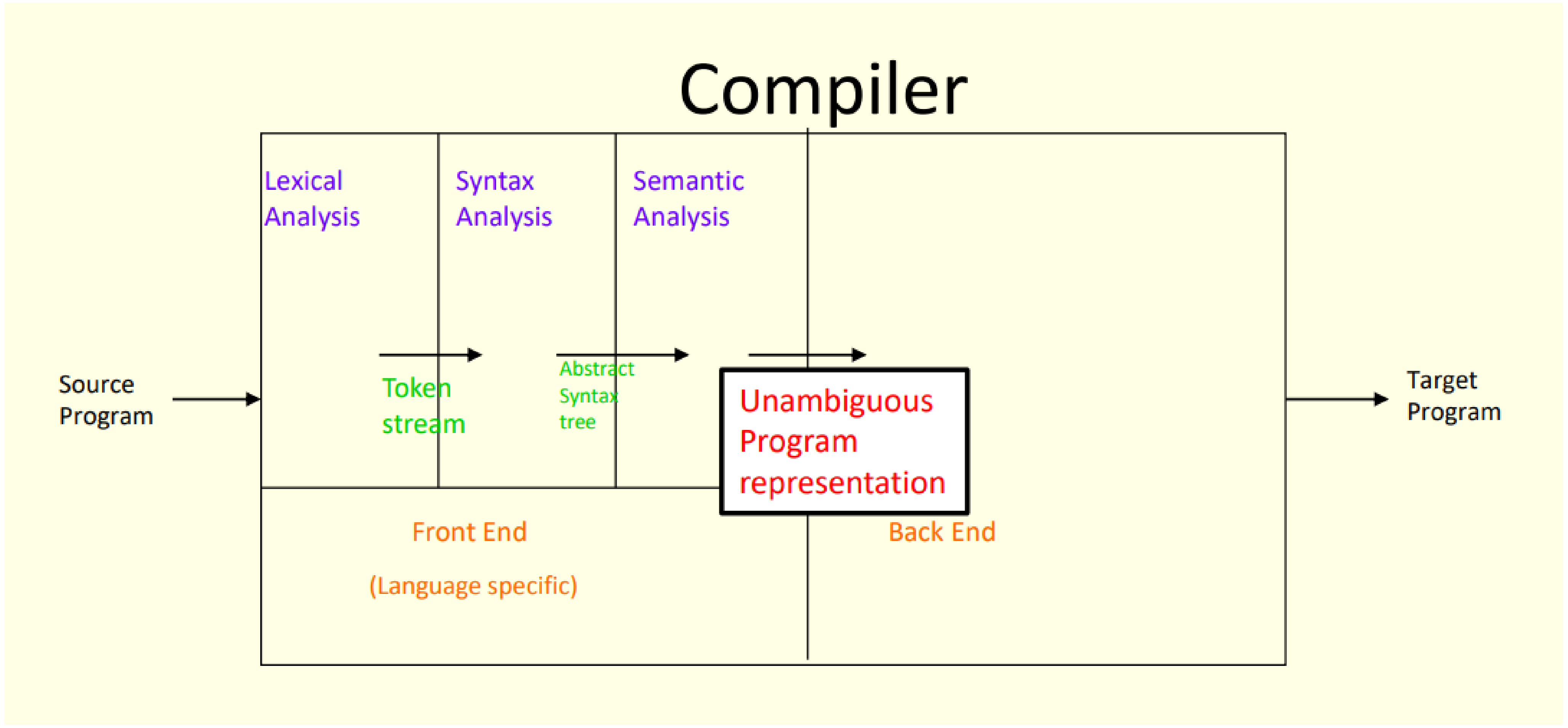
CS 4031
Compiler Construction
Lecture 11

Mahzaib Younas

Lecturer, Department of Computer Science

FAST NUCES CFD

Intermediate Representation



Intermediate Code

- Similar terms: *Intermediate representation, intermediate language*
- Ties the front and back ends together
- Language and Machine neutral
- Many forms
- Level depends on how being processed
- More than one intermediate language may be used by a compiler

Intermediate Representation

- More of a wizardry rather than science
- Compiler commonly use 2-3 IRs
  - HIR (high level IR) preserves loop structure and array bounds
  - MIR (medium level IR)
    - reflects range of features in a set of source languages
    - language independent
    - good for code generation for one or more architectures
    - appropriate for most optimizations
  - LIR (low level IR) low level similar to the machines

