Syntax Directed Definition

- Syntax Direct Translation (SDT) method is used in compiler design, associating the translation rules with grammar production. Syntax Directed Translation can identify informal notations, called semantic rules along with the grammar.

- Syntax Directed Translation in compiler design requires some information to convert the parse tree into a code. This information cannot be represented by the CFG(Context Free Grammar), hence attributes are attached to the variables of CFG.

We can represent it as:

Grammar + semantic rule = SDT

- Every non-terminal syntax direct translation can get one or more than one attribute depending on its type. The values of these attributes are set by semantic rules associated with the production rule.
- The 'val' attribute in the semantic rules may contain strings, numbers, memory location or a complex record.

The table below shows production and semantic rules.

S.No	Production	Semantic Rules
1.	E → E + T	E.val := E.val + T.val
2.	E → T	E.val := T.val
3.	T → T * F	T.val := T.val * F.val
4.	T → F	T.val := F.val
5.	F → (F)	F.val := F.val
6.	F → num	F.val := num.lexval

The right side of the translation rule is always in correspondence to the attribute values of the right side of the production rule. From this, we come to the conclusion that SDT in compiler design associates:

- A set of attributes to every node in grammar.
- A set of translation rules to every production rule with the help of attributes.

Application of SDT in Compiler Design

Here are some applications of SDT in Compiler Design:

1. Syntax Directed Translation is used for executing arithmetic expressions
2. Conversion from infix to postfix expression
3. Conversion from infix to prefix expression
4. For Binary to decimal conversion
5. Counting the number of Reductions
6. Creating a Syntax Tree
7. Generating intermediate code
8. Storing information into the symbol table
9. Type checking

Types of Attributes

There are two types of attributes:

1. Synthesized Attributes: These are those attributes which derive their values from their children nodes i.e. value of synthesized attribute at node is computed from the values of attributes at children nodes in parse tree.

Example:

```
E --> E1 + T { E.val = E1.val + T.val}
```

In this, E.val derive its values from E_1.val and T.val

Computation of Synthesized Attributes

•Write the SDD using appropriate semantic rules for each production in given grammar.

•The annotated parse tree is generated and attribute values are computed in bottom up manner.

•The value obtained at root node is the final output.

Example: Consider the following grammar

S --> E

E --> E1 + T

E --> T

T --> T1 * F

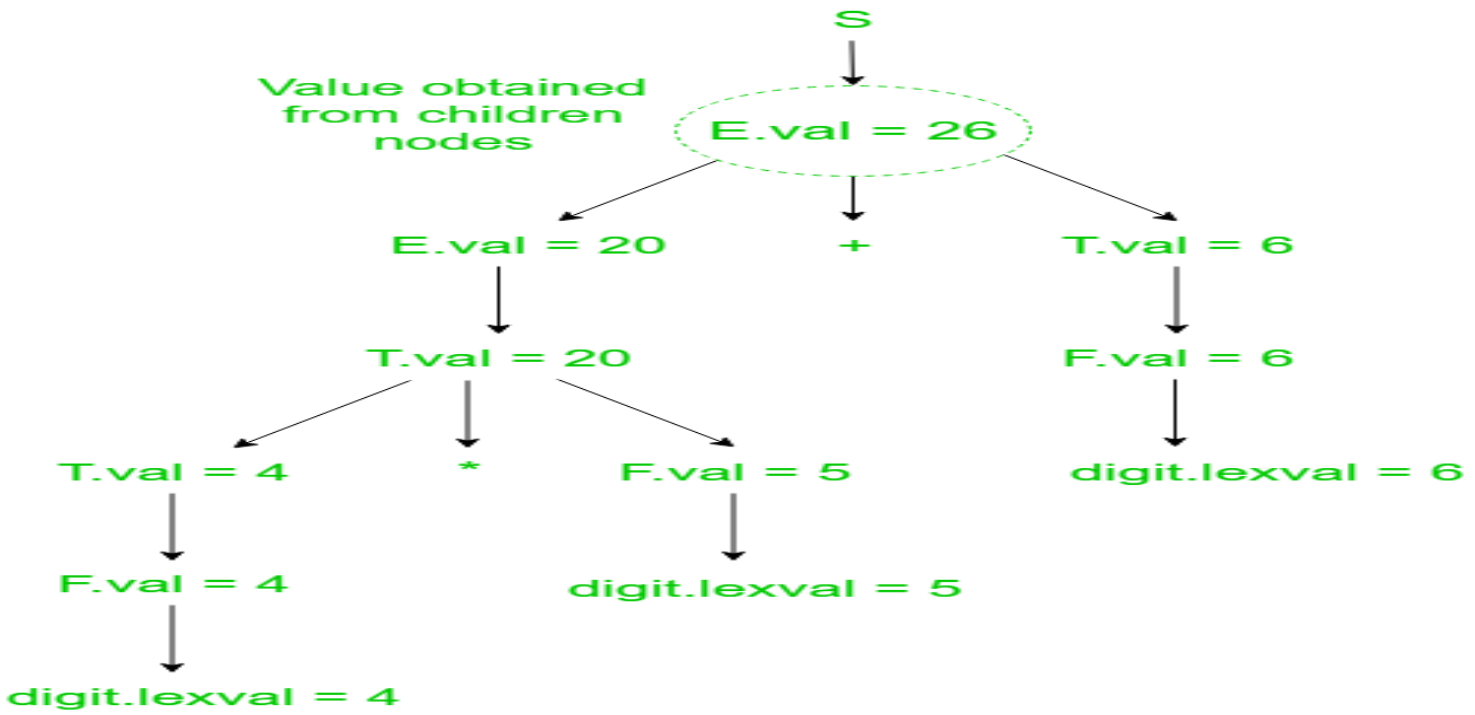
T --> F

F --> digit

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is



Annotated Parse Tree

2. Inherited Attributes: These are the attributes which derive their values from their parent or sibling nodes i.e. value of inherited attributes are computed by value of parent or sibling nodes.

