SYSTEM SOFTWARE and COMPILERS-18CS61

Prof. Sampada K S, Assistant Professor
DEPT. OF CSE | RNSIT

MODULE-5

Syntax Directed Translation, Intermediate code generation, Code generation Text book 2: Chapter 5.1, 5.2, 5.3, 6.1, 6.2, 8.1, 8.2

SEMANTIC ANALYSIS

Semantic analysis is the third phase of the compiler which acts as an interface between syntax analysis phase and code generation phase. It accepts the parse tree from the syntax analysis phase and adds the semantic information to the parse tree and performs certain checks based on this information. It also helps constructing the symbol table with appropriate information. Some of the actions performed semantic analysis phase are:

- Type checking i.e., number and type of arguments in function call and in function header of function definition must be same. Otherwise, it results in semantic error.
- Object binding i.e., associating variables with respective function definitions
- Automatic type conversion of integers in mixed mode of operations
- Helps intermediate code generation.
- Display appropriate error messages

The semantics of a language can be described very easily using two notations namely:

- Syntax directed definition (SDD)
- Syntax directed translation (SDT)

1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).

2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions:

The syntax-directed definition (SDD) is a CFG that includes attributes and rules. In an augmented CFG, the attributes are associated with the grammar symbols (i.e. nodes of the parse tree). And the rules are associated with the productions of grammar.

• **Syntax Directed Definitions** are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of **Attributes**;

For example, a simple SDD for the production $E \rightarrow E_1 + T$ can be written as shown below:

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

Attributes

Attribute is a property of a programming language construct. Associated with grammar symbols. Such formalism generates Annotated **Parse-Trees** where each node of the tree is a record with a field for each attribute. If X is a grammar symbol and 'a' is a attribute then X.a denote the value of attribute 'a' at a particular node X in a parse tree.

- Ex 1: If val is the attribute associated with a non-terminal E, then E.val gives the value of attribute val at a node E in the parse tree.
- Ex 2: If lexval is the attribute associated with a terminal digit, then digit.lexval gives the value of attribute lexval at a node digit in the parse tree.
- Ex 3: If syn is the attribute associated with a non-terminal F, then F.syn gives the value of attribute syn at a node F in the parse tree.

Typical examples of attributes are:

- The data types associated with variable such as int, float, char etc
- The value of an expression
- The location of a variable in memory
- The object code of a function or a procedure
- The number of significant digits in a number and so on.

2. Productions are associated with **Semantic Rules** for computing the values of attributes.

The rule that describe how to compute the attribute values of the attributes associated with a grammar symbol using attribute values of other grammar symbol is called semantic rule.

Example:

E.val= E1.val+T.val //Semantic rule

where E.val on RHS can be computed using E1.val and T.val on RHS

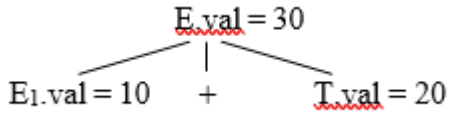
The attribute value for a node in the parse tree may depend on information from its children nodes or its sibling nodes or parent nodes. Based on how the attribute values are obtained we can classify the attributes. Now, let us see "What are the different types or classifications of attributes?" There are two types of attributes namely:

- Synthesized attribute
- Inherited attribute

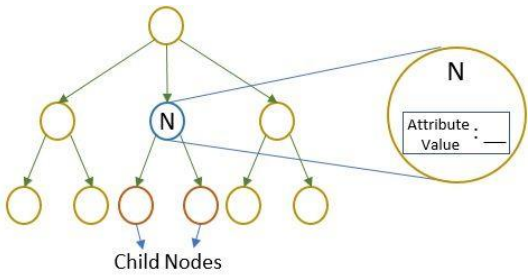
Synthesized Attributes: The attribute value of a non-terminal A derived from the attribute values of its children or itself is called synthesized attribute. Thus, the attribute values of synthesized attributes are passed up from children to the parent node in bottom-up manner.

For example, consider the production: E → E1 + T. Suppose, the attribute value val of E on LHS (head) of the production is obtained by adding the attribute values E1.val and T.val appearing on the RHS (body) of the production as shown below:

Production	Semantic Rule	Parse tree with attribute values
E → E + T	E.val = E_1.val + T.val	



Now, attribute val with respect to E appearing on head of the production is called synthesized attribute. This is because, the value of E.val which is 30, is obtained from the children by adding the attribute values 10 and 20 as shown in above parse tree.



SDD with Synthesized Attributes

PRODUCTION	SEMANTIC RULE
$L \rightarrow En$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ digit	$F.val :=$ digit.$lexval$

Observe the following points from the above parse tree:

- The type int obtained from the lexical analyzer is already stored in T.type whose value is

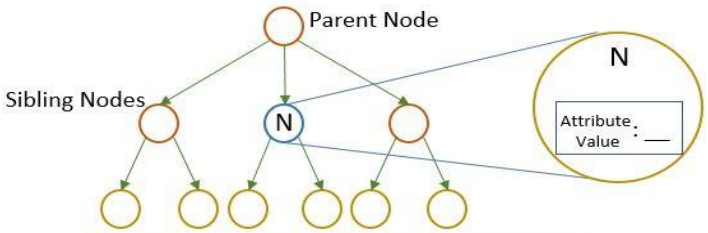
transferred to its sibling V. This can be done using:

V.inh = T.type

Since attribute value for V is obtained from its sibling, it is inherited attribute and its attribute is denoted by *inh.*

- On similar line, the value int stored in V.inh is transferred to its child id.entry and hence entry is inherited attribute of id and attribute value is denoted by id.entry

Note: With the help of the annotated parse tree, it is very easy for us to construct SDD for a given grammar.



SDD with Inherited Attributes

Let us consider the syntax directed definition with both inherited and synthesized attributes for the grammar for "type declarations":

PRODUCTION	SEMANTIC RULE
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow$ int	$T.type :=$ *integer*
$T \rightarrow$ real	$T.type :=$ *real*
$L \rightarrow L_1$, id	$L_1.in := L.in$; *addtype*(id.*entry, L.in*)
$L \rightarrow$ id	*addtype*(id.*entry, L.in*)

• The non terminal T has a synthesized attribute, type, determined by the keyword in the declaration.
• The production D → T L is associated with the semantic rule L.in := T .type which set the inherited attribute L.in.
• Note: The production L → L_1, id distinguishes the two occurrences of L.

