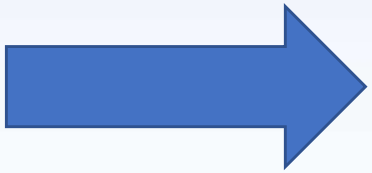
- A check is made to find if there exists any node with the same value.
- A new node is created only when there does not exist any node with the same value.
- This action helps in detecting the common sub-expressions and avoiding the re-computation of the same.

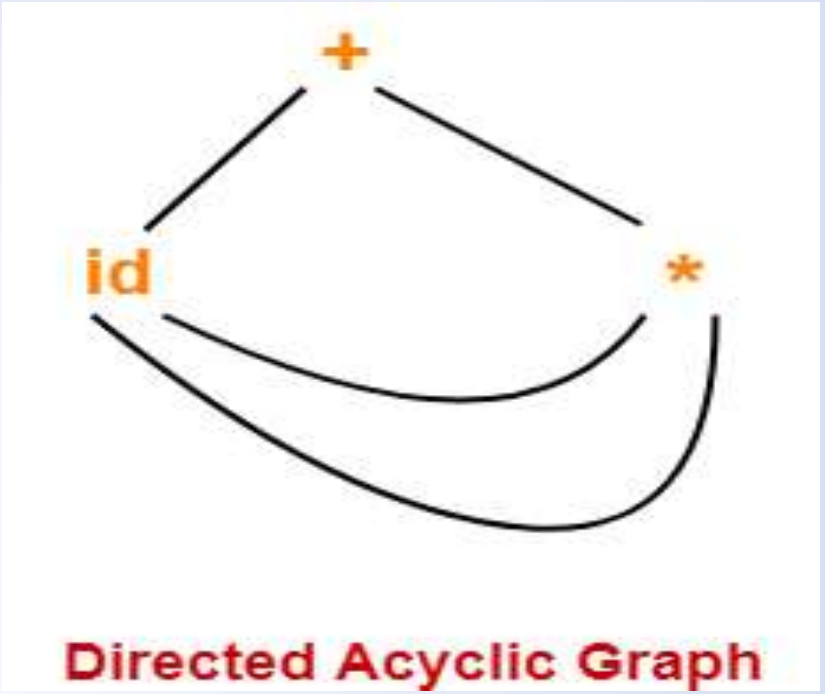
Rule-03:

- The assignment instructions of the form x:=y are not performed unless they are necessary.

DAG Example



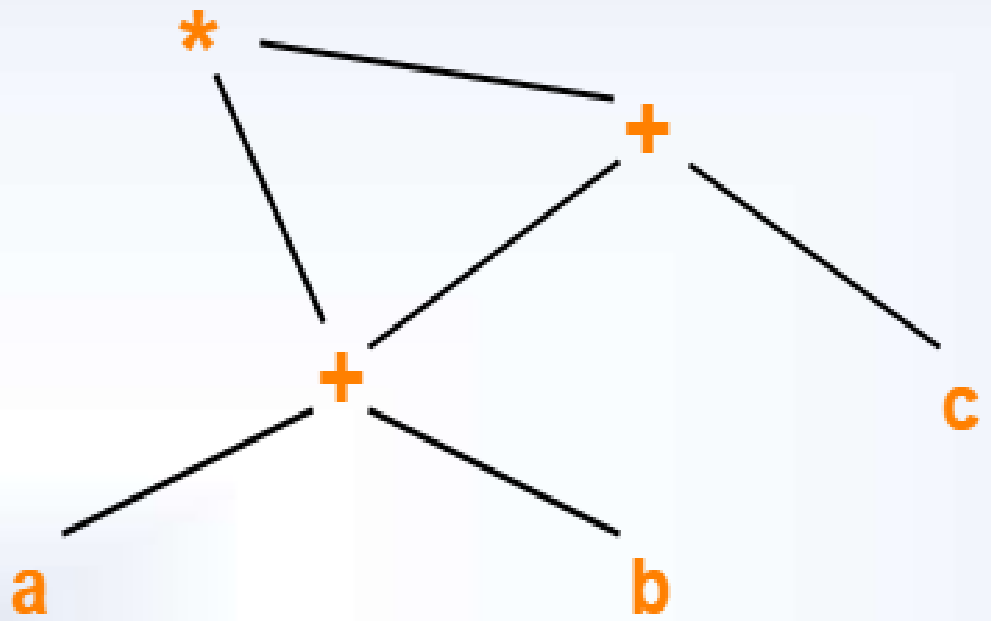




DAG Example

Consider the following expression and construct a DAG for it-

(a + b) * (a + b + c)

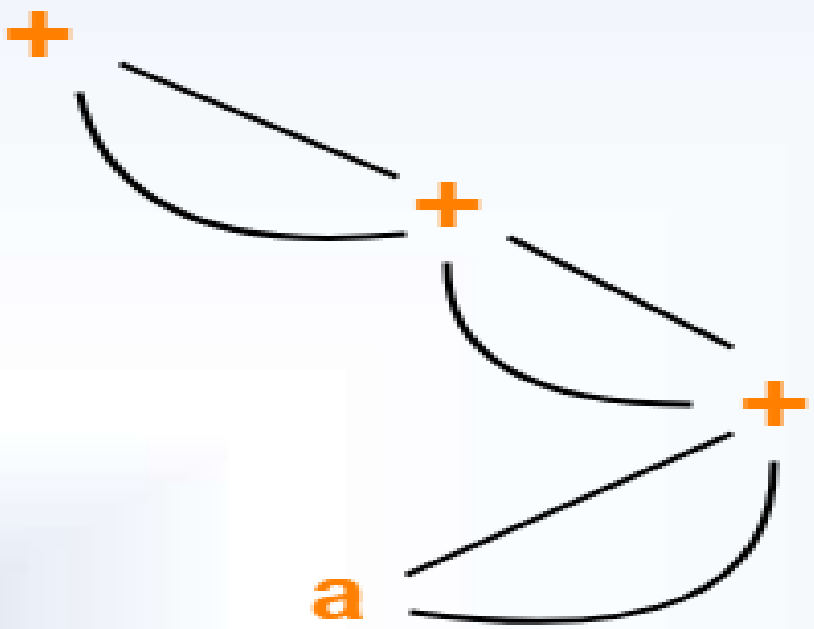


Directed Acyclic Graph

DAG Example

Consider the following expression and construct a DAG for it-

(((a + a) + (a + a)) + ((a + a) + (a + a)))