Example:

A --> BCD { C.in = A.in, C.type = B.type }

Computation of Inherited Attributes

- Construct the SDD using semantic actions.
- The annotated parse tree is generated and attribute values are computed in top down manner.

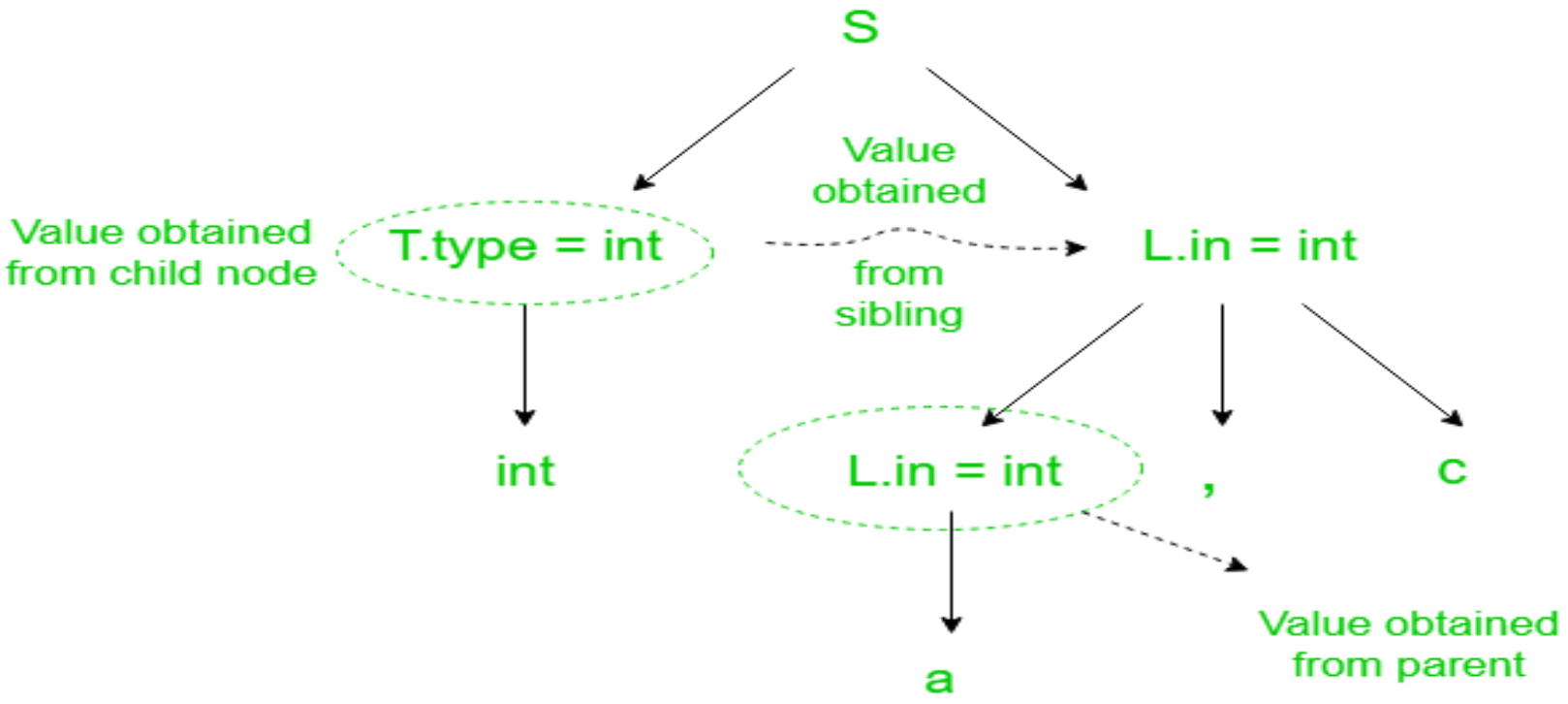
Example: Consider the following grammar

S --> T L
T --> int
T --> float
T --> double
L --> L1, id
L --> id

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> T L	L.in = T.type
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

Let us assume an input string **int a, c** for computing inherited attributes. The annotated parse tree for the input string is



Annotated Parse Tree

Two types of SDT:

- **S – Attributed SDT**
- **L – Attributed SDT**

1. S – Attributed SDT:

- An S-attributed SDT **(Synthesized Attributed SDT)** is one of the Syntax-Directed Translation schemes in which all attributes are synthesized.
- Predictive attributes are calculated as a result of attributes of the parse tree children nodes, their values are defined.
- Normally, the value of a synthesized attribute is produced at the leaf nodes and then passed up to the root of the parse tree.

Key Features:

•Bottom-Up Evaluation: Similarly, synthesized attributes are assessed in the bottom-up approach.

•Suitable for Bottom-Up Parsing: Thus, S-attributed SDTs are more suitable to the approaches to bottom-up parsing, including the shift-reduce parsers.

•Simple and Efficient: As all attributes are generated there are no inherited attributes involved thus making it easier to implement.

2. L – Attributed SDT:

- An L-Attributed SDT **(Left-Attributed SDT)** also permits synthesized attributes as well as inherited attributes.
- Some of these attributes are forced attributes, which are inherited from the parent node, other attributes are synthetic attributes, which are calculated like S-attributed SDTs.
- L-attributed SDTs is an algebra of system design and the key feature of the algebra is that attributes can only be inherited on the left side of a particular production rule.

Key Features:

•Top-Down Evaluation: Evaluations of the inherited attributes are carried out in a manner that is top-down while those of the synthesized attributes are bottom–up.

•Suitable for Top-Down Parsing: L-attributed SDTs are typical for the top-down approaches to parsing such as the recursive descent parsers.

•Allows More Complex Dependencies: Since a language that has the capability of supporting both, inherent as well as synthesized attributes for its terms define more sophisticated semantic rules, then it is appropriate for a semantic network.

Construction of Syntax Tree

- A syntax tree is a tree in which each leaf node represents an operand, while each inside node represents an operator. The Parse Tree is abbreviated as the syntax tree. The syntax tree is usually used when representing a program in a tree structure.

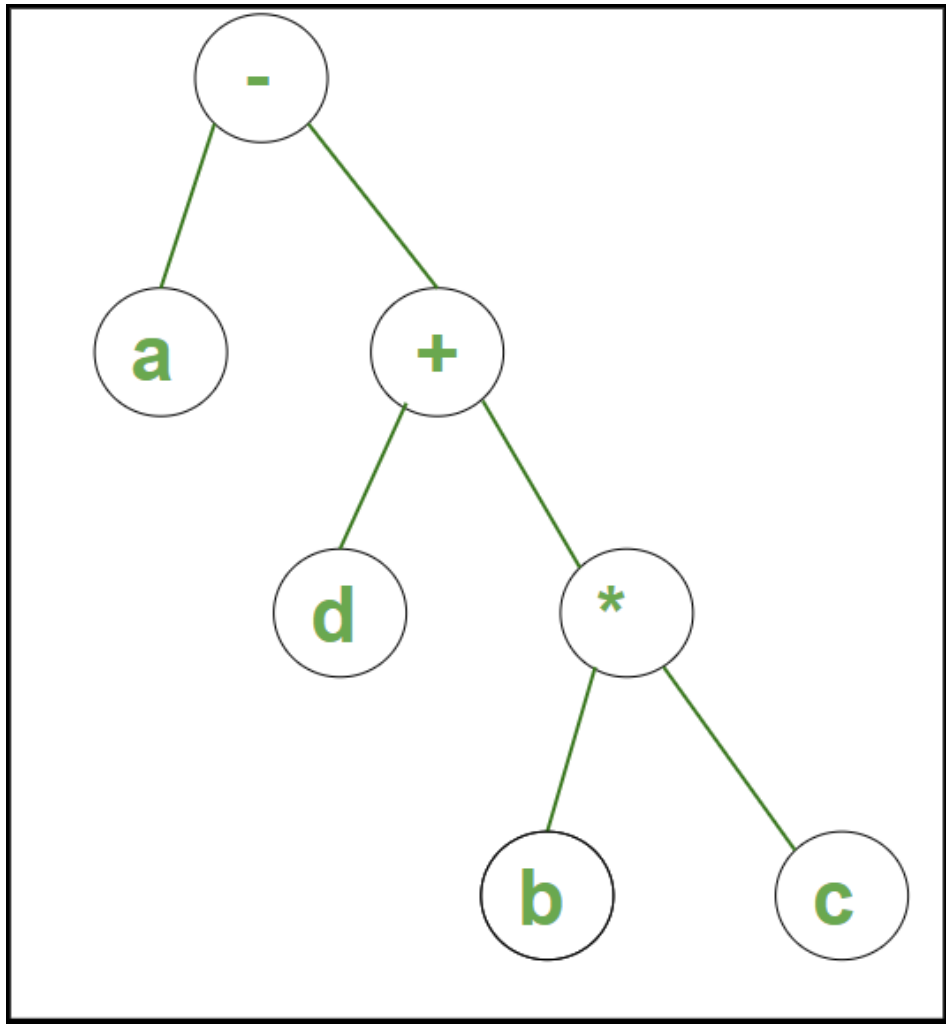
Rules of Constructing a Syntax Tree

1. mknode (op, left, right): It creates an operator node with the name op and two fields, containing left and right pointers.

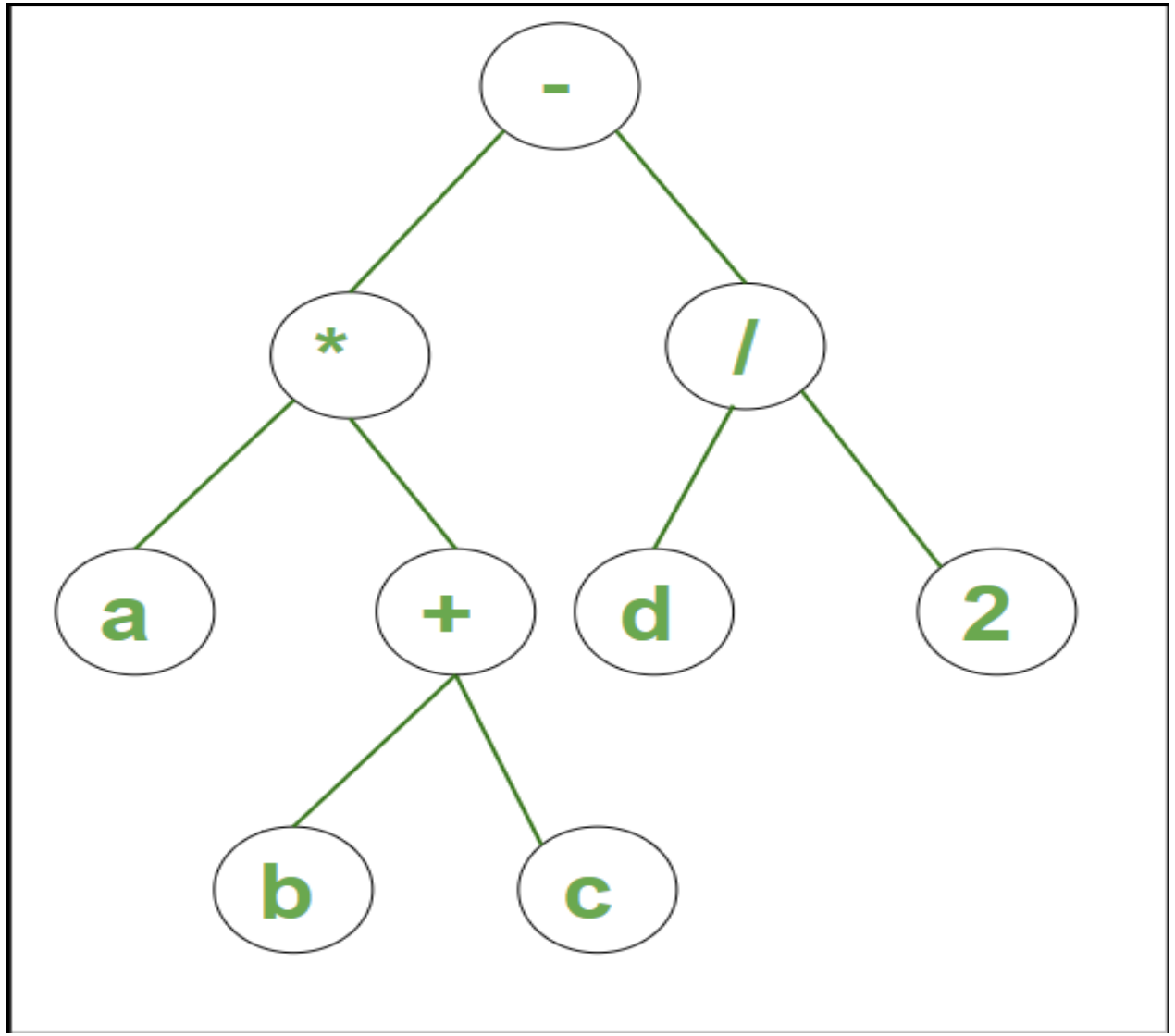
2. mkleaf (id, entry): It creates an identifier node with the label id and the entry field, which is a reference to the identifier's symbol table entry.

3. mkleaf (num, val): It creates a number node with the name num and a field containing the number's value, val. Make a syntax tree for the expression a 4 + c, for example. p1, p2,…, p5 are pointers to the symbol table entries for identifiers 'a' and 'c', respectively, in this sequence.

Example 1: Syntax Tree for the string **a – b ∗ c + d** is:



Example 2: Syntax Tree for the string **a * (b + c) – d /2** is:



Variants of syntax tree:

A syntax tree basically has two variants which are described below:

1. Directed Acyclic Graphs for Expressions (DAG)
2. The Value-Number Method for Constructing DAGs

Directed Acyclic Graphs for Expressions (DAG):

- The Directed Acyclic Graph (DAG) is a tool that shows the structure of fundamental blocks, allows you to examine the flow of values between them, and also allows you to optimize them. DAG allows for simple transformations of fundamental pieces.

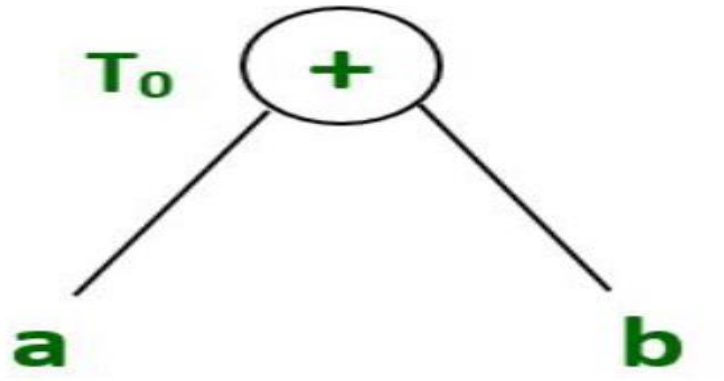
Properties of DAG are:

1.Leaf nodes represent identifiers, names, or constants.
2.Interior nodes represent operators.
3.Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

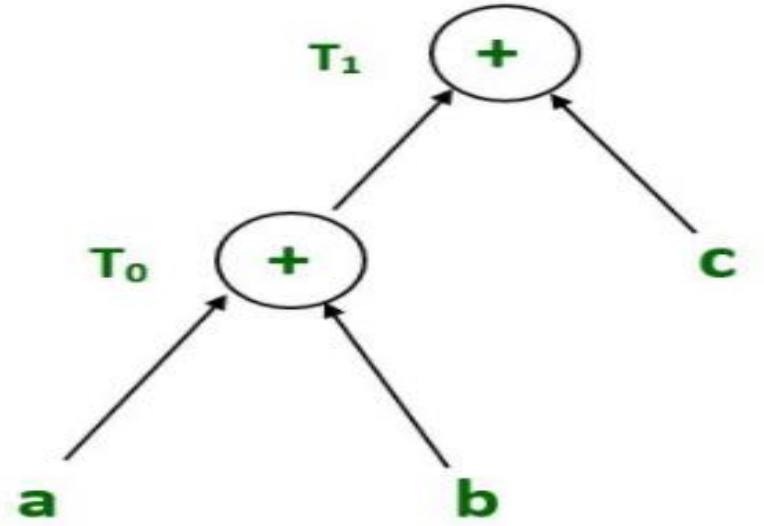
Examples:

T0 = a+b --- Expression 1
T1 = T0 +c --- Expression 2

Expression 1: $T_0 = a+b$

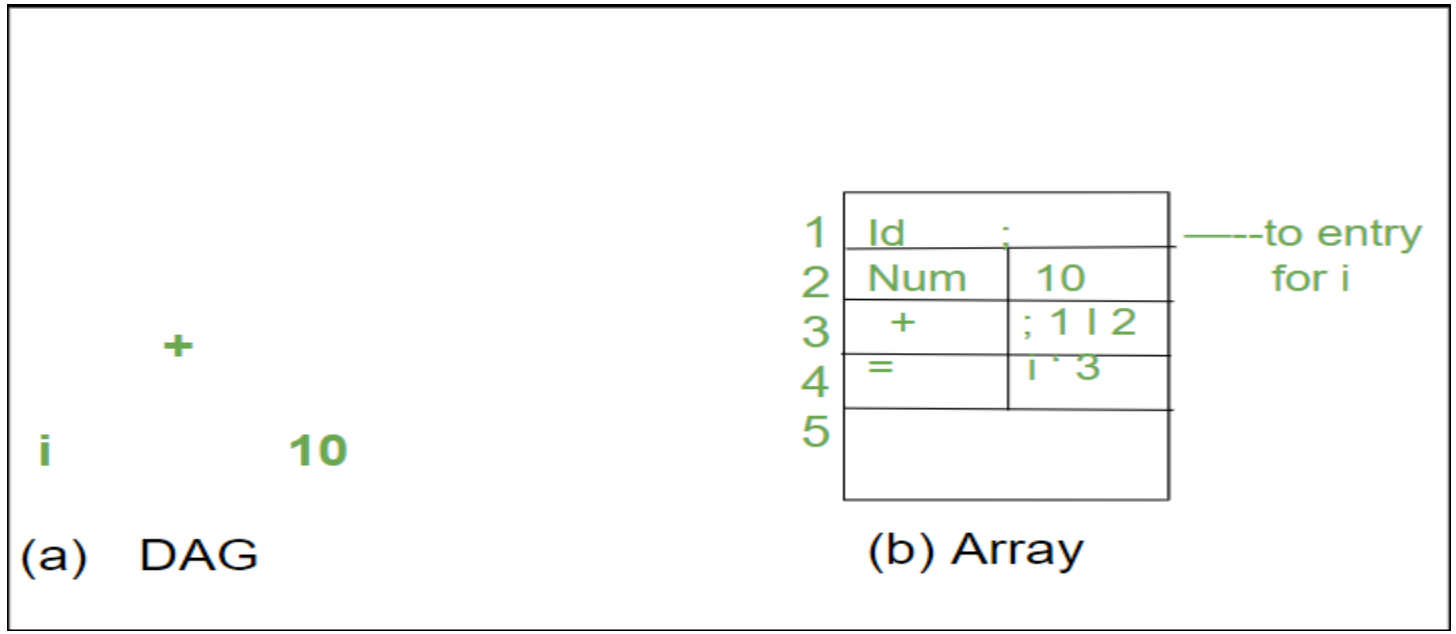


Expression 2: $T_1 = T_0 + c$



The Value-Number Method for Constructing DAGs:

- An array of records is used to hold the nodes of a syntax tree or DAG.
- Each row of the array corresponds to a single record, and hence a single node.
- The first field in each record is an operation code, which indicates the node's label. In the given figure below, Interior nodes contain two more fields denoting the left and right children, while leaves have one additional field that stores the lexical value (either a symbol-table pointer or a constant in this instance).



Intermediate Code Generation

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine). The benefits of using machine-independent intermediate code are:

- Because of the machine-independent intermediate code, portability will be enhanced. For example, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Retargeting is facilitated.
- It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.

What is Intermediate Code Generation?

Intermediate Code Generation is a stage in the process of compiling a program, where the compiler translates the source code into an intermediate representation. This representation is not machine code but is simpler than the original high-level code. Here's how it works:

- **Translation:** The compiler takes the high-level code (like C or Java) and converts it into an intermediate form, which can be easier to analyze and manipulate.
- **Portability**: This intermediate code can often run on different types of machines without needing major changes, making it more versatile.
- **Optimization:** Before turning it into machine code, the compiler can optimize this intermediate code to make the final program run faster or use less memory.

Postfix Notation

- Also known as reverse Polish notation or suffix notation.
- In the infix notation, the operator is placed between operands, e.g., *a* + *b.* Postfix notation positions the operator at the right end, as in *ab* +.
- For any postfix expressions *e1* and *e2* with a binary operator *(+)* , applying the operator yields *e1e2+.*
- Postfix notation eliminates the need for parentheses, as the operator's position and arity allow unambiguous expression decoding.
- In postfix notation, the operator consistently follows the operand.

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

ab + c *

Example 2: The postfix representation of the expression (a – b) * (c + d) + (a – b) is : **ab – cd + *ab -+**

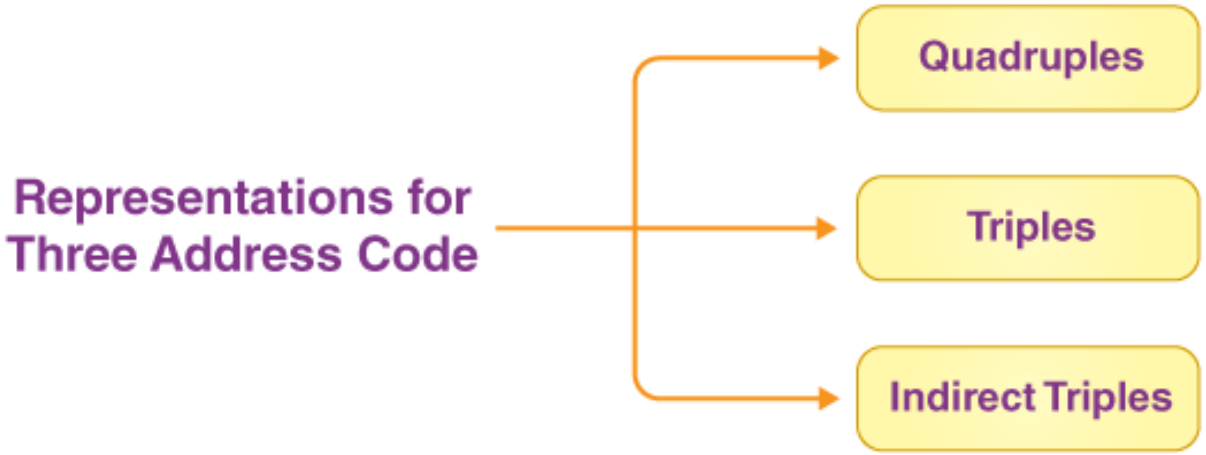
Three-Address Code

- A three address statement involves a maximum of three references, consisting of two for operands and one for the result.
- A sequence of three address statements collectively forms a three address code.
- The typical form of a three address statement is expressed as *x = y op z*, where *x, y*, and *z* represent memory addresses.
- Each variable *(x, y, z)* in a three address statement is associated with a specific memory location.

While a standard three address statement includes three references, there are instances where a statement may contain fewer than three references, yet it is still categorized as a three address statement.

Example: The three address code for the expression

```
a = b + c * d;
r1 = c * d;
r2 = b + r1;
a = r2
```



Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

```
r1 = c * d;
r2 = b + r1;
a = r2
```

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

#	Op	arg1	arg2
(0)	*	c	d
(1)	+	b	(0)
(2)	+	(1)	(0)
(3)	=	(2)	

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

#	Op	arg1	arg2
(0)	*	c	d
(1)	+	b	(0)
(2)	+	(1)	(0)
(3)	=	(2)	

Statement	
(0)	35
(1)	36
(2)	37
(3)	38

List of Pointers to table

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 non-terminal E on the left side of the production $E \rightarrow E_1$ op E_2. we associate two attributes place and code associated with each non-terminal.
- E.place, the name that will hold the value of E.
- E.code, the sequence of three-address statements evaluating E.

Syntax-Directed Translation into Three Address Code

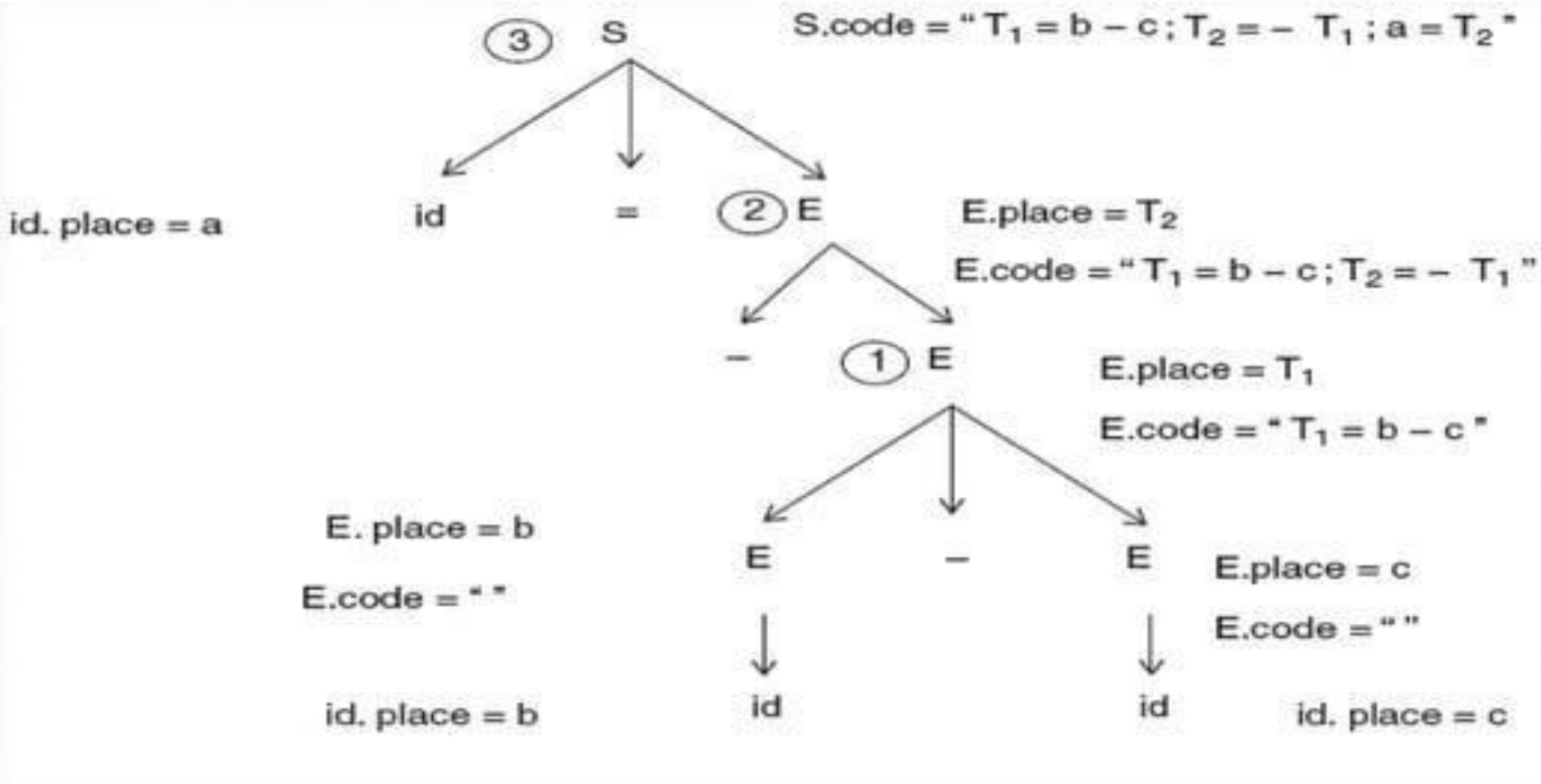
- To generate intermediate code for SDD, first searching is applied to get the information of the identifier from the symbol table. After searching, the three address code is generated for the program statement. Function lookup will search the symbol table for the lexeme and store it in id.place.
- Function **newtemp** is defined to return a new temporary variable when invoked and
- **gen** function generates the three address statement in one of the above standard forms depending on the arguments passed to it.

Three-address code for expressions

Production	Semantic rule
$S \rightarrow id = E$	{id.place = lookup(id.name); if id.place ≠ 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 = " "}

Three-address code for expressions

Example :- Syntax tree for **a = –(b – c)**



Translation of Boolean Expression

Boolean Expression have 2 primary purpose.

1. Used to computing logical values.
2. Used to computing conditional expression using if then else or while-do.

$E \rightarrow E_1$ or E_2

$E \rightarrow E_1$ and E_2

$E \rightarrow$ not E_1

$E \rightarrow id_1$ relop id_2

$E \rightarrow (E_1)$

$E \rightarrow$ true

$E \rightarrow$ false

Boolean Expression: Grammar and Actions

Production	Semantic Rule
$E \rightarrow E_1$ or E_2	{E.place = newtemp(); gen(E.place "=" E_1.place "or" E_2.place)}
$E \rightarrow E_1$ and E_2	{E.place = newtemp(); gen(E.place "=" E_1.place "and" E_2.place)}
$E \rightarrow$ not E_1	{E.place = newtemp(); gen(E.place "=" "not" E_1.place)}
$E \rightarrow (E_1)$	{E.place = E_1.place}
$E \rightarrow id_1$ relop id_2	{E.place = newtemp(); gen("if" id_1.place relop.op id_2.place "goto" nextstat + 3) gen(E.place "=" "0") gen("goto" nextstat + 2) gen(E.place "=" "1")}
$E \rightarrow$ true	{E.place = newtemp(); gen(E.place "=" "1")}
$E \rightarrow$ false	{E.place = newtemp(); gen(E.place "=" "0")}

Boolean Expression: Example

Example1 :-
Given Boolean Expression
if x < y then 1 else 0

The address code for the above expression is

1. if x < y go to 3
2. $t_1 = 0$
3. go to 4
4. $t_1 = 1$

Production	Semantic Rule
$E \rightarrow E_1$ or E_2	{E.place = newtemp(); gen(E.place "=" E_1.place "or" E_2.place)}
$E \rightarrow E_1$ and E_2	{E.place = newtemp(); gen(E.place "=" E_1.place "and" E_2.place)}
$E \rightarrow$ not E_1	{E.place = newtemp(); gen(E.place "=" "not" E_1.place)}
$E \rightarrow (E_1)$	{E.place = E_1.place}
$E \rightarrow id_1$ relop id_2	{E.place = newtemp(); gen("if" id_1.place relop.op id_2.place "goto" nextstat + 3) gen(E.place "=" "0") gen("goto" nextstat + 2) gen(E.place "=" "1")}
$E \rightarrow$ true	{E.place = newtemp(); gen(E.place "=" "1")}
$E \rightarrow$ false	{E.place = newtemp(); gen(E.place "=" "0")}

Boolean Expression: Example

Example 2 :-
Given Boolean Expression

x or y and not z

The address code for
the above expression is

$t_1 = \text{not } z$
$t_2 = y \text{ and } t_1$
$t_3 = x \text{ or } t_2$

Production	Semantic Rule
$E \rightarrow E_1$ or E_2	{E.place = newtemp(); gen(E.place "=" E_1.place "or" E_2.place)}
$E \rightarrow E_1$ and E_2	{E.place = newtemp(); gen(E.place "=" E_1.place "and" E_2.place)}
$E \rightarrow$ not E_1	{E.place = newtemp(); gen(E.place "=" "not" E_1.place)}
$E \rightarrow (E_1)$	{E.place = E_1.place}
$E \rightarrow id_1$ relop id_2	{E.place = newtemp(); gen("if" id_1.place relop.op id_2.place "goto" nextstat + 3) gen(E.place "=" "0") gen("goto" nextstat + 2) gen(E.place "=" "1")}
E → true	{E.place = newtemp(); gen(E.place "=" "1")}
E → false	{E.place = newtemp(); gen(E.place "=" "0")}

Boolean Expression: Example

Example : Give three address code for the following: **while(P < Q)do**

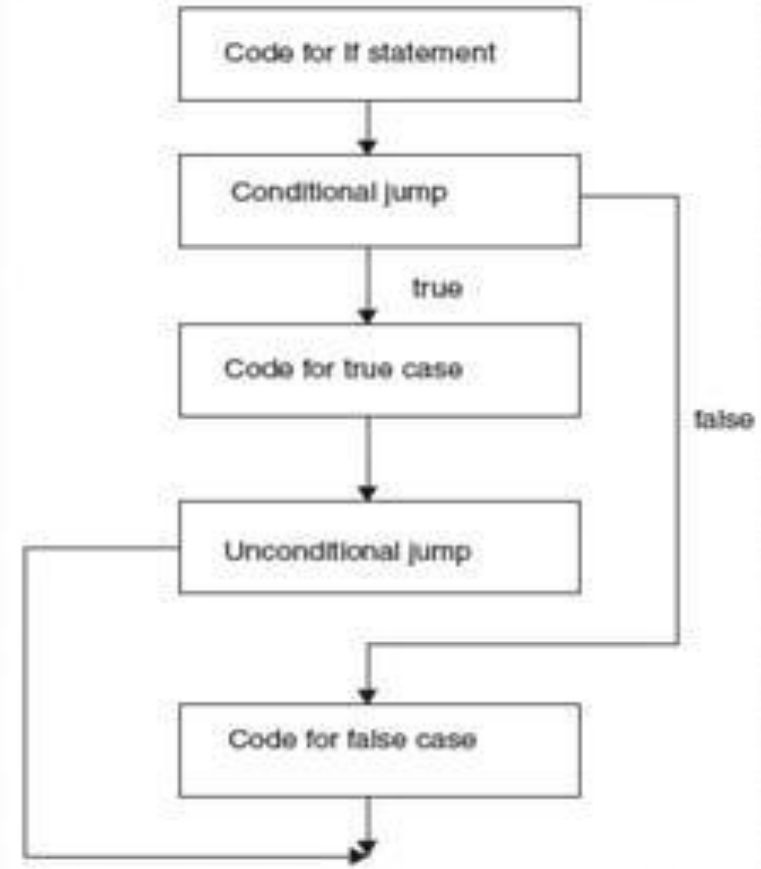
if(R < S) then a = b + c;

Solution 1. if P < Q goto (3)

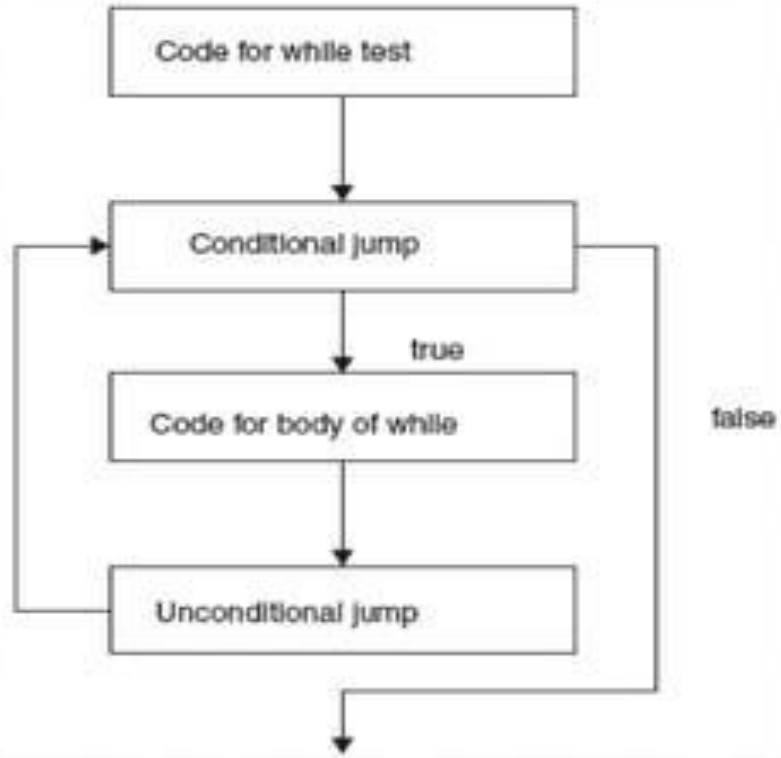
2. goto (8)
3. if R < S goto (5)
4. goto (1)
5. t_1: = b + c
6. a: = t_1
7. goto (1)
8. Next

Control Statements

Flow of control statements can be shown pictorially as



Control low for if-then-else statement



Control low for while statement

Translation of Control Flow Statements

Production	Semantic Rule
$S \rightarrow$ if E then S_1	{E.true = newlabel(); E.false = S.next; S_1.next = S.next; S.code = E.code | | gen(E.true,":") | | S_1.code}
$S \rightarrow$ if E then S_1 else S_2	{E.true = newlabel(); E.false = newlabel(); S_1next = S.next; S_2.next = S.next; S.code = E.code | | gen(E.true,":") | | S_1.code | | gen(" GOTO ", S.next) | | gen(E.false, " :") | | S_2.code}
$S \rightarrow$ while E do S_1	{S.begin = newlabel(); E.true = newlabel(); E.false = S.next; S_1.next = S.next; S.code = gen(S.begin":") | | E.code | | gen(E.true,":") | | S_1.code | | gen("GOTO",S.begin)}

Control Flow Statements

Example : Give three address code for the following:

While (a < 5) do a: = b + 2

Solution:-

L_1:

If a < 5 goto L_2

goto last

L_2:

$t_1 = b + 2$

$a = t_1$

goto L_1

last:

Control Flow Statements

Example : Give three address code for the following:

```
while a<b
do
if c < d then
    x = y + z
else
    x = y - z
done
```

Solution:-

L1. if a<b then GOTO L2

GOTO LNEXT

L2. if c<d then GOTO L3

GOTO L4

L3. tl = y + z

x = tl

GOTO L1

L4. tl= y - z

x = tl

GOTO L1

LNEXT.