Understanding Compiler Design

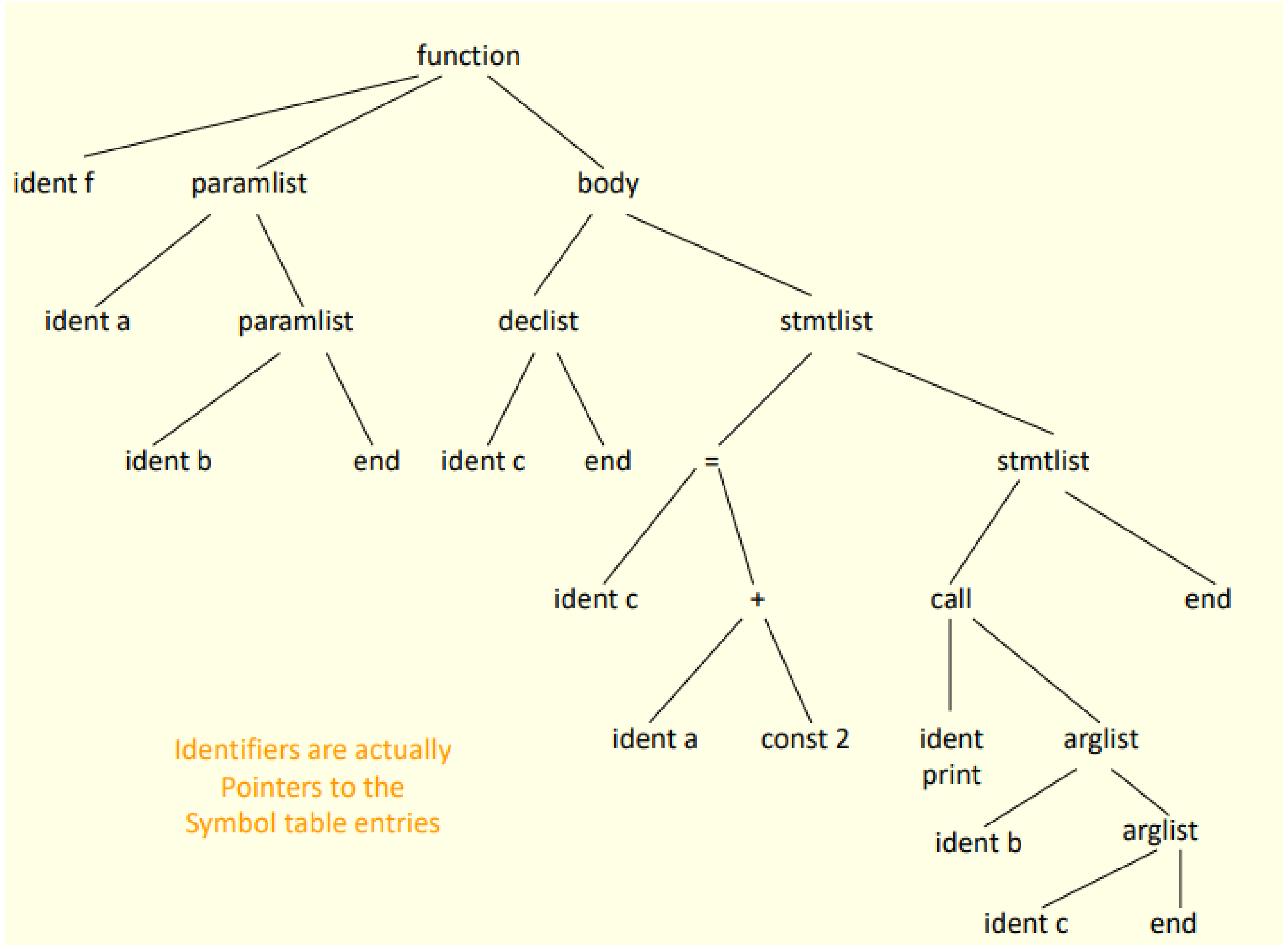
- Source and Target Language
  - A compiler converts a source language (high-level code) into a target language (machine code).
- Porting Cost and Reuse
  - Reusing an existing compiler design can save time and resources but may come with porting challenges (adjusting to a new system).
- Optimization
  - Some compiler architectures allow more efficient code optimizations than others.

High Level IR

```
int f(int a, int b) {
int c;
c = a + 2;
print(b, c);
}
```

- Abstract syntax tree
  - keeps enough information to reconstruct source form
  - keeps information about symbol table

High Level IR



Medium Level IR

- Medium level IR
  - reflects range of features in a set of source languages
  - language independent
  - good for code generation for a number of architectures
  - appropriate for most of the optimizations
  - normally three address code

Low Level IR

- corresponds one to one to target machine instructions
- architecture dependent

Intermediate Language

- Intermediate code can be represented in the following four ways.
  1. Syntax trees
  2. Directed acyclic graph(DAG)
  3. Postfix notation
  4. Three address code

Three address Code

- Three address code is a linear representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.
- Three address code is a sequence of statements of the form A = B OP C where A, B and C are the names of variables, constants or the temporary variables generated by the compiler.
- OP is any arithmetic operation or logical operation applied on the operands B and C.

Example

- Consider the expression a + b * c;
- this expression is expressed as follows:

T1 = b * c

T2 = a + T1

Here T1 and T2 are compiler-generated temporary names.

Example:

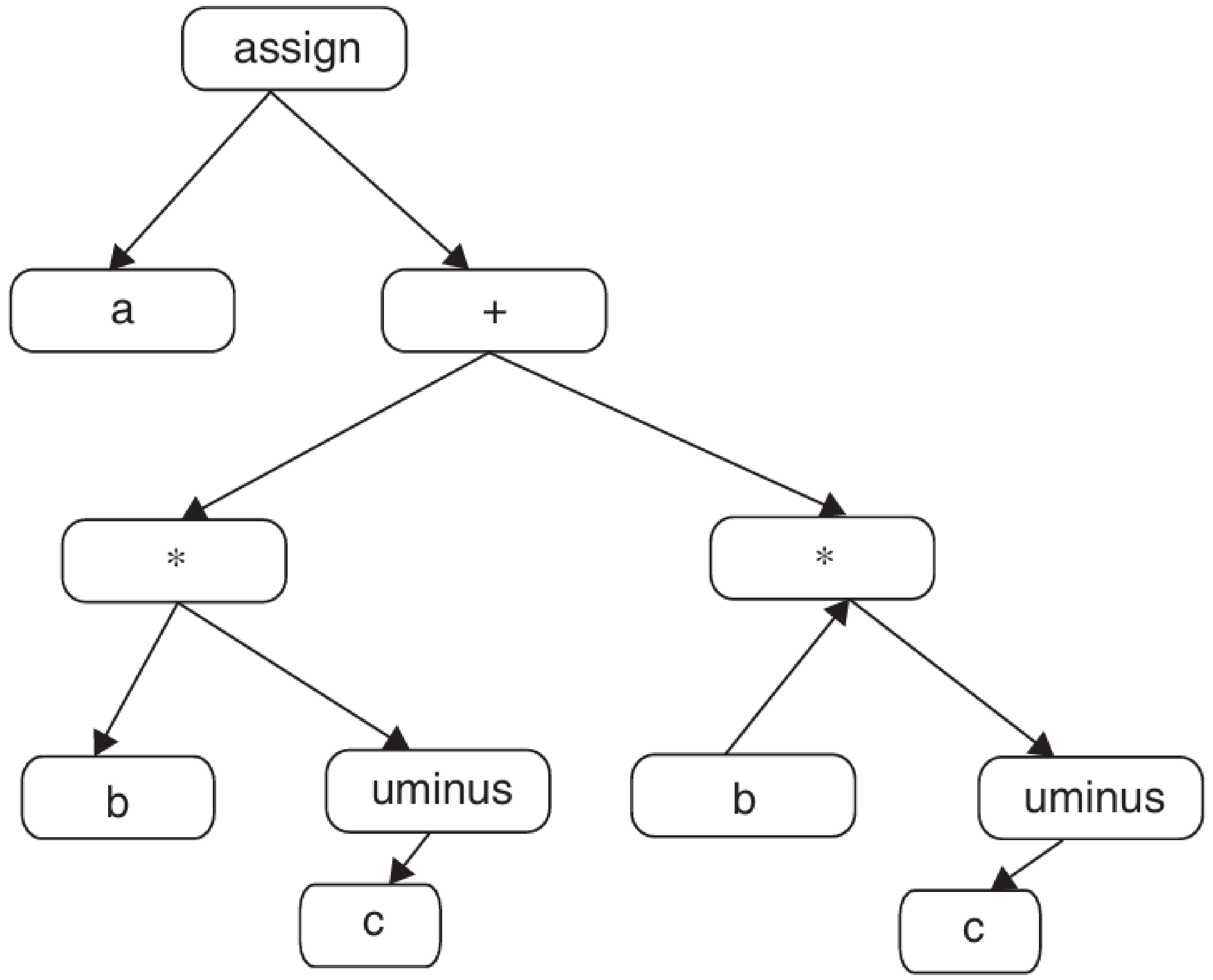
- Generate the three address code for the following C statement

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

- Solution:

```
T1 = c − d
T2 = −T1
T3 = b * T2
T4 = c − d
T5 = −T4
T6 = b * T5
T7 = T3 + T6
a = T7
```

Consider the assignment a: = b* − c + b* − c.
Draw the syntax tree



Types of Three Address Statement

- The three address statements can be written in different standard formats and these formats are used based on the expression.
    1. Assignment statements with binary operator
    2. Assignment statements with unary operator
    3. Copy statements
    4. Unconditional Jumps
    5. Conditional Jumps
    6. Functional calls
    7. Indexed assignments
    8. Address assignments
    9. Pointer assignment

Types of Three Address Statement

1. Assignment statements with binary operator

   They are of the form A := B op C

   where op is a binary arithmetic or logical operation.

2. Assignment statement with unary operator

Assignment statement **x = *op* y**

   where ***op*** is a unary operation, e.g., unary minus, logical negation, shift operators

Types of Three Address Statement

3**. Copy Statement**

Copy statement **x = y**

where value of ***y*** is assigned to **x**

4. Unconditional Jump

The statement such as goto L:

The label L with three address statement is the next statement number to be executed.

Types of Three Address Statements

5. Conditional Jumps

if X relop Y goto L.

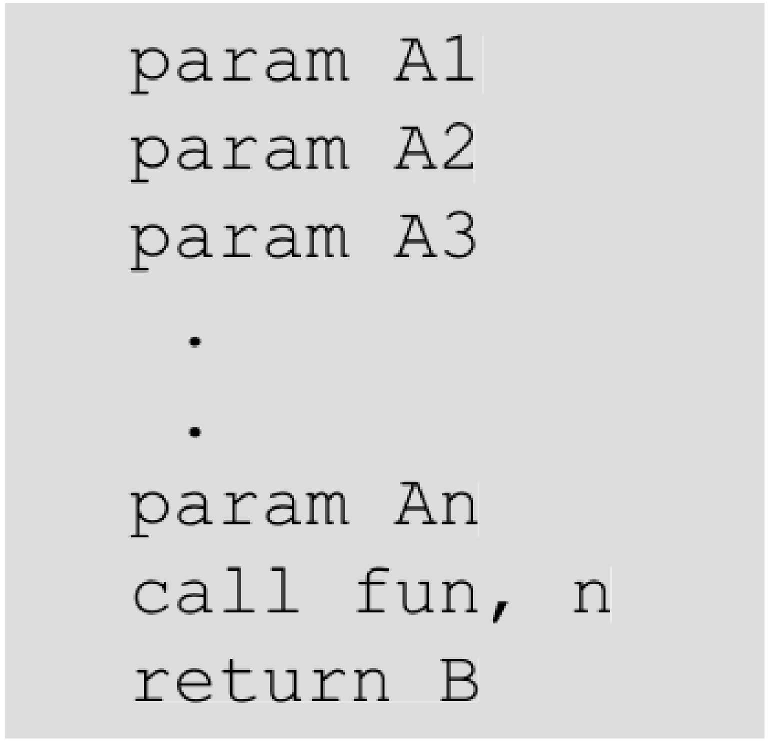
If the condition is satisfied, then this instruction applies a relational operator (<=,>=,) to X and Y and executes the statement with label L else the statement following if X relop Y goto L is executed.

