

UNIT – IV

Intermediate code generation

Objective

To understand various intermediate code forms and machine independent optimization techniques.

Syllabus

Intermediate code- Three address code, quadruples, triples, abstract syntax trees.

Machine independent code optimization- Common sub expression elimination, constant folding, copy propagation, dead code elimination, liveness analysis, loop optimization- strength reduction, code motion.

Learning Outcomes

Students will be able to

- Understand intermediate code forms such as polish notation, syntax trees and three-address code .
- Write three address code for various language constructs.
- Construct basic blocks for the given language constructs.
- Construct Flow graph for the given basic block.
- Construct DAG for the given basic block and expression.

Learning Material

4.1 Intermediate Code

- 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.
- Ideally, details of the source language are confined to the front end, and details of the target machine to the back end. The source code is translated into a language, called **Intermediate code or intermediate text** which is intermediate in complexity between a high level programming language and a machine code.

Benefits of using a machine independent intermediate form are:

- Retargeting is facilitated, a compiler for different machine can be created by attaching a back end for the new machine to an existing front end.
- A machine-independent code optimizer can be applied to the intermediate representation.

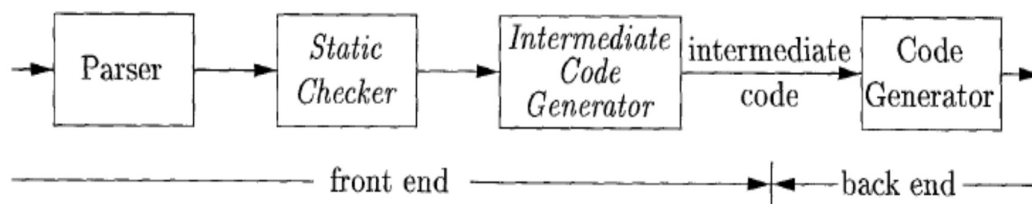


Fig1: Position of Intermediate Code Generator

4.1.1 Intermediate code representations

- Postfix (or) reverse polish notation
- Syntax tree or parse tree
- Three address code

1)Postfix (or) reverse polish notation:

- The postfix notation for the expression places the operator at the right end.
- If e_1 and e_2 are any postfix expressions, and Θ is any binary operator, the result of applying Θ to the values denoted by e_1 and e_2 is indicated in postfix notation by $e_1e_2\Theta$.

Example: $(a+b)*c$ in postfix notation is $ab+c*$.

2)Syntax tree or Parse tree:

- A syntax tree depicts the natural hierarchical structure of a source program.
- A syntax tree for the assignment statement $a := b * -c + b * -c$ is

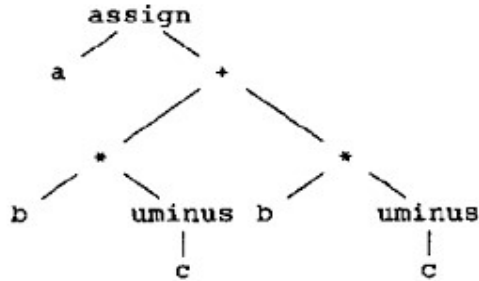


Fig2: Parse Tree

3)Three Address Code:

- Three address code is a sequence of statements of the general form $x := y \text{ op } z$
- where x, y and z are names, constants or compiler-generated temporaries;
- **op** stands for any operator.
- In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.
- A source language expression like $x+y*z$ might be translated into the sequence of three-address instructions
 - $t1=y*z$
 - $t2=x+t1$
- where **t1** and **t2** are compiler-generated temporary names.

Symbolic labels will be used by instructions that alter the flow of control. A symbolic label represents the index of a three-address instruction in the sequence of instructions. Actual indexes can be substituted for the labels, either by making a separate pass or by "back patching".

4.2 Three address Code

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**
- **Indirect triples**

- **Abstract Syntax Tree**

4.2.1 QUADRUPLES

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

$res = arg1 \text{ op } arg2$

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 arg2. Operators like param do not use arg2 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	Result
-	b		t1
*	t1	d	t2
+	t2	c	t3
-	b		t4
*	t4	d	t5
+	t3	t5	t6
=	t6		a

Table 1: Quadruples table for Three address code

4.2.2 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)

Table 2: Triple Table

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

Table 3: Triples for statement $x[i] = y$ which generates two records

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

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

Table 4: Triples for statement $x = y[i]$ which generates two records

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

4.2.3 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		Op	Arg1	Arg2
(0)	(10)	-	b	
(1)	(11)	*	d	(0)
(2)	(12)	+	c	(1)
(3)	(13)	-	b	
(4)	(14)	*	d	(3)
(5)	(15)	+	(2)	(4)
(6)	(16)	=	a	(5)

Table 5: Indirect triples

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

Note: The conversion of all labels in three address statements to addresses of instructions is known as backpatching.

4.2.4 Abstract syntax tree (AST):

An **abstract syntax tree (AST)** is a way of representing the **syntax** of a programming language as a hierarchical **tree**-like structure. This structure is used for generating symbol tables for compilers and later code generation. The **tree** represents all of the constructs in the language and their subsequent rules.

4.3 Machine independent code optimization

The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

Optimizations are classified into two categories. They are

- Machine independent optimizations
- Machine dependant optimizations

Machine independent optimizations:

- Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

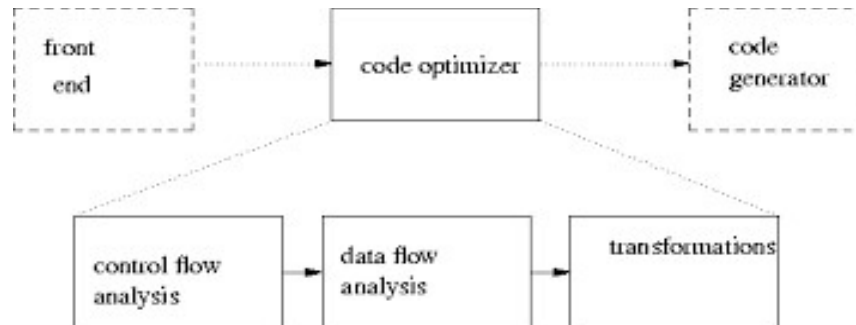


Fig: Organization for an Optimizing Compiler

4.3.1 Common Sub expressions elimination:

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.

Example 1:

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5
```

The above code can be optimized using the common sub-expression elimination as

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

4.3.2 Constant folding:

- Folding is the replacement of expressions that can be evaluated at compile time by their computed values. That is, if the values of all the operands of expression are known to the compiler at compile time, the expression can be replaced by its computed value.
- One advantage of copy propagation is that it often turns the copy statement into dead code.
- Constant folding is also known as constant prorogation.
- Folding is a local optimization technique; folding operation can take place involving a variable that is common to a set of contiguous block in a program.

Example 1:

The statement $A=2+3+A+C$

can be replaced by $A=5+A+C$, where the value "5" has replaced the expression "2+3"

Example 2:

$Pi=3.14157$

$a=pi/2$

Can be rewritten as

$Pi=3.14157$

$a=3.14157/2$ can be replaced by

$Pi= 3.14157$

$a=1.570$ thereby eliminating a division operation.

4.3.3 Copy Propagation:

- Assignments of the form $f := g$ called copy statements, or copies for short.
- The idea behind the copy-propagation transformation is to use g for f , whenever possible after the copy statement $f := g$.
- Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x .

For example:

In block B

$x := t3$ $a[t2] := t5$ $a[t4] := x$ goto B2

The assignment $x:=t3$ in block B is a copy.

Copy propagation applied to B yields:


```
x := t3
a[t2] := t5
a[t4] := t3
goto B2
```

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 into:

```
a[t2] := t5
a[t4] := t3
goto B2
```

4.3.4 Dead-Code Elimination:

- 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.
- An optimization can be done by eliminating dead code.

Example 1:

```
i=0;
if(i=1)
{
a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

Example 2:

The use of debug that is set to true or false at various points in the program and used in statements like

```
debug := false
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.

If copy propagation replaces debug by false, then the print statement is dead because it cannot be reached. We eliminate both the test and printing from the object code

4.4 Loop Optimization:

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Three techniques are important for loop optimization:
 - code motion, which moves code outside a loop;
 - Induction-variable elimination, which we apply to replace variables from inner loop.
 - Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Various steps to Loop Optimization:

- 1) Convert the program into Three-address code
- 2) Break the code into basic blocks
- 3) Construct flow graph
- 4) Apply optimization
 - a) Code Motion
 - b) Induction variable elimination
 - c) Strength reduction

Example:

Consider the following source code for dot product of two vectors a and b of length 20

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

Step1: Convert the program into Three-address code

The three-address code for the above source program is given as :

```
(1) prod := 0
(2) i := 1
(3) t1 := 4* i
(4) t2 := a[t1] /*compute a[i] */
(5) t3 := 4* i
(6) t4 := b[t3] /*compute b[i] */
(7) t5 := t2*t4
(8) t6 := prod+t5
(9) prod := t6
(10) t7 := i+1
(11) i := t7
(12) if i<=20 goto (3)
```

Step2: Break the code into basic blocks

Basic Block:

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

Basic Block Construction:

Algorithm: 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 of the following:

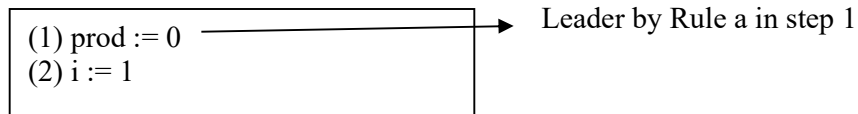
- The first statement is a leader.
- Any statement that is the target of a conditional or unconditional goto is a leader.

- 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.
 3. Any statement not placed in a block can never be executed and may now be removed if desired.

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (11)

Basic block B1



Basic block B2

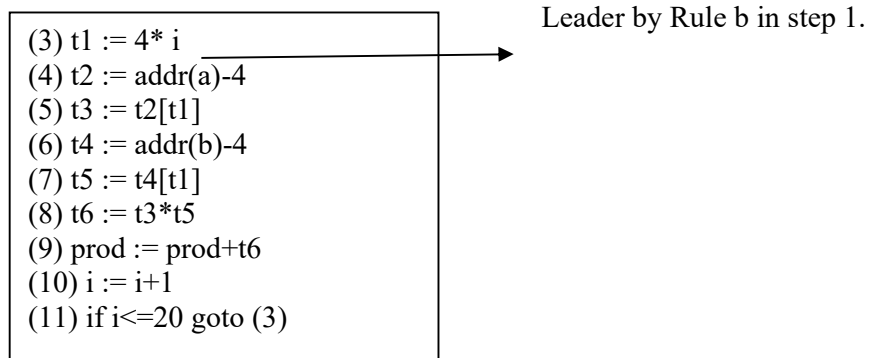


Fig: Basic block

Step3: Construct flow graph

Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.
- The nodes of the flow graph are basic blocks. It has a distinguished initial node.
- E.g.: Flow graph for the vector dot product is given as follows:

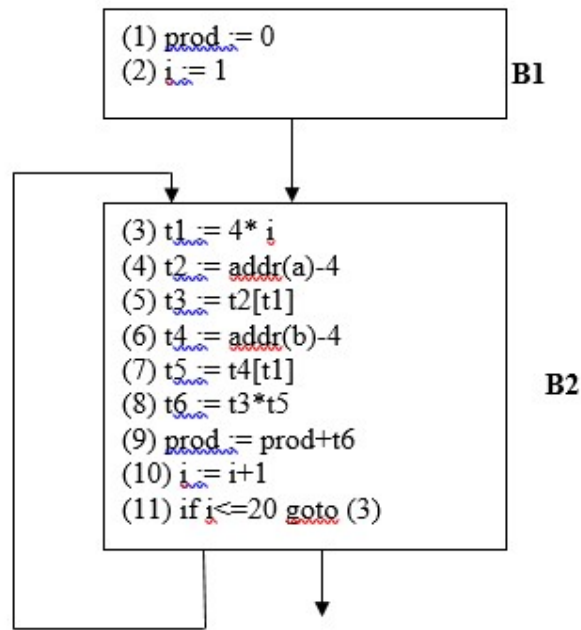


Fig 3:Flow graph for the vector dot product

- B1 is the initial node. B2 immediately follows B1, so there is an edge from B1 to B2. The target of jump from last statement of B1 is the first statement B2, so there is an edge from B1 (last statement) to B2 (first statement).
- B1 is the predecessor of B2, and B2 is a successor of B1.

Step 4: Apply optimization

i) Code motion, which moves code outside a loop;

- An important modification that decreases the amount of code in a loop is code motion.
- Identify loop invariant computations and move them out of the block.
- A **loop-invariant** computation takes an expression that yields the same result independent of the number of times a loop is executed.

ii) Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition

Reduction In Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

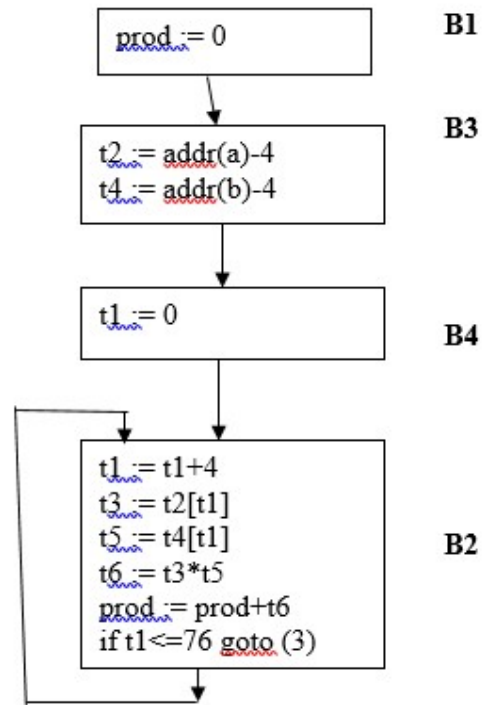


Fig: Flow graph after code motion and Reduction in Strength