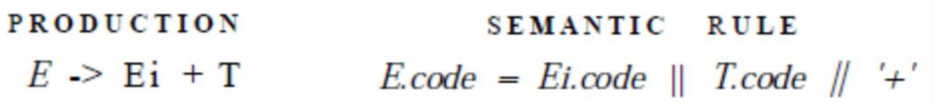
MODULE -5

Syntax Directed Definitions

A *syntax-directed definition* (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions



PRODUCTION	SEMANTIC RULE
E -> Ei + T	$E.code = Ei.code \parallel T.code \parallel$ '+'

A syntax-directed translation (SDT) scheme embeds program fragments called semantic actions within production bodies

E -» Ei + T { print '+' }

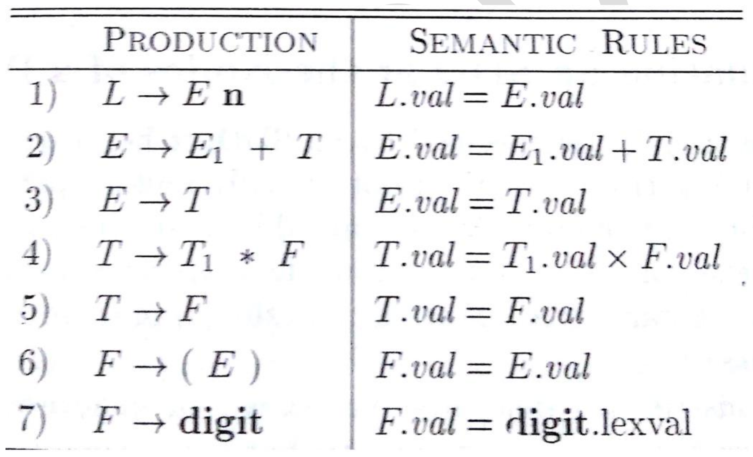
A *synthesized attribute* for a nonterminal *A* at a parse-tree node *N* is defined by a semantic rule associated with the production at *N*. Note that the production must have *A* as its head. A synthesized attribute at node *N* is defined only in terms of attribute values at the children of *N* and at *N* itself.

Production	Semantic rule
E->E+T	E.val=E.val+T.val

2. An *inherited attribute* for a nonterminal B at a parse-tree node *N* is defined by a semantic rule associated with the production at the parent of *N*. Note that the production must have *B* as a symbol in its body. An inherited attribute at node *N* is defined only in terms of attribute values at N's parent, *N* itself, and *N's* siblings.

Production	Semantic rule
E->TE'	E'.inh=T.val
	E.val=E'.syn

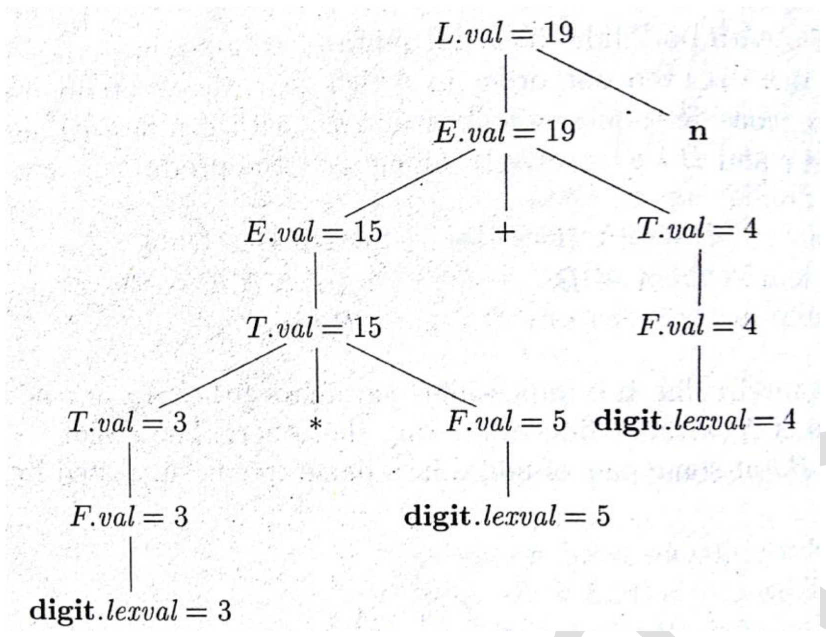
SDD for Simple Desk Calculator



	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

Annotated Parse tree for 3*5+4n

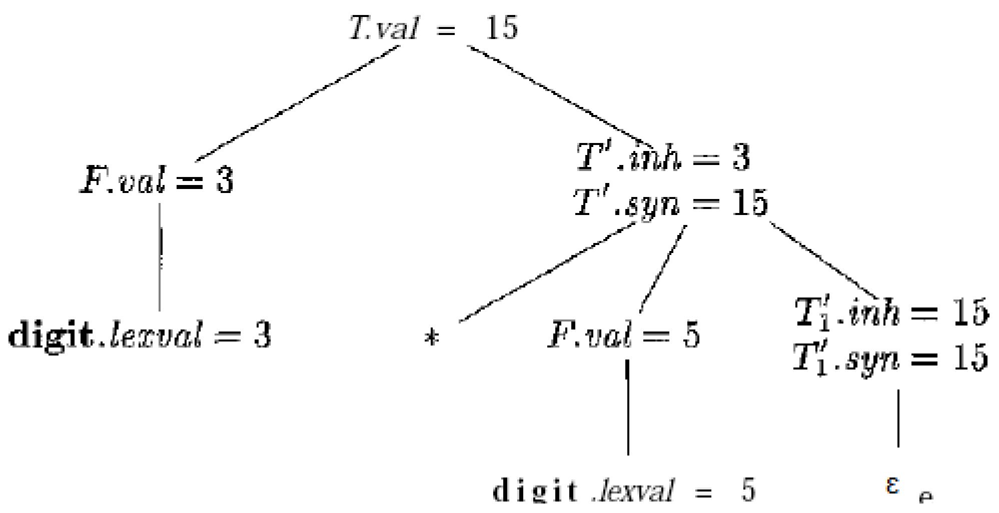
A parse tree, showing the value(s) of its attribute(s) is called an *annotated parse tree.*



An SDD based on a grammar suitable for top-down parsing

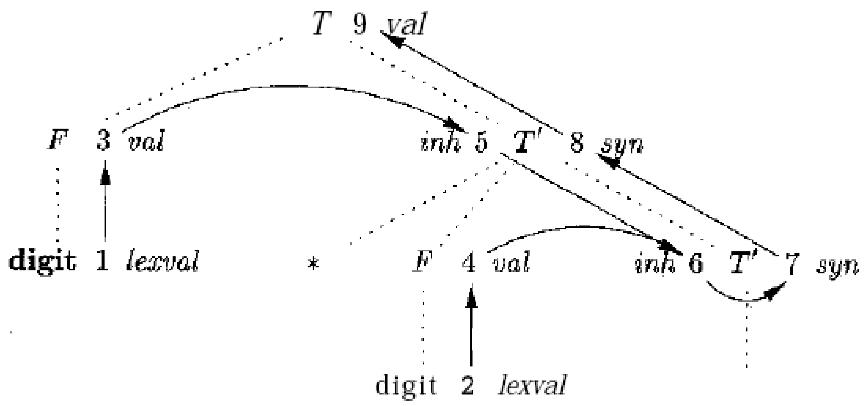
Production	Semantic rule
T→FT’	T'.inh = F.val T.val = T'.syn
T '→*FT$_1$’	T$_1$‘.inh = T'.inh x F.val T'.syn = T$_1$’.syn
T’→ε	T'.syn = T'.inh
F → **digit**	F.val = **digit,** lexval

Annotated parse tree for 3 * 5



Dependency Graphs
A *dependency graph* depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules. In more detail:
• For each parse-tree node, say a node labeled by grammar symbol *X*, the dependency graph has a node for each attribute associated with *X*.
• Suppose that a semantic rule associated with a production *p* defines the value of synthesized attribute *A.b* in terms of the value of *X.c* (the rule may define *A.b* in terms of other attributes in addition to *X.c).* Then, the dependency graph has an edge from *X.c* to *A.b.* More precisely, at every node *N* labeled *A* where production *p* is applied, create an edge to attribute *b* at *N*, from the attribute c at the child of *N* corresponding to this instance of the symbol *X* in the body of the production.2
• Suppose that a semantic rule associated with a production *p* defines the value of inherited attribute *B.c* in terms of the value of *X.a.* Then, the dependency graph has an edge from *X.a* to *B.c.* For each node *N* labelled *B* that corresponds to an occurrence of this *B* in the body of production *p*, create an edge to attribute c at *N* from the attribute *a* at the node *M*.

Dependency Graph for 3 * 5



Ordering the Evaluation of Attributes

The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of a parse tree. If the dependency graph has an edge from node *M* to node *N,* then the attribute corresponding to *M* must be evaluated before the attribute of *N.* Thus, the only allowable orders of evaluation are those sequences of nodes *N1, N2,... ,Nk* such that if there is an edge of the dependency graph from *Ni* to *Nj,* then *i < j.* Such an ordering embeds a directed graph into a linear order, and is called a *topological sort* of the graph.

If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort. To see why, since there are no cycles, we can surely find a node with no edge entering. For if there were no such node, we could proceed from predecessor to predecessor until we came back to some node we had already seen, yielding a cycle. Make this node the first in the topological order, remove it from the dependency graph, and repeat the process on the remaining nodes.

S-Attributed Definitions

An SDD is S-attributed if every attribute is synthesized. When an SDD is S-attributed, we can evaluate its attributes in any bottom up order of the nodes of the parse tree. It is often especially simple to evaluate the attributes by performing a postorder traversal of the parse tree and evaluating the attributes at a node *N* when the traversal leaves *N* for the last time.

```
postorder(N)    {
        for ( each child C of N, from the left ) postorder(C);
        evaluate the attributes associated with node N;
}
```

L-Attributed Definitions

The second class of SDD's is called L-attributed definitions. The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed"). More precisely, each attribute must be either

1. Synthesized, or

2. Inherited, but with the rules limited as follows. Suppose that there is a production A ->X1X2 • • • Xn, and that there is an inherited attribute Xi.a computed by a rule associated with this production. Then the rule may use only:

(a) Inherited attributes associated with the head A.

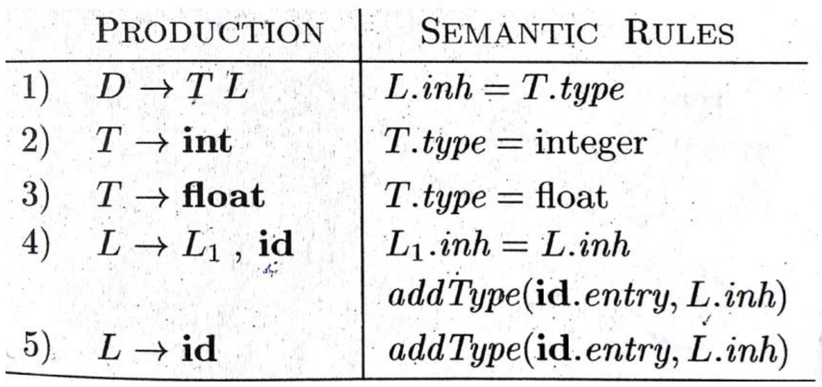
(b) Either inherited or synthesized attributes associated with the occurrences of symbols X1,X2,... , Xi located to the left of Xi.

c) Inherited or synthesized attributes associated with this occurrence of Xi itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X j

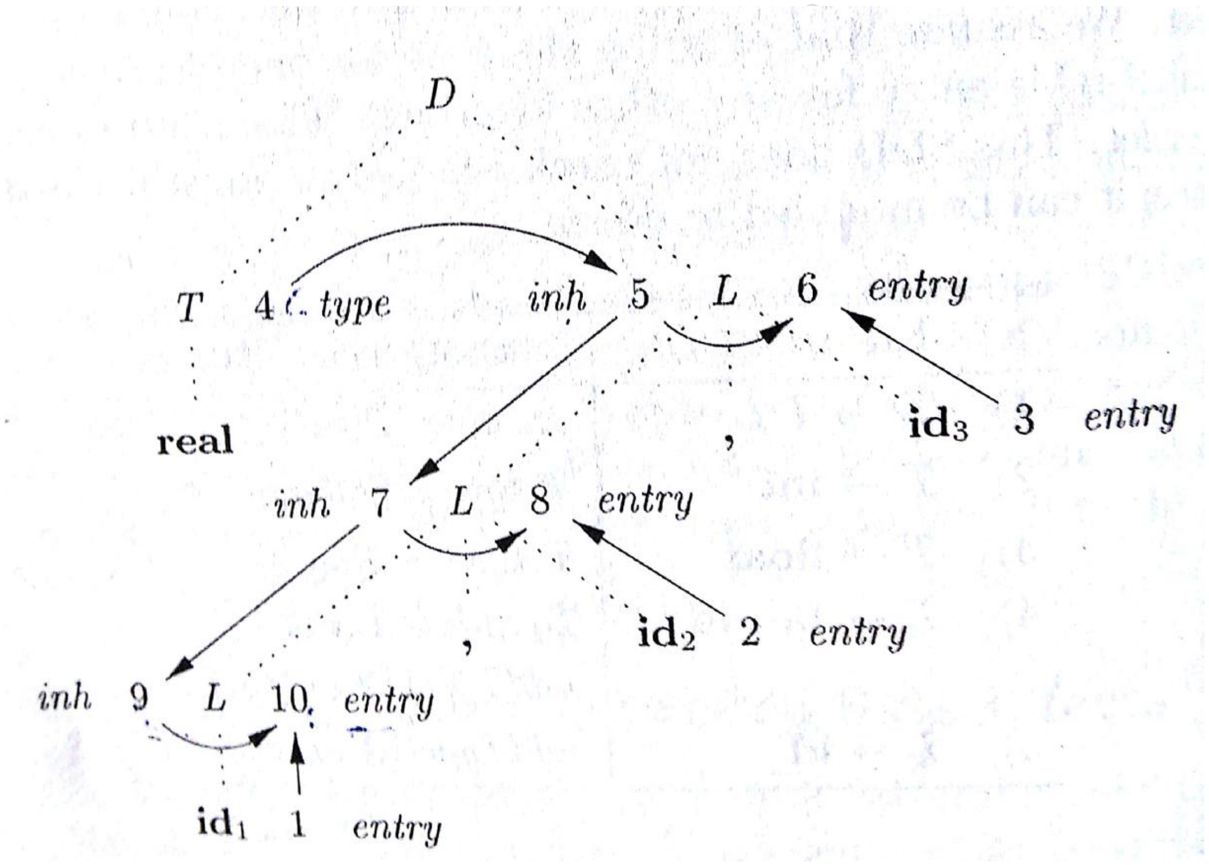
SDD with controlled side effects

In practice, translations involve side effects: a desk calculator might print a result; a code generator might enter the type of an identifier into a symbol table.

SDD to process sample variable declaration in C and the dependency graph for the input float id_1, id_2, id_3



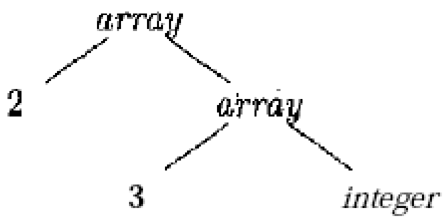
	PRODUCTION	SEMANTIC RULES
1)	$D \to T\ L$	$L.inh = T.type$
2)	$T \to \textbf{int}$	$T.type = \text{integer}$
3)	$T \to \textbf{float}$	$T.type = \text{float}$
4)	$L \to L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$
		$addType(\textbf{id}.entry, L.inh)$
5)	$L \to \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

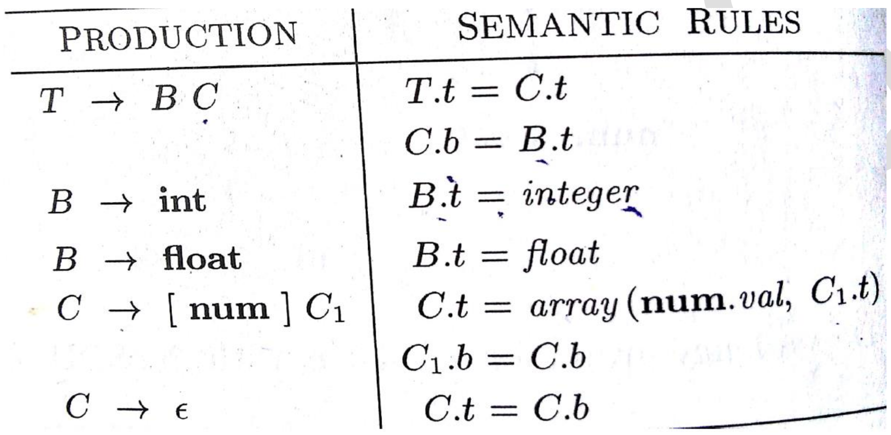


Applications of SDD
The main application in this section is the construction of syntax trees

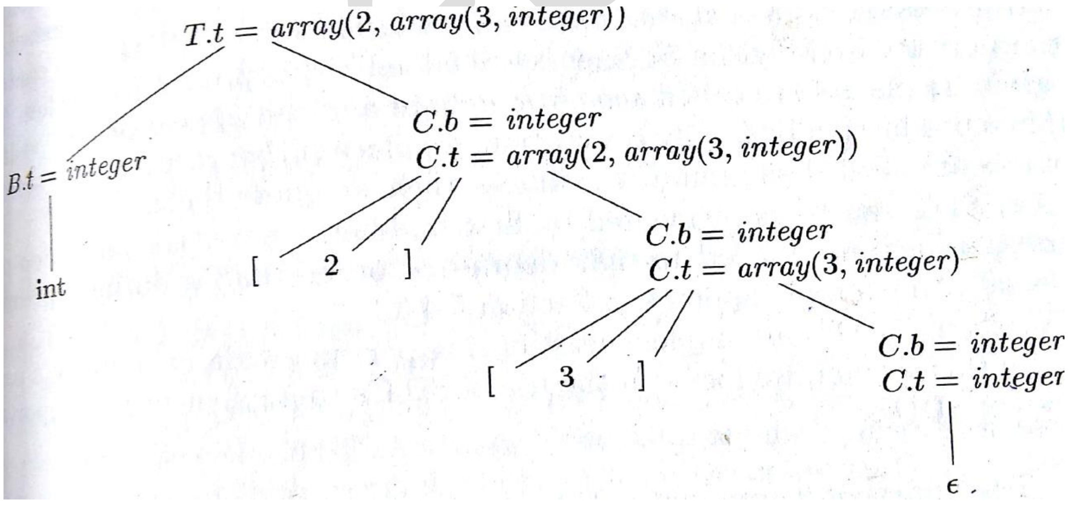
1. SDD of array types and annotated parse tree for int[2][3]

The corresponding type expression *array(2,* array(3, *integer))*

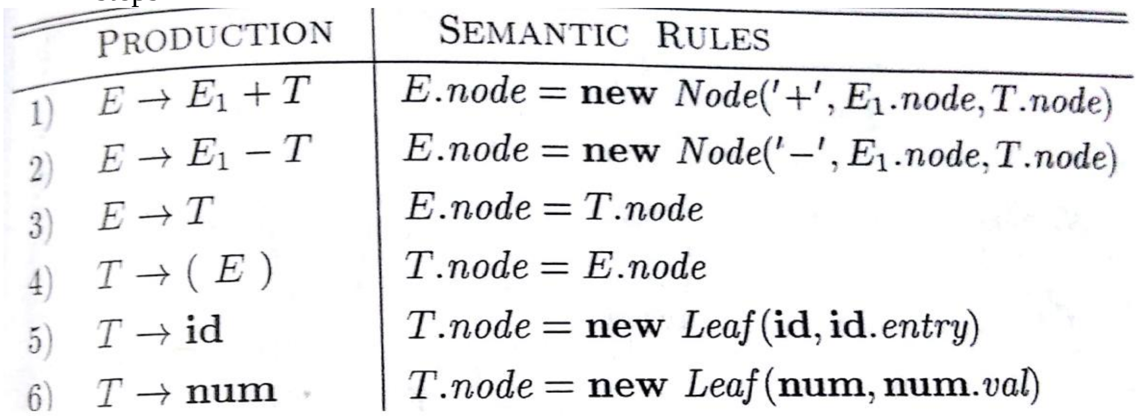




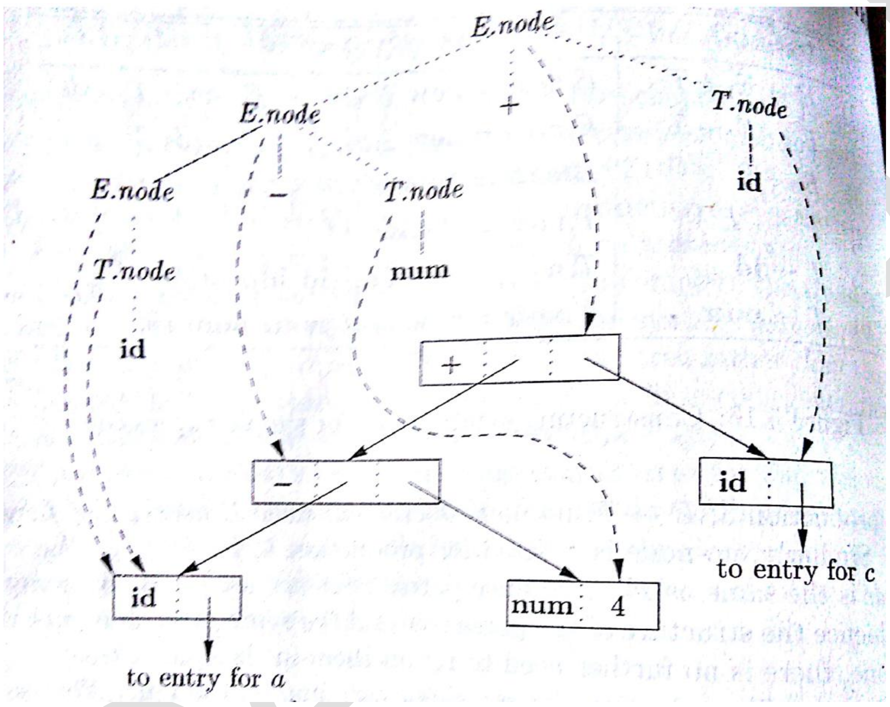
PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [$ **num** $]\ C_1$	$C.t = array($**num**$.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$



2. SDD for syntax tree construction and construct the syntax tree for a-4+c by giving the steps



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



Steps

P1=new Leaf(id,entry-a);

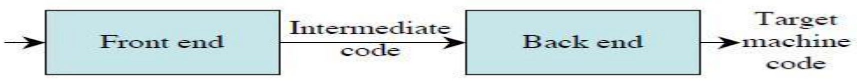
P2=new Leaf(Num,4);

P3=new Node('-',P1,P2);

P4=new Leaf(id,entry-c);

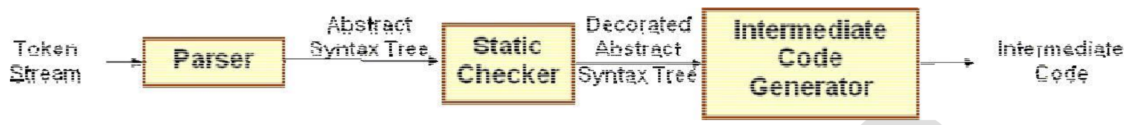
P5=new Node('+',P3,P4);

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 Stucture 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 optimisation 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

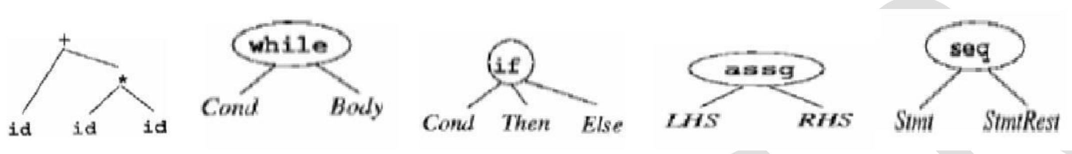
- Specific to the language being implemented o P-
  code for Pascal
  o Bytecode for Java
- Language independent:

o 3-address code

IR can be either an actual language or a group of internal data structures that are shared by the phases of the compiler. C used as intermediate language as it is flexible, compiles into efficient machine code and its compilers are widely available.

In all cases, the intermediate code is a linearization of the syntax tree produced during syntax and semantic analysis. It is formed by breaking down the tree structure into sequential instructions, each of which is equivalent to a single, or small number of machine instructions. Machine code can then be generated (access might be required to symbol tables etc).

Syntax Trees



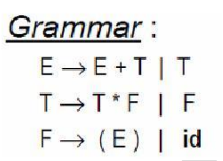
Syntax trees are high level IR. They depict the natural hierarchical structure of the source program. Nodes represent constructs in source program and the children of a node represent meaningful components of the construct. Syntax trees are suited for static type checking.

Variants of Syntax Trees: DAG

A directed acyclic graph (*DAG)* for an expression identifies the *common sub expressions* (sub expressions that occur more than once) of the expression. DAG's can be constructed by using the same techniques that construct syntax trees.

A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. A node N in a DAG has more than one parent if N represents a common sub expression, so a DAG represents expressions concisely. It gives clues to compiler about the generating efficient code to evaluate expressions.

Example 1: Given the grammar below, for the input string **id + id * id** , the parse tree, syntax tree and the DAG are as shown.

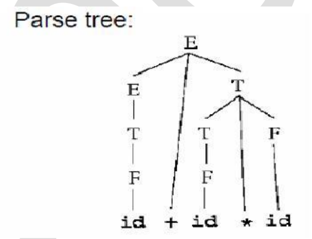


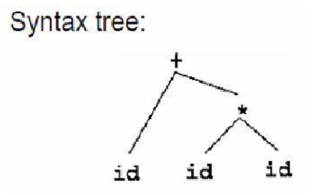
Grammar :

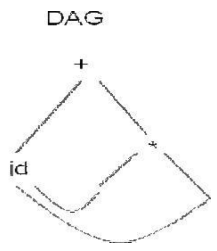
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

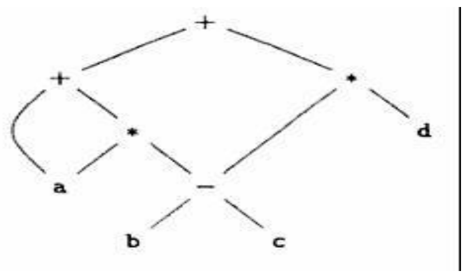
$F \rightarrow (E) \mid$ id







Example 2: DAG for the expression a + a * (b - c) + (b - c) * d is shown below.



Using the SDD to draw syntax tree or DAG for a given expression:-

- Draw the parse tree
- Perform a post order traversal of the parse tree
- Perform the semantic actions at every node during the traversal
  - – Creates a syntax tree if a new node is created each time functions Leaf and Node are called
  - – Constructs a DAG if before creating a new node, these functions check whether an identical node already exists. If yes, the existing node is returned.

SDD to produce Syntax trees or DAG is shown below.

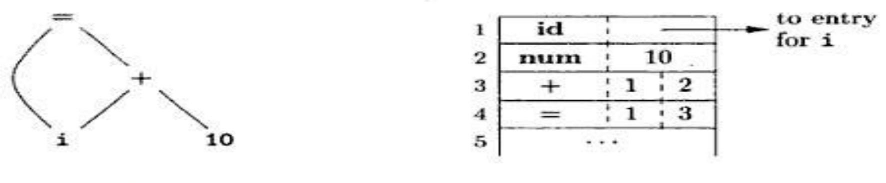
	Production	Semantic Rules
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

For the expression a + a * (b – c) + (b - c) * d, steps for constructing the DAG is as below.

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) $p_{13} = Node('+', p_7, p_{12})$

Value-Number Method for Constructing DAGs

Nodes of a syntax tree or DAG are stored in an array of records. The integer index of the record for a node in the array is known as the **value number** of that node.



(a) DAG (b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

The signature of a node is a triple < op, l, r> where op is the label, l the value number of its left child, and r the value number of its right child. The value-number method for constructing the nodes of a DAG uses the signature of a node to check if a node with the same signature already exists in the array. If yes, returns the value number. Otherwise, creates a new node with the given signature.

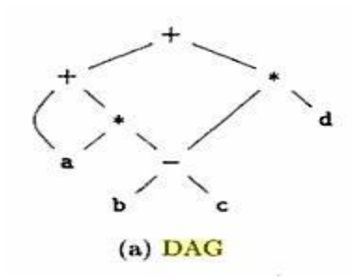
Since searching an unordered array is slow, there are many better data structures to use. Hash tables are a good choice.

Three Address Code(TAC)

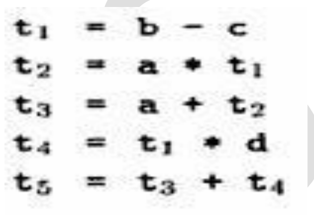
TAC can range from high- to low-level, depending on the choice of operators. In general, it is a statement containing at most 3 addresses or operands.

The general form is x := y op z, where "op" is an operator, x is the result, and y and z are operands. **x**, **y**, **z** are variables, constants, or "temporaries". A three-address instruction consists of at most 3 addresses for each statement.

It is a linearized representation of a binary syntax tree. Explicit names correspond to interior nodes of the graph. E.g. for a looping statement , syntax tree represents components of the statement, whereas three-address code contains labels and jump instructions to represent the flow-of-control as in machine language.



(a) DAG



$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

A TAC instruction has at most one operator on the RHS of an instruction; no built-up arithmetic expressions are permitted.

e.g. x + y * z can be translated as t_1 =
y * z t_2 = x +
t_1

where t_1 & t_2 are compiler–generated temporary names.

Since it unravels multi-operator arithmetic expressions and nested control-flow statements, it is useful for target code generation and optimization.

Addresses and Instructions

- TAC consists of a sequence of instructions, each instruction may have up to three addresses, prototypically t1 = t2 op t3
- Addresses may be one of:

  – A name. Each name is a symbol table index. For convenience, we write the names as the identifier.

  – A constant.

  – A compiler-generated temporary. Each time a temporary address is needed, the compiler generates another name from the stream t1, t2, t3, etc.

    - Temporary names allow for code optimization to easily move instructions
    - At target-code generation time, these names will be allocated to registers or to memory.

- TAC Instructions
  - Symbolic labels will be used by instructions that alter the flow of control.

    L: t1 = t2 op t3

  - Assignment instructions: x = y op z
    - Includes binary arithmetic and logical operations
  - Unary assignments: x = op y
    - Includes unary arithmetic op (-) and logical op (!) and type conversion
  - Copy instructions: x = y
  - Unconditional jump: goto L
    - L is a symbolic label of an instruction
  - Conditional jumps:

    if x goto L If x is true, execute instruction L next ifFalse x goto L If x is false, execute instruction L next
  - Conditional jumps:

    if x relop y goto L
  - Procedure calls. For a procedure call p(x_1, …, x_n)

    param x_1
    …
    param x_n

    call p, n
  - Function calls : y= p(x_1, …, x_n)   y = call p,n , return y
  - Indexed copy instructions: x = y[i]   and x[i] = y
    - Left: sets x to the value in the location i memory units beyond y
    - Right: sets the contents of the location i memory units beyond x to y
  - Address and pointer instructions:
    - x = &y sets the value of x to be the location (address) of y.
    - x = *y, presumably y is a pointer or temporary whose value is a location. The value of x is set to the contents of that location.
    - *x = y sets the value of the object pointed to by x to the value of y.

Example: Given the statement **do i = i+1; while (a[i] < v); ,** the TAC can be written as below in two ways, using either symbolic labels or position number of instructions for labels.

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

(a) Symbolic labels.

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

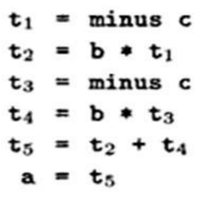
(b) Position numbers.

Three Address Code Representations

Data structures for representation of TAC can be objects or records with fields for operator and operands. Representations include quadruples, triples and indirect triples.

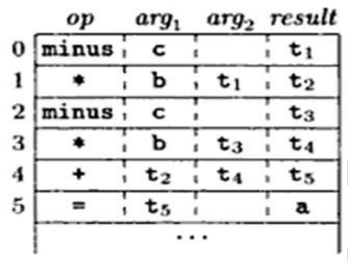
Quadruples

- In the quadruple representation, there are four fields for each instruction: *op, arg1, arg2, result*
  - Binary ops have the obvious representation
  - Unary ops don't use arg2
  - Operators like param don't use either arg2 or result
  - Jumps put the target label into result
- The quadruples in Fig (b) implement the three-address code in (a) for the expression a = b * - c + b * - c



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

(a) Three-address code



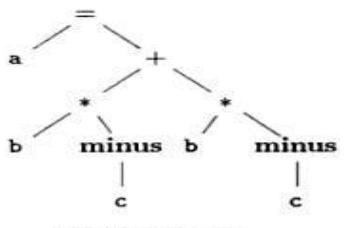
	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

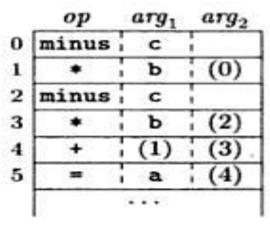
Triples

- A triple has only three fields for each instruction: *op, arg1, arg2*
- The *result* of an operation *x op y* is referred to by its position.
- Triples are equivalent to signatures of nodes in DAG or syntax trees.
- Triples and DAGs are equivalent representations only for expressions; they are not equivalent for control flow.
- Ternary operations like ***x[i]*** = *y* requires two entries in the triple structure, similarly for x = y[i].
- Moving around an instruction during optimization is a problem

Example: Representations of a =b *– c + b * – c



(a) Syntax tree

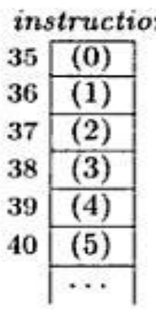


	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

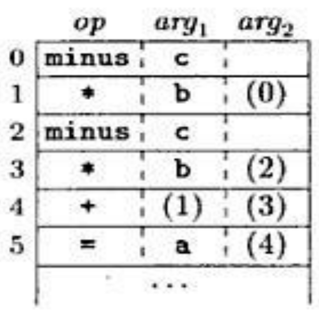
(b) Triples

Indirect Triples

These consist of a listing of pointers to triples, rather than a listing of the triples themselves. An optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.



	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)
		...	

Construct the quadruple, triple and indirect triple for a+-(b+c)

TAC
t1=b+c
t2=-t1
t3=a+t2

Quadruples

	op	Arg1	Arg2	Result
0	+	b	c	t1
1	-	t1		t2
2	+	a	t2	t3

Triples

	op	Arg1	Arg2
0	+	b	c
1	-	(0)	
2	+	a	(1)

Indirect Triples

41	(0)
42	(1)
43	(2)

	op	Arg1	Arg2
0	+	b	c
1	-	(0)	
2	+	a	(1)

Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code.

- All assignments in SSA are to variables with distinct names; hence *static single-assignment.*
- Φ-FUNCTION

Same variable may be defined in two different control-flow paths. For example,

if (flag) x = -1; else x = 1; y = x * a;

using Φ-function it can be written as

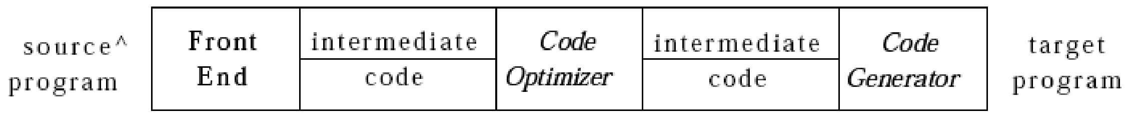
if (flag) x_1 = -1; else x_2 = 1;

$x_3 = \Phi(x_1,x_2)$;

y = x3 * a;

Code generation

The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program



A code generator has three primary tasks: instruction selection, register allocation and assignment, and instruction ordering.

Issues in the design of code generator

Instruction selection involves choosing appropriate target-machine instructions to implement the IR statements. Register allocation and assignment involves deciding what values to keep in which registers. Instruction ordering involves deciding in what order to schedule the execution of instructions.

Input to the Code Generator

The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR. The many choices for the IR include three-address representations such as quadruples, triples, indirect triples; virtual machine representations such as bytecodes and stack-machine code; linear representations such as postfix notation; and graphical representations such as syntax trees and DAG's.

The 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 architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based. A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture. In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers. Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many swap and copy operations.
Producing an absolute machine-language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Programs can be compiled and executed quickly.
Producing a relocatable machine-language program (often called an *object module)* as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added

expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program modules.
Producing an assembly-language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate code. The price paid is the assembly step after code generation.

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 performing this mapping is determined by a factors such as
• the level of the IR
• the nature of the instruction-set architecture
• the desired quality of the generated code.
If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates.

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. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. On some machines, for example, floating-point operations are done using separate registers.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct.

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

```
a = b + c
d = a + e
```

would be translated into

```
LD   RO, b        // RO = b
ADD  RO, RO, c    // RO = RO
ST   a, RO        // a =RO
LD   RO, a        // RO = a
ADD  RO, RO, e    // RO = RO
ST   d, RO        // d =RO
```

Register Allocation
A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

The use of registers is often subdivided into two sub problems:
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 machines. Mathematically, the problem is NP-complete.

Three Address code

```
t = a + b
t = t * c
t = t/ d
```

Optimal machine code

```
L   R1,a
A   R1,b
M   R0,c
D   R0,d
ST  R1,t
```

Evaluation Order
The order in which computations are performed can affect the efficiency of the target code. As we shall see, some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem. Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

The Target Language
Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. Unfortunately, in a general discussion of code generation it is not possible to describe any target machine in sufficient detail to generate good code for a complete language on that machine.

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, R0,R1,... ,Rn - 1. A full-fledged assembly language would have scores of instructions. To avoid hiding the concepts in a myriad of details, we shall use a very limited set of instructions and assume that all operands are integers. Most instructions consists of an operator, followed by a target, followed by a list of source operands. A label may precede an instruction.

We assume the following kinds of instructions are available:
•*Load* operations: The instruction **LD** *dst, addr* loads the value in location *addr* into location *dst* This instruction denotes the assignment *dst* = *addr*. The most common form of this instruction is **LD** *r, x* which loads the value in location *x* into register r. An instruction of the form **LD** *r±,r2* is a *register-to-register copy* in which the contents of register *r2* are copied

into register *r\.*
• *Store* operations: The instruction **ST** *x, r* stores the value in register *r* into
the location *x.* This instruction denotes the assignment *x = r.*
• *Computation* operations of the form *OP dst, src\,src2,* where *OP* is a operator like **ADD** or **SUB,** and *dst,* ***srcy,*** and *src2* are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by *OP to* the values in locations ***srci*** and *src2,* and place the result of this operation in location *dst.* For example, **SUB** *n,r2,*r3 computes *n = r2 - r3.* Any value formerly stored in *n* is lost, but if *r* is
r2 or r 3 , the old value is read first. Unary operators that take only one
operand do not have a *src*
•*Unconditional jumps:* The instruction BR *L* causes control to branch to
the machine instruction with label *L.* (BR stands for *branch.)*
• *Conditional jumps* of the form *Bcond* r, *L,* where *r* is a register, *L* is a label,
and *cond* stands for any of the common tests on values in the register r. For example, BLTZ r, *L* causes a jump to label *L* if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.
In instructions, a location can be a variable name *x* referring to the memory location that is reserved for *x* (that is, the /-value of *x).*
• A location can also be an indexed address of the form *a(r),* where *a* is a variable and r is a register. The memory location denoted by *a(r)* is computed by taking the I-value of *a* and adding to it the value in register r. For example, the instruction LD Rl, a**(R2)** has the effect of setting Rl = *contents* (a + *contents* **(R2)),** where *contentsix)* denotes the contents of the register or memory location represented by *x.* This addressing mode is useful for accessing arrays, where *a* is the base address of the array (that is, the address of the first element), and r holds the number of bytes past that address we wish to go to reach one of the elements of array *a.*
• A memory location can be an integer indexed by a register. For example, LD Rl, **100(R2)** has the effect of setting Rl = *contents(100 + content*s**(R2)),** that is, of loading into Rl the value in the memory location obtained by adding **100** to the contents of register **R2.** This feature is useful for following pointers, as we shall see in the example below-
• We also allow two indirect addressing modes: **r* means the memory location found in the location represented by the contents of register r and ***100**(r) means the memory location found in the location obtained by adding **100** to the contents of r. For example, LD Rl, ***100(R2)** has the effect of setting Rl = *contents(contents(10Q + content*s**(R2))),** that is, of loading into Rl the value in the memory location stored in the memory location obtained by adding **100** to the contents of register **R2.**
• Finally, we allow an immediate constant addressing mode. The constant is prefixed by #. The instruction LD Rl, **#100** loads the integer **100** into register Rl, and ADD Rl, Rl, **#100** adds the integer **100** into register Rl. '

The three-address statement x = y - z can be implemented by
the machine instructions:

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

Suppose a is an array whose elements are 8-byte values, perhaps real numbers.
Also assume elements of a are indexed starting at 0. We may execute the

three-address instruction b = a [i] by the machine instructions:

```
LD   R1, i          // R1 = i
MUL  R1, R1, 8      // R1 = R1 * 8
LD   R2, a(R1)      // R2 = contents(a + contents(R1))
ST   b, R2          // b = R2
```

Similarly, the assignment into the array a represented by three-address instruction
a [j] = **c** is implemented by:

```
LD   R1, c          // R1 = c
LD   R2, j          // R2 = j
MUL  R2, R2, 8      // R2 = R2 * 8
ST   a(R2), R1      // contents(a +  contents(R2)) = R1
```

To implement a simple pointer indirection, such as the three-address statement
x = *p, we can use machine instructions like:

```
LD   R1, p          // R1 = p
LD   R2, y          // R2 = y

ST   0(R1), R2      // contents(0 + contents(R1)) = R2
```

Finally, consider a conditional-jump three-address instruction like
`if x < y goto L`

The machine-code equivalent would be something like:

```
LD    R1, x         // R1 = x
LD    R2, y         // R2 = y
SUB   R1, R1, R2    // R1 = R1 - R2
BLTZ  R1, M         // if R1 < 0 jump to M
```

Here, M is the label that represents the first machine instruction generated from the three-address instruction that has label L.

Program and instruction costs

Determining the actual cost of compiling and running a program is a complex

problem. Finding an optimal target program for a given source program is an undecidable problem in general, and many of the subproblems involved are

NP-hard.
For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands. This cost corresponds to the length in words of the instruction.
Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction.
Some examples:
• The instruction LD RO, Rl copies the contents of register Rl into register RO. This instruction has a cost of one because no additional memory words are required.
• The instruction LD RO, M loads the contents of memory location M into register RO. The cost is two since the address of memory location M is in the word following the instruction.
• The instruction LD Rl, *100(R2) loads into register Rl the value given by *contents(contents(100 + contents(K2)))*. The cost is three because the constant 100 is stored in the word following the instruction.