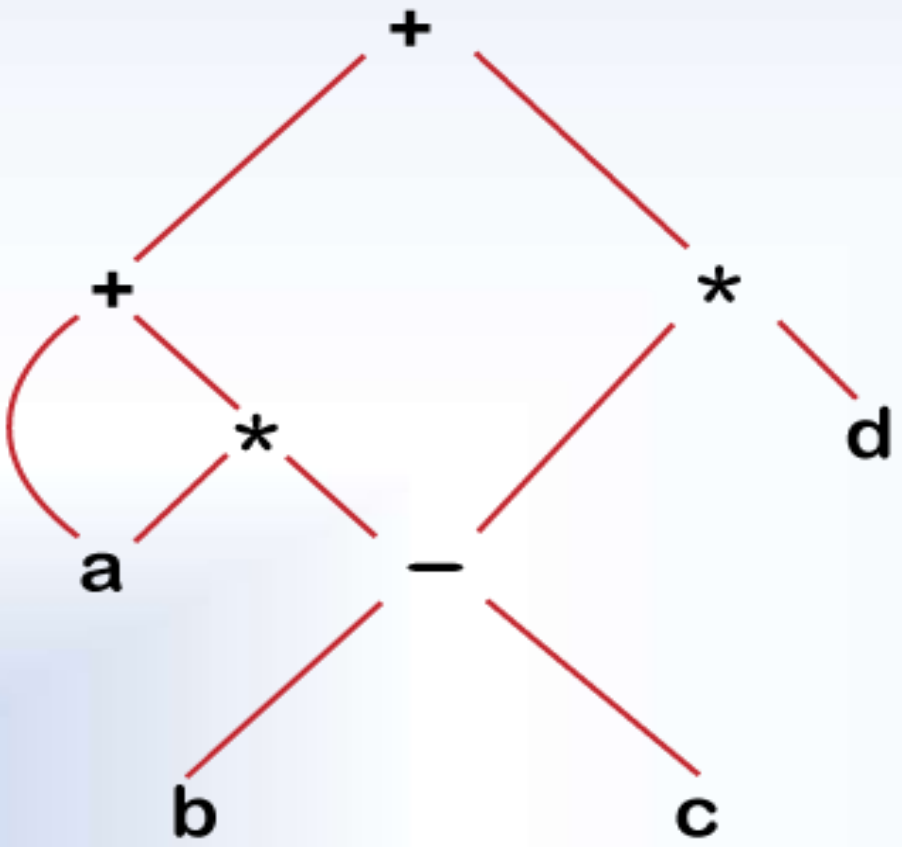


Directed Acyclic Graph

DAG Example

Consider the following expression and construct a DAG for it-

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



SDD for creating DAG's

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 -> (E)	T.node = E.node
5) T -> id	T.node = new Leaf(id, id.entry)
6) T -> num	T.node = new Leaf(num, num.val)

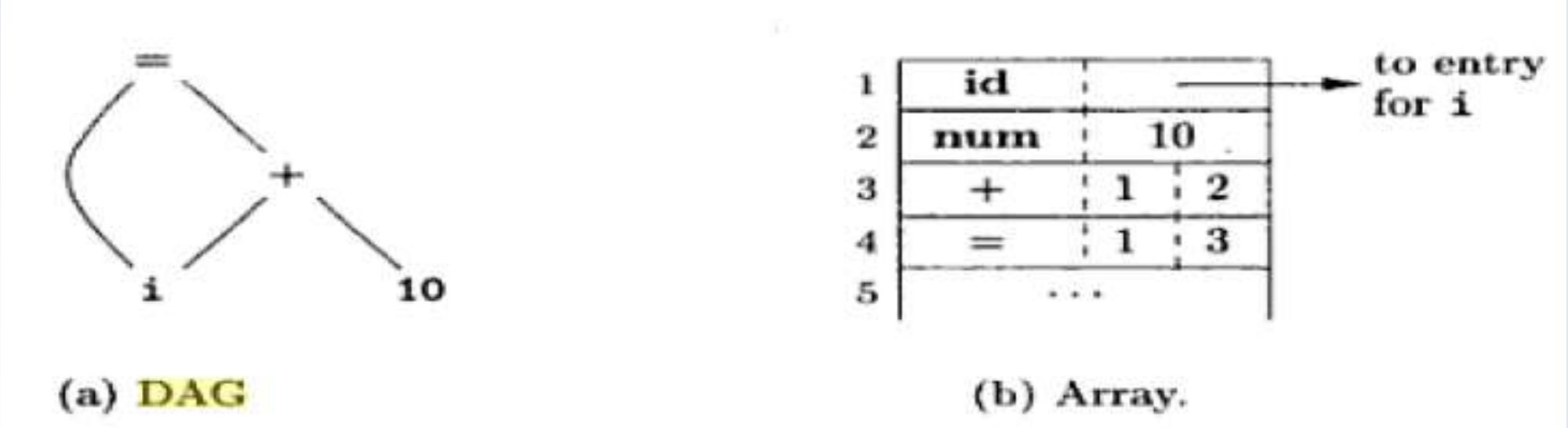
Example:

1)p1=Leaf(id, entry-a)
2)p2=Leaf(id, entry-a)=p1
3)p3=Leaf(id, entry-b)
4)p4=Leaf(id, entry-c)

5)p5=Node('-',p3,p4)
6)p6=Node('*',p1,p5)
7)p7=Node('+',p1,p6)
8) p8=Leaf(id,entry-b)=p3

9) p9=Leaf(id,entry-c)=p4
10) p10=Node('-',p3,p4)=p5
11) p11=Leaf(id,entry-d)
12) p12=Node('*',p5,p11)
13) p13=Node('+',p7,p12)

The Value-Number Method for Constructing DAG's



- The tree or DAG can be stored in an array, one entry per node.
- The array index, rather than a pointer, is used to reference a node.
- This index is called the node's value-number and the triple
  <op, value-number of left, value-number of right> is called the signature of the node.
- When Node(op,left,right) needs to determine if an identical node exists, it simply searches the table for an entry with the required signature.
- Searching an unordered array is slow. Hash tables are a good choice.

Postfix Notation

- Postfix notation is the useful form of intermediate code if the given language is expressions.
- Postfix notation is also called as 'suffix notation' and 'reverse polish'.
- Postfix notation is a linear representation of a syntax tree.
- In the postfix notation, any expression can be written unambiguously without parentheses.
- The ordinary (infix) way of writing the sum of x and y is with operator in the middle: x * y. But in the postfix notation, we place the operator at the right end as xy *.
- In postfix notation, the operator follows the operand.

