THREE ADDRESS CODE

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

  x := y op z

  where x, y, 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.
- In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like x+y*z might be translated into the sequence of three-address instructions

  t1 := y * z

  t2 := x + t1

  where t1 and t2 are compiler-generated temporary names. The use of names for the intermediate values computed by a program allows three address code to be easily rearranged unlike postfix notation.

Implementations 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 we call op, argl, 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.
- Statements with unary operators like x: = – y or x: = y do not use arg2.
- Operators like param use neither arg2 nor result.
- Conditional and unconditional jumps put the target label in result.
- The quadruples for the assignment a: = b * –c + b * –c

$t_1 = \text{minus } c$
$t_2 = b * t_1$
$t_3 = \text{minus } c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

	op	Arg1	Arg2	Result
(0)	minus	c		t1
(1)	*	b	t1	t2
(2)	minus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

- 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.

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

	op	Arg1	Arg2
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

- In Triples, 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).
- In practice, the information needed to interpret the different kinds of entries in the arg1 and arg2 fields can be encoded into the op field or some additional fields.

Indirect Triples:

- Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves.
- This implementation is naturally called indirect triples. For example, let us use an array statement to list pointers to triples in the desired order.

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

	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)

Quadruples Vs Triples Vs Indirect Triples:

- A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around.
- With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change.
- With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result.
- This problem does not occur with indirect triples. Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.