6. Functional Calls

The functional calls are written as a sequence of param A, call fun, n , and return B statements, where A indicates one of the input argument in n arguments to be passed to the function fun that returns B. The return statement is optional.

Example of Function Call

- if the statement is B fun($A_1, A_2, A_3, \ldots, A_n$), then three address statements for it are as follows:



```
param A1
param A2
param A3
  .
  .
param An
call fun, n
return B
```

Indexed Assignment

- The statements of the form A: = B[i] and A [i]: = B are indexed assignments.
    - In the A: = B[i] statement, A is set to the value in the location i memory units beyond location B.
    - A[i]:=B sets the contents of the location i units beyond A to the value of B. In both these instructions, A, B, and i refer to data objects.

Types of Three Address Statement

- Address assignments

  Statement of the form A:= &B, which sets A to the location B.

- Pointer assignment

  Statements of the form A:= *B and *A: = B are included.

For instance, A:= *B sets the value of A to the value pointed to by B. *A:=B changes the location of the value in A to the address pointed by B.

Example of each Statement

Type of Statement	Statement in C	Statement in TAC

Assignment Statements	x = a + b;	t1 = a + b
		x = t1

Assignment with Unary Operator	x = -a;	t1 = -a
		x = t1

Copy Statements	x = a;	x = a
Unconditional Jumps	goto L1;	goto L1

Conditional Jumps (if else)	if (a > b)
	goto L1;
		if a > b
		goto L1

Function Calls	y = call func(x);	param x y = call func

Indexed Assignment	arr[i] = x;	t1 = i * width
		t2 = base + t1
		*t2 = x

Address Assignment	p = &x;	p = &x

Pointer Assignment	x = *c;	t1 = *c
		x = t1

Representation of Three Address Code

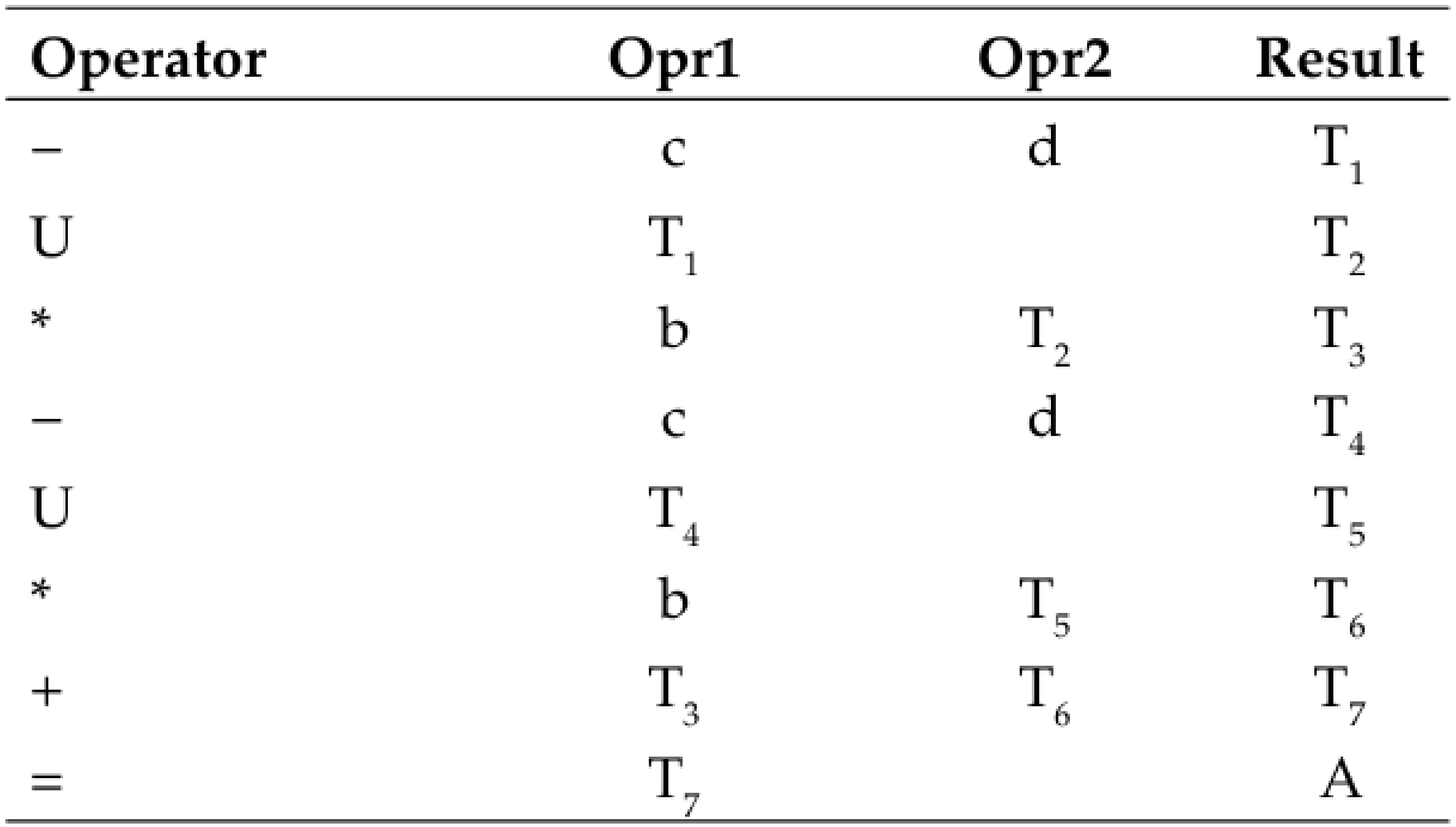
- Three address codes can be represented in special structures