Postfix Notation-Example

Example

1. The postfix representation of the expression (a – b) * (c + d) + (a – b) is :

ab – cd + *ab -+.

2.The postfix representation of the expression (((1 + 2) * 3) + 6) / (2 + 3) is :

1 2 + 3 * 6 + 2 3 + /

Three Address Code

- Three-address code is an intermediate code. It is used for optimizing compilers.
- In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

a = b op c

Here,

- a, b and c are the operands.
- Operands may be constants, names, or compiler generated temporaries.
- op represents the operator.

Examples of Three Address instructions are-

a = b + c

c = a * b

Three Address Code-Examples

1.Write Three Address Code for the following expression-
a = b + c + d

Three Address Code for the given expression is-
(1) T1 = b + c
(2) T2 = T1 + d
(3) a = T2

Three Address Code-Examples

2. Write Three Address Code for the following expression-
-(a * b) + (c + d) – (a + b + c + d)

Three Address Code for the given expression is-

(1) T1 = a * b
(2) T2 = uminus T1
(3) T3 = c + d
(4) T4 = T2 + T3
(5) T5 = a + b
(6) T6 = T3 + T5
(7) T7 = T4 – T6

Three Address Code-Examples

3) GivenExpression: a := (-c * b) + (-c * d)

Three-address code is as follows:

t1 := -c
t2 := b*t1
t3 := -c
t4 := d * t3
t5 := t2 + t4
a := t5

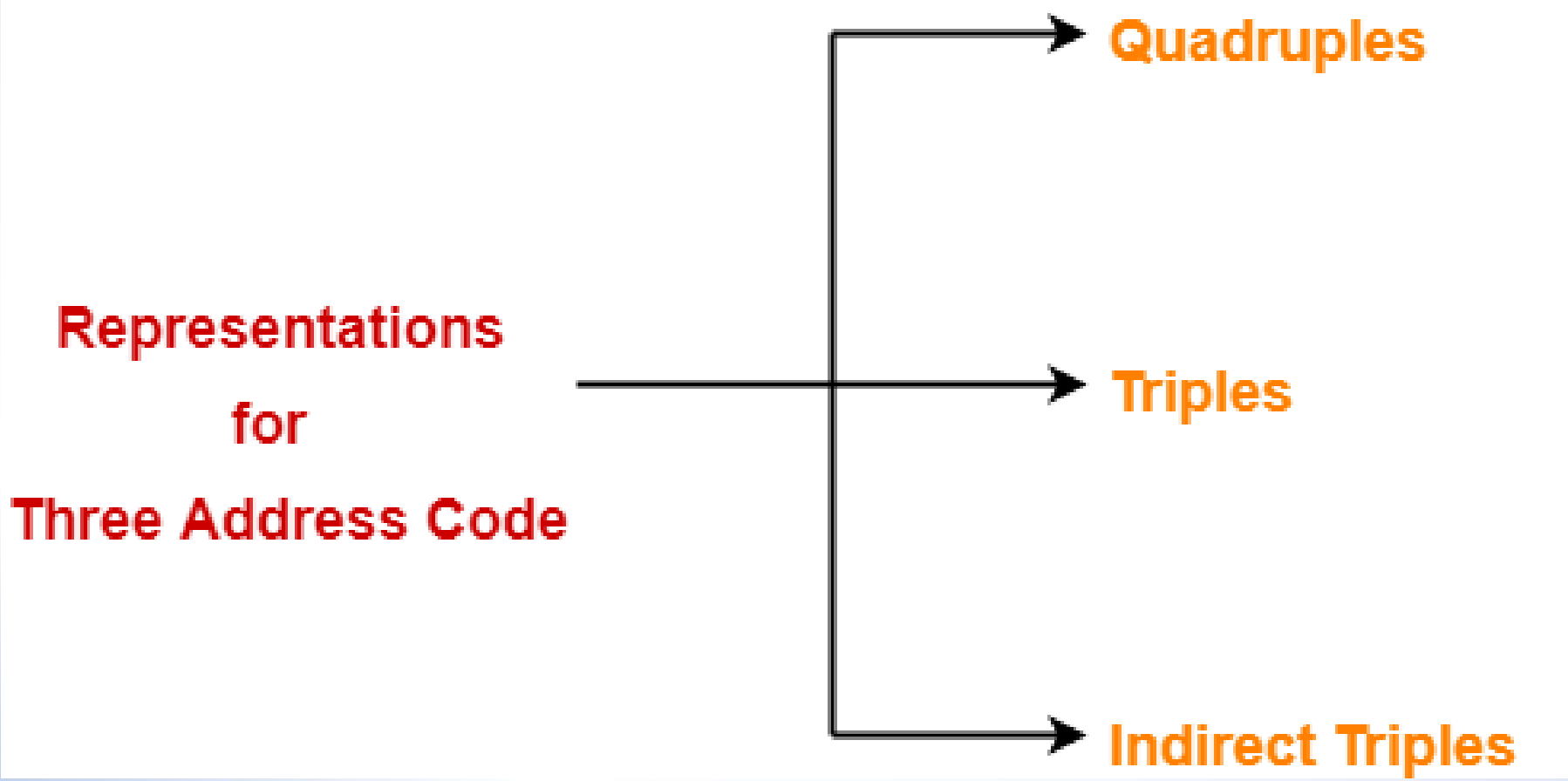
Types of Three Address Statements

For expressing the different programming constructs, the three address statements can be written in different standard formats and these formats are used based on the expression.

- x = y op z
- x = op y
- x = y
- goto L
- if x relop y goto L
- Procedure calls using:
  - param x
  - call p,n
  - y = call p,n
- x = y[i] and x[i] = y
- x = &y and x = *y and *x =y

Representation of Three Address Statements

The commonly used representations for implementing Three Address Code are-



Quadruples

In quadruples representation, each instruction is splitted into the following 4 different fields-

op, arg1, arg2, result

Example

a := -b * c + d

Three-address code is as follows:

```
t1 := -b
t2 := c + d
t3 := t1 * t2
a := t3
```

Quadruple Representation

	Operator	Source 1	Source 2	Destination
(0)	uminus	b	-	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3	-	a

Triples

In triples representation,

- References to the instructions are made.
- Temporary variables are not used.

Example

a := -b * c + d

Three-address code is as follows:

```
t1 := -b
t2 := c + d
t3 := t1 * t2
a := t3
```

Triple Representation

	Operator	Source 1	Source 2
(0)	uminus	b	-
(1)	+	c	d
(2)	*	(0)	(1)
(3)	:=	(2)	-

Indirect Triples

- This representation is an enhancement over triples representation.
- It uses an additional instruction array to list the pointers to the triples in the desired order.
- Thus, instead of position, pointers are used to store the results.
- It allows the optimizers to easily re-position the sub-expression for producing the optimized code.

Indirect Triples

Example

a := -b * c + d

Three-address code is as follows:

```
t1 := -b
t2 := c + d
t3 := t1 * t2
a := t3
```

Indirect Triple Representation

	Statement
41	(0)
42	(1)
43	(2)
44	(3)

	Operator	Source 1	Source 2
(0)	uminus	b	-
(1)	+	c	d
(2)	*	(0)	(1)
(3)	:=	(2)	-

Example-1

Example

b * minus c + b * minus c

Three address code

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

Quadruples

op	arg1	arg2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Triples

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

Indirect Triples

	op
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

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

Example-2

a = b* – (c – d) + b* – (c – d)

Quadruple

Operator	Opr1	Opr2	Result
–	c	d	T_1
U	T_1		T_2
*	b	T_2	T_3
–	c	d	T_4
U	T_4		T_5
*	b	T_5	T_6
+	T_3	T_6	T_7
=	T_7		A

Triple

Statement No	Operator	Arg 1	Arg 2
(0)	–	c	d
(1)	U	(0)	
(2)	*	b	(1)
(3)	–	c	d
(4)	U	(3)	
(5)	*	b	(4)
(6)	+	(2)	(5)
(7)	=	a	(6)

Example-2

a = b* – (c – d) + b* – (c – d)

Indirect Triple

Sequence	Statement No
(0)	(20)
(1)	(21)
(2)	(22)
(3)	(23)
(4)	(24)
(5)	(25)
(6)	(26)
(7)	(27)

Statement No	Operator	Arg 1	Arg 2
(20)	–	c	d
(21)	U	(20)	
(22)	*	b	(21)
(23)	–	c	d
(24)	U	(23)	
(25)	*	b	(24)
(26)	+	(22)	(25)
(27)	=	a	(26)