To evaluate translation rules, identify the best-suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals (because SDT rules don't impose any specific order on evaluation)

Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Features –

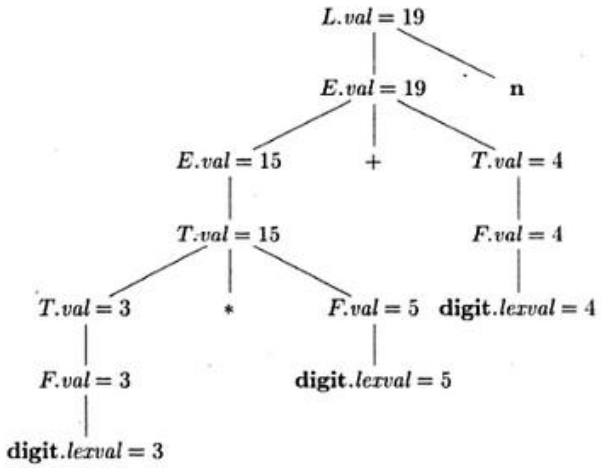
- High level specification
- Hides implementation details
- Explicit order of evaluation is not specified

A parse tree showing the attribute values of each node is called *annotated parse tree.* The terminals in the annotated parse tree can have only synthesized attribute values and they are obtained directly from the lexical analyzer. So, there are no semantic rules in SDD (short form **S**yntax **D**irected **D**efinition) to get the lexical values into terminals of the annotated parse tree. The other nodes in the annotated parse tree may be either synthesized or inherited attributes. **Note:** Terminals can never have inherited attributes

Consider the SDD

L→ En where *n* represent end of file marker
E → E + T | T
T → T * F | F
F → (E) | **digit**

Here we can see the production rules of grammar along with the semantic actions. And the input string provided by the lexical analyzer is 3 * 5 + 4 n.



the final SDD along with productions and semantic rules is shown below:

Productions	**Semantic Rules**
L → En	L.val = E.val
E → E + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = digit.lexval

Dependency Graph

A dependency graph is used to represent the flow of information among the attributes in a parse tree. In a parse tree, a dependency graph basically helps to determine the evaluation order for the attributes. The main aim of the dependency graphs is to help the compiler to check for various types of dependencies between statements in order to prevent them from being executed in the incorrect

sequence, i.e. in a way that affects the program's meaning. This is the main aspect that helps in identifying the program's numerous parallelizable components.

Example of Dependency Graph:

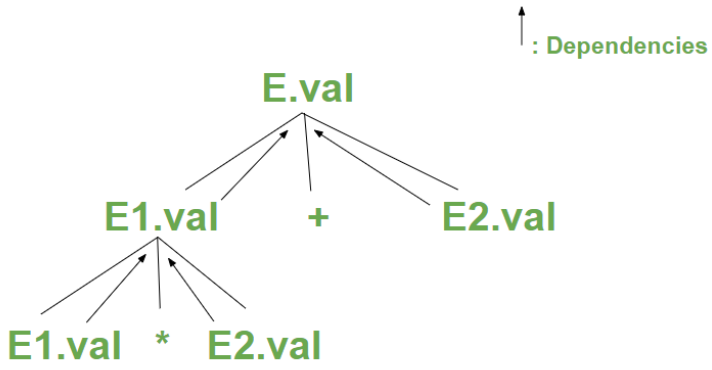
Design dependency graph for the following grammar:

E -> E1 + E2

E -> E1 * E2

PRODUCTIONS	SEMANTIC RULES
E -> E1 + E2 E -> E1 * E2	E.val -> E1.val + E2.val E.val -> E1.val * E2.val

Required dependency graph for the above grammar is represented as –



Evaluation Orders for SDD

There can be two classes of syntax-directed translations S-attributed translation and L-attributed translation.

S-ATTRIBUTED DEFINITIONS

Definition. An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

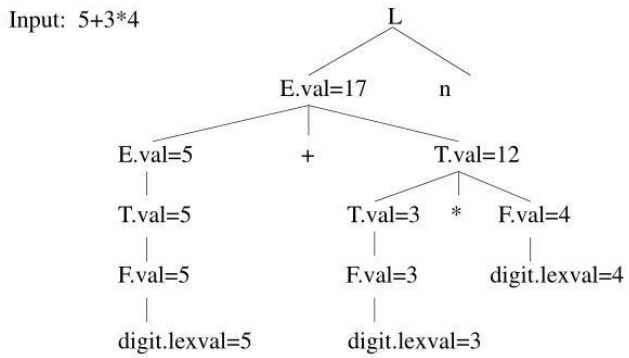
• **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.

• **Example.** The arithmetic grammar is an example of an S-Attributed Definition.

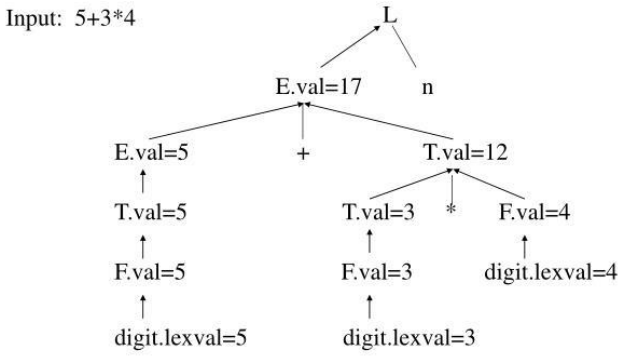
PRODUCTION	SEMANTIC RULE
$L \rightarrow En$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ digit	$F.val :=$digit.$lexval$

The annotated parse-tree for the input 5+3*4 is:

Annotated Parse Tree -- Example



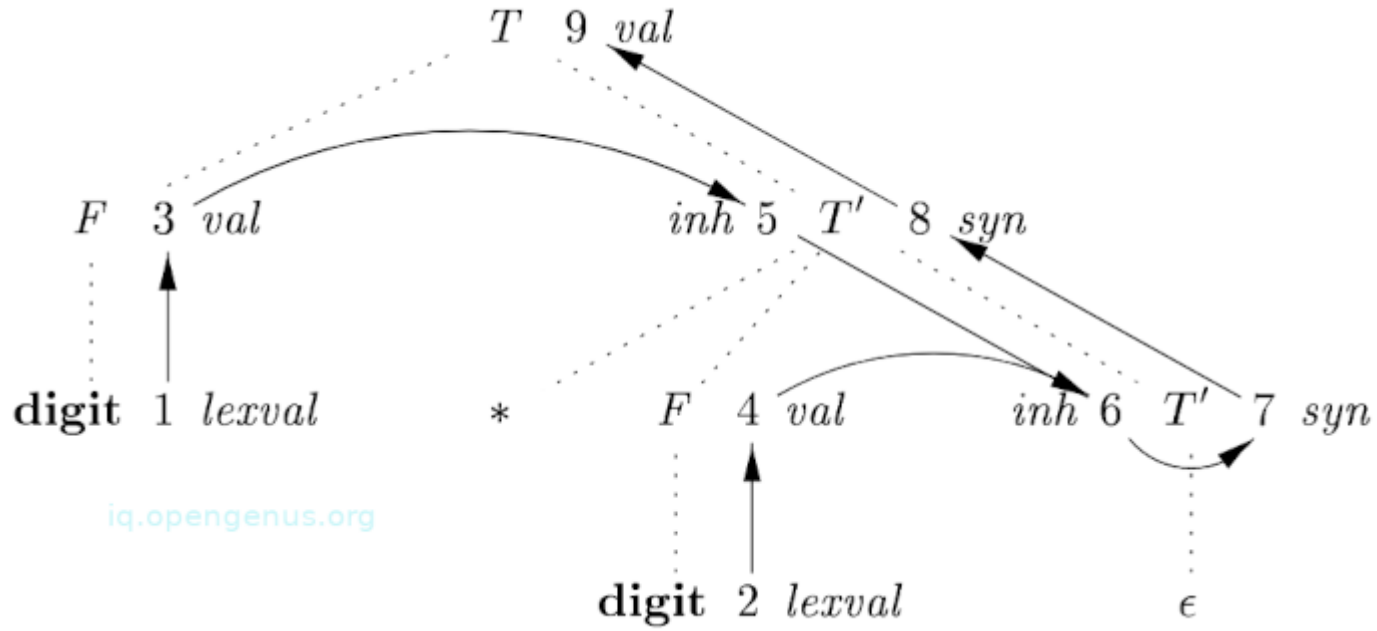
Dependency Graph

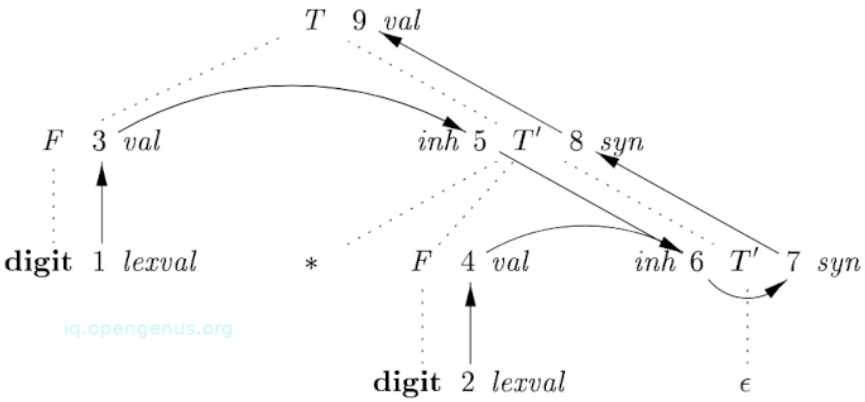


L-attributed definition

Definition: A SDD its *L-attributed* if each inherited attribute of Xi in the RHS of A ! X1 :Xn depends only on

1. attributes of X1;X2; : : : ;Xi□1 (symbols to the left of Xi in the RHS)





To evaluate the synthesized attribute of a node we have to parse the tree in a bottom-up fashion. As its value depends on the value of the concerned node's child attributes and the node itself.
To evaluate the inherited attribute of a node we have to parse the tree in a top-down fashion. As its value depends on the attribute value of its parent, its siblings and the node itself.
Some nodes in a parse tree may involve both synthesized and inherited attributes. In such a case, we are not sure that there exists even one order in which the attributes at the nodes can be evaluated.
Even a dependency graph can determine the order of evaluation of the attributes.

Syntax Directed Translation Scheme

1. Postfix Translation Scheme

Here, we construct SDT in such as manner that each semantic action is executed at the end of that production. Thus execution of the semantic action takes place along with the reduction of that production to its head.Thus, we refer to SDT's with all the semantic actions at the right end of the production rules as postfix SDT's.

- Simplest SDDs are those that we can parse the grammar bottom-up and the SDD is s- attributed
- For such cases we can construct SDT where each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production
- SDT's with all actions at the right ends of the production bodies are called postfix SDT's

2. Parser-Stack Implementation of Postfix SDT's

In this scheme, we place the grammar symbol (node) along with its attribute onto the stack. Thus it becomes easy to retrieve attributes and symbols together when the reduction of the corresponding node occurs. As the stack operates in the first-in-last-out mode.

- In a shift-reduce parser we can easily implement semantic action using the parser stack
- For each nonterminal (or state) on the stack we can associate a record holding itsattributes
- Then in a reduction step we can execute the semantic action at the end of a productionto evaluate the attribute(s) of the non-terminal at the leftside of the production
- And put the value on the stack in replace of the rightside of productionEXAMPLE

```
L -> E n    {print(stack[top-1].val); top=top-1;}

E -> E1 + T {stack[top-2].val=stack[top-2].val+stack.val;top=top-2;}

E -> T

T -> T1 * F {stack[top-2].val=stack[top-2].val+stack.val;top=top-2;}

T -> F

F -> (E)  {stack[top-2].val=stack[top-1].val top=top-2;}

F -> digit
```

3. SDT's With Action Inside Productions

In one of the methods, you can even place the semantic action at any position within a production body. The action takes place just after processing all the grammar symbols on the left side of the

action.
Such as consider the production B -> X {a} Y;
Here we can perform the semantic action 'a' after the processing grammar symbol X (in case X is a terminal). Or after processing all the symbols derived from X (in case the X is a non-terminal).

4. Eliminating Left Recursion from SDT's

It is not possible to parse the grammar with left recursion in a top-down fashion. Thus, we must eliminate the left recursion from the grammar.

5. SDT's for L-Attributed Definition

To translate the L-attributed SDD into the SDT we have to follow:

1. First, perform the semantic action evaluating the value of the inherited attribute.
2. Secondly, perform the semantic action evaluating the value of the synthesized attribute.

We can relate these steps for the SDT with the L-attributed translation.
Applications of Syntax Directed Translation

- SDT is useful in evaluating an arithmetic expression
- It helps in converting infix to postfix or converting infix to prefix.
- The syntax-directed translation is helpful in converting binary to decimal.
- SDT facilitates help in creating the syntax tree.
- The syntax-directed translation can help in generating the intermediate code and even in type checking.
- SDT stores the type info in the symbol table.

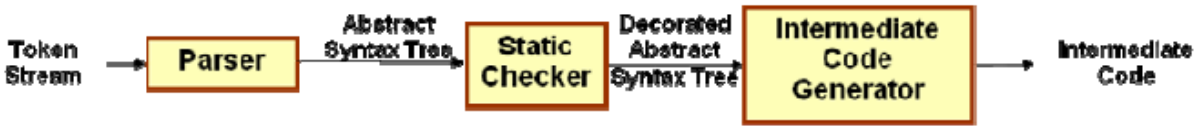
So this is all about syntax-directed translation. We have learnt about its implementation with the help of an example. We have also discussed some SDT schemes and their applications. The SDT also helps in type checking and even in generating intermediate code. The translation technique also helps in implementing little languages for some specialized tasks.

INTERMEDIATE CODE

INTERMEDIATE CODE GENERATION

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. This facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end.

Logical Structure of a Compiler Front End



A compiler front end is organized as in figure above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and then walking the tree.

Static Checking

This includes type checking which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing like

- flow–of-control checks
  - Ex: Break statement within a loop construct
- Uniqueness checks
  - Labels in case statements
- Name-related checks

Intermediate Representations

We could translate the source program directly into the target language. However, there are benefits to having an intermediate, machine-independent representation.

- A clear distinction between the machine-independent and machine-dependent parts of the compiler
- Retargeting is facilitated the implementation of language processors for new machines will require replacing only the back-end.
- We could apply machine independent code optimization techniques

Intermediate representations span the gap between the source and target languages.

• ***High Level Representations***

- closer to the source language
- easy to generate from an input program
- code optimizations may not be straightforward

• ***Low Level Representations***

- closer to the target machine
- Suitable for register allocation and instruction selection
- easier for optimizations, final code generation

There are several options for intermediate code. They can be either specific to the language being implemented

- P-code for Pascal
- Byte code for Java

There are three types of intermediate representation:-

1. Syntax Trees
2. Postfix notation
3.Three Address Code

Variants of Syntax Trees

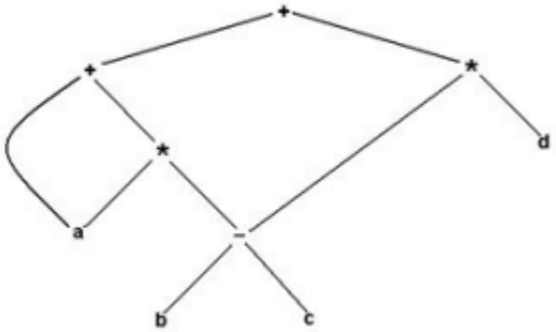
• Nodes of syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct.

• A directed acyclic graph (DAG) for an expression identifies the common subexpressions of the expression. (subexpressions that appears more than once).

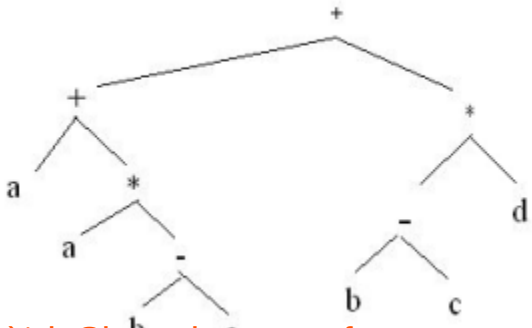
Directed Acyclic Graphs for Expressions

7. Syntax tree for the expression a+a*(b-c)+(b-c)*d

Fig 6.3:DAG **a + a * (b - c) + (b - c) * d**

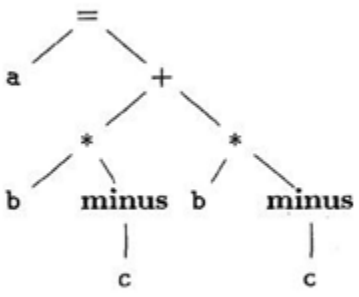


Syntax tree for **a + a * (b - c) + (b - c) * d**



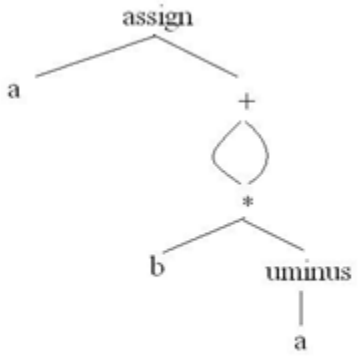
8. DAG for the expression a+a*(b-c)+(b-c)*d. Show the steps for constructing the same.

Eg2: syntax tree for assignment statement a=b*-c + b*-c

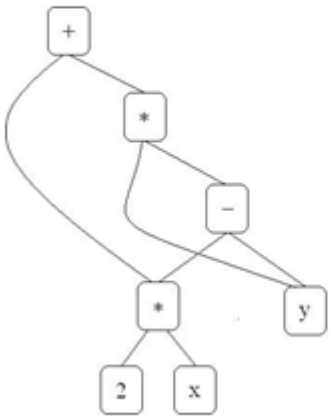


(a) Syntax tree

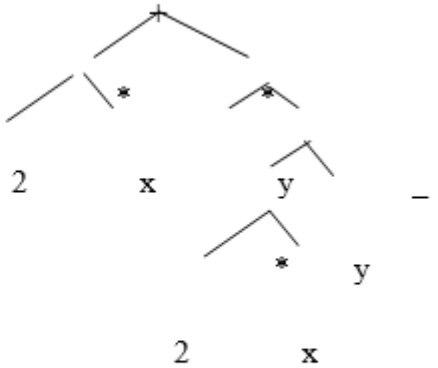
DAG for a=b*-c + b*-c



Eg 3 : For example, the expression 2*x + y * (2*x - y) could be represented by DAG



Syntax tree for 2*x + y * (2*x - y)



The DAG representation may expose instances where redundancies can be eliminated.

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**$,$ **id**$.entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**$,$ **num**$.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

1) $p_1 = Leaf(\textbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\textbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\textbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

-----steps for dag

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

SDD to construct DAG for the expression **a + a * (b - c) + (b - c) * d.**

The Value-Number Method for Constructing DAG's

In many applications, nodes are implemented as records stored in an array, as in Figure 7.In the figure; each record has a label field that determines the nature of the node. We can refer to a node by its index in the array. The integer index of a node is often called *value number*. For example, using value numbers, we can say node 3 has label +, its left child is node 1, and its right child is node 2.The following algorithm can be used to create nodes for a *dag* representation of an expression.

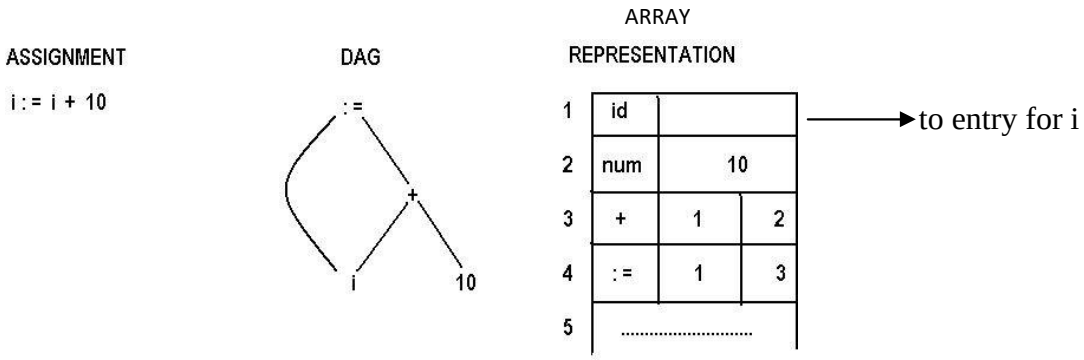


Figure : Nodes o fa DAG for i=i+10 allocated in an array

Algorithm: *The value-number method for constructing the nodes of a DAG*
Input: *Label* ***op****, node* ***l****, and node* ***r***

Output: *The value number of a node in the array with signature <**op, l, r**>*

Method: *Search the array for a node M with label* ***op****, left child* ***l****, and right child* ***r****. If there is such node, return the value number of M. If not, create in the array a new node N with label* ***op****, left child* ***l,*** *and right child* ***r,*** *and return its value number.*

Three-Address Code

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.

x+y*z ⇒ t_1 = y * z

t_2 = x + t_1

- ***Example* 6.4**:

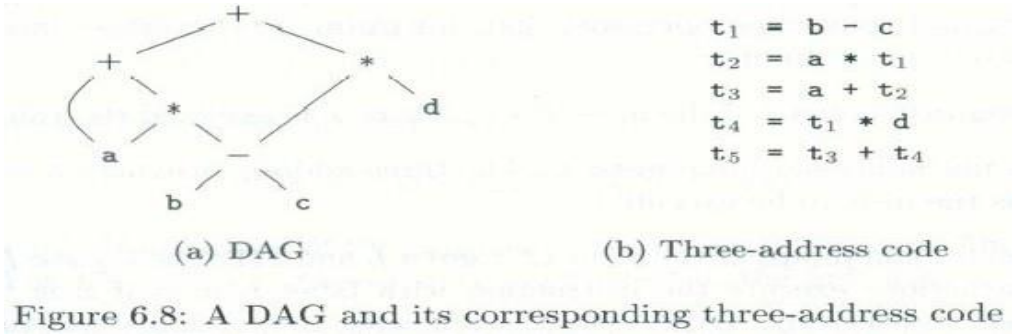


Figure 6.8: A DAG and its corresponding three-address code

Problems: write the 3-address code for the following expression

1. if(x + y * z > x * y +z)a=0;
2. (2 + a * (b – c / d)) / e
3. A :=b * -c + b * -c

Address and Instructions

- Three-address code is built from two concepts: addresses and instructions.
- An address can be one of the following:

–***A name****: A source name is replaced by a pointer to its symbol table entry.*

A name: For convenience, allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.

–***A constant***

A constant: In practice, a compiler must deal with many different types of constants and variables

–***A compiler-generated temporary***

A compiler-generated temporary. It is useful, especially in optimizing

compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, then registers are allocated to variables.

A list of common three-address instruction forms:

Assignment statements

- **x= y op z**, where op is a binary operation
- **x= op y**, where op is a unary operation
- **Copy statement**: x=y
- **Indexed assignments**: x=y[i] and x[i]=y
- **Pointer assignments**: x=&y, *x=y and x=*y

Control flow statements

- **Unconditional jump**: goto L
- **Conditional jump**: if x relop y goto L ; if x goto L; if False x goto L
- **Procedure calls**: call procedure p with n parameters and **return y**, is optional

param x1param x2
... param xn
call p, n

- ***Example* 6.5**:

```
L:  t1 = i + 1                 100:  t1 = i + 1
    i = t1                     101:  i = t1
    t2 = i * 8                 102:  t2 = i * 8
    t3 = a [ t2 ]              103:  t3 = a [ t2 ]
    if t3 < v goto L           104:  if t3 < v goto 100
```

(a) Symbolic labels. (b) Position numbers.

- **do i = i +1; while (a[i]<v);**

The multiplication i * 8 is appropriate for an array of elements that each take 8 units of space.

4. Translate the arithmetic expression a + -(b + c) into quadruples, triples and indirect triples.

Quadruples

- Three-address instructions can be implemented as objects or as record with fields for the operator and operands.
- Three such representations
  - *Quadruple*, *triples*, and *indirect triples*
- A *quadruple* (or *quad*) has four fields: *op*, arg_1, arg_2, and *result*.
- ***Example* 6.6**:

6. About the (i) Quadruples. (ii) Triples (iii) Indirect triples

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

(a) Three-address code

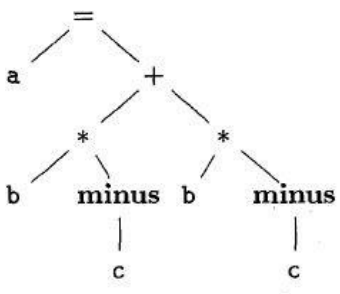
	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

(b) Quadruples

Triples

- A *triple* has only three fields: *op*, arg_1, and arg_2
- Using triples, we refer to the result of an operation *x op y* by its position, rather by an explicit temporary name.

***Example* 6.7**



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

Fig6.11: Representations of a = b * - *c* + **b** * - c

Indirect Triples

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

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Fig6.12: Indirect triples representation of 3-address code

- The 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 changing all references to that result. This problem does not occur with indirect triples.

Static Single-Assignment Form

• Static single assignment form (SSA) is an intermediate representation that facilitates certain code optimization.
• Two distinct aspects distinguish SSA from three –address code.
– All assignments in SSA are to variables with distinct names; hence the term static single-assignment.

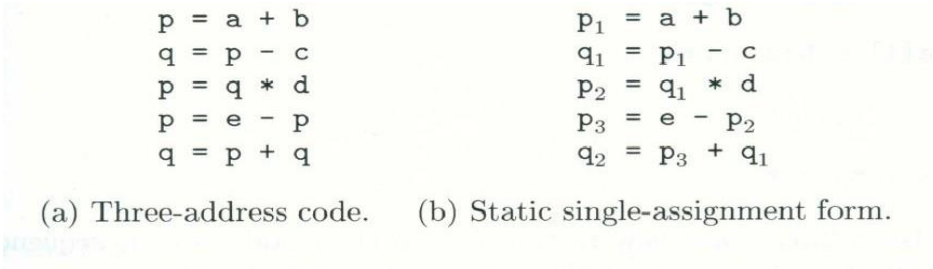


Figure 6.13: Intermediate program in three-address code and SSA

if (flag) x = -1; else x = 1; **y = x * ;a**	If we use different names for X in true part and false part, then conflict arises which name should be considered in y=x*a
if (flag) x_1 = -1; else x_2 = 1; **X_3 = ϕ(x_1, x_2)**	SSA uses a conventional natation Φ to combine the 2 definitions of x. Here Φ(x1,x2) has the value of x1 if true else the value of x2

Translate the arithmetic expression a+_(b+c) into

i. **Syntax tree**
ii. **Quadruples**
iii. **Triples**
iv. **Indirect Triples**

5. About code generator and the three primary task of code generator.

CODE GENERATION

- **The final phase in our compiler model**

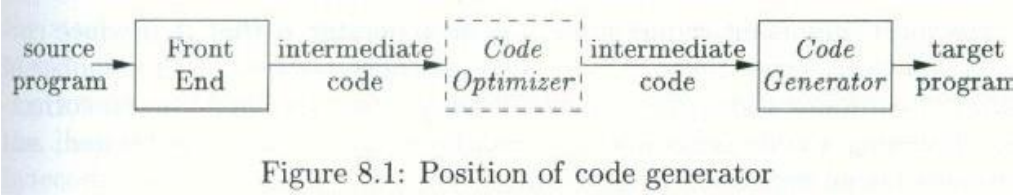


Figure 8.1: Position of code generator

- Requirements imposed on a code generator
  - Preserving the semantic meaning of the source program and being of high quality
  - Making effective use of the available resources of the target machine
  - The code generator itself must run efficiently.
- A code generator has three primary tasks:

Instruction selection, register allocation, and instruction ordering

2. About the issues in the design of a code generator.

Issue in the Design of a Code Generator

- General tasks in almost all code generators: instruction selection, register allocation and assignment.
  - The details are also dependent on the specifics of the intermediate representation, the target language, and the run-tie system.
- The most important criterion for a code generator is that it produces correct code.

Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design

1. Input to the Code Generator

- The input to the code generator is: ***IR+SYMBOL TABLE***
  - the Intermediate representation (IR) of the source program produced by the frontend along with information in the symbol table that is used to determine the run-time address of the data objects denoted by the names in the IR.

Choices for the IR

- Three-address representations: quadruples, triples, indirect triples
- Virtual code machine representations such as bytecodes and stack-machine
- Linear representations such as postfix notation
- Graphical representation such as syntax trees and DAG's
- Assumptions
  - Relatively lower level IR
  - All syntactic and semantic errors are detected.

2. The Target Program

- The output of the code generation is **target program**.
- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code.
- The most common target-machine architecture are RISC, CISC, and stack based.
    - A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
    - A CISC machine typically has few registers, two-address instructions, and variety of addressing modes, several register classes, variable-length instructions, and instruction with side effects.
- Producing the target program as
  - *An absolute machine-language program*: It can be placed in a fixed location in memory and can be executed.
  - *Relocatable machine-language program*: allows subprograms to be compiled seperately
  - *An assembly-language program*: makes the process of code generator much easier

3. Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by the target machine.
- The complexity of the mapping is determined by the factors such as
  - **The level of the IR**
    - IR is high level, use code templates to translate each IR statement into a sequence of machine instruction. Produces poor code, needs further optimization.
    - If the IR reflects some of the low-level details of the underlying machine, then it can use this information to generate more efficient code sequence.
  - **The nature of the instruction-set architecture**
    - Has strong effect on difficulty of instruction selection