1. Quadruple
2. triple
3. indirect triple.

Quadruple

- A quadruple is a record structure with four fields.
- First field is to store the operator, the second and third fields are for the operands used in the operation, and the fourth field is for the result. F.

Example = $a = b^* - (c - d) + b^* - (c - d)$



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

We use U for unary minus and – as it is for binary minus.

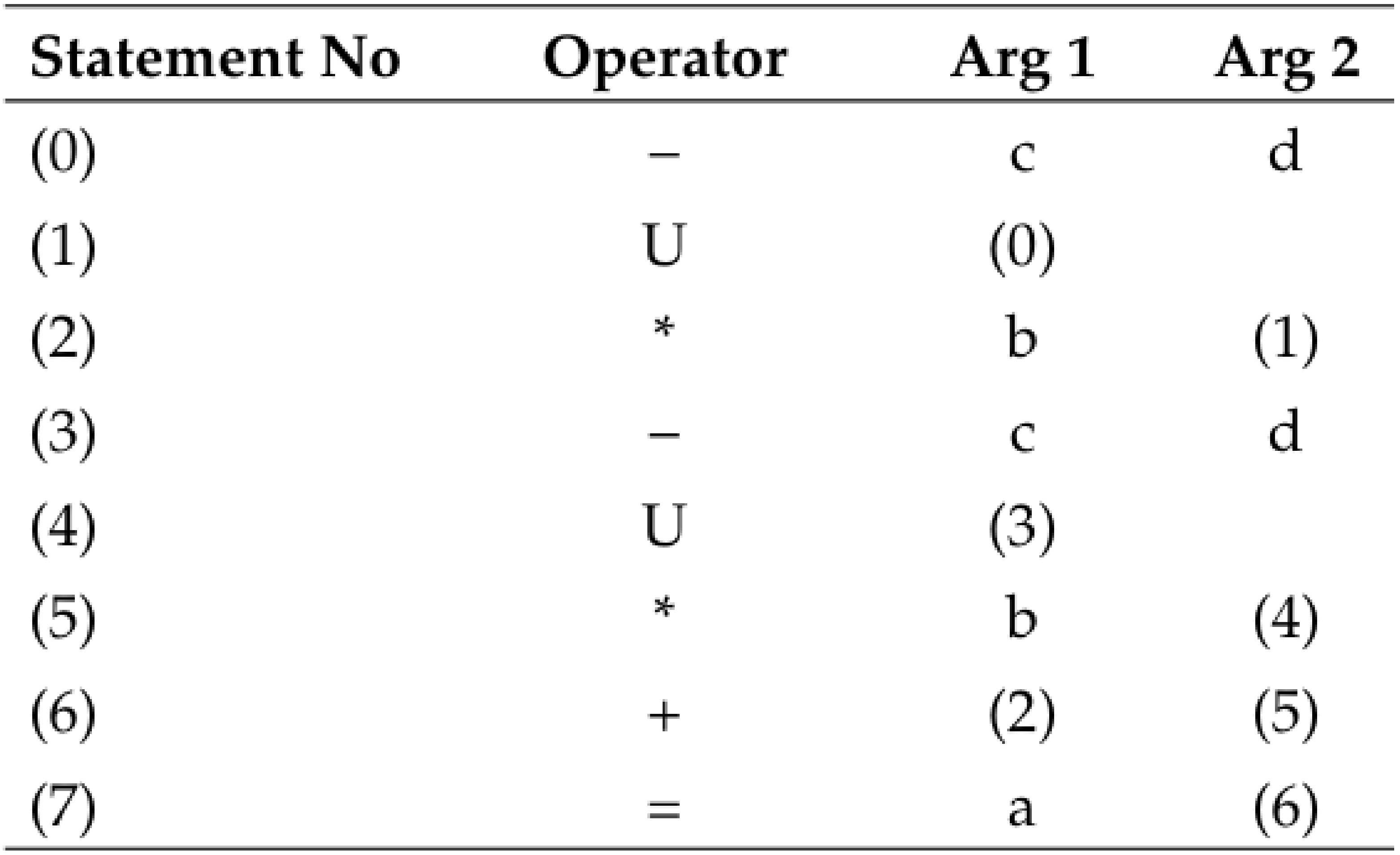
Problem in Quadruple

- In quadruples there is an overhead for managing the temporary variables created. This problem can be reduced by referring the position of the statement that computes the value of the sub expression.

