UNIT-I

INTRODUCTION TO LANGUAGE PROCESSING:

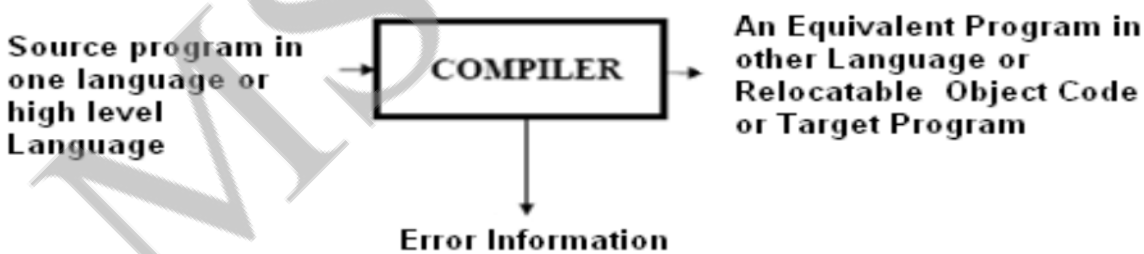
As Computers became inevitable and indigenous part of human life, and several languages with different and more advanced features are evolved into this stream to satisfy or comfort the user in communicating with the machine , the development of the translators or mediator Software‘s have become essential to fill the huge gap between the human and machine understanding. This process is called Language Processing to reflect the goal and intent of the process. On the way to this process to understand it in a better way, we have to be familiar with some key terms and concepts explained in following lines.

LANGUAGE TRANSLATORS :

Is a computer program which translates a program written in one (Source) language to its equivalent program in other [Target]language. The Source program is a high level language where as the Target language can be any thing from the machine language of a target machine (between Microprocessor to Supercomputer) to another high level languageprogram.

- Two commonly Used Translators are Compiler and Interpreter

1. **Compiler :** Compiler is a program, reads program in one language called Source Language and translates in to its equivalent program in another Language called Target Language, in addition to this its presents the error information to the User.



- If the target program is an executable machine-language program, it can then be called by the users to process inputs and produce outputs.

Figure1.1: Running the target Program

2. **Interpreter:** An interpreter is another commonly used language processor. Instead of producing a target program as a single translation unit, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by theuser.

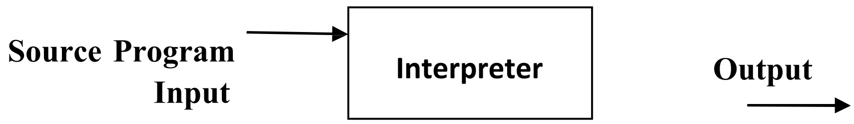


Figure 1.2: Running the target Program

LANGUAGE PROCESSING SYSTEM:

Based on the input the translator takes and the output it produces, a language translator can be called as any one of the following.

Preprocessor: A preprocessor takes the skeletal source program as input and produces an extended version of it, which is the resultant of expanding the Macros, manifest constants if any, and including header files etc in the source file. For example, the C preprocessor is a macro processor that is used automatically by the C compiler to transform our source before actual compilation. Over and above a preprocessor performs the following activities:

- ∑ Collects all the modules, files in case if the source program is divided into different modules stored at different files.
- ∑ Expands short hands / macros into source languagestatements.

Compiler: Is a translator that takes as input a source program written in high level language and converts it into its equivalent target program in machine language. In addition to above the compiler also

- ∑ Reports to its user the presence of errors in the source program.
- ∑ Facilitates the user in rectifying the errors, and execute the code.

Assembler: Is a program that takes as input an assembly language program and converts it into its equivalent machine language code.

Loader / Linker: This is a program that takes as input a relocatable code and collects the library functions, relocatable object files, and produces its equivalent absolute machine code.

Specifically,

- ∑ **Loading** consists of taking the relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper locations.
- ∑ **Linking** allows us to make a single program from several files of relocatable machine code. These files may have been result of several different compilations, one or more may be library routines provided by the system available to any program that needs them.

In addition to these translators, programs like interpreters, text formatters etc., may be used in language processing system. To translate a program in a high level language program to an executable one, the Compiler performs by default the compile and linking functions.

Normally the steps in a language processing system includes Preprocessing the skeletal Source program which produces an extended or expanded source program or a ready to compile unit of the source program, followed by compiling the resultant, then linking / loading , and finally its equivalent executable code is produced. As I said earlier not all these steps are mandatory. In some cases, the Compiler only performs this linking and loading functions implicitly.

The steps involved in a typical language processing system can be understood with following diagram.

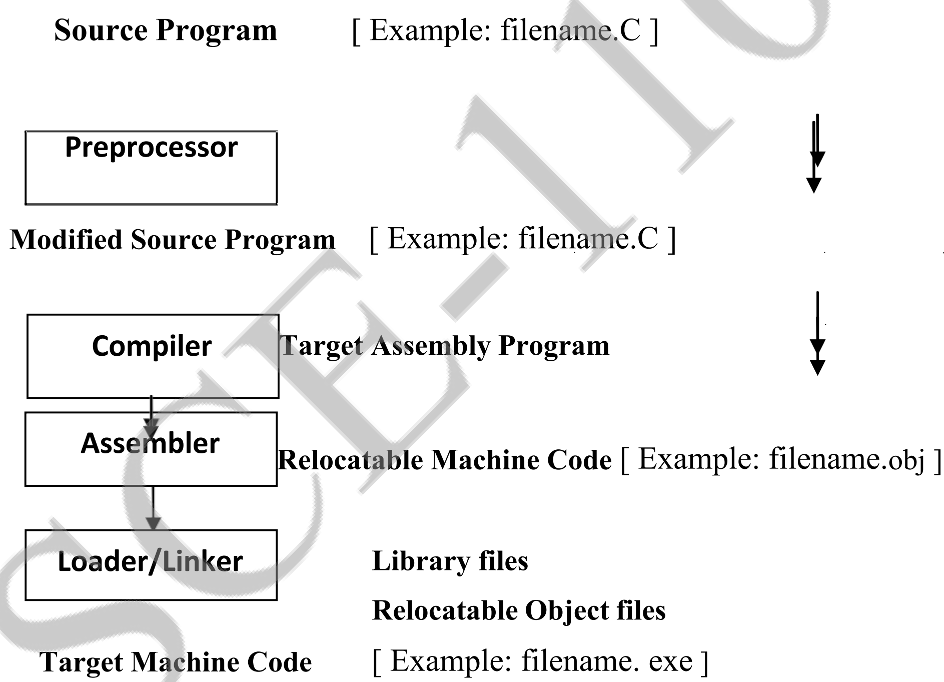


Figure1.3 : Context of a Compiler in Language Processing System

TYPES OF COMPILERS:

Based on the specific input it takes and the output it produces, the Compilers can be classified into the following types;

Traditional Compilers(C, C++, Pascal): These Compilers convert a source program in a HLL into its equivalent in native machine code or object code.

Interpreters(LISP, SNOBOL, Java1.0): These Compilers first convert Source code into intermediate code, and then interprets (emulates) it to its equivalent machine code.

Cross-Compilers: These are the compilers that run on one machine and produce code for another machine.

Incremental Compilers: These compilers separate the source into user defined–steps; Compiling/recompiling step- by- step; interpreting steps in a given order

Converters (e.g. COBOL to C++): These Programs will be compiling from one high level language to another.

Just-In-Time (JIT) Compilers (Java, Micosoft.NET): These are the runtime compilers from intermediate language (byte code, MSIL) to executable code or native machine code. These perform type –based verification which makes the executable code more trustworthy

Ahead-of-Time (AOT) Compilers (e.g., .NET ngen): These are the pre-compilers to the native code for Java and .NET

Binary Compilation: These compilers will be compiling object code of one platform into object code of another platform**.**

PHASES OF A COMPILER:

Due to the complexity of compilation task, a Compiler typically proceeds in a Sequence of compilation phases. The phases communicate with each other via clearly defined interfaces. Generally an interface contains a Data structure (e.g., tree), Set of exported functions. Each phase works on an abstract **intermediate representation** of the source program, not the source program text itself (except the first phase)

Compiler Phases are the individual modules which are chronologically executed to perform their respective Sub-activities, and finally integrate the solutions to give target code.

It is desirable to have relatively few phases, since it takes time to read and write immediate files. Following diagram (Figure1.4) depicts the phases of a compiler through which it goes during the compilation. There fore a typical Compiler is having the following Phases:

1. Lexical Analyzer (Scanner), 2. Syntax Analyzer (Parser), 3.Semantic Analyzer, 4.Intermediate Code Generator(ICG), 5.Code Optimizer(CO) , and 6.Code Generator(CG)

In addition to these, it also has **Symbol table management**, and **Error handler** phases. Not all the phases are mandatory in every Compiler. e.g, Code Optimizer phase is optional in some

cases. The description is given in next section.

The Phases of compiler divided in to two parts, first three phases we are called as Analysis part remaining three called as Synthesis part.

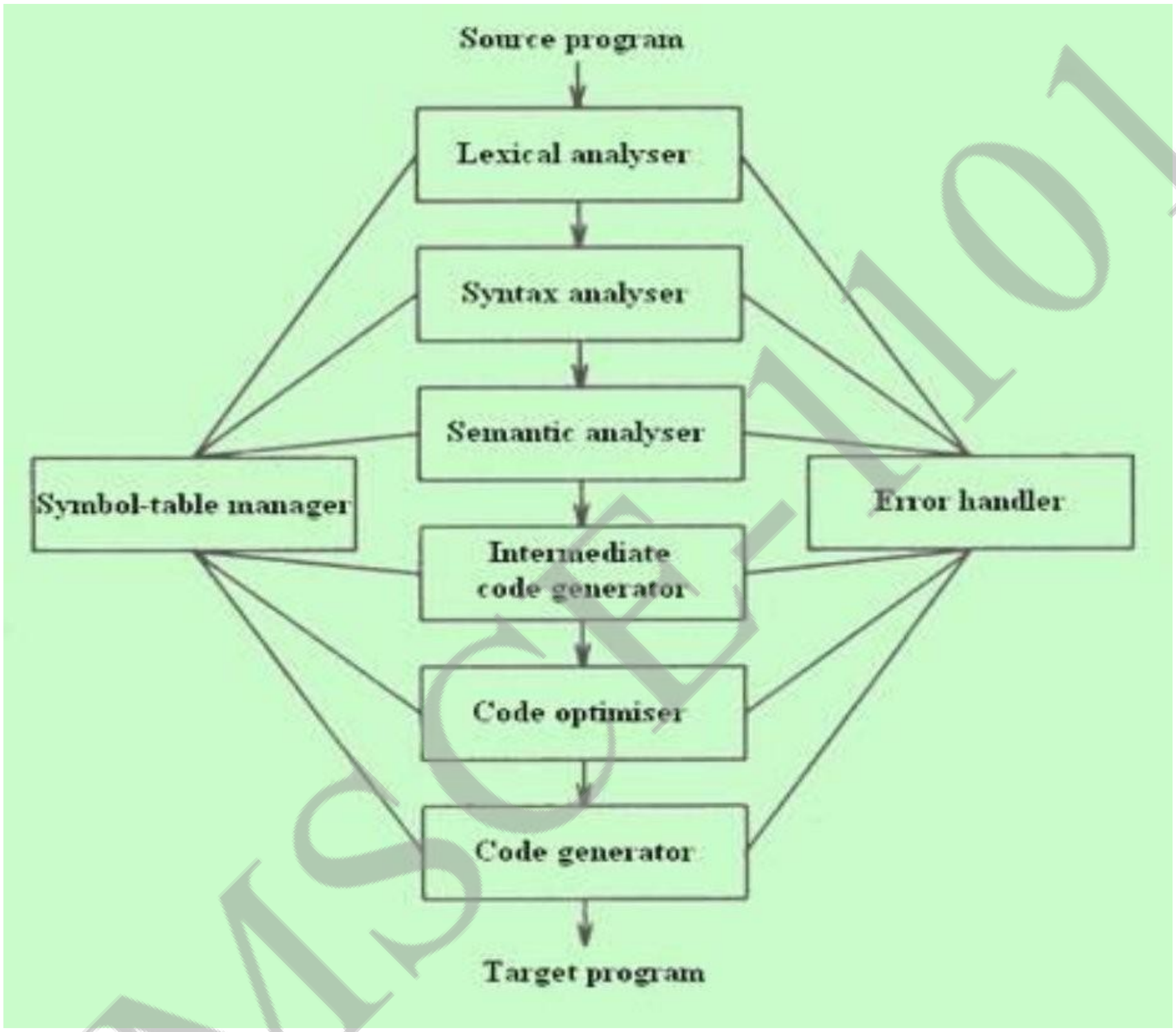


Figure1.4 : Phases of a Compiler

PHASE, PASSES OF A COMPILER:

In some application we can have a compiler that is organized into what is called passes. Where a pass is a collection of phases that convert the input from one representation to a completely deferent representation. Each pass makes a complete scan of the input and produces its output to be processed by the subsequent pass. For example a two pass Assembler.

THE FRONT-END & BACK-END OF A COMPILER

All of these phases of a general Compiler are conceptually divided into **The Front-end**, and **The Back-end**. This division is due to their dependence on either the Source Language or the Target machine. This model is called an Analysis & Synthesis model of a compiler.

The **Front-end** of the compiler consists of phases that depend primarily on the Source language and are largely independent on the target machine. For example, front-end of the compiler includes Scanner, Parser, Creation of Symbol table, Semantic Analyzer, and the Intermediate Code Generator.

