UNIT 7
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

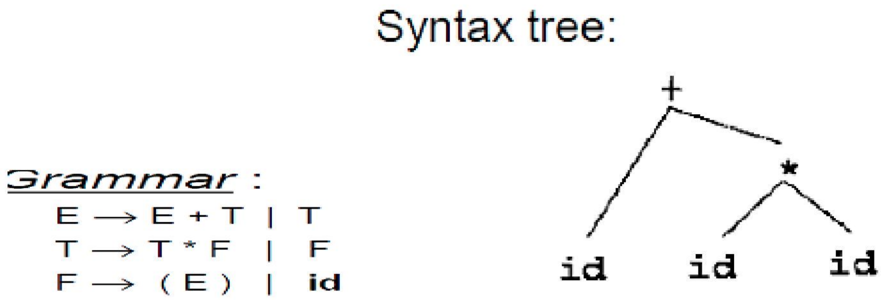
Syntax trees are high level IR. They depict the natural hierarchical structure of the source program. Nodes represent constructs in source program and the children of a node represent meaningful components of the construct. Syntax trees are suited for static type checking.

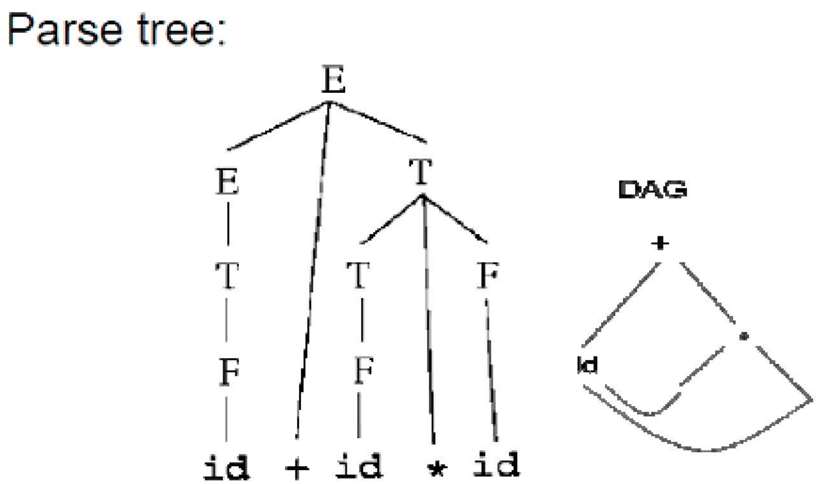
Variants of Syntax Trees: DAG

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.

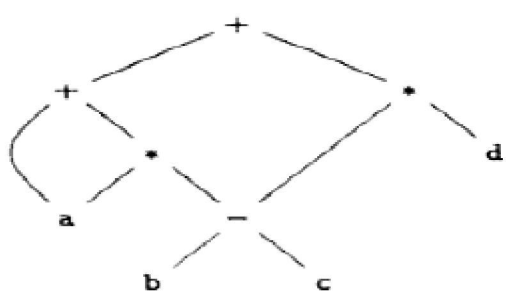
A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. A node N in a DAG has more than one parent if N represents a common sub expression, so a DAG represents expressions concisely. It gives clues to compiler about the generating efficient code to evaluate expressions.

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.

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

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.)