Triple

- In triples there are only three fields, one for the operation and the remaining two for operands that may have a variable or a constant or the statement position number that computes the value of the operand. Since there are only three fields, it is called “triples.”

Example : $a = b * -(c - d) + b * -(c - d)$

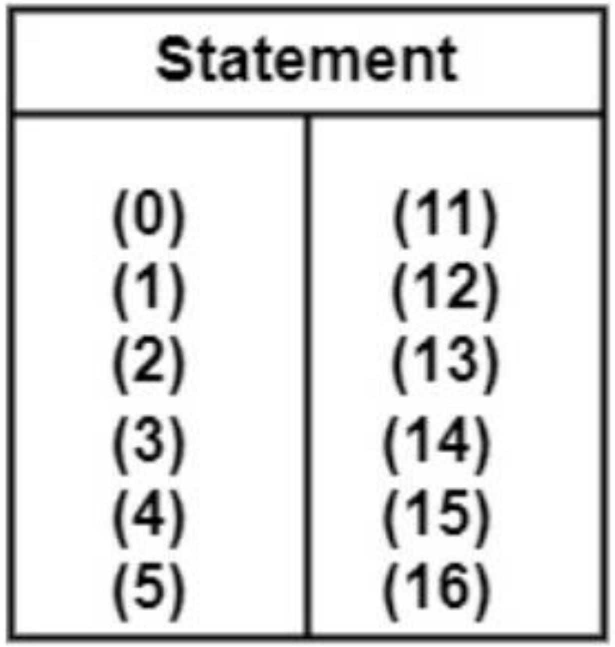


Statement No	Operator	Arg 1	Arg 2
(0)	–	c	d
(1)	U	(0)	
(2)	*	b	(1)
(3)	–	c	d
(4)	U	(3)	
(5)	*	b	(4)
(6)	+	(2)	(5)
(7)	=	a	(6)

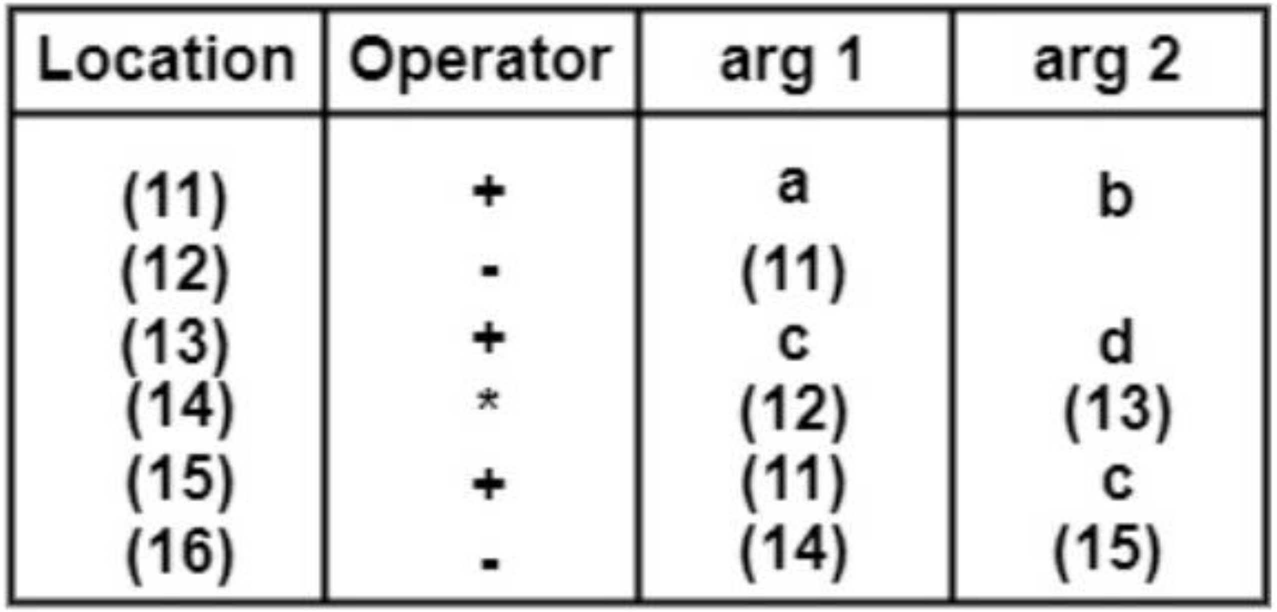
Indirect Triples

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

Example : **A + B * C + D**



Statement	
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)



Location	Operator	arg 1	arg 2
(11)	+	a	b
(12)	-	(11)	
(13)	+	c	d
(14)	*	(12)	(13)
(15)	+	(11)	c
(16)	-	(14)	(15)

Syntax-Directed Translation into Three Address Code

- Syntax-directed translation rules can be defined to generate the three address code while parsing the input.
- It may be required to generate temporary names for interior nodes which are assigned to nonterminal E on the left side of the production E→E1 op E2.
- While processing, the variable names and code generated so far are tracked till they reach the starting nonterminal.
- For this purpose, we associate two attributes place and code associated with each nonterminal.