The **Back-end** of the compiler consists of phases that depend on the target machine, and those portions don‘t dependent on the Source language, just the Intermediate language. In this we have different aspects of Code Optimization phase, code generation along with the necessary Error handling, and Symbol table operations.

LEXICAL ANALYZER (SCANNER): The Scanner is the first phase that works as interface between the compiler and the Source language program and performs the following functions:

- Reads the characters in the Source program and groups them into a stream of tokens in which each token specifies a logically cohesive sequence of characters, such as an identifier , a Keyword , a punctuation mark, a multi character operator like := .
- The character sequence forming a token is called a **lexeme** of the token.
- The Scanner generates a token-id, and also enters that identifiers name in the Symbol table if it doesn‘t exist.
- Also removes the Comments, and unnecessary spaces.

The format of the token is **< Token name, Attribute value>**

SYNTAX ANALYZER (PARSER): The Parser interacts with the Scanner, and its subsequent phase Semantic Analyzer and performs the following functions:

- Groups the above received, and recorded token stream into syntactic structures, usually into a structure called **Parse Tree** whose leaves are tokens.
- The interior node of this tree represents the stream of tokens that logically belongs together.
- It means it checks the syntax of program elements.

SEMANTIC ANALYZER: This phase receives the syntax tree as input, and checks the semantically correctness of the program. Though the tokens are valid and syntactically correct, it may happen that they are not correct semantically. Therefore the semantic analyzer checks the semantics (meaning) of the statements formed.

- The Syntactically and Semantically correct structures are produced here in the form of a Syntax tree or DAG or some other sequential representation like matrix.

INTERMEDIATE CODE GENERATOR(ICG): This phase takes the syntactically and semantically correct structure as input, and produces its equivalent intermediate notation of the source program. The Intermediate Code should have two important properties specified below:

- It should be easy to produce,and Easy to translate into the target program. Example intermediate code forms are:
- Three address codes,
- Polish notations, etc.

CODE OPTIMIZER: This phase is optional in some Compilers, but so useful and beneficial in terms of saving development time, effort, and cost. This phase performs the following specific functions:

- Attempts to improve the IC so as to have a faster machine code. Typical functions include –Loop Optimization, Removal of redundant computations, Strength reduction, Frequency reductions etc.
- Sometimes the data structures used in representing the intermediate forms may also be changed.

CODE GENERATOR: This is the final phase of the compiler and generates the target code, normally consisting of the relocatable machine code or Assembly code or absolute machine code.

- Memory locations are selected for each variable used, and assignment of variables to registers is done.
- Intermediate instructions are translated into a sequence of machine instructions.

The Compiler also performs the **Symbol table management** and **Error handling** throughout the compilation process. Symbol table is nothing but a data structure that stores different source language constructs, and tokens generated during the compilation. These two interact with all phases of the Compiler.

For example the source program is an assignment statement; the following figure shows how the phases of compiler will process the program.

The input source program is **Position=initial+rate*60**

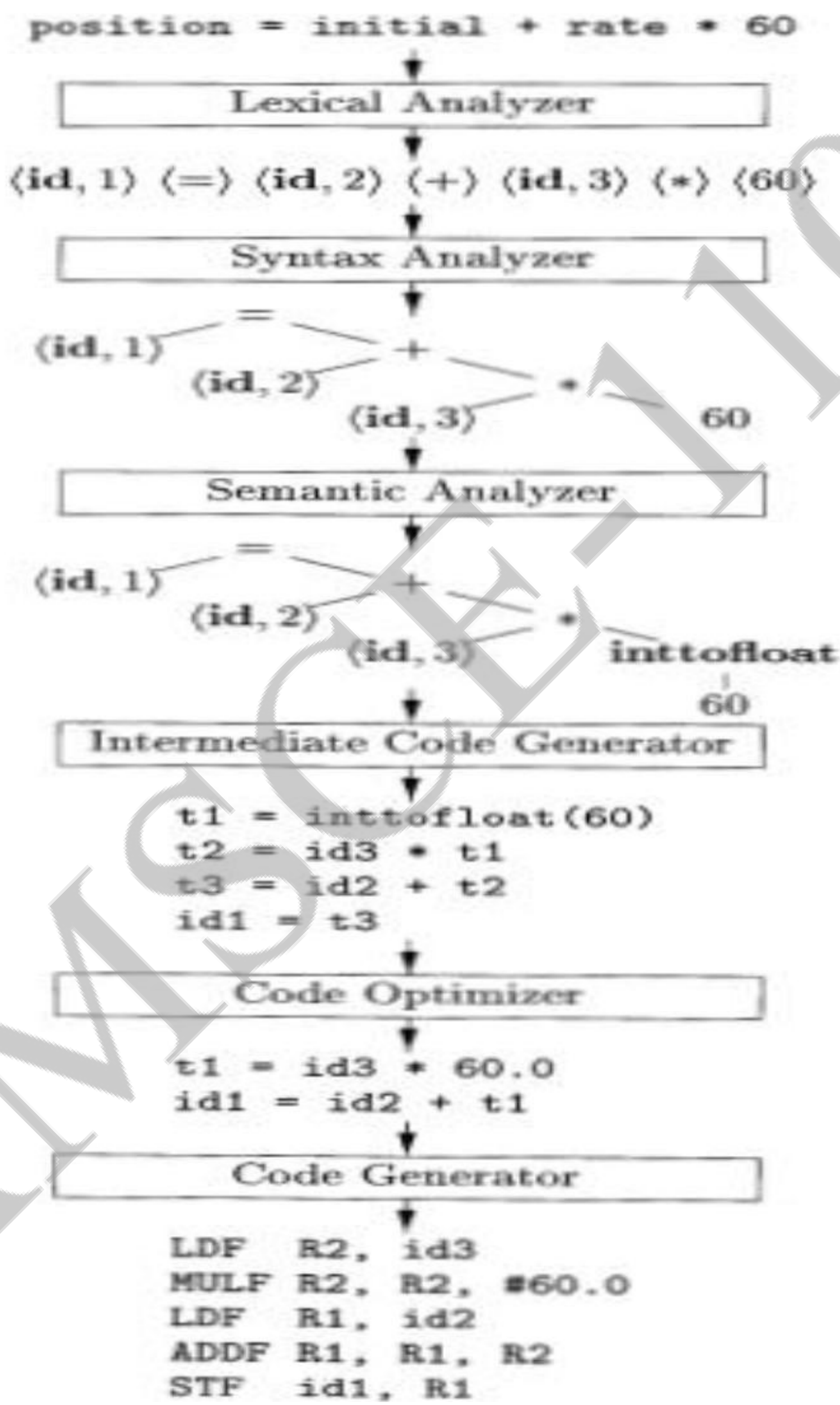


Figure1.5: Translation of an assignment Statement

LEXICAL ANALYSIS:

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output tokens for each lexeme in the source program. This stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well.

When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. This process is shown in the following figure.

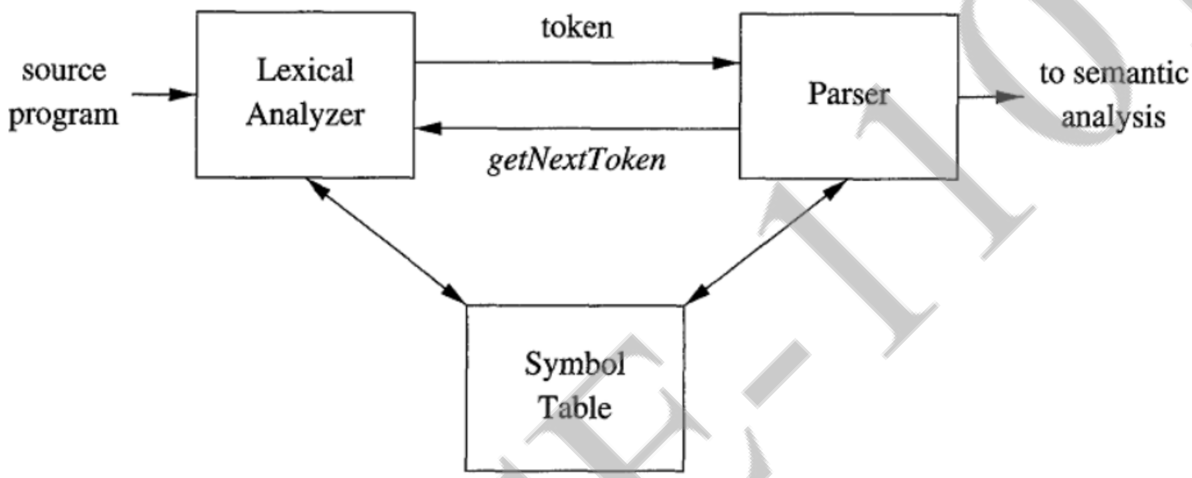


Figure 1.6 : Lexical Analyzer

. When lexical analyzer identifies the first token it will send it to the parser, the parser receives the token and calls the lexical analyzer to send next token by issuing the **getNextToken()** command. This Process continues until the lexical analyzer identifies all the tokens. During this process the lexical analyzer will neglect or discard the white spaces and comment lines.

TOKENS, PATTERNS AND LEXEMES:

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

A pattern is a description of the form that the lexemes of a token may take [or match]. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example: In the following C language statement ,

printf ("Total = %d\n‖, score) ;

both **printf** and **score** are lexemes matching the **pattern** for token **id**, and **"Total = %d\n‖** is a lexeme matching **literal [or string]**.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure 1.7: Examples of Tokens

LEXICAL ANALYSIS Vs PARSING:

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

∑ 1. **Simplicity of design is the most important consideration.** The separation of Lexical and Syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.

∑ **2. Compiler efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

∑ **3. Compiler portability is enhanced**: Input-device-specific peculiarities can be restricted to the lexical analyzer.

INPUT BUFFERING:

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. There are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded.

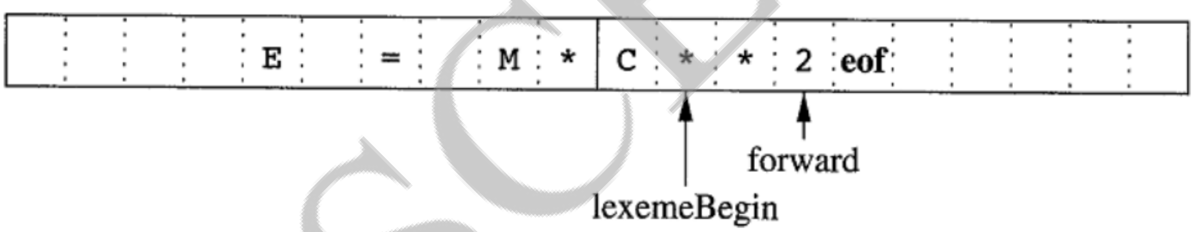


Figure1.8 : Using a Pair of Input Buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters in to a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program.

- Two pointers to the input are maintained:

1. The Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig, we see forward has passed the end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retracted one position to its left.

Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels To Improve Scanners Performance:

If we use the above scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multi way branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a **sentinel** character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**. Figure 1.8 shows the same arrangement as Figure 1.7, but with the sentinels added. Note that eof retains its use as a marker for the end of the entire input.

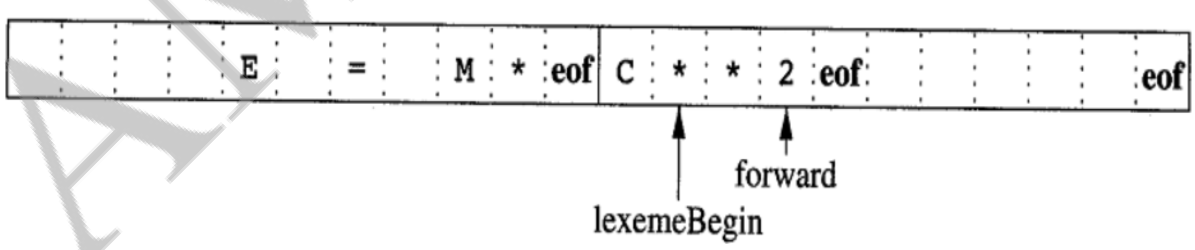


Figure1.8 : Sentential at the end of each buffer

Any eof that appears other than at the end of a buffer means that the input is at an end. Figure 1.9 summarizes the algorithm for advancing forward. Notice how the first test, which can be part of a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

```
switch ( *forward++ )
{
        case eof: if (forward is at end of first buffer )
                    {
                      reload second buffer;
                      forward = beginning of second buffer;
                    }
                    else if (forward is at end of second buffer )
                    {
                      reload first buffer;
                      forward = beginning of first buffer;
                    }
                      else    /* eof within a buffer marks the end of input */
                       terminate lexical analysis;
           break;
}
```

Figure 1.9: use of switch-case for the sentential

SPECIFICATION OF TOKENS:

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.

LEX the Lexical Analyzer generator

Lex is a tool used to generate lexical analyzer, the input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler. Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex .yy .c, it is a c program given for C Compiler, gives the Object code. Here we need to know how to write the Lex language. The structure of the Lex program is given below.

Structure of LEX Program : A Lex program has the following form:

Declarations
%%
Translation rules
%%
Auxiliary functions definitions

The declarations section : includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions. It appears between %{. . .%}

In the **Translation rules** section, We place Pattern Action pairs where each pair have the form

Pattern {Action}

The auxiliary function definitions section includes the definitions of functions used to install identifiers and numbers in the Symbol tale.