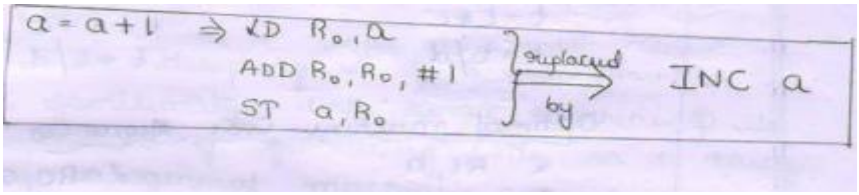
Instruction Selection

- The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For example,
  - The uniformity and completeness of the instruction set are important factors.
  - Instruction speeds and machine idioms are another important factor.
    - If we do not care about the efficiency of the target program, instruction selection is straightforward.

```
x = y + z ⇒    LD   R0, y
               ADD  R0, R0, z
               ST   x, R0

a = b + c ⇒    LD   R0, b
d = a + e      ADD  R0, R0, c
               ST   a, R0
               LD   R0, a      Redundant
               ADD  R0, R0,e
               ST   d, R0
```

– **The quality of the generated code** is usually determined by its speed and size.
  - A given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations.
  - A naïve translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.
  - For example use **INC for a=a+1** instead of

LD R0,a
ADD R0, R0, #1
ST a, R0



- We need to know instruction costs in order to design good code sequences but, unfortunately, accurate cost information is often difficult to obtain.

4. Register Allocation

- A key problem in code generation is deciding what values to hold in what registers.
- Efficient utilization is particularly important.
- The use of registers is often subdivided into two subproblems:

  1.**Register Allocation**, during which we select the set of variables that will reside in registers at each point in the program.

  2.**Register assignment**, during which we pick the specific register that a variable will reside in.

- Finding an optimal assignment of registers to variables is difficult, even with single-register machine.
  - Register pairs (even/odd numbered) for some operands & results
- Multiplication instruction is in the form M x, y where x, the multiplicand, is the even register

of even/odd register pair and y, the multiplier, is the odd register. The product is occupies the entire even/odd register pair.

- D x, y where the dividend occupies an even/odd register pair whose even register is x, the divisor is y. After division, the even register holds the remainder and the odd register the quotient.

Figure 8.2: Two three-address code sequences

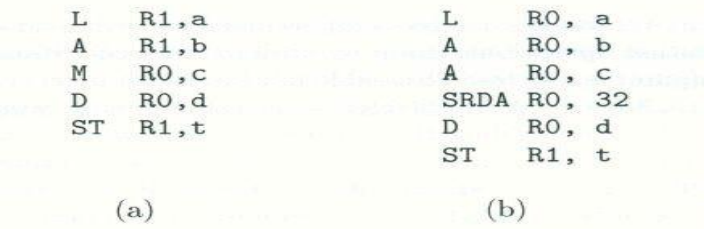
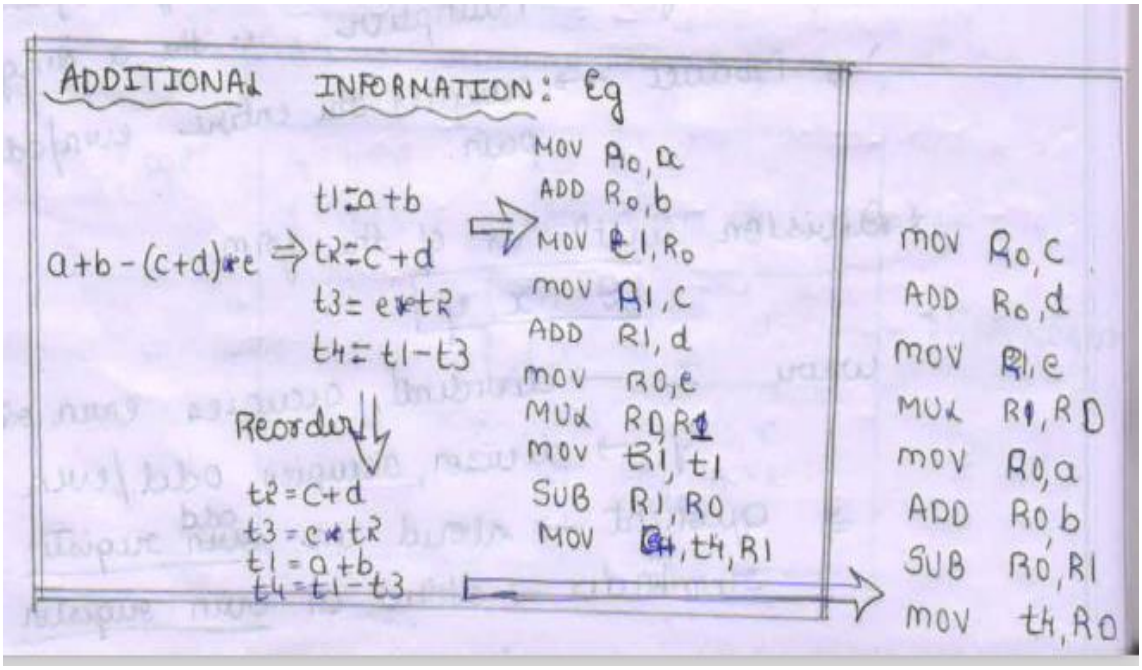


Figure 8.3: Optimal machine-code sequences

1. Generate code for the following expression using the code generator algorithm X := (a + b) * (c - d)

5. *Evaluation Order*

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- However, problem of picking a best order in the general case is a difficult NP-complete



A Simple Target Machine Model

- Our target computer models a **three-address machine** with load and store operations, computation operations, jump operations, and conditional jumps.
- The underlying computer is a byte-addressable machine with n general-purpose registers.
- Assume the following kinds of instructions are available:

– Load operations	**LD dst, addr**
– Store operations Instruction like	**ST x, r**
– Computation operations of the form	**OP dst,src1,src2**
– Unconditional jumps The instruction	**BR L**
– Conditional jumps	**BLTZ r, L**

A Simple Target Machine Model

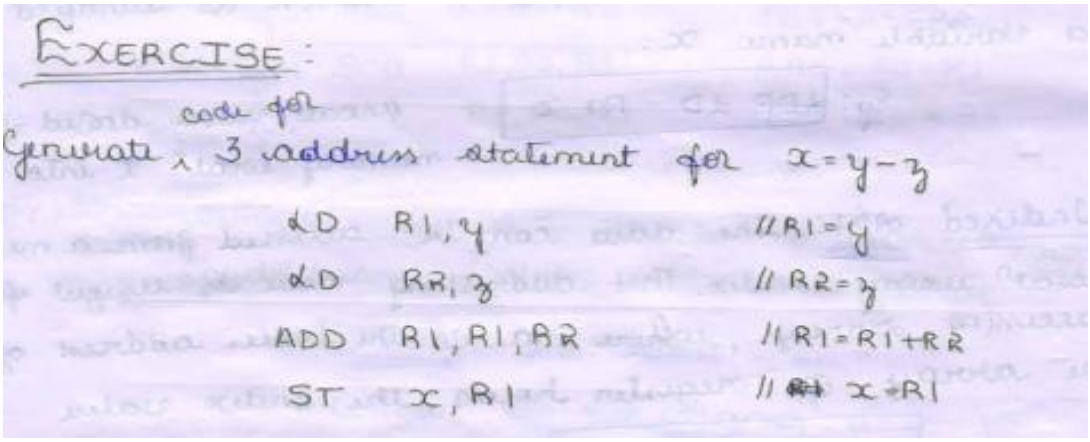
- *Example* **8.2**:

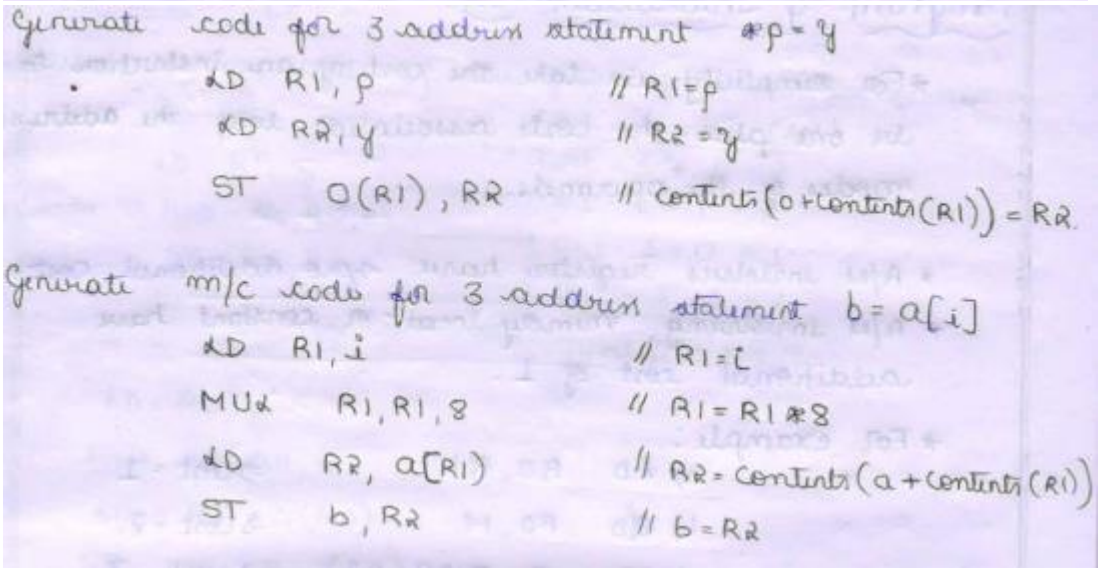
```
x = y -z ⇒ LD  R1, y
           LD  R2, z
           SUB R1, R1, R2
           ST  x, R1

b = a[i] ⇒ LD  R1, i
           MUL R1, R1, 8
           LD  R2, a(R1)
           ST  b, R2

a[j] = c ⇒ LD  R1, c
           LD  R2, j
           MUL R2, R2, 8
           ST  a(R2), R1

x = *p ⇒   LD  R1, p
           LD  R2, 0(R1)
           ST  x, R2

*p = y ⇒   LD  R1, p
           LD  R2, y
           ST  0(R1), R2

if x < y goto L ⇒ LD   R1, x
                  LD   R2, y
                  SUB  R1, R1, R2
                  BLTZ R1, L
```

3. Generate code for the following three-address statements.
x=a[i]
y=b[x]
a[i]=y

Different Addressing Modes supported by generalized target machines

Direct Addressing	LD R1, x	variable name: x
indexed address	LD R1, a(R2)	data is accessed from memory a(r)
integer indexed by a register	LD R1, 100(R2)	R1=contents(a+contents(R2))
Indirect addressing mode	LD R1,*(R2)	Data stored in memory location pointed by R2 is loaded on to reg R1
immediate constant addressing mode	LD R1, #100	R1<- 100





6. Program and Instruction Costs

- For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands.

- Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one.
- For example,
  - **LD R0, R1** **cost = 1**
  - **LD R0, M** **cost = 2**
  - **LD R1, *100(R2)** **cost = 3**

* Cost of Addressing mode:

	Mode	Form	Address	Added Cost
1	Absolute direct A/M	M	M	1
R	Register direct A/M	R	R	0
3	Indexed A/M	C(R)	C+contents(R)	1
R	Indirect register A/M	*R	contents(R)	0
5	Indirect indexed A/M	*C(R)	contents(C+contents(R))	1
6	Immediate A/M	#C	N/A	1

NOTE: Cost of each statement = 1 + cost (Addressing mode)

EXERCISES (8.2)

Determine the costs of the following instruct^{n} sequence

LD R0, y	Cost = 1 + cost(AM)
LD R1, z	Cost = 1 + 1 = 2
ADD R0, R0, R1	Cost = 1 + 1 = 2
ST x, R0	Cost = 1 + 0 = 1
	Cost = 1 + 1 = 2
	Total Cost = 7

```
LD   R0, i
MUL  R0, R0, 8
LD   R1, a(R0)
ST   b, R1
```

	Cost = Cost (AM) + 1
LD R0, i	Cost = 1 + 1 = 2
MUL R0, R0, 8	Cost = 1 + 1 = 2
LD R1, a(R0)	Cost = 1 + 1 = 2
ST b, R1	Cost = 1 + 1 = 2
	Total cost = 8

```
LD   R0, C
LD   R1, i
MUL  R1, R1, 8
ST   a(R1), R0
```

	Cost = cost(A.M) + 1
LD R0, C	1 + 1 = 2
LD R1, i	1 + 1 = 2
MUL R1, R1, 8	1 + 1 = 2
ST a(R1), R0	1 + 1 = 2
	Total cost = 8