Three-Address Code:

Three-address code is a sequence of statements of the general form

$$x := y \mathit{op} z$$

where x, y and z are names, constants, or compiler-generated temporaries;*op*stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into a sequence

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

where t_1 and t_2 are compiler-generated temporary names.

Advantages of three-address code:

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.

- The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged – unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three-address statements.

Three-address code corresponding to the syntax tree and dag given above

(a) Code for the syntax tree	(b) Code for the dag
$t_1 := -c$	$t_1 := -c$
$t_2 := b * t_1$	$t_2 := b * t_1$
$t_3 := -c$	$t_5 := t_2 + t_2$
$t_4 := b * t_3$	$a := t_5$
$t_5 := t_2 + t_4$	
$a := t_5$	

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three-Address Statements:

The common three-address statements are:

1. Assignment statements of the form**x : = yopz**, where*op*is a binary arithmetic or logical operation.

2. Assignment instructions of the form**x : =opy**, where*op*is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.

3.*Copy statements*of the form**x : = y**where the value of*y*is assigned to*x*.

4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.

5. Conditional jumps such as **if** ***x relop y*** **goto L**. This instruction applies a relational operator (<, =, >=, etc.) to*x*and*y*, and executes the statement with label L next if*x*stands in relation

relop to y. If not, the three-address statement following if*x relop* ygoto L is executed next, as in the usual sequence.

6.*param x* and *call p, n*for procedure calls and *return y*, where y representing a returned valueis optional. For example,

param x_1
param x_2
. . .
param x_n
call p,n

generated as part of a call of the procedure p(x_1, x_2, …. ,x_n).

7. Indexed assignments of the form x : = y[i] and x[i] : = y.

8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

Syntax-Directed Translation into Three-Address Code:

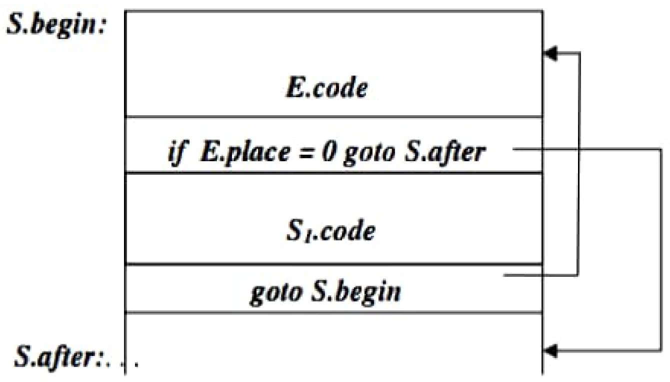
When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. For example,**id : =*E***consists of code to evaluate***E***into some temporary t, followed by the assignment**id**.*place*: =**t.**

Given input a : = b * - c + b * - c, the three-address code is as shown above. The synthesized attribute*S.code*represents the three-address code for the assignment*S.*
The nonterminal*E*has two attributes :
1.*E.place*, the name that will hold the value of*E*, and
code, the sequence of three-address statements evaluating*E.*

Syntax-directed definition to produce three-address code for assignments

PRODUCTION	SEMANTIC RULES
S* → *id : = E	***S.code : = E.code‖gen(id.place ':=' E.place)***
E* → *E₁ + E₂	***E.place := newtemp;* *E.code := E₁.code‖E₂.code‖gen(E.place ':=' E₁.place '+' E₂.place)***
E* → *E₁ * E₂	***E.place := newtemp;* *E.code := E₁.code ‖ E₂.code ‖ gen(E.place ':=' E₁.place '*' E₂.place)***
E* → *- E₁	***E.place := newtemp;* *E.code := E₁.code ‖ gen(E.place ':=' 'uminus' E₁.place)***
E* → *(E₁)	***E.place : = E₁.place;* *E.code : = E₁.code***
E* → *id	***E.place : = id.place;* *E.code : = ' '***

Semantic rules generating code for a while statement



PRODUCTION	SEMANTIC RULES
S→ **while** *E* **do** S_1	*S.begin := newlabel;* *S.after := newlabel;* *S.code := gen(S.begin ':') ||* *E.code ||* *gen ('if' E.place '=' '0' 'goto' S.after)||* *S_1.code ||* *gen ('goto' S.begin) ||* *gen (S.after ':')*

- ➢The function*newtemp*returns a sequence of distinct names $t_1, t_2, \ldots$.. in response tosuccessive calls.
- ➢Notation*gen(x ':=' y '+' z)*is used to represent three-address statement x := y + z. Expressions appearing instead of variables like*x, y*and*z*are evaluated when passed to *gen*, and quoted operators or operand, like '+' are taken literally.
- ➢Flow-of–control statements can be added to the language of assignments. The code for*S* →**while***E***do***S* $_1$ is generated using new attributes*S.begin*and*S.after*to mark the first statement in the code for*E*and the statement following the code for S, respectively.
- ➢The function*newlabel*returns a new label every time it is called.
- ➢We assume that a non-zero expression represents true; that is when the value of*E* becomes zero, control leaves the while statement.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

- A quadruple is a record structure with four fields, which are,***op, arg1, arg2***and***result.***
- The*op*field contains an internal code for the operator. The three-address statement**x : = y op z**is represented by placing*y*in*arg1*,*z*in*arg2*and*x*in*result.*
- The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

- To avoid entering temporary names into the symbol table, we might refer to 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.*
- The fields*arg1*and*arg2*, for the arguments of*op*, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as*triples*.

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

(a) Quadruples

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

(b) Triples

Quadruple and triple representation of three-address statements given above

A ternary operation like x[i] : = y requires two entries in the triple structure as shown as below while x : = y[i] is naturally represented as two operations.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

(a) x[i] : = y

	op	*arg1*	*arg2*
(0)	= [ ]	y	i
(1)	assign	x	(0)

(b) x : = y[i]

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements