UNIT 6

SEMANTIC ANALYSIS

6.1 SEMANTIC ANALYSIS

- Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a Representation Formalism and an Implementation Mechanism.
- As representation formalism this lecture illustrates what are called Syntax Directed Translations.

6.2 SYNTAX DIRECTED TRANSLATION

- The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
    - We associate Attributes to the grammar symbols representing the language constructs.
    - Values for attributes are computed by Semantic Rules associated with grammar productions.
- Evaluation of Semantic Rules may:
    - Generate Code;
    - Insert information into the Symbol Table;
    - Perform Semantic Check;
    - Issue error messages;
    - etc.

There are two notations for attaching semantic rules:

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

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

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

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

- Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g.,X.a indicates the attribute a of the grammar symbol X).
- The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

We distinguish between two kinds of attributes:

1. **Synthesized Attributes.** They are computed from the values of the attributes of the children nodes.

2. **Inherited Attributes.** They are computed from the values of the attributes of both the siblings and the parent nodes

Syntax Directed Definitions: An Example

• **Example.** Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

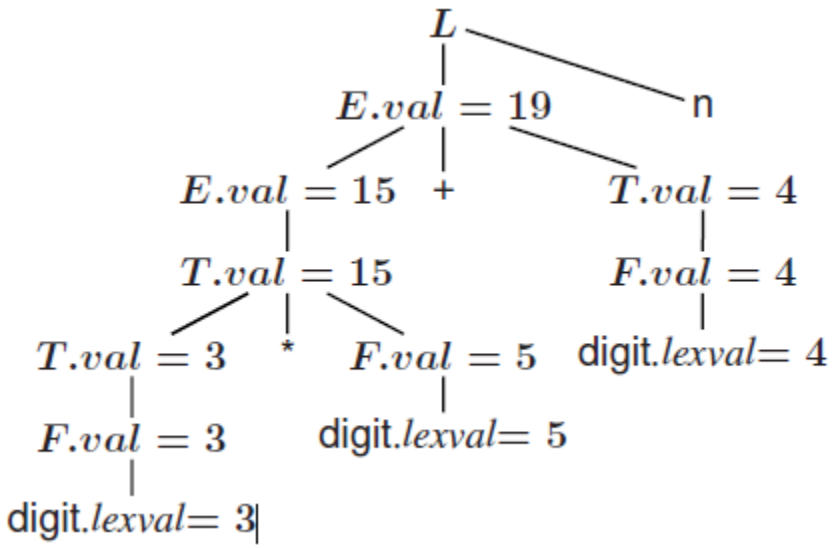
PRODUCTION	SEMANTIC RULE
$L \rightarrow E\text{n}$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \text{digit}$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

6.3 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 above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input 3*5+4n is:



6.4 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$\square$1 (symbols to the left of Xi in the RHS)

2. inherited attributes of A.

Restrictions for translation schemes:

1. Inherited attribute of Xi must be computed by an action before Xi.

2. An action must not refer to synthesized attribute of any symbol to the right of that action.

3. Synthesized attribute for A can only be computed after all attributes it references have been completed (usually at end of RHS).

6.5 SYMBOL TABLES

A symbol table is a major data structure used in a compiler. Associates attributes with identifiers used in a program. For instance, a type attribute is usually associated with each identifier. A symbol table is a necessary component Definition (declaration) of identifiers appears once in a program .Use of identifiers may appear in many places of the program text Identifiers and attributes are entered by the analysis phases. When processing a definition (declaration) of an identifier. In simple languages with only global variables and implicit declarations. The scanner can enter an identifier into a symbol table if it is not already there In block-structured languages with scopes and explicit declarations:

- The parser and/or semantic analyzer enter identifiers and corresponding attributes
- Symbol table information is used by the analysis and synthesis phases
- To verify that used identifiers have been defined (declared)
- To verify that expressions and assignments are semantically correct – type checking
- To generate intermediate or target code

✓ Symbol Table Interface

The basic operations defined on a symbol table include:

- allocate – to allocate a new empty symbol table
- free – to remove all entries and free the storage of a symbol table
- insert – to insert a name in a symbol table and return a pointer to its entry

- lookup – to search for a name and return a pointer to its entry
- set_attribute – to associate an attribute with a given entry
- get_attribute – to get an attribute associated with a given entry

Other operations can be added depending on requirement For example, a delete operation removes a name previously inserted Some identifiers become invisible (out of scope) after exiting a block

- This interface provides an abstract view of a symbol table
- Supports the simultaneous existence of multiple tables
- Implementation can vary without modifying the interface

Basic Implementation Techniques

- First consideration is how to insert and lookup names
- Variety of implementation techniques
- Unordered List
- Simplest to implement
- Implemented as an array or a linked list
- Linked list can grow dynamically – alleviates problem of a fixed size array
- Insertion is fast $O(1)$, but lookup is slow for large tables – $O(n)$ on average
- Ordered List
- If an array is sorted, it can be searched using binary search – $O(\log 2\ n)$
- Insertion into a sorted array is expensive – $O(n)$ on average
- Useful when set of names is known in advance – table of reserved words
- Binary Search Tree
- Can grow dynamically
- Insertion and lookup are $O(\log 2\ n)$ on average

6.6 HASH TABLES AND HASH FUNCTIONS

- ✓ A hash table is an array with index range: 0 to *TableSize* – 1
- ✓ Most commonly used data structure to implement symbol tables
- ✓ Insertion and lookup can be made very fast – $O(1)$
- ✓ A hash function maps an identifier name into a table index

- ✓ A hash function, *h*(*name*), should depend solely on *name*
- ✓ *h*(*name*) should be computed quickly
- ✓ *h* should be uniform and randomizing in distributing names
- ✓ All table indices should be mapped with equal probability.
- ✓ Similar names should not cluster to the same table index

6.7 HASH FUNCTIONS

_ Hash functions can be defined in many ways . . .

_ A string can be treated as a sequence of integer words

_ Several characters are fit into an integer word

_ Strings longer than one word are folded using exclusive-or or addition

_ Hash value is obtained by taking integer word modulo *TableSize*

_ We can also compute a hash value character by character:

_ *h*(*name*) = (*c*0 + *c*1 + … + *cn*–1) mod *TableSize*, where *n* is *name* length

_ *h*(*name*) = (*c*0 * *c*1 * … * *cn*–1) mod *TableSize*

_ *h*(*name*) = (*cn*–1 + ___ *cn*–2 + … + ___ *c*1 + __*c*0))) mod *TableSize*

_ *h*(*name*) = (*c*0 * *cn*–1 * *n*) mod *TableSize*

6.8 RUNTIME ENVIRONMENT

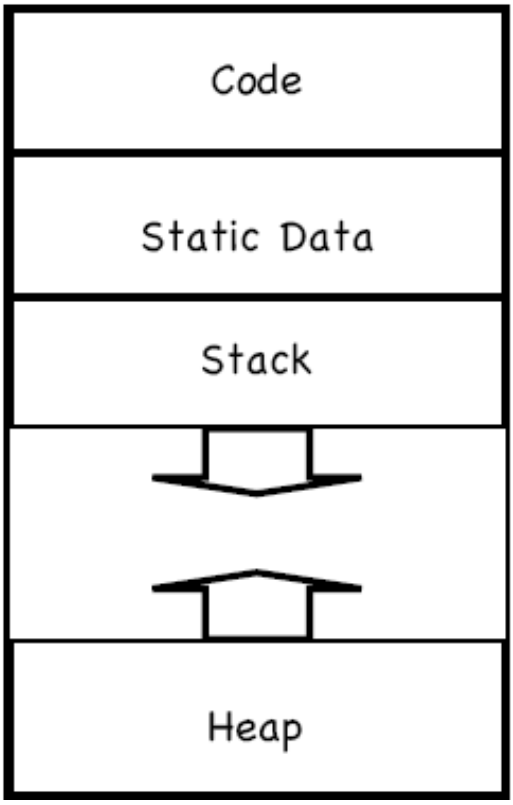
- ➢ Runtime organization of different storage locations
- ➢ Representation of scopes and extents during program execution.
- ➢ Components of executing program reside in blocks of memory (supplied by OS).
- ➢ Three kinds of entities that need to be managed at runtime:
  - o Generated code for various procedures and programs.
- forms text or code segment of your program: size known at compile time.
  - o Data objects:
- Global variables/constants: size known at compile time
- Variables declared within procedures/blocks: size known
- Variables created dynamically: size unknown.
  - o Stack to keep track of procedure activations.
- Subdivide memory conceptually into code and data areas:

- Code: Program

- instructions
  - Stack: Manage activation of procedures at runtime.
  - Heap: holds variables created dynamically

6.9 STORAGE ORGANIZATION

1*Fixed-size objects can be placed in predefined locations.*



2. Run-time stack and heap

The STACK is used to store:

- Procedure activations.
  - The status of the machine just before calling a procedure, so that the status can be restored when the called procedure returns.
  - The HEAP stores data allocated under program control (e.g. by malloc() in C).

Activation records

Any information needed for a single activation of a procedure is stored in the ACTIVATION RECORD (sometimes called the STACK FRAME). Today, we'll assume the stack grows DOWNWARD, as on, e.g., the Intel architecture. The activation record gets pushed for each procedure call and popped for each procedure return.

6.9 STATIC ALLOCATION

Statically allocated names are bound to storage at compile time. Storage bindings of statically allocated names never change, so even if a name is local to a procedure, its name is always bound to the same storage. The compiler uses the type of a name (retrieved from the symbol table) to determine storage size required. The required number of bytes (possibly aligned) is set aside for the name.The address of the storage is fixed at compile time.

Limitations:

- The size required must be known at compile time.
- Recursive procedures cannot be implemented as all locals are statically allocated.
- No data structure can be created dynamically as all data is static.

❖ **Stack-dynamic allocation**

- ✓ Storage is organized as a stack.
- ✓ Activation records are pushed and popped.
- ✓ Locals and parameters are contained in the activation records for the call.
- ✓ This means locals are bound to fresh storage on every call.
- ✓ If we have a stack growing downwards, we just need a stack_top pointer.
- ✓ To allocate a new activation record, we just increase stack_top.
- ✓ To deallocate an existing activation record, we just decrease stack_top.

❖ **Address generation in stack allocation**

The position of the activation record on the stack cannot be determined statically. Therefore the compiler must generate addresses RELATIVE to the activation record. If we have a downward-growing stack and a stack_top pointer, we generate addresses of the form stack_top + offset

6.10 HEAP ALLOCATION

Some languages do not have tree-structured allocations. In these cases, activations have to be allocated on the heap. This allows strange situations, like callee activations that live longer than their callers' activations. This is not common Heap is used for allocating space for objects created at run timeFor example: nodes of dynamic data structures such as linked lists and trees

Dynamic memory allocation and deallocation based on the requirements of the program*malloc()* and *free()* in C programs

*new()*and *delete()*in C++ programs

*new()*and garbage collection in Java programs

Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic*(Java), or *fully automatic* (Lisp)

6.11 PARAMETERS PASSING

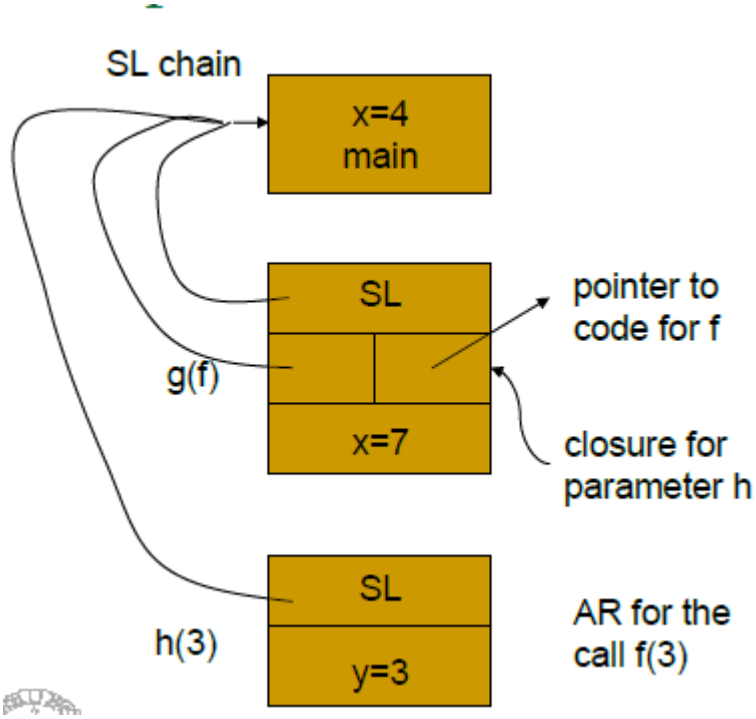
A language has first-class functionsif functions can bedeclared within any scope passed as arguments to other functions returned as results of functions.In a language with first-class functions and static scope, a function value is generally represented by a closure. a pair consisting of a pointer to function code a pointer to an activation record.Passing functions as arguments is very useful in structuring of systems using upcalls

An example:

```
main()
{ int x = 4;
int f (int y) {
return x*y;
}
int g (int →int h){
int x = 7;
return h(3) + x;
}
```

g(f);//returns 12
}

Passing Functions as Parameters – Implementation with Static Scope



An example:

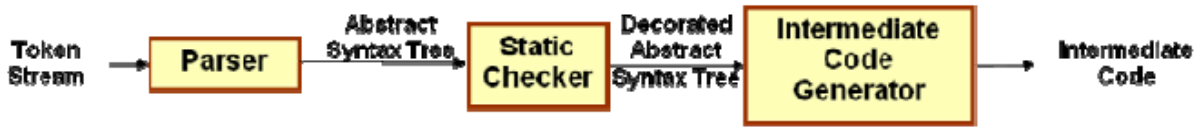
```
main()
{ int x = 4;
  int f (int y) {
     return x*y;
  }
  int g (int → int h){
     int x = 7;
     return h(3) + x;
   }
   g(f);// returns 12
}
```

INTERMEDIATE CODE

7.1. 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

7.2 LANGUAGE INDEPENDENT 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). 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 linear zed 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 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

t1 = y * z

t2 = x + t1

Where t1 & t2 are compiler–generated temporary names.

5Since 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 writethe 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.

The instruction addresses of labels will be filled in later.

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(x1, …, xn)

param x1

…

param xn

call p, n

– Function calls : y= p(x1, …, xn) 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.

Types of three address code

There are different types of statements in source program to which three address code has to be generated. Along with operands and operators, three address code also use labels to provide flow of control for statements like if-then-else, for and while. The different types of three address code statements are:

Assignment statement

a = b op c

In the above case b and c are operands, while op is binary or logical operator. The result of applying op on b and c is stored in a.

Unary operation

a = op b This is used for unary minus or logical negation.

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

Three address code for the above example will be

t1 = -c

t2 = t1 * b

t3 = t2 + d

a = t3

Copy Statement

a = b

The value of b is stored in variable a.

Unconditional jump

goto L

Creates label L and generates three-address code ‘goto L’

v. Creates label L, generate code for expression exp, If the exp returns value true then go to the statement labelled L. exp returns a value false go to the statement immediately following the if statement.

Function call

For a function fun with n arguments a1,a2,a3….an ie.,

fun(a1, a2, a3,…an),

the three address code will be

Param a1

Param a2

…

Param an

Call fun, n

Where param defines the arguments to function.

Array indexing

In order to access the elements of array either single dimension or

multidimension, three address code requires base address and offset value. Base address consists of the address of first element in an array. Other elements of the array can be accessed using the base address and offset value.

Example: x = y[i]

Memory location m = Base address of y + Displacement i

x = contents of memory location m

similarly x[i] = y

Memory location m = Base address of x + Displacement i

The value of y is stored in memory location m

Pointer assignment

x = &y x stores the address of memory location y

x = *y y is a pointer whose r-value is location

*x = y sets r-value of the object pointed by x to the r-value of y

Intermediate representation should have an operator set which is rich to implement most of the

operations of source language. It should also help in mapping to restricted instruction set of target machine.

Data Structure

Three address code is represented as record structure with fields for operator and operands. These

records can be stored as array or linked list. Most common implementations of three address code are-

Quadruples, Triples and Indirect triples.

7.3 QUADRUPLES-

Quadruples consists of four fields in the record structure. One field to store operator op, two fields to store operands or arguments arg1and arg2 and one field to store result res. res = arg1 op arg2

Example: a = b + c

b is represented as arg1, c is represented as arg2, + as op and a as res.

Unary operators like '-'do not use agr2. Operators like param do not use agr2 nor result. For conditional and unconditional statements res is label. Arg1, arg2 and res are pointers to symbol table or literal table for the names.

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

Three address code for the above statement is as follows

t1 = - b

t2 = t1 * d

t3 = t2 + c

t4 = - b

t5 = t4 * d

t6 = t3 + t5

a = t6

Quadruples for the above example is as follows

Op	Arg1	Arg2	Res
-	B		t1
*	t1	d	t2
+	t2	c	t3
-	B		t4
*	t4	d	t5
+	t3	t5	t6
=	t6		a

7.4 TRIPLES

Triples uses only three fields in the record structure. One field for operator, two fields for operands named as arg1 and arg2. Value of temporary variable can be accessed by the position of the statement the computes it and not by location as in quadruples.

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

Triples for the above example is as follows

Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Arg1 and arg2 may be pointers to symbol table for program variables or literal table for constant or pointers into triple structure for intermediate results.

Example: Triples for statement x[i] = y which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	[]=	x	i
(1)	=	(0)	y

Triples for statement x = y[i] which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	=[]	y	i
(1)	=	x	(0)

Triples are alternative ways for representing syntax tree or Directed acyclic graph for program defined names.

Indirect Triples

Indirect triples are used to achieve indirection in listing of pointers. That is, it uses pointers to triples than listing of triples themselves.

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

	Stmt no
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

Stmt no	Op	Arg1	Arg2
(10)	-	b	
(11)	*	d	(0)
(12)	+	c	(1)
(13)	-	b	
(14)	*	d	(3)
(15)	+	(2)	(4)
(16)	=	a	(5)

Conditional operator and operands. Representations include quadruples, triples and indirect triples.

7.5 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.

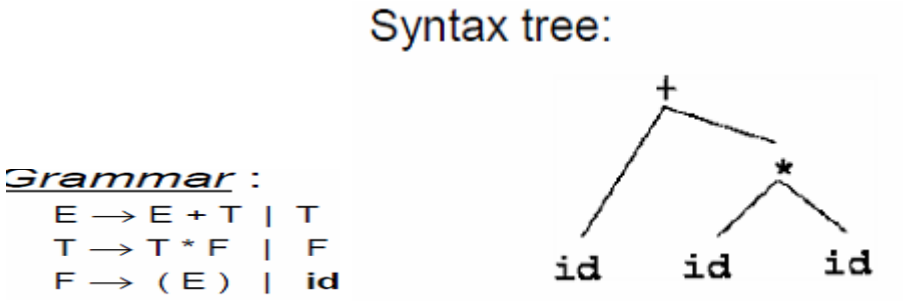
Variants of Syntax Trees: DAG

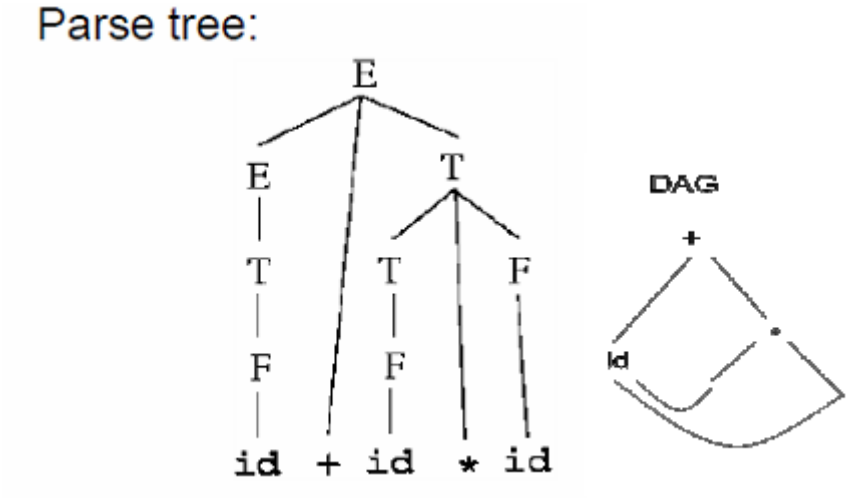
A directed acyclic graph (DAG) for an expression identifies the common sub expressions (sub expressions that occur more than once) of the expression. DAG's can be constructed by using the same techniques that construct syntax trees.

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.

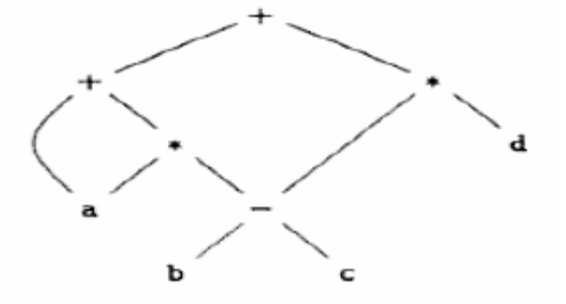
Example 1: Given the grammar below, for the input string id + id * id , the parse tree,

syntax tree and the DAG are as shown.





Example : 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

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

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

7.6 BASIC BLOCKS AND FLOW GRAPHS

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control. Flow graph of a program can be used as a vehicle to collect information about the intermediate program. Some register-assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

BASIC BLOCKS

A basic block is a sequence of consecutive statements in which flow of control
enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block:

```
t1  :=  a*a
t2  :=  a*b
t3 := 2*t2
t4 := t1+t3
t5 := b*b
t6 := t4+t5
```

A three-address statement x := y+z is said to define x and to use y or z. A name in a basic block is said to live at a given point if its value is used after that point in the program, perhaps in another basic block.

The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

Algorithm 1: Partition into basic blocks.

Input: A sequence of three-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. We first determine the set of leaders, the first statements of basic blocks.

The rules we use are the following:

I) The first statement is a leader.

II) Any statement that is the target of a conditional or unconditional goto is a leader.

III) Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example 3: Consider the fragment of source code shown in fig. 7; it computes the dot product of two vectors a and b of length 20. A list of three-address statements performing this computation on our target machine is shown in fig. 8.

```
begin
prod := 0;
i := 1;
do begin
prod := prod + a[i] * b[i];
i := i+1;
end
while i<= 20
end
```

Let us apply Algorithm 1 to the three-address code in fig 8 to determine its basic blocks. statement (1) is a leader by rule (I) and statement (3) is a leader by rule (II), since the last statement can jump to it. By rule (III) the statement following (12) is a leader. Therefore, statements (1) and (2) form a basic block. The remainder of the program beginning with statement (3) forms a second basic block.

```
(1) prod := 0
(2) i := 1
(3) t1 := 4*i
(4) t2 := a [ t1 ]
(5) t3 := 4*i
(6) t4 :=b [ t3 ]
(7) t5 := t2*t4
(8) t6 := prod +t5
(9) prod := t6
(10) t7 := i+1
```

(11) i := t7

(12) if i<=20 goto (3)

7.7 TRANSFORMATIONS ON BASIC BLOCKS

A basic block computes a set of expressions. These expressions are the values of the names live on exit from block. Two basic blocks are said to be equivalent if they compute the same set of expressions. A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Many of these transformations are useful for improving the quality of code that will be ultimately generated from a basic block. There are two important classes of local transformations that can be applied to basic blocks; these are the structure-preserving transformations and the algebraic transformations.

7.8 STRUCTURE-PRESERVING TRANSFORMATIONS

The primary structure-preserving transformations on basic blocks are:

1. Common sub-expression elimination
2. Dead-code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent statements

We assume basic blocks have no arrays, pointers, or procedure calls.

1. **Common sub-expression elimination**

Consider the basic block

a:= b+c

b:= a-d

c:= b+c

d:= a-d

The second and fourth statements compute the same expression, namely b+c-d, and hence this basic block may be transformed into the equivalent block

a:= b+c

b:= a-d

c:= b+c d:= b

Although the 1st and 3rd statements in both cases appear to have the same expression

on the right, the second statement redefines b. Therefore, the value of b in the 3rd statement is different from the value of b in the 1st, and the 1st and 3rd statements do not compute the same expression.

2**. Dead-code elimination**

Suppose x is dead, that is, never subsequently used, at the point where the statement x:= y+z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

3. **Renaming temporary variables**

Suppose we have a statement t:= b+c, where t is a temporary. If we change this statement to u:= b+c, where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed.

4.Interchange of statements

Suppose we have a block with the two adjacent statements

t1:= b+c

t2:= x+y

Then we can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2. A normal-form basic block permits all statement interchanges that are possible.

7.9 DAG REPRESENTATION OF BASIC BLOCKS

The goal is to obtain a visual picture of how information flows through the block. The leaves will show the values entering the block and as we proceed up the DAG we encounter uses of these values defs (and redefs) of values and uses of the new values.

Formally, this is defined as follows.

1. Create a leaf for the initial value of each variable appearing in the block. (We do not know what that the value is, not even if the variable has ever been given a value).

2. Create a node N for each statement s in the block.

i. Label N with the operator of s. This label is drawn inside the node.

ii. Attach to N those variables for which N is the last def in the block. These additional labels are drawn along side of N.

iii. Draw edges from N to each statement that is the last def of an operand used by N.

2. Designate as output nodes those N whose values are live on exit, an officially-mysterious term meaning values possibly used in another block. (Determining the live on exit values requires global, i.e., inter-block, flow analysis.) As we shall see in the next few sections various basic-block optimizations are facilitated by using the DAG.

Finding Local Common Subexpressions

As we create nodes for each statement, proceeding in the static order of the tatements, we might notice that a new node is just like one already in the DAG in which case we don't need a new node and can use the old node to compute the new value in addition to the one it already was computing. Specifically, we do not construct a new node if an existing node has the same children in the same order and is labeled with the same operation.

Consider computing the DAG for the following block of code.

a = b + c

c = a + x

d = b + c

b = a + x

The DAG construction is explain as follows (the movie on the right accompanies the explanation).

1. First we construct leaves with the initial values.
2. Next we process a = b + c. This produces a node labeled + with a attached and having b0 and c0 as children.
3. Next we process c = a + x.
4. Next we process d = b + c. Although we have already computed b + c in the first statement, the c's are not the same, so we produce a new node.
5. Then we process b = a + x. Since we have already computed a + x in statement 2, we do not produce a new node, but instead attach b to the old node.
6. Finally, we tidy up and erase the unused initial values.

You might think that with only three computation nodes in the DAG, the block could be reduced to three statements (dropping the computation of b). However, this is wrong. Only if b is dead on exit can we omit the computation of b. We can, however, replace the last statement with the simpler b = c. Sometimes a combination of techniques finds

improvements that no single technique would find. For example if a-b is computed, then both a and b are incremented by one, and then a-b is computed again, it will not be recognized as a common subexpression even though the value has not changed. However, when combined with various algebraic transformations, the common value can be recognized.

7.10 DEAD CODE ELIMINATION

Assume we are told (by global flow analysis) that certain values are dead on exit. We examine each root (node with no ancestor) and delete any that have no live variables attached. This process is repeated since new roots may have appeared.

For example, if we are told, for the picture on the right, that only a and b are live, then the root d can be removed since d is dead. Then the rightmost node becomes a root, which also can be removed (since c is dead).

The Use of Algebraic Identities

Some of these are quite clear. We can of course replace x+0 or 0+x by simply x. Similar Considerations apply to 1*x, x*1, x-0, and x/1.

Strength reduction

Another class of simplifications is strength reduction, where we replace one operation by a cheaper one. A simple example is replacing 2*x by x+x on architectures where addition is cheaper than multiplication. A more sophisticated strength reduction is applied by compilers that recognize induction variables (loop indices). Inside a for i from 1 to N loop, the expression 4*i can be strength reduced to j=j+4 and 2^i can be strength reduced to j=2*j (with suitable initializations of j just before the loop). Other uses of algebraic identities are possible; many require a careful reading of the language

reference manual to ensure their legality. For example, even though it might be advantageous to convert ((a + b) * f(x)) * a to ((a + b) * a) * f(x)

it is illegal in Fortran since the programmer's use of parentheses to specify the order of operations can not be violated.

Does

a = b + c

x = y + c + b + r

contain a common sub expression of b+c that need be evaluated only once?

The answer depends on whether the language permits the use of the associative and commutative law for addition. (Note that the associative law is invalid for floating point numbers.)

OPTIMIZATION

8.1 PRINCIPLE SOURCES OF OPTIMIZATION

A transformation of a program is called local if it can be performed by looking only at the statements in a bas9ic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations There are a number of ways in which a compiler can improve a program without changing the function it computes. Common sub expression elimination, copy propagation, deadcode elimination, and constant folding are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed. Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of these duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language. For example, block B5 recalculates 4*i and 4*j.

Common Sub expressions An occurrence of an expression E is called a common sub expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid re computing the expression if we can use the previously computed value. For example, the assignments to t7 and t10 have the common sub expressions 4*I and 4*j, respectively, on the right side in Fig. They have been eliminated in Fig by using t6 instead of t7 and t8 instead of t10. This change is what would result if we reconstructed the intermediate code from the dag for the basic block.

Example: the above Fig shows the result of eliminating both global and local common sub expressions from blocks B5 and B6 in the flow graph of Fig. We first discuss the transformation of B5 and then mention some subtleties involving arrays.

After local common sub expressions are eliminated B5 still evaluates 4*i and 4*j, as

Shown in the earlier fig. Both are common sub expressions; in particular, the three statements t8:= 4*j; t9:= a[t[8]; a[t8]:=x in B5 can be replaced by t9:= a[t4]; a[t4:= x using t4 computed in block B3. In Fig. observe that as control passes from the evaluation of 4*j in B3 to B5, there is no change in j, so t4 can be used if 4*j is needed.

Another common sub expression comes to light in B5 after t4 replaces t8. The new expression a[t4] corresponds to the value of a[j] at the source level. Not only does j retain its value as control leaves b3 and then enters B5, but a[j], a value computed into a temporary t5, does too because there are no assignments to elements of the array a in the interim. The statement t9:= a[t4]; a[t6]:= t9 in B5 can therefore be replaced by

a[t6]:= t5 The expression in blocks B1 and B6 is not considered a common sub expression although t1 can be used in both places. After control leaves B1 and before it reaches B6,it can go through B5,where there are assignments to a. Hence, a[t1] may not have the same value on reaching B6 as it did in leaving B1, and it is not safe to treat a[t1] as a common sub expression.

Copy Propagation

Block B5 in Fig. can be further improved by eliminating x using two new transformations. One concerns assignments of the form f:=g called copy statements, or copies for short. Had we gone into more detail in Example 10.2, copies would have arisen much sooner, because the algorithm for eliminating common sub expressions introduces them, as do several other algorithms. For example, when the common sub expression in c:=d+e is eliminated in Fig., the algorithm uses a new variable t to hold the value of d+e. Since control may reach c:=d+e either after the assignment to a or after the assignment to b, it would be incorrect to replace c:=d+e by either c:=a or by c:=b. The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy statement f:=g. For example, the assignment x:=t3 in block B5 of Fig. is a copy. Copy propagation applied to B5 yields:

x:=t3

a[t2]:=t5

a[t4]:=t3

goto B2 Copies introduced during common subexpression elimination. This may not appear to be an improvement, but as we shall see, it gives us the opportunity to eliminate the assignment to x.

8.2 DEAD-CODE ELIMINATIONS

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. For example, we discussed the use of debug that is set to true or false at various points in the program, and used in statements like If (debug) print. By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement, the value of debug is false. Usually, it is because there is one particular statement Debug :=false

That we can deduce to be the last assignment to debug prior to the test no matter what sequence of branches the program actually takes. If copy propagation replaces debug by false, then theprint statement is dead because it cannot be reached. We can eliminate both the test and printing from the o9bject code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to x and transforms 1.1 into

a [t2] := t5

a [t4] := t3

goto B2

8.3 PEEPHOLE OPTIMIZATION

A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying “optimizing” transformations to the target program.

A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this. We shall give the following examples of program transformations that are characteristic of peephole optimizations:

• Redundant-instructions elimination
• Flow-of-control optimizations
• Algebraic simplifications
• Use of machine idioms

REDUNTANT LOADS AND STORES

If we see the instructions sequence

(1) (1) MOV R0,a

(2) (2) MOV a,R0

-we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

UNREACHABLE CODE

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1.In C, the source code might look like:

```
#define debug 0
….
If ( debug ) {
Print debugging information
}
```

In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L2
Goto L2
L1: print debugging information
```

L2: ……(a)

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of debug; (a) can be replaced by:

If debug ≠1 goto L2

Print debugging information

L2: ……(b)

As the argument of the statement of (b) evaluates to a constant true it can be replaced by

If debug ≠0 goto L2

Print debugging information

L2: ……(c)

As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

8.4 FLOW-OF-CONTROL OPTIMIZATIONS

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L2

….

L1 : gotoL2

by the sequence

goto L2

….

L1 : goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1

….

L1 : goto L2

can be replaced by

if a < b goto L2

….

L1 : goto L2

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

……..

L1:if a<b goto L2

L3: ……..(1)

may be replaced by

if a<b goto L2

goto L3

…….

L3: …….(2)

While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

8.5 REGISTER ALLOCATION

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

1. During register allocation, we select the set of variables that will reside in registers at a point in the program.

2. During a subsequent register assignment phase, 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 values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.

Certain machines require register pairs (an even and next odd numbered register) for some operands and results. For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs. The multiplication instruction is of the form M x, y where x, is the multiplicand, is the even register of an even/odd register pair.

The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The division instruction is of the form D x, y where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor. After division, the even register holds the remainder and the odd register the quotient. Now consider the two three address code sequences (a) and (b) in which the only difference is

the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c). Ri stands for register i. L, ST and A stand for load, store and add respectively. The optimal choice for the register into which 'a' is to be loaded depends on what will ultimately happen to e.

t := a + b t := a + b

t := t * c t := t + c

t := t / d t := t / d

(a) (b)

Two three address code sequences

L R1, a L R0, a

A R1, b A R0, b

M R0, c A R0, c

D R0, d SRDA R0,

ST R1, t D R0, d

ST R1, t

(a) (b)

8.6 CHOICE OF OF 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. Picking a best order is another 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.

8.7 APPROCHES TO CODE GENERATION

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that code generator must face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal Reference Counting Garbage Collection The difficulty in garbage collection is not the actual process of collecting the garbage--it is the problem of finding the garbage in the first place. An object is considered to be garbage when no references to that object exist. But how can we tell when no references to an object exist? A simple expedient is to keep track in each object of the total number of references to that object. That is, we add a special field to each object called a reference count . The idea is that the reference count field is not accessible to the Java program. Instead, the reference count field is updated by the Java virtual machine itself.

Consider the statement

Object p = new Integer (57);

which creates a new instance of the Integer class. Only a single variable, p, refers to the object. Thus, its reference count should be one.

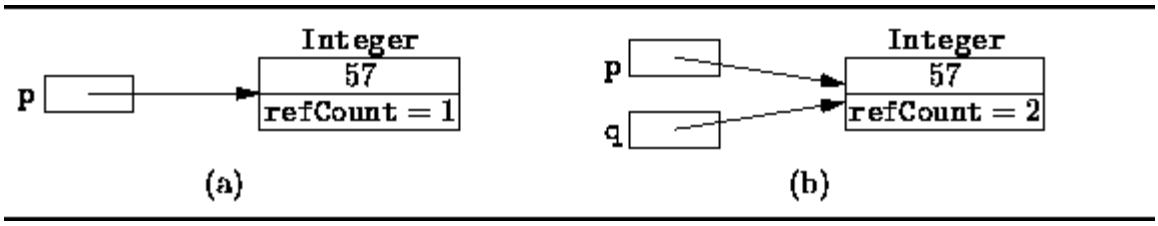


Figure: Objects with reference counters.

Now consider the following sequence of statements:

Object p = new Integer (57);

Object q = p;

This sequence creates a single Integer instance. Both p and q refer to the same object. Therefore, its reference count should be two.

In general, every time one reference variable is assigned to another, it may be necessary to update several reference counts. Suppose p and q are both reference variables. The assignment

p = q;

would be implemented by the Java virtual machine as follows:

```
if (p != q)
{
   if (p != null)
         --p.refCount;
   p = q;
   if (p != null)
         ++p.refCount;
}
```

For example suppose p and q are initialized as follows:

Object p = new Integer (57);

Object q = new Integer (99);

As shown in Figure ❑ (a), two Integer objects are created, each with a reference count of one. Now, suppose we assign q to p using the code sequence given above. Figure ❑ (b) shows that after the assignment, both p and q refer to the same object--its reference count is two. And the reference count on Integer(57) has gone to zero which indicates that it is garbage.

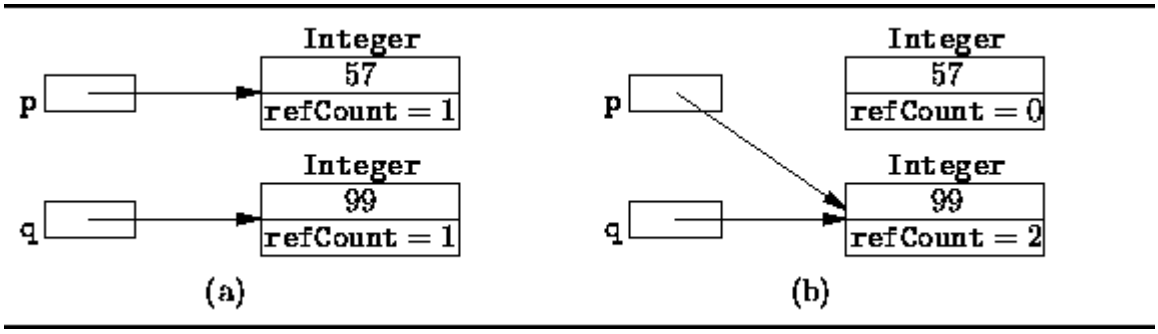


Figure: Reference counts before and after the assignment p = q.

The costs of using reference counts are twofold: First, every object requires the special reference count field. Typically, this means an extra word of storage must be allocated in each object. Second, every time one reference is assigned to another, the reference counts

must be adjusted as above. This increases significantly the time taken by assignment statements.

The advantage of using reference counts is that garbage is easily identified. When it becomes necessary to reclaim the storage from unused objects, the garbage collector needs only to examine the reference count fields of all the objects that have been created by the program. If the reference count is zero, the object is garbage.

It is not necessary to wait until there is insufficient memory before initiating the garbage collection process. We can reclaim memory used by an object immediately when its reference goes to zero. Consider what happens if we implement the Java assignment p = q in the Java virtual machine as follows:

```
if (p != q)
{
   if (p != null)
         if (--p.refCount == 0)
            heap.release (p);
   p = q;
   if (p != null)
         ++p.refCount;
}
```

Notice that the release method is invoked immediately when the reference count of an object goes to zero, i.e., when it becomes garbage. In this way, garbage may be collected incrementally as it is created.

TEXT BOOKS:

1. Compilers, Principles Techniques and Tools- Alfred V Aho, Monical S Lam, Ravi Sethi, Jeffrey D. Ullman,2nd ed, Pearson,2007.
2. Principles of compiler design, V. Raghavan, 2nd ed, TMH, 2011.
3. Principles of compiler design, 2nd ed, Nandini Prasad, Elsevier

REFERENCE BOOKS:

1. http://www.nptel.iitm.ac.in/downloads/106108052/
2. Compiler construction, Principles and Practice, Kenneth C Louden, CENGAGE
3. Implementations of Compiler, A new approach to Compilers including the algebraic methods, Yunlinsu, SPRINGER