LEX Program Example:

```
%{

/* definitions of manifest constants LT,LE,EQ,NE,GT,GE, IF,THEN, ELSE,ID, NUMBER,
RELOP */

%}

/* regular definitions */

delim           [ \t\n]

ws       {      delim}+

letter          [A-Za-z]

digit           [o-91

id              {letter} ({letter} | {digit})  *

number          {digit}+ (\ . {digit}+)? (E  [+-I]?{digit}+)?

%%

{ws}            {/* no action and no return */}

if              {return(1F) ; }
```

```
then            {return(THEN) ; }
else            {return(ELSE) ; }
(id)            {yylval = (int) installID(); return(1D);}
(number)        {yylval = (int) installNum() ; return(NUMBER) ; }
‖< ‖            {yylval = LT; return(REL0P) ; )}
— <=‖           {yylval = LE; return(REL0P) ; }
—=‖             {yylval =  EQ  ; return(REL0P) ;  }
—<>‖            {yylval = NE;  return(REL0P);}
—<‖             {yylval =  GT;  return(REL0P);)}
—<=‖            {yylval  =  GE;  return(REL0P);}
%%
int installID0() {/* function to install the lexeme, whose first character is pointed to by yytext,
                    and whose length is yyleng, into the symbol table and return a pointer
                    thereto */
int installNum() {/* similar to installID, but puts numerical constants into a separate table */}
```

Figure 1.10 : Lex Program for tokens common tokens

SYNTAX ANALYSIS (PARSER)

THE ROLE OF THE PARSER:

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in the below Figure, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

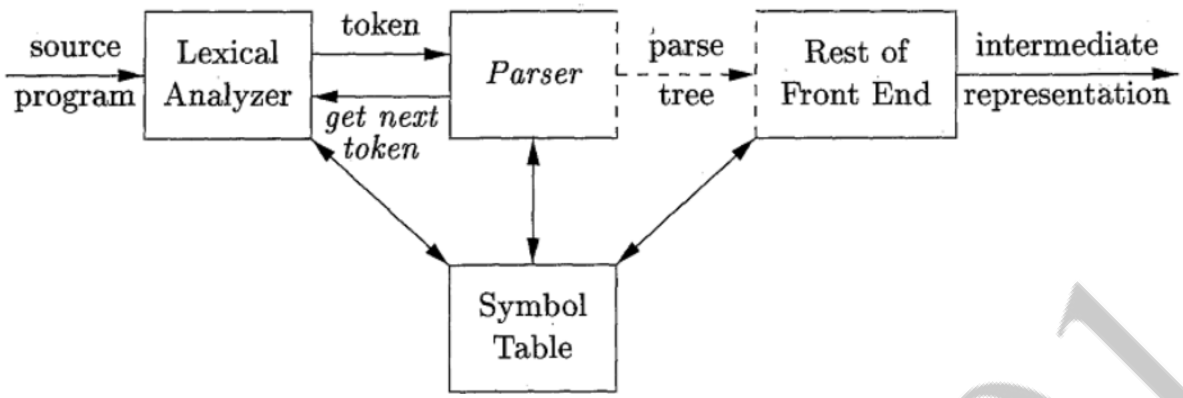


Figure2.1: Parser in the Compiler

During the process of parsing it may encounter some error and present the error information back to the user

Syntactic errors include misplaced semicolons or extra or missing braces; that is, —{" or "}." As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

Based on the way/order the Parse Tree is constructed, **Parsing** is basically **classified** in to following two types:

1. **Top Down Parsing :** Parse tree construction start at the root node and moves to the children nodes (i.e., top down order).
2. **Bottom up Parsing:** Parse tree construction begins from the leaf nodes and proceeds towards the root node (called the bottom up order).

IMPORTANT (OR) EXPECTED QUESTIONS

1. What is a Compiler? Explain the working of a Compiler with your own example?
2. What is the Lexical analyzer? Discuss the Functions of Lexical Analyzer.
3. Write short notes on tokens, pattern and lexemes?
4. Write short notes on Input buffering scheme? How do you change the basic input buffering algorithm to achieve better performance?
5. What do you mean by a Lexical analyzer generator? Explain LEX tool.

ASSIGNMENT QUESTIONS:

1. Write the differences between compilers and interpreters?
2. Write short notes on token reorganization?
3. Write the Applications of the Finite Automata?
4. Explain How Finite automata are useful in the lexical analysis?
5. Explain DFA and NFA with an Example?

Unit - II

Syntax Analysis

Need and role of parser - CFG - Topdown parsing - General strategies
Recursive Descent parser - predictive parser - LL(1) parser

* Syntax Analysis

⇒ Syntax analysis is the second phase of the compiler.

⇒ It gets the input from the lexical analyzer as tokens and generates a syntax tree or a parse tree.

Advantages of Grammar

* Grammar gives a precise and easy to understand Syntactic specification of a programming language.

* An efficient parser can be constructed automatically from a properly designed grammar.

* A grammar imparts a structure to a programming language that is useful for the translation of source program into correct object code and for the detection of errors.

* New constructs can be added to a language more easily when there is an existing implementation based on a grammatical description of the languages.

Role of the Parser

⇒ Parser obtains the string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.

⇒ It reports any syntax errors in an intelligible fashion.

⇒ It should also recover from commonly occuring errors, so that it can continue processing the remainder of its input.

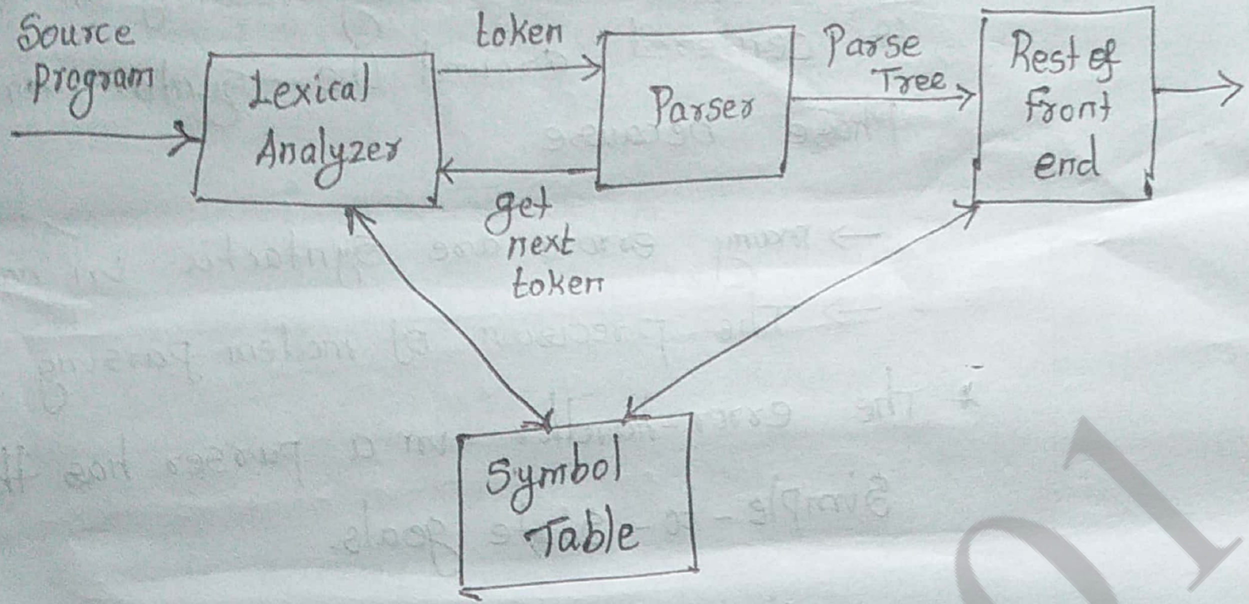
* There are 3 general types of parsers.

i) Universal parsing methods such as Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar. But this is too inefficient to use in production compilers.

ii) Top down parser : built parse trees from top (root) to the bottom (leaves)

iii) Bottom up parsers : built parse trees from the leaves and work up to the root.

In both cases the input is scanned from left to right, one symbol at a time.



Syntax Error Handling

* A good compiler should assist the programmer in identifying and locating errors.

* Programs can contain errors at many different levels.

→ Lexical - Such as mispelling an identifier, keyword or operator.

→ Syntatic - Such as arithmetic expression with unbalanced parenthesis

→ Semantic - Such as an operator applied to an incompatible operand.

→ Logical - Such as an infinitely recursive call.

* The error detection any recovery in compiler is centered around the syntax analysis phase because

→ many errors are syntactic in nature

→ The precision of modern parsing methods

* The error-handler in a parser has the following simple-to-state goals.

→ It should report the presence of errors clearly and accurately.

→ It should recover from each error quickly enough to be able to detect subsequent errors.

→ It should not significantly slow down the processing of correct programs.

→ Several parsing methods such as LL and LR methods have the viable-prefix property meaning that they detect that an error has occured as soon as they see a prefix of the input that is not a prefix of any string in the language.

Error Recovery Strategies

* The different strategies that a parser can employ to recover from a syntatic error are

→ Panic mode recovery
→ Phrase level recovery
→ Error productions
→ Global corrections.

1) Panic mode recovery

* This is the simplest method to implement and can be used by most parsing methods.
* On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing token is found.
* The synchronizing tokens are usually delimeters such as semicolon or end.
* It has the advantage of simplicity and does not go into an infinite loop.
* In the case of multiple errors in the same statement, this method may be quite adequate.

Phase Level Recovery

* On discovering an error, a parser may perform local correction on the remaining input and allows the parser to continue.

* A typical local correction would be to replace a comma by a semicolon, delete an exteraneous semicolon or insert a missing semicolon.

* The replacement should not lead to infinite loops.

* This type of replacement can correct any input string and has been used in several error-repairing compilers.

* This method was first used in top-down parsing

* The drawback is that it has difficulty in coping with the situations in which the actual error has occured.

Error productions

* Error produtions generate the erroneous constructs by using augmented grammar.

* If an error production is used by the parser appropriate error diagnostics can be generated to indicate the erroneous constructs that has

been recognized in the input.

Global Correction

* Given an incorrect input string x and grammar G, certain algorithm can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible.
* However these methods are in general too costly in terms of time and space.
*

Context Free Grammars (CFG)

* A Context free grammar consists of terminals, non-terminals, a start symbol and productions.
* Terminals – are the basic symbols from which strings are formed.
* Non-Terminals – are the syntatic variables that denote a set of strings. These help to define the language generated by the grammar.
* Start symbol – is one non-terminal in the grammar and the set of strings it denotes is the language defined by the grammar

Produtions - of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings.

Eg: Simple arithmetic expressions

expr ⟶ expr op expr | (expr) | - (expr) | id
op ⟶ + | - | * | / | ↑

id, *, -, +, /, ↑, (,) — terminals
expr, op — non terminals
expr — start symbol

each line is a production

eg: op ⟶ +, expr ⟶ (expr)

Derivations

* Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with string on the right side of the production.

eg $E \rightarrow E+E \,/\, E*E \,/\, (E) \,/\, -(E) \,/\, id$

i/p string - (id + id)

$E \Rightarrow -(E)$
$\Rightarrow -(E+E)$
$\Rightarrow -(id+E)$
$\Rightarrow -(id+id)$
$E \overset{*}{\Rightarrow} -(id+id)$

$\Rightarrow$ derives one step
$\overset{*}{\Rightarrow}$ derives zero or more steps
$\overset{+}{\Rightarrow}$ derives one or more steps

Types of Derivations

→ There are 2 types
i) Left most derivation (LMD)
ii) Rightmost derivation (RMD)

* In Left most derivation, the left most Non terminal in any sentential form is replaced at each step

* In right most derivation, the right most Non terminal is replaced in each step.

eg: $E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$

i/p string – $(id+id)$

LMD

	Apply
$E \Rightarrow -E$	
$E \Rightarrow -(E)$	$E \rightarrow (E)$
$E \Rightarrow -(E+E)$	$E \rightarrow E+E)$
$E \Rightarrow -(id+E)$	$E \rightarrow id$
$E \Rightarrow -(id+id)$	$E \rightarrow id$.

RMD

	Apply
$E \Rightarrow -E$	
$E \Rightarrow -(E)$	$E \rightarrow (E)$
$E \Rightarrow -(E+E)$	$E \rightarrow E+E$
$E \Rightarrow -(E+id)$	$E \rightarrow id$
$E \Rightarrow -(id+id)$	$E \rightarrow id$.

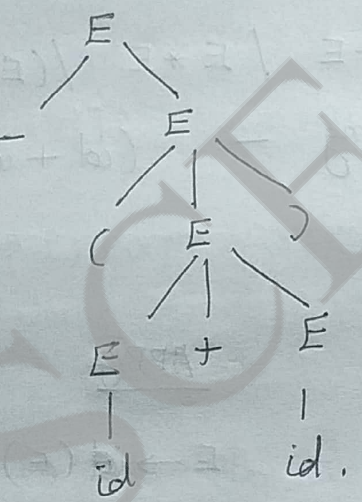
Parse Tree

* A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order.

Eg

$E \rightarrow E+E \,/\, E*E \,/\, (E) \,/\, -E \,/\, id$

i/p string: → (id + id)



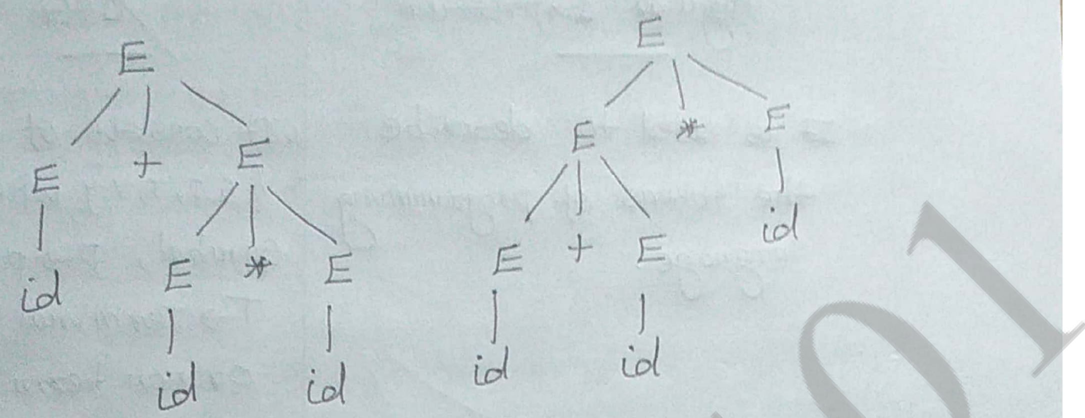
Ambiguity

* A grammar that produces more than one parse tree for same sentence is said to be ambiguous.

* A grammar that produces more than one left most derivation or more than one right most derivation is an ambiguous grammar.

Eg $E \rightarrow E+E \,/\, E*E \,/\, (E) \,/\, -E \,/\, id$

i/p string id + id * id



There are 2 parse tree for same input so the grammar is ambiguous.

Writing a grammar

* Each parsing method can handle grammars only of a certain form, hence the initial grammar may have to be rewritten to make it parsable.
* There are 4 categories in rewriting a grammar
  1) Regular Expressions VS CFG
  2) Eliminating Ambiguity
  3) Eliminating left recursion
  4) Left factoring.

Regular Expressions VS CFG

Regular Expression	CFG
1. It is used to describe the tokens of programming language.	1. It consists of a quadruple {S, P, T, V} where S → Start symbol, P → production, T → Terminal, V → variable or non terminal.
2. It is used to check whether the given input is valid or not using transition diagram.	2. It is used to check whether the given input is valid or not using derivation.
3. The transition diagram has set of states and edges	3. The CFG has set of productions.
4. It has no start symbol	4. It has a start symbol
5. It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords and so on.	5. It is useful in describing nested structures such as balanced parenthesis, matching begin-ends and so on.

Advantages of using RE

* The lexical rules of a language are simple and RE is used to describe them.
* They provide a more concise and easier to understand notation for tokens than grammars.
* Efficient lexical analyzers can be constructed automatically from RE than from grammars.
* Seperating the syntatic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler into 2 managable sized components.

Eliminating ambiguity

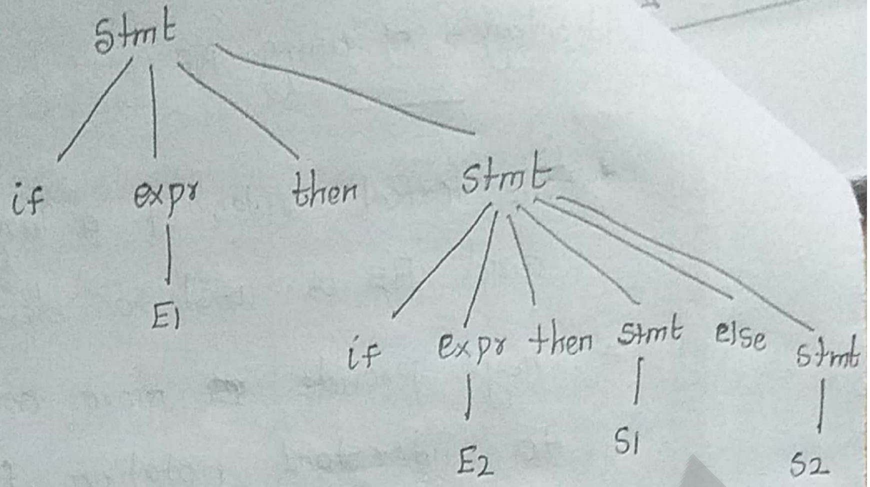
* An ambiguous grammar can be rewritten to eliminate the ambiguity.

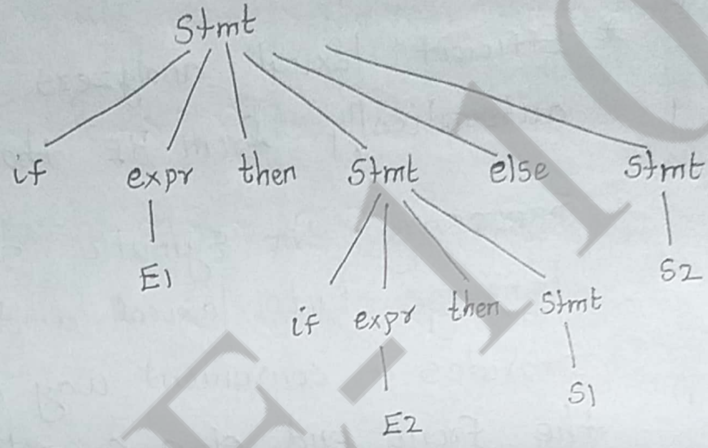
eg

stmt → if expr then stmt
 | if expr then stmt else stmt
 | other

input string - if E1 then if E2 then S1 else S2

Two parse trees can be generated for this.





* To Eliminate ambiguity the following grammar may be used dangling else grammar.

Stmt ⟶ matched Stmt / unmatched Stmt

matched Stmt ⟶ if expr then matched Stmt else matched Stmt / other

unmatched Stmt ⟶ if expr then Stmt / if expr then matched Stmt else unmatched Stmt.

Eliminating Left Recursion

* A grammar is left recursive, if it has a non terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α

* Top down parsing methods cannot handle left recursive grammars, hence left recursion has to be eliminated.

* If there is a production $A \rightarrow A\alpha / \beta$

then it can be replaced with a sequence

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' / \varepsilon$$

without changing the set of strings derivable from A

eg

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

After Eliminating left recursion

$E \rightarrow TE'$

$E' \rightarrow +TE' / \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

$F \rightarrow (E) / id$

Algorithm for eliminating left recursion

1. Arrange the non terminals in some order
   $A1, A2 \ldots An$

2. for $i=1$ to n do begin
   for $j:=1$ to $i-1$ do begin
   replace each production of the form
   $A_i \rightarrow A_j \gamma$ by the productions
   $A_i \rightarrow \delta_1 \gamma / \delta_2 \gamma / \ldots \delta_k \gamma$
   where $A_j = \delta_1 / \delta_2 / \ldots \delta_k$ are all the current A_j productions

   end
   eliminate the immediate left recursion among A_i productions.

end.

Left factoring

* Left factoring is a transformation that is useful for producing a grammar suitable for predictive parsing.

* When it is not clear which of two alternative productions to use to expand a non terminal A

We may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

* If there is any production of the form

$$A \rightarrow \alpha\beta_1 / \alpha\beta_2$$

, it can be rewritten as

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 / \beta_2$$

Eg

$S \rightarrow iEtS / iEtSeS / a$

$E \rightarrow b$

After left factoring

$S \rightarrow iEtSS' / a$

$S' \rightarrow eS / \varepsilon$

$E \rightarrow b$

Top down parsing

* Top down parsing can be viewed as an attempt to find a leftmost derivation for an input string.
* It constructs a parse tree for the input starting from the root and creating the nodes of the parse tree in pre-order.

Recursive Descent Parsing

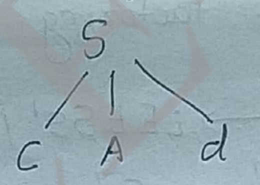
* A general form of top down parsing called recursive descent, that may involve backtracking that is making repeated scans of the input.

eg

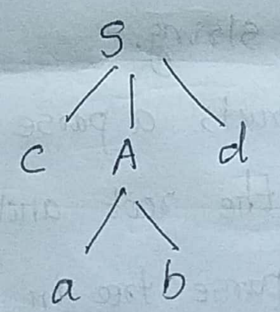
$S \rightarrow cAd$

$A \rightarrow ab/a$ input string $w = cad$

Step1 : Initially create a tree with single node labeled S, An input pointer points to 'c' the first symbol of w expand the tree with the production of S.

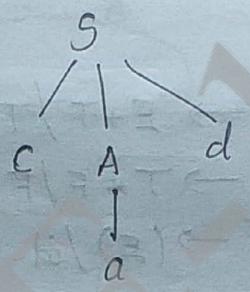


Step2 : The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the next symbol of w 'a' and consider the next leaf 'A' Expand A using first alternative



Step 3: The second symbol 'a' of w also matches with the second leaf. So advance the input pointer to the next symbol of w 'd'. But the third leaf does not match with the input symbol 'd'. Hence discard the choosen production and reset the pointer to the second. This is known as backtracking.

Step 4: Now the second alternative for A



Now we can halt and announce successful completion of parsing.

Predictive Parsers

* Predictive parsing is a special case of recursive descent parsing where no backtracking is required.

Transition diagram for predictive parsers

* To construct the transition diagram of a predictive parser from the grammar,

first eliminate left recursion from the grammar then left factor the grammar.

* Then for each non-terminal A do the following

1) Create an initial and final state

2) For each production $A \rightarrow X_1, X_2, \ldots X_n$ create a path from the initial to the final state, with edges labeled $X_1, X_2 \ldots X_n$

eg

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

By eliminating left recursion, we get

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

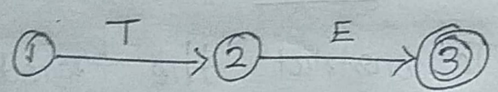
$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

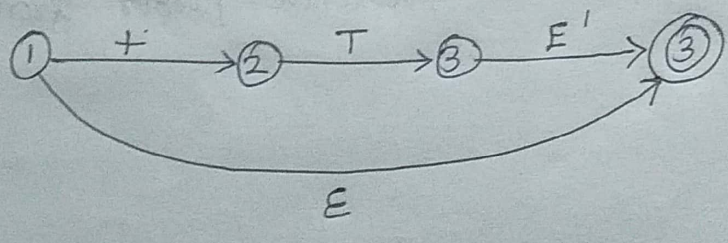
$F \rightarrow (E) / id$

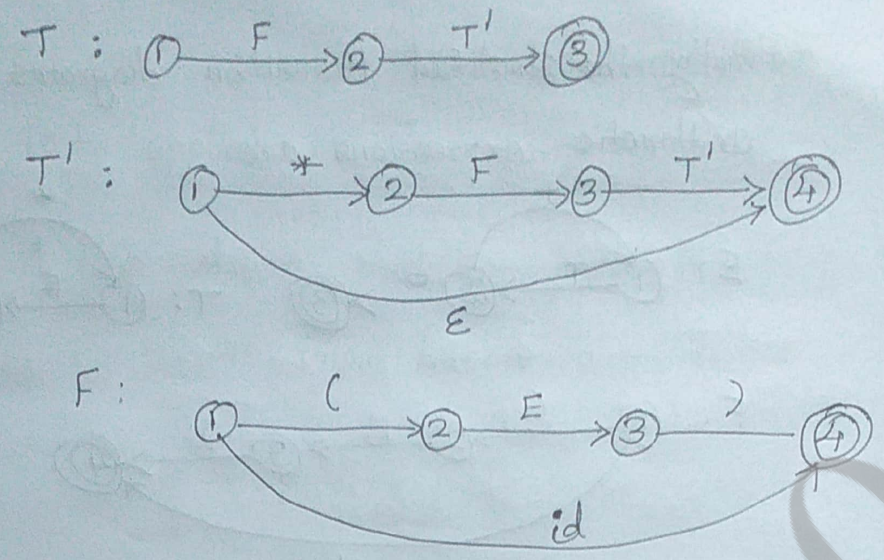
* There is no left factoring
* The transition diagram is as follows

E :