Assignment Statement

- To generate intermediate code for assignment statement, first searching is applied to get the information of the identifier from the symbol table.
- These identifiers are simple or multidimensional array or a constant value that is stored in a literal table.
- After searching, the three address code is generated for the program statement.
- Function lookup will search the symbol table for the lexeme and store it in id.place.
- Function newtemp is defined to return a new temporary variable when invoked and gen function generates the

Example:

- Let us consider the example arithmetic statement $a = -(b - c)$. When it is represented as three address code, the statements are

$$T1 = b - c$$

$$T2 = - T1$$

$$a = T2$$

Example:

- While parsing the possible rules used for this expression are

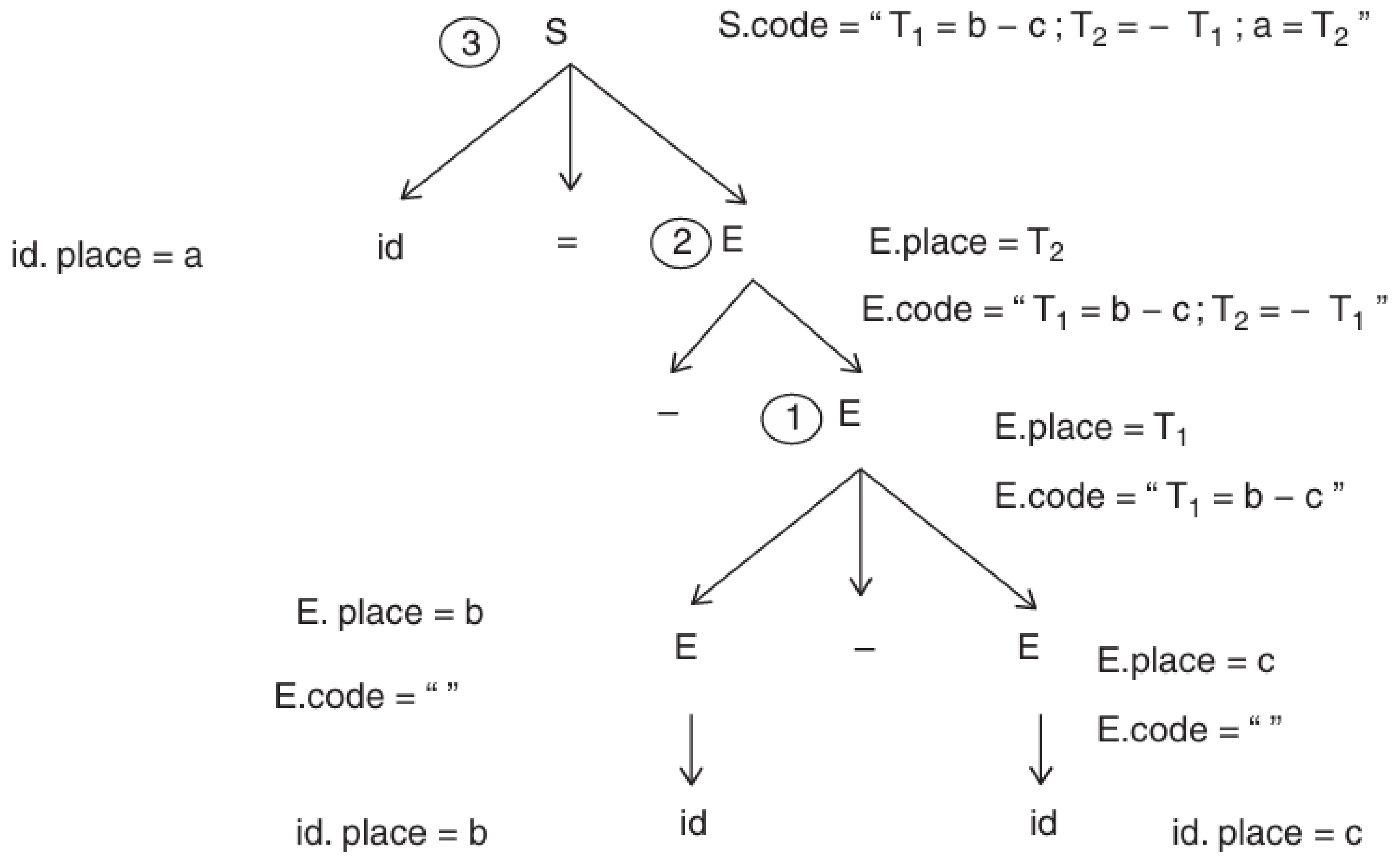
$S \rightarrow id = E$

$E \rightarrow - E1$

$E \rightarrow E1 - E2$

$E \rightarrow id$

Syntax Tree



3
S
S.code = “ T$_1$ = b – c ; T$_2$ = – T$_1$; a = T$_2$ ”
id. place = a
id
=
2
E
E.place = T$_2$
E.code = “ T$_1$ = b – c ; T$_2$ = – T$_1$ ”
–
1
E
E.place = T$_1$
E.code = “ T$_1$ = b – c ”
E. place = b
E.code = “ ”
E
–
E
E.place = c
E.code = “ ”
id. place = b
id
id
id. place = c

Solution

- Consider the syntax tree for the expression $a = -(b - c)$
- **Step1:**
  - Using the production $E \longrightarrow id$, that is, $E \longrightarrow b$ and $E \longrightarrow c$.
  - Check for entries b and c in the symbol table; if these entries are not present an error message is displayed.
  - If these entries are present then E1 .place = b and E2 .place = c (pointer to symbol table for the entry b and c).
- **Step2:**
  - At node 1 in the figure using the production $E \longrightarrow E1 - E2$ it creates a temporary variable T1 using newtemp(). Three address code E.place = E1 .place + E2 .place is generated.

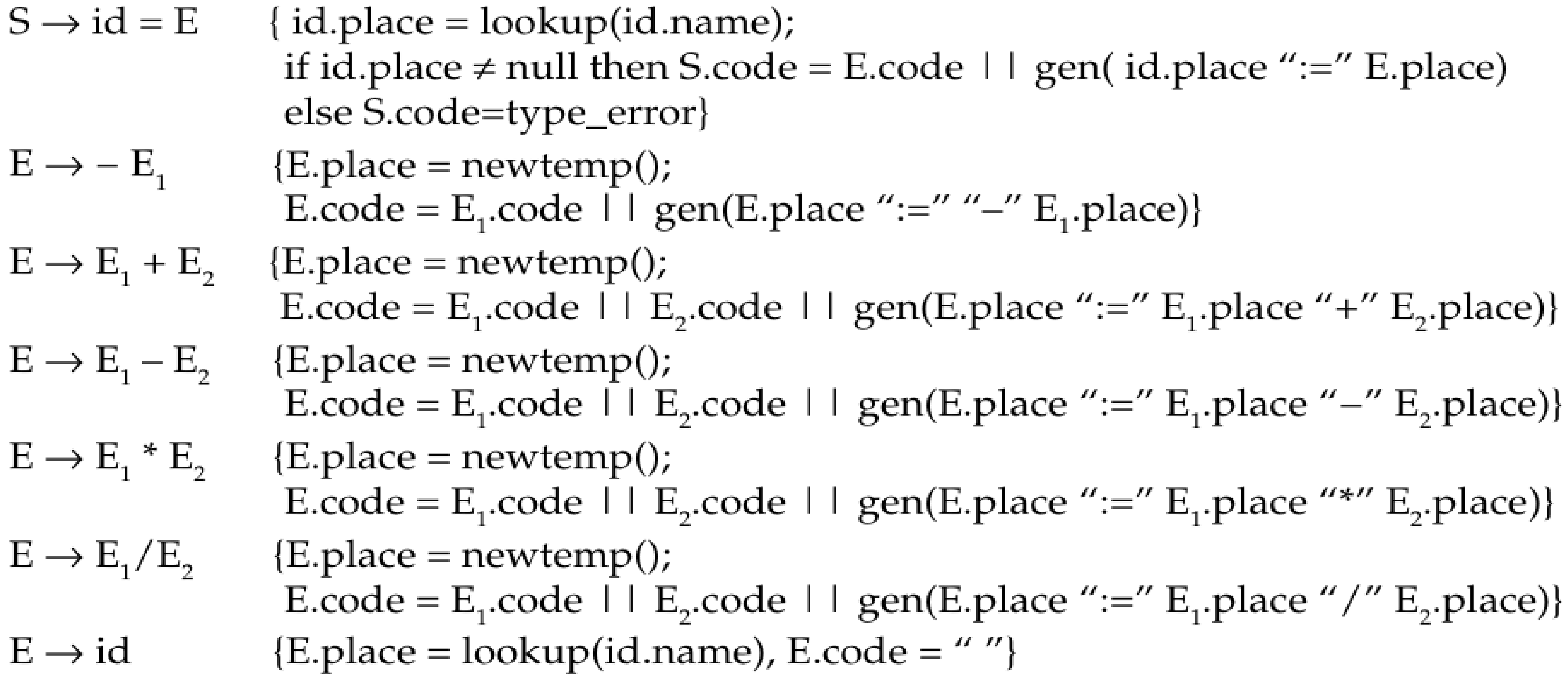
Solution:

- **Step3:**
  - At node 2 in the fi gure using the production $E \longrightarrow - E1$, it creates a temporary variable T2 using newtemp(). Three address code $E.place = - E1 .place$ is generated.

- **Step4:**
  - At node 3 using the production $S \longrightarrow id = E$ searches for a in the symbol table, assuming it the code produced is $a = E.place$

Type Checking

- The syntax-directed translation for arithmetic statements can thus be written as follows:



$S \rightarrow id = E$	{ id.place = lookup(id.name); if id.place $\neq$ null then S.code = E.code || gen(id.place ":=" E.place) else S.code=type_error}
$E \rightarrow - E_1$	{E.place = newtemp(); E.code = E_1.code || gen(E.place ":=" "–" E_1.place)}
$E \rightarrow E_1 + E_2$	{E.place = newtemp(); E.code = E_1.code || E_2.code || gen(E.place ":=" E_1.place "+" E_2.place)}
$E \rightarrow E_1 - E_2$	{E.place = newtemp(); E.code = E_1.code || E_2.code || gen(E.place ":=" E_1.place "–" E_2.place)}
$E \rightarrow E_1 * E_2$	{E.place = newtemp(); E.code = E_1.code || E_2.code || gen(E.place ":=" E_1.place "*" E_2.place)}
$E \rightarrow E_1 / E_2$	{E.place = newtemp(); E.code = E_1.code || E_2.code || gen(E.place ":=" E_1.place "/" E_2.place)}
$E \rightarrow id$	{E.place = lookup(id.name), E.code = " "}