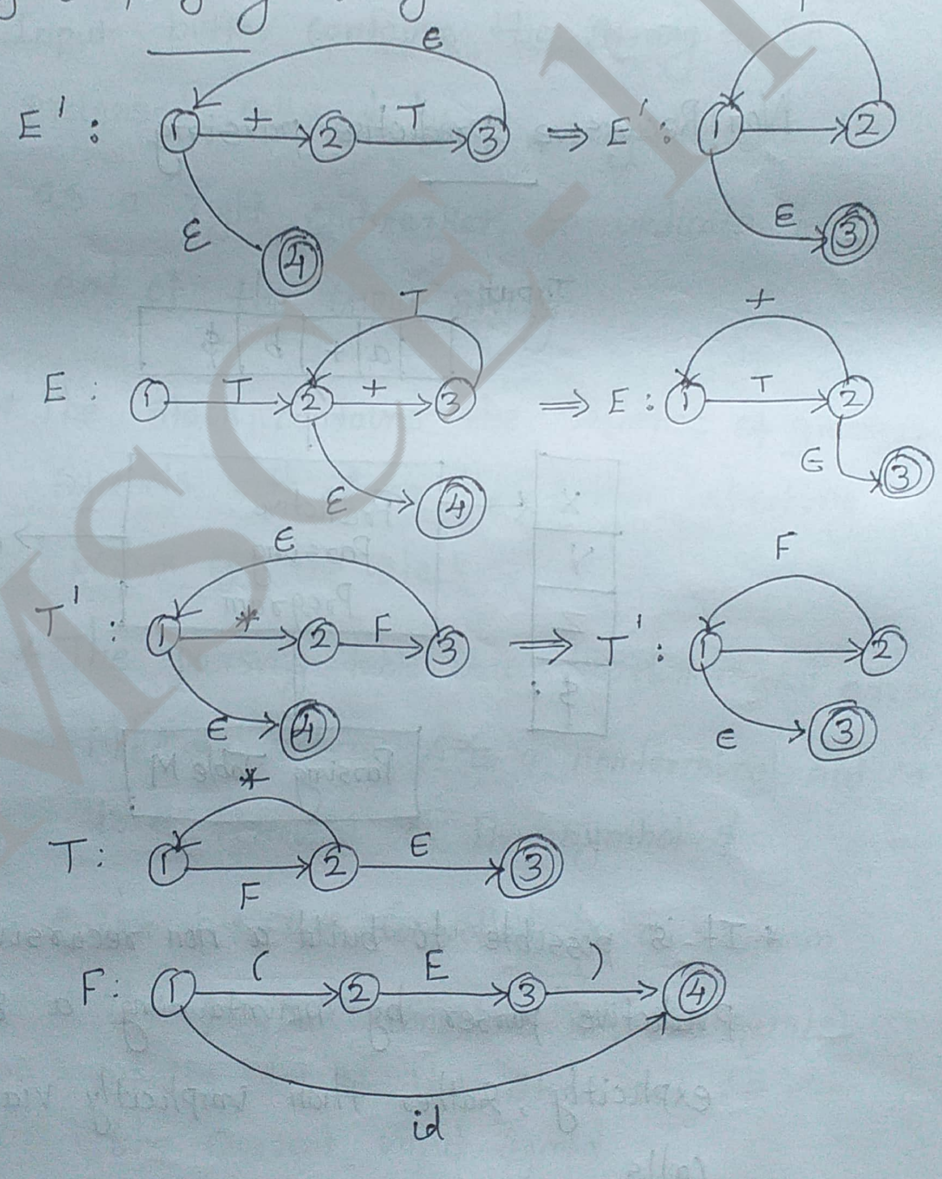


E' :

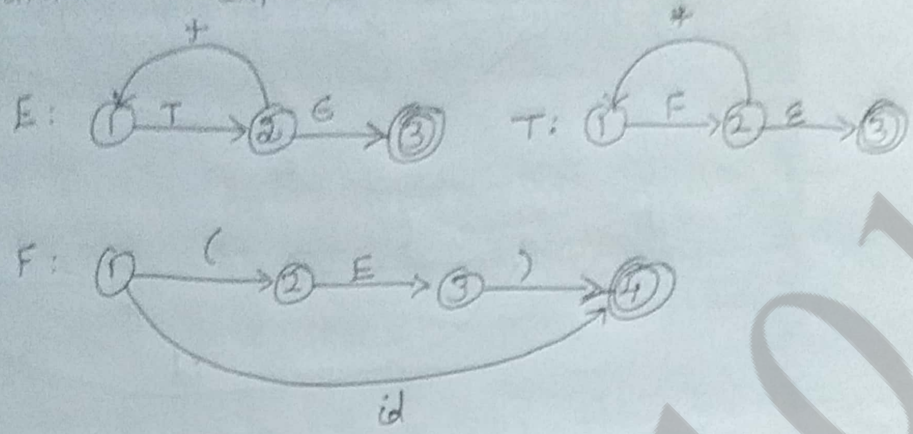




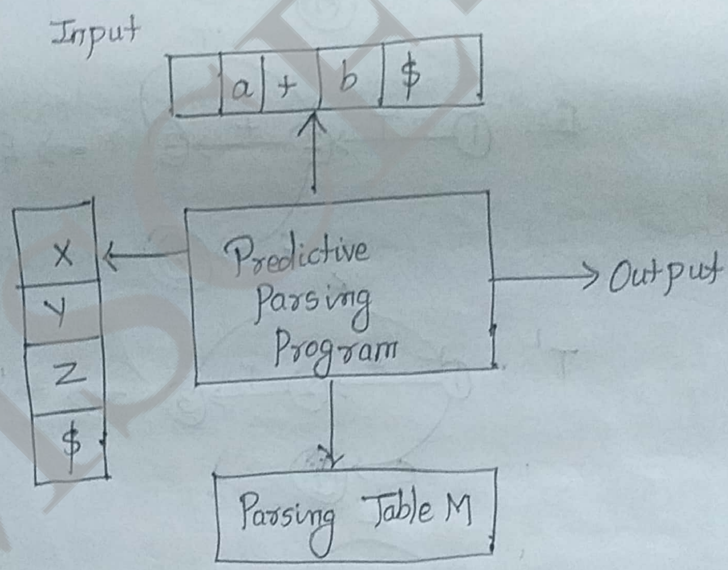
* By simplifying we get



* Finally the simplified transition diagrams for arithmetic expressions are



Non Recursive predictive parsing



* It is possible to build a non recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls.

* The non recursive parser looks up the production to be applied in the parsing table

* A table driven predictive parser has
  - → an input buffer
  - → a stack
  - → a parsing table
  - → an output stream

* Input buffer contains the string to be processed, followed by $, a symbol used as a right endmarker to indicate the end of the input string.

* The stack contains the sequence of grammar symbols with $ on the bottom, indicating bottom of the stack.

* The parsing table is a two dimensional array M[A, a], where A is a nonterminal and 'a' is a terminal or the symbol $

* The parser is Controlled by a program

* The program considers 'x', the symbol on the top of the stack and 'a' the current input symbol.

There are 3 possibilities

→ If x = a = $, parser halts and announces the successful completion of parsing.

→ If x = a = $, the parser pops 'x' off the stack and advances input pointer to the next input symbol.

→ If M[x, a] = {x → uvw}, then the parser replaces x on top of the stack by uvw

If M[x, a] = error, the parser calls an error recovery routine.

Algorithm: Non Recursive Predictive Parsing

Set the input pointer to point the first symbol of w $

repeat

let x be the top stack symbol and 'a' the symbol pointed by i/p pointer.

if x is a terminal or $ then

if x=a then

pop x from the stack and advance i/p pointer

else error()

else

if $M[X,a] = X \rightarrow Y_1, Y_2, \ldots, Y_K$ then begin

Pop X from the stack

Push $Y_K, Y_{K-1}, \ldots Y_1$ onto the stack with Y_1 on top

Output the production $X \rightarrow Y_1 Y_2 \ldots Y_K$

end

else error()

until $X = \$$

First and Follow

* The construction of a predictive parser is aided by 2 functions associated with a grammar G

1. FIRST
2. FOLLOW

* If α is any string of grammar symbols FIRST(α) be the set of terminals that begin the strings derived from α

FOLLOW(A), for non-terminal A, be the set of terminals that can appear immediately to the right of A in some sentential form.

Rules for FIRST()

1. If X is a terminal, then FIRST (X) is {X}
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X)
3. If X is a non terminal and $X \rightarrow Y_1 Y_2 \ldots Y_K$ is a production, then place 'a' in FIRST(X) if for some i, a is in FIRST(Y_i) and ε is in all of FIRST(Y_1) .... FIRST(Y_{i-1})

   ie) $Y_1, \ldots Y_{i-1} \Rightarrow \varepsilon$. If ε is FIRST(Y_j) for all $j = 1, 2 \ldots k$, then add ε to FIRST(X)

Rules For Follow ()

1. If S is a start symbol, then FOLLOW(S) contains $
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is placed in FOLLOW(B)
3. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

Construction of predictive parsing Tables

1. For each production $A \rightarrow \alpha$ of the grammar do steps 2 and 3
2. For each terminal 'a' in FIRST (α), add $A \rightarrow \alpha$ to M[A,a]
3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to M[A,b] for each terminal b in FOLLOW(A)

   If ε is in FIRST (α) and \$ is in FOLLOW (A), add $A \rightarrow \alpha$ to M[A, \$]
4. Make each undefined entry of M to be error.

Example

Construct the predictive parser for the grammar

$E \longrightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

Steps to construct non recursive predictive parser

1. Elimination of left recursion
2. Left factoring
3. Computation of FIRST and FOLLOW
4. Construction of parsing table
5. Parsing the input string.

Construction of predictive parsing Tables

1. For each production $A \to \alpha$ of the grammar do steps 2 and 3

2. For each terminal 'a' in FIRST (α), add $A \to \alpha$ to M[A,a]

3. If ε is in FIRST(α), add $A \to \alpha$ to M[A,b] for each terminal b in FOLLOW(A)

   If ε is in FIRST (α) and \$ is in FOLLOW (A), add $A \to \alpha$ to M[A, \$]

4. Make each undefined entry of M to be error.

Example

Construct the predictive parser for the grammar

$E \to E+T \,/\, T$

$T \to T*F \,/\, F$

$F \to (E) \,/\, id$

Steps to construct non recursive predictive parser

1. Elimination of left recursion
2. Left factoring
3. Computation of FIRST and FOLLOW
4. Construction of parsing table
5. Parsing the input string.

Elimination of Left Recursion

$E \rightarrow TE'$

$E' \rightarrow +TE' / \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

$F \rightarrow (E) / id$

Left factoring

No left factoring is required.

Computation of FIRST and FOLLOW

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

FIRST(E') = { +, ε }

FIRST(T') = { *, ε }

~~FIRST (E) = { $,) }~~

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { +, $,) }

FOLLOW(T') = { +, $,) }

FOLLOW(F) = { *, +, $,) }

Construction of Parsing Table

Non Terminal / Terminal	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Parsing the input String

i/p → id+id *id

Stack	input	Output
$E	id+id *id $	
$E'T	id+id *id $	E → TE'
$E'T'F	id+id *id $	T → FT'
$E'T'id	id+id *id $	F → id
$E'T'	+id *id $	
$E'	+id *id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id *id$	
$E'T'F	id *id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id$	
$E'T'F*	*id$	T' → *FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε
		Accept

Sucessful parsing

LL(1) grammar

* A grammar whose parsing table has no multiple-defined entries, can be called as LL(1) grammar.

Properties of LL(1) grammar

* No ambiguous left recursive grammar can be LL(1)

* A Grammar G is LL(1) whenever $A \to \alpha / \beta$ are two distinct productions of G, the following condition hold.

→ For no terminal 'a' do both α and β derive string beginning with 'a'

→ Almost one of α and β can derive the empty string

→ If $\beta \overset{*}{\Rightarrow} \epsilon$ then α does not derive any string begining with a terminal in FOLLOW(A)

Bottom-up Parsing

* Bottom up parser attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root.

* Various types of bottom up parsers are
  1. Shift reduce parser
  2. Operator precedence parser
  3. LR Parser
     - SLR
     - CLR
     - LALR

Shift-reduce parsing

* Shif reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up toward the root (the top).

* At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production and if the substring is chosen correctly at each step, a right most derivation is traced out in reverse.

eg

S → aABe
A → Abc/b
B → d

The sentence abbcde can be reduced to S by the following steps

abbcde	S ⇒ aABe	↑ reduction
aAbcde	⇒ aAde	
aAde	⇒ aAbcde	
aABe	⇒ abbcde	
S		
reduction leftmost	rightmost derivation	

Right

Handles

* A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the leftside of the production represents one step along the reverse of a rightmost derivation.

$$E \rightarrow E+E \mid E*E \mid (E) \mid id$$

input String: $id_1 + id_2 * id_3$

Right most derivation

$E \overset{rm}{\Rightarrow} E+E$

$\overset{rm}{\Rightarrow} E+E*E$

$\overset{rm}{\Rightarrow} E+E*id3$

$\overset{rm}{\Rightarrow} E+id_2*id_3$

$\overset{rm}{\Rightarrow} id1+id2*id3$

⇒ The underlined substrings are called handles.

Handle Pruning

* The rightmost derivation in reverse can be obtained by "handle purning"

Stack implementation of shift reduce parsing

* Shift-reduce parser uses the following data structures.
  1. Stack – used to hold grammar symbols
  2. Input buffer – used to hold the i/p string

* Initially the stack is empty and the string 'w' is in the input buffer.

Stack	i/p
$	w$

* Finally the input is empty and the stack have the starting symbol to indicate successful parsing

Stack	i/p
$S	$

Actions of shift reduce parser

i) shift - The next i/p symbol is shifted onto the top of the stack.

ii) reduce - The parser replaces the handle within a stack with non-terminal

iii) accept - Parser announces successful completion of parsing

iv) error - Parser discovers that a syntax error has occured and calls an error recovery routine.

* Viable prefixes

⇒ The set of prefixes of right sentential form that can appear on the stack of a shift reduce parser are called Viable prefixes.

Eg:

$$E \rightarrow E+E \mid E*E \mid (E) \mid id$$

input string $id_1 + id_2 * id_3$

Stack	input	Action
$		
$id1	id1 + id2 * id3 $	Shift
$E	+ id2 * id3 $	reduce by E→id
$E+	+ id2 * id3 $	Shift
$E+id2	id2 * id3 $	Shift
$E+E	* id3 $	reduce by E→id
	* id3 $	~~Shift~~ reduce by E→E+E
$E	* id3 $	Shift
$E*	id3 $	Shift
$E*id3	$	reduce E→id
$E*E	$	reduce E→E*E
$E	$	Accept.

LR Parser

Input

* It is an efficient bottom up syntax analysis technique that can be used to parse a large class of CFG. This technique is Called LR(K) parsing.

  L → Left to right Scanning of the input

  R → Rightmost derivation in reverse

  K – number of input Symbols of lookahead

* When k is omitted k is assumed to be 1

* LR Parsing is attractive for a variety of reasons

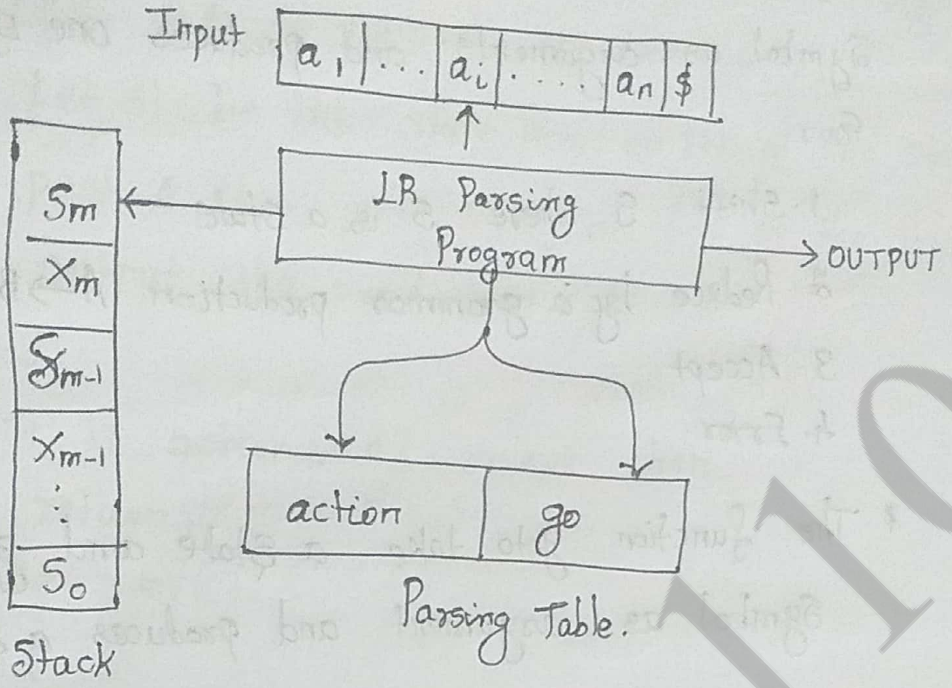
1. LR Parser can be constructed to recognize large CFG
2. LR Parsing method is the most general non-backtracking shift reduce parsing method.
3. LR Parsers can parse all languages parsed by predictive parsers
4. An LR parser can detect a syntactic error as soon as it is possible to do.

* The drawbacks of the method are.

1. Construction of LR Parser is too much work by hand for a typical programming language grammar.

2. Difficult to parse ambigious grammar by using LR Parsers.

Model of a LR Parser



* It consist of input, output, stack, driver program and parsing table (action and goto).

* The driver program is same for all LR Parser. Only the parsing table changes from one parser to another.

* Parsing program reads characters from an input buffer one at a time.

* The program uses a stack to store a string of the form

$$S_0 X_1 S_1 X_2 S_2 \ldots X_m S_m$$

Where

$S_m \rightarrow$ is on top

$S_i \rightarrow$ a symbol called state

$X_i \rightarrow$ a grammar symbol.

* The function action takes a state and i/p symbol as arguments and produces one of the four.

1. Shift S where S is a state
2. Reduce by a grammar production A → B
3. Accept
4. Error.

* The function goto take a state and grammar symbol as argument and produces a state.

LR Parsing algorithm

Set input to point to the first symbol of w$

repeat forever begin.

Let 's' be the state on top of the stack and 'a' the symbol pointed by i/p.

if action [s,a] = shift s' then begin

push 'a' then 's' on top of the stack;

advance i/p to the next input symbol;

end

else if action [S, a] = ~~Shift~~ ~~S~~ reduce $A \to B$ then begin

pop 2 * |B| symbol ~~off~~ the stack;

Let S' be the state now on top of the stack;

push A then goto[S', A] on top of the stack;

Output the production $A \to B$

end

else if action [S, a] = accept then

return;

else error()

end

Types of LR Parsers

1. Simple LR Parser (SLR)
2. Canonical LR Parser (CLR)
3. Lookahead LR Parser (LALR)

Constructing SLR(1) parsing table

* To perform SLR Parsing, take grammar as input and do the following

1. Find LR(0) items
2. Completing the closure
3. Compute goto(I, X) where
   I is the set of items
   X is the grammar Symbol.

1. LR(0) items

* An LR(0) item of a grammar G is a production of G with a dot at some position of the right side.

Eg:

$A \rightarrow XYZ$

LR(0) items

$A \rightarrow \cdot XYZ$
$A \rightarrow X \cdot YZ$
$A \rightarrow XY \cdot Z$
$A \rightarrow XYZ \cdot$

2. Closure Operation

* If I is a set of items for a grammar G then closure(I) is the set of items constructed from I by the 2 rules.

1. Initially every item in I is added to closure (I)
2. If $A \rightarrow \alpha . B \beta$ is in closure (I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow . \gamma$ to closure (I). apply rule until no more new items can be added to closure (I)

3. Goto operation

* Goto (I, X) is defined to be the closure of the set of all items $[A \to \alpha X . \beta]$ such that $[A \to \alpha . X \beta]$ is in I.

Steps to construct SLR Parsing table for grammar G.

1. Augment G and produce G'
2. Construct the Canonical collection of set of items C for G.
3. Construct the parsing action functions 'action' and goto using the algorithm that requires FOLLOW(A) for each non terminal of grammar.

Algorithm for constructing SLR Parsing table

1. Construct $C = \{I_0, I_1, \ldots I_n\}$ the collection of sets of LR(0) items for G'
2. State i is constructed from I_i, the parsing actions for state 'i' determined as follows

a) If $[A \to \alpha . a\beta]$ is in I_i and

goto$(I_i, a) = I_j$ then set action$[i, a]$

to shift j. Here 'a' must be a terminal.

b) If $[A \to \alpha .]$ is in I_i, then set action$[i, a]$

to reduce $A \to \alpha$ for all a in FOLLOW(A)

c) if $(S' \to S.)$ is in I_i, then set action$[i, \$]$

to accept.

if any conflicting actions are generated

by the above rules, the grammar is

not SLR(1).

3. The goto transitions for state i are constructed

for all non terminal A using the rule

If goto$(I_i, A) = I_j$ then goto$(i, A) = j$

4. All entries not defined by the rules ②

and ③ are made as errors.

5. The initial state of the parser is the

one constructed from the set of items

containing $[S' \to .S]$

Eg

G: F → E+T/T
T → T*F/F
F → (E)/id

G: E → E+T —— (1)
E → T —— (2)
T → T*F —— (3)
T → F —— (4)
F → (E) —— (5)
F → id —— (6)

Step 1 augumented grammar

E' → E.
E → E+T
E → T
T → T*F
T → F
F → (E)
F → id

Step 2 LR(0) items

I_0 : E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

GOTO(I_0, E)

I_1 : E' → E.
E → E.+T

GOTO (I0, T)

I2 : E → T.
T → T. * F

GOTO (I0, F)

I3 : T → F.

GOTO (I0, ( )

I4 : F → (.E)
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id

GOTO (I0, id)

I5 : F → id

GOTO (I1, +)

I6 : E → E + .T
T → .T * F
T → .F
F → .(E)
F → .id

GOTO (I2, *)

I7 : T → T * .F
F → .(E)
F → .id

GOTO (I4, E)

I8 : F → (E.)
E → E. + T

GOTO (I_4, T)

I_2 : E → T.
T → T . * F

GOTO (I_4, F)

I_3 : T → F.

GOTO (I_4, ( )

I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_4, id)

I_8 : F → id

GOTO (I_6, T)

I_9 : E → E + T.
T → T . * F

GOTO (I_6, F)

I_3 : T → F

GOTO (I_6, ( )

I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_6, id)

I_5 : F → id.

GOTO (I_7, F)

I_{10} : T → T * F

GOTO (I_7, ( )

I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_7, id)

I_5 : F → id

GOTO (I_8,))

I_{11} : F → (E).

GOTO (I_8, +)

I_6 : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_9, *)

I_7 : T → T * . F
F → . (E)
F → . id

FOLLOW(E) = { $,), + }

FOLLOW(T) = { $, +,), * }

FOLLOW(F) = { *, +,), $ }

SLR Parsing Table

	Action						GoTo		
	id	+	*	(	)	$	E	T	F
I_0	S5			S4			1	2	3
I_1		S6				Acc			
I_2		r2	S7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	S5			S4			8	2	3
I_5		r6	r6		r6	r6			
I_6	S5			S4				9	3
I_7	S5			S4					10
I_8		S6			S11				
I_9		r1	S7		r1	r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Parsing the input string id+id

Stack	input	Action.
0	id+id $	Action[0,id] = S5; shift id and push5
0id5	+id $	Action[5,+] = r6, reduce by F→id Pop 2 symbols off the stack and goto[0,F] = 3, so push F&3
0F3	+id$	Action[3,+] = r4; reduce by T→F Pop 2 symbols. goto[0,T] = 2, so push T&2
0T2	+id$	Action[2,+] = r2, reduce by E→T Pop 2 symbols, goto[0,E] = 1, So push E&1
0E1	+id$	Action[1,+] = S6, shift '+', push 6
0E1+6	id$	Action[6,id] = S5, shift id, push 5
0E1+6id5	$	Action[5,$] = r6, reduce by F→id Pop 2 symbols, goto [6,F] = 3, push F&3

stack	input	Action.
0E1+6F3	$	Action[3,$] = r4, reduce by T→F pop 2 symbols goto[6,T] = 9 Push T&9
0E1+6T9	$	Action[9,$] = r1 reduce by E→E+T Pop 6 symbols, goto[0,E] = 1, Push E&1
0E1	$	Action[1,$] = accept, Parsing is Completed Successfully.

CLR Parser (Canonical LR Parser)

* It is a Canonical Collection of LR(1) item.
* General form of an item is $[A \to \alpha \cdot \beta, a]$ where $A \to \alpha \cdot \beta$ is a production, 'a' is a terminal or $ symbol.
* Such an object is called LR(1) item and 1 indicates the length of the second Component 'Lookahead' of the item.

Construction of the sets of LR(1) items

Function closure(I)
begin
    repeat
        For each item $[A \to \alpha \cdot B\beta, a]$ in I
        each production $B \to \gamma$ in G'

and each terminal b in FIRST(β,a)

Such that [B → .γ, b) is not in I do

add [B → .γ, b] to I

until no more items can be added to I

return I

end.

procedure items(G');

begin

C := {closure ({[S' → .S, $]})};

repeat

for each set of items I in C and

each grammar symbol X

such that goto(I, X) is not empty

and not in C do

add goto(I, X) to C

until nomore set of items can be

added to C.

Constructing CLR Parsing table

1. Construct C = {$I_0, I_1 \ldots I_n$}, the collection of set of LR(1) items for G'

2. State i of the parser is constructed from I_i. The parsing actions for state i are determined as follows.

a) If $[A \rightarrow \alpha \cdot a\beta, b]$ is in I_i and $goto(I_i, a) = I_j$, then set action$[i,a]$ to "shift j". Here a is a terminal

b) If $[A \rightarrow \alpha \cdot, a]$ is in I_i, $A \neq S'$ then set action$[i,a]$ to reduce $A \rightarrow \alpha$

c) If $[S' \rightarrow S \cdot, \$]$ is in I_i, then set action$[i, \$]$ to "accept"

if a conflict results from the above rules the grammar is not LR(1).

3. The goto transitions for state i are determined as follows

if $goto(I_i, A) = I_j$ then $goto[i, A] = j$

4. All entries not defined by rules ② & ③ are made "error"

5. The initial state of the parser is the one constructed from the set containing item

$$[S' \rightarrow \cdot S, \$]$$

Example

$S \rightarrow CC$ —— ①

$C \rightarrow cC$ —— ②

$C \rightarrow d$ —— ③

Augmented grammar

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d.$

LR(1) items

I_0 : $S' \rightarrow .S, \$$

$S \rightarrow .CC, \$$

$C \rightarrow .cC, c/d$

$C \rightarrow .d, c/d$

goto(I_0, S)

I_1 : $S' \rightarrow S., \$$

goto(I_0, C)

I_2 : $S \rightarrow C.C, \$$

$C \rightarrow .cC, \$$

$C \rightarrow .d, \$$

goto(I_0, c)

I_3 : $C \rightarrow c.C, c/d$

$C \rightarrow .cC, c/d$

$C \rightarrow .d, c/d$

goto (I0, d)

I4 : C → d. , c/d

goto (I2, C)

I5 : S → cC. , $

goto (I2, c)

I6 : C → c.C, $

C → .cC, $

C → .d, $

goto (I2, d)

I7 : C → d., $

goto (I3, C)

I8 : C → cC., c/d

goto (I3, c)

I3 : C → c.C, c/d

C → .cC, c/d

C → .d, c/d

goto (I3, d)

I4 : C → d., c/d

goto (I6, C)

I9 : C → cC., $

goto (I6, c)

I6 : C → c.C, $

C → .cC, $

C → .d, $

goto (I6, d)

I7 : C → d., $

Parsing table

State	Action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			Accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
			r2		

Parsing input : cdd

Stack	Input	Action
0	cdd$	shif S3
0c3	dd$	shift S4
0c3d4	d$	reduce C→d (r3) pop 2 symbols goto(3, C) = 8
0c3C8	d$	reduce C→cC (r2) pop 4 symbols goto(0, C) = 2
0C2	d$	shift S7
0C2d7	$	reduce by C→d (r3) pop 2 symbols goto(2, C) = 5

$0 C_2 C_5$	\$	reduce $S \rightarrow CC$ Pop 4 symbols goto $(0, S) = 1$
$0 S_1$	\$	Accept

LALR Parser (Look ahead LR Parser)

* It is a Look-ahead LR Technique.
* This is often used in practice because the tables obtained by it are considerably smaller the CLR tables.

Eg

$S \rightarrow CC$ —— (1)

$C \rightarrow cC$ —— (2)

$C \rightarrow d$ —— (3)

Augmented grammar G'

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

LR(1) items

I_0 : $S' \rightarrow .S, \$$

$S \rightarrow .CC, \$$

$C \rightarrow .cC, c/d$

$C \rightarrow .d, c/d$

goto(I_0, S)

I_1 : $S' \longrightarrow S.,\ \$$

goto(I_0, C)

I_2 : $S \rightarrow C.C,\ \$$

$C \rightarrow .cC,\ \$$

$C \rightarrow .d,\ \$$

goto(I_0, c)

I_3 : $C \longrightarrow c.C,\ c/d$

$C \rightarrow .cC,\ c/d$

$C \rightarrow .d,\ c/d$

goto(I_0, d)

I_4 : $C \rightarrow .d,\ c/d$

goto(I_2, C)

I_5 : $S \rightarrow CC.,\ \$$

goto(I_2, c)

I_6 : $C \rightarrow c.C,\ \$$

$C \rightarrow .cC,\ \$$

$C \rightarrow .d,\ \$$

goto(I_2, d)

I_7 : $C \rightarrow d.,\ \$$

goto(I_3, C)

I_8 : $C \rightarrow cC,\ c/d$

goto(I_3, c)

I_3 : $C \rightarrow c.C,\ c/d$

$C \rightarrow .cC,\ c/d$

$C \rightarrow .d,\ c/d$

goto(I_3, d)

I_4

goto(I_6, C)

I_9 : $C \rightarrow cC.,\ \$$

goto(I_6, c)

I_6 : $C \longrightarrow c.C,\ \$$

$C \rightarrow .cC,\ \$$

$C \rightarrow .d,\ \$$

goto(I_6, d)

I_7 : $C \rightarrow d.,\ \$$

LALR Parsing table

State	Action			goto	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Parsing the input cdd

Stack	Input	Action
0	cdd$	shift ~~S6~~ S36
0c36	dd$	shift S47
0c36d47	d$	reduce by C→d pop 2 symbols goto[36, c]=89
0c36C89	d$	reduce C→cC Pop 4 symbols goto[0, c]=2
0C2	d$	shift S47
0C2d47	$	reduce C→d pop 2 symbols goto[2, c]=5
0C2C5	$	reduce S→CC pop 4 symbols goto[0, S]=1
0S1	$	Accept

UNIT-III SYNTAX DIRECTED TRANSLATION

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.

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*.

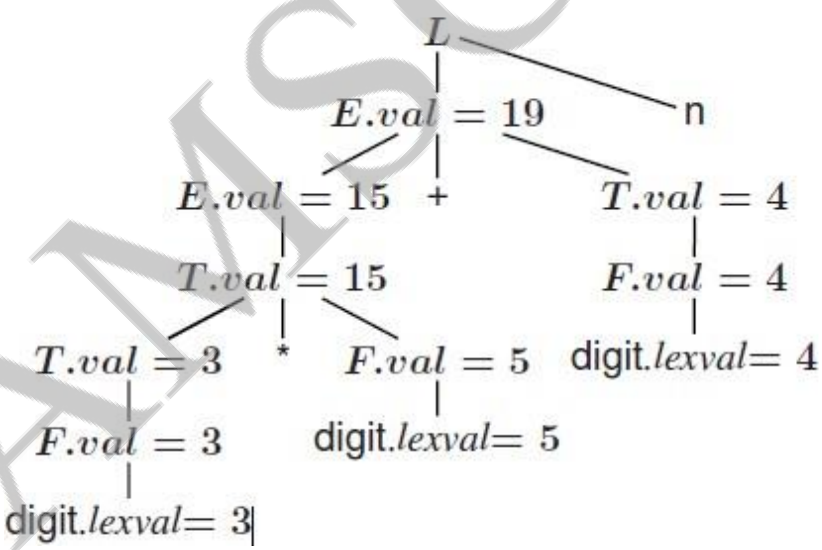
PRODUCTION	SEMANTIC RULE
$L \to E\mathrm{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathrm{digit}$	$F.val := \mathrm{digit}.lexval$

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:



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; : : : ;Xi1 (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).

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 – $O(\log2\ n)$
- 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 $O(\log2\ n)$ on average

RUNTIME ENVIRONMENT

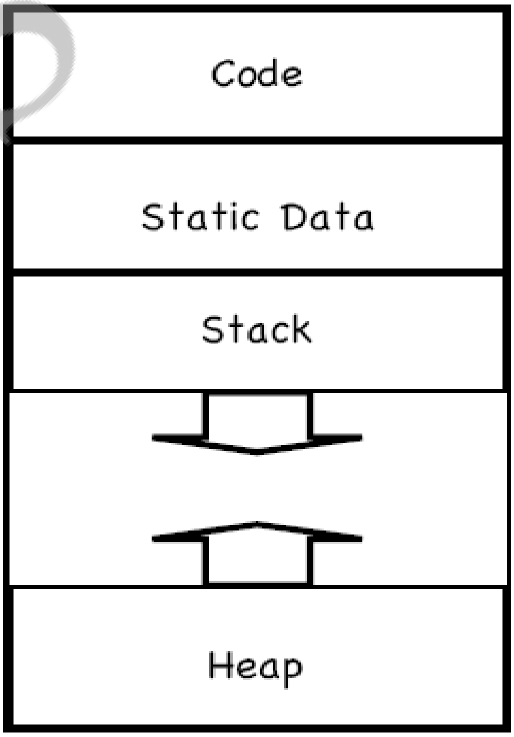
- Runtime organization of different storage locations
- Representation of scopes and extents during program execution.
- Components of executing program reside in blocks of memory (supplied by OS).
- Three kinds of entities that need to be managed at runtime:
    - Generated code for various procedures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

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.

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's static.

```
float f(int k)
{
float c[10],b;
b = c[k]*3.14;
return b;
}
```

Return value	offset = 0
Parameter k	offset = 4
Local c[10]	offset = 8
Local b	offset = 48

❖ **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

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)

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) {
retur
n
x*y;
}
int g (int →int h){
int x = 7;
```

```
return h(3) + x;
}
```

Most closely nested rule

```
main () {
int a = 0, b=0; {
    int b = 1;    {
                  int a = 2;
                  print(a,b); }        2, 1
                  {
                  int b = 3;
                  print(a,b); }        0, 3
    print(a,b);}   0, 1
print(a,b); }      0, 0
```

Call-by-Value

The actual parameters are evaluated and their r-values are passed to the called procedure

A procedure called by value can affect its caller either through nonlocal names or through pointers.

Parameters in C are always passed by value. Array is unusual, what is passed by value is a pointer.

Pascal uses pass by value by default, but var parameters are passed by reference.

Call-by-Reference

Also known as call-by-address or call-by-location. The caller passes to the called procedure the l-value of the parameter.

If the parameter is an expression, then the expression is evaluated in a new location, and the address of the new location is passed.

Parameters in Fortran are passed by reference an old implementation bug in Fortran

```
func(a,b) { a = b};
call func(3,4); print(3);
```

Copy-Restore

A hybrid between call-by-value and call-by reference.

The actual parameters are evaluated and their r-values are passed as in call- by-value. In addition, l values are determined before the call.

When control returns, the current r-values of the formal parameters are copied back into the l-values of the actual parameters.

Call-by-Name

The actual parameters literally substituted for the formals. This is like a macro- expansion or in-line expansion Call-by-name is not used in practice. However, the conceptually related technique of in-line expansion is commonly used. In-lining may be one of the most effective optimization transformations if they are guided by execution profiles.

UNIT - 4

CODE GENERATION

[Issues in the design of a code generator - The target machi Run-time storage mangement - Basic blocks and flow graphs - Next-use information - A simple code generator - Register allocation and assignment - The dag representation of basic blocks - Generation code from dags.]

Introduction :-

→ The final phase of the compiler is the code generation ph

→ The code generator takes the intermediate code representa of the source program as input and produces an equivalent target program as output

→ The symbolic representation is shown below,

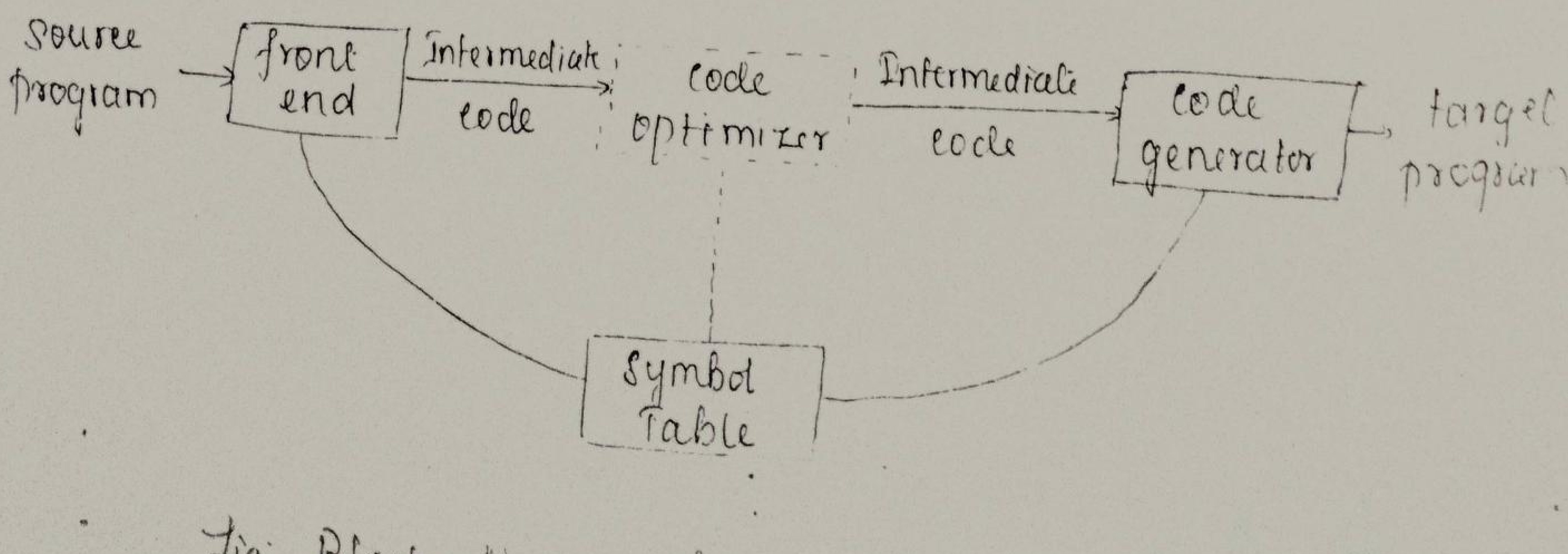


fig Block diagram of code generator

Issues in the design of a code generator

→ The output of the code generator must have high quality that is it should make effective use of the resources.

→ While designing the code generator, there are several problems encountered. Since the code generation phase is system dependent, the following issues arises during code generation phase

1. Input to the code generator (intermediate code)
2. Target programs
3. Memory management
4. Instruction selection
5. Register allocation
6. Choice of evaluation order
7. Approaches of code generation

4.1.1. Input to the code generator:-

→ The input to the code generation phase are as follows

* Intermediate code of the source program.
* Symbol table information.
  ↳ It is used to determine the run addresses of the data objects denoted by names in the intermediate code

→ The representations of the intermediate code are

i) Linear representation - postfix notation

ii) Three address representation – quadruples.

iii) Virtual machine representation – stack machine code

iv) Graphical representation – syntax trees, DAG's.

→ The input to the code generator must be free of errors. In some compilers, the semantic checking is done together with code genera

4.1.2. Target Programs:-

→ The output of the code generator is the target program co

It may be in several forms

(ie) i) Absolute machine language:

* It has the advantage of making absolute machine language as the output, that it can be placed in a fixed memory location & executed immediately.

Eg: WATFIV and PL/C → compilers that produce absolute code as target progra

* The small programs can be compiled and executed fastly.

ii) Relocatable machine language:

* The advantage of making relocatable machine language as the output; allows subprograms to be compiled separately.

* It needs loader and linker for explicit loading and linking. The set of relocatable object modules

can be linked together and loaded for execution by a linking loader. By using this, It is possible to call other previously compiled program from an object module.

iii) Assembly language:

* The advantage of making assembly language program as output, makes the process of code generation easier.

* It is easy to generate symbolic instructions & use the macro facilities of the assembler to generate the target code.

* It works well for machines with smaller memory

4.1.3. Memory Management:-

→ Mapping the names in the source program to addresses of the data objects in run-time memory is done by the front end and code generator.

→ The names in the three-address statement refers to a symbol table entry for the name.

→ The symbol table entries corresponding to names are created whenever there is a declaration in a procedure are examined.

* From the symbol table information, the relative address can be determined for the name in the data area for the procedure

three-address statements have to be converted into required addre of instructions. This process is same as backpatching techniques:

→ Let us assume labels refers to the quadraple numbers in the quadruple array.

* while scanning each quadruple, a count has been maintained for storing the list of symbols used. This count can be kept in quadruple array.

→ Eg: when we encounter

j: goto i generate the instruction as follows.

* if i < j , (ie) backward jump.

– generate a jump instruction with the target address = machine location of the first instruction in the code for quadruple i.

* if i > j , (ie) forward jump

– We must store the location of the first instruc for quadraple j on quadruple i's list. When we process quadruple i, all the instructions that refers memory locations of i are filled.

4.1.4. Instruction Selection:-

→ The instruction set of the target machine decides the instruction selection.

→ The important factors of the instruction selection are,

1. Uniformity and completeness of the instruction set.
2. Instruction speed and machine idioms
3. Size of the instruction set.

* The target machine supports all data types, only if the instruction set follows uniformity & completeness. Otherwise some special exception handling is needed.

* The efficiency of the target program depends on the instruction selection

→ For each type of three-address statement, a skeleton code can be generated.

For example,

- A three-address statement of the form,

$x := y + z$

where x, y, z are statically allocated.

- The target code for the three-address statement is,

```
MOV y, R0    // load y into register R0 (ie. R0 ← y)
ADD z, R0    // add z to R0 (ie. R0 ← R0+z)
MOV R0, x    // Store R0 into x (ie. x ← R0)
```

- But the above code is statement-by-statement code generation often produces poor code

Eg: * Let us see the following statement sequence,

$a := b+c$
$d := a+e$

- The translated code sequence is,

```
MOV b, R0      // R0 <- b
ADD c, R0      // R0 <- R0+c
MOV R0, a   }  unnecessary moves
MOV a, R0   }
ADD e, R0      // R0 <- R0 + e
MOV R0, d      // d <- R0
```

- In the above code, the 4th statement is redundant, because it is moved to the memory variable only in the last step.

→ The quality of the target code is based on the speed & size of the code. Likewise the way of implementing an operation with suitable instructions, decides the efficiency of a code generator.

* For example,

- The three instructions for incrementing a variable can be replaced by a single INC instruction.

- Let us see the code for, a := a+1.

(ie)

```
MOV a, R0
ADD #1, R0
MOV R0, a
```

→ The above target code results in poor code. Instead of having three statement, we can use single instruction named Increment instruction as,

```
INC a
```

//

→ The instruction speed is also needed to design go[od] code sequence, but accurate timing information of the ins[truction] & the ... is difficult to obtain.

- The target code should be more efficient & complete.

4.1.5 Register Allocation:-

→ The registers in the target machine affects the design of code generation. The instructions involving register operands are usually shorter & faster than those involving operands in memory.

→ The efficient utilization of registers is needed in generating the target code. The use of registers involves a phases

(ie) (i) Register allocation

→ In this phase, we select the set of variables that will reside in registers at a point in the program.

(ii) Register assignment

→ In this phase, we pick the specific registers that a variable will reside in.

→ It is difficult to assign registers for variables. Because the hardware and operating systems may require certain register-usage conventions.

→ Some machines uses register-pairs for storing the operands and results.

For eg:- the IBM system/370 machine uses register pairs for integer multiplication.

* The multiplication instruction is of the form,

| M x, y |

where, M → Multiply instruction

x → Multiplicand value taken from even register

y → Multiplier value taken from odd register.

* The division instruction is of the form,

| D x, y |

where, x – 64 bit dividend even register in even/odd register

y – divisor value.

even register – remainder value.

odd register – quotient value.

→ Now consider the two three-address statements sequence below,

(a)	(b)
t := a + b	t := a + b
t := t * c	t := t + c
t := t / d	t := t / d

Fig two three-address code sequence

The optimal code sequences for the above three-address code generated are as follows,

(a)		(b)	
L	R1, a // Load a to R1	L	R0, a
A	R1, b // R1 ← R1 + b	A	R0, b
M	R0, c // R0 ← R0 * c	A	R0, c
D	R0, d // R0 ← R0/d	SRDA	R0, 32 // shifts the c into R1 & cl
ST	R1, t	D	R0, d
		ST	R1, t

Fig Optimal machine code sequences

→ The instruction SRDA R_0, 32 Shifts the dividend in[illegible] and clears R_0, so that all the bits equal its sign bit.

→ L Refers Load, ST refers to store and A refers [illegible] D refers to Divide.

4.1.6. Choice of Evaluation Order:-

→ The order of evaluating ~~of~~ the expressions (ie. instruction execution) can affect the efficiency of the target code. But identifying the best order is an optimization problem (difficul[t]). Many problems are 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.

4.1.7. Approaches 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 a code generator might face.

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

The Target machine :-

→ The instruction set plays a vital role in the code generation. For generating a good code for a target machine, the instruction set is the prerequisite, which has been used under different ways.

→ The target machine always differs by the memory and the operating system.

→ The target code depends on the instruction type, addressing modes and instruction cost.

4.2.1. Instructions & its Types :-

→ Since the target machine is a byte addressable machine with four bytes to a word and 'n' general registers $R_0, R_1, \ldots, R_{n-1}$

* The instruction format is,

op source, destination

where, op → refers to op-code,

source, destination → refers to data fields.

* The Op-code types are as follows,

- MOV (move source to destination)
- ADD (add source to destination)
- SUB (subtract source from destination)

→ The source and destination fields are not long enough to hold memory addresses

* A Simple Target machine Model:-

→ The target computer designed has load and s[illegible] operations, computation operations, jump operations and conditional jumps.

→ It is a byte-addressable machine with 'n' general purpose registers. Each machines language has a specific set of instructions.

→ Most of the instructions consists of an operator, target register or location followed by a list of source operands. Some of the instructions are as follows,

i) Load Operations:

1. `LD dest, addr` // dest ← [addr]

- This instruction loads the content in location addr into location dest.

* dest ← content in addr.

2. `LD reg, x` // reg ← x

- This instruction loads the cont[illegible] in location x into register reg.

3. `LD reg1, reg2` // reg1 ← reg2

- This instruction loads the value in register 2 into register 1.

ii) Store Operations:-

[ST x, reg] // $x \leftarrow reg$.

- This instruction stores the values in register reg into the location x.

iii) Computation operations.

[OP dest, s_1, s_2] // dest $\leftarrow s_1$ OP s_2

- This operation performs the operations specified by the operator OP. The operations include ADD, SUB, MUL etc.

- Any operation is performed on two source operands s_1 and s_2 and is stored in the destination dest.

iv) Unconditional jumps:

[BR L] // unconditional jump to L.

- This instruction makes the control to branch to the machine instruction with label L

v) Unconditional jumps:-

[Bcond r, L] // L - Label

- This instruction performs the test on register r and jumps to the location specified by L.

eg: BLTZ r, L

→ If the value in register r is less than zero, then the jump to the location specified as L

4.2.2. Addressing Modes:-

→ The source and destination of an instruction specified by combining the registers and memory locat with the address modes.

→ The target machine has its own addressing modes. The contents (a) denotes the contents of the register or memory address represented by 'a'.

* The addressing modes and its costs are as follows,

Mode	Form	Address	Cost
Absolute	M	M	1
Register	R	R	0
Indexed	C(R)	c + contents(R)	1
Indexed Register	*R	contents (R)	0
Indexed indexed	*C(R)	contents(c + contents(R))	1
literal	#c	c	0

→ The memory location M or Register R represents its when used as a source or destination

* For example,

i) MOV R_0, M — stores the contents of register into memory location M.

ii) MOV 4(R_0), M — stores the values contents(4 + contents (R_0)) into memory location M.

iii) MOV *4 (R0), M — Stores the value (contents (4 + contents(R0))) into memory location M.

iv). MOV #1, R0 — loads the constant 1 into the register R0

4.2.3. Instruction Costs:

→ Instruction cost = 1 + cost for source & destination address modes.

→ The instruction cost corresponds to the length of the instruction

→ The address modes involving registers have cost zero while those with a memory location or literal have cost 1.

* We can minimize the time taken to perform the instruction by choosing the instruction with small length

* A good code generating algorithm should not generate duplicating steps like moving the content to memory location and then moving it then to register

addr mode	cost
Register →	0
m/y, constants →	1

Examples:

1. MOV R0, R1 — has cost one, since it involves only registers

2. MOV R5, M — has cost 2, since it has memory location M.

3. ADD #1, R4 — has cost 2, since it has the constant value

4. SUB 5(R_0), *4(R_1) - has cost 3, since both source & destination operands have constan[t]

5. MOV b, R_0
ADD c, R_0 --- cost = 6
MOV R_0, a

(ie) MOV b, R_0 = 1+1+0 = 2
ADD c, R_0 = 1+1+0 = 2
MOV R_0, a = 1+0+1 = 2
= 6

6. MOV b, a
ADD c, a — cost = 6

7. MOV *R_1, *R_0
ADD *R_2, *R_0 — cost = 2

8. MOV R_1, R_0
ADD R_2, R_1 — cost = 4
MOV R_1, a

9. ADD R_2, R_1
MOV R_1, a — cost = 5
MOV R_1, b

→ The cost associated with each and every process of compiling and running a program. The common cost measures are the length of compilation time and the size, running time and power consumption of the target program

- It is not an easy job to find the actual cost of compiling & running a program. Finding an optimal target program for a given source program is an undecid[able] problem.

Run-time Storage Management

→ The meaning (semantics) of procedure in a language determines how names are bound to storage during execution.

→ Information needed during an execution of a procedure is kept in a block of storage called activation record.

* An activation record is a datastructure maintaining a block storage to keep the information required during an execution of a procedure.

* It contains the fields like temporaries, local data, saved machine status, access link information, actual parameters and returned value.

→ In this phase, the code must be generated to manage the activation record at run-time.

→ There are 2 standard storage allocation strategies namely,

1. Static allocation
2. Stack allocation.

→ The activation record has the fields to hold the result, parameters, local data, temporaries.

→ Since the run-time allocation and deallocation of activation records occurs as part of the procedure call & return sequences.

→ The three-address statements associated with procedure call and return sequences, which are us... for the storage allocation strategies are,

* call
* return
* halt
* action — it is a place holder for other statem...

→ We assume that run-time memory is divided into areas for,

1. code
2. Static data
3. Stack
4. Heap.

→ The size and the layout of the activation recor... are communicated to the code generator through the information about names that is present in the symbol ta...

4.3.1. Static Allocation:

→ In static allocation, the position of an activatio... record in memory is fixed at compile time.

* A caller is a one who makes the call for a ... sequ...
* A callee is a one who has the required cal... seque...

→ The call statement, in the intermediate code is implemented by a sequence of two target machine instructions. That is,

```
MOV   #here + 20, callee.static-area
GOTO  callee.code-area
```

where,

MOV - instruction saves the return address

GOTO - instruction transfers control to the target code for the called procedure.

callee.static-area - is the attribute, that refers the address of the activation record.

callee.code-area - refers to the address of the 1st instruction of the called procedure

#here + 20 - return address (ie. address of the instruction following the GOTO instruction

20 - 5 words or 20 bytes (ie. three constant (callee.static-area, callee.code-area, #here+20) plus 2 instructions in the calling sequence (mov and Goto) cost is 5 words or 20 bytes).

(MOV, #here+20, callee.static-area 1+1+1 = 3
GOTO, callee.code-area 1+1 = 2
5)

→ The return statement, from the procedure callee is implemented by,

```
GOTO  *callee.static-area
```

where,

* callee.static_arec – move to the address stored at beginning of the activation record

→ Implementation of action statement,

ACTION – we use pseudo-instruction ACTION to implement action statement

→ Implementation of halt statement,

HALT – the last statement of each procedure is HALT, which returns controls to the operating system

Example:-

- Consider the following pseudo code & generate the assembly codes using static allocation method.

Three-address code

```
/* code for c */
    action 1
    call p
    action 2
    halt
/* code for p */
    action 3
    return
```

Activation Record for c (64 bytes)

0:	return address
8:	arr
56:	i
60:	j

Activation record for p (88 bytes)

0:	return address
4:	buf
84:	n

fig Input to a code generator.

for this, let us start the procedures at address 100 and 200 respectively. (ie) c:100, p:200.

→ Assume each action instruction takes 20 bytes. The activation records for the procedures are statically allocated at starting addresses 300 & 364 respectively. (ie) c:300 p:364

* The target code for the above three address code is shown below,

```
// code for c

100 : ACTION1
120 : MOV #140, 364   /* save return address 140 at
                         activation record of p*/
132 : GOTO 200        /* call p */
140 : ACTION2
160 : HALT
...

// code for p
200 : ACTION3
220 : GOTO *364       /* return to address saved in location
                         364
...

// 300-363 occupies the activation record for c

300 :                 /* return address */
304 :                 /* local data for c */
...

// 364-451 holds activation record for p
364 :                 /* return address */
368 :                 /* local data for p */
```

For procedure c, execution starts with the instruction ACTION, at address 100. The next instruction "MOV" stores the return address 140 in the machine status field, at address 364, the first word of the activation record of p

4.3.2. Stack allocation:

→ In stack allocation, a new activation record is pushed onto the stack for each procedure execution & popped when the activation ends.

→ This stack allocation uses relative addresses for storage in activation records, because the position of the record for an activation of a procedure is not known until run time

→ Relative addresses in an activation record can be taken as offsets from known position in the activation record that is stored in registers.

* The stack pointer 'SP' maintains the beginning of the activation record on top of the stack.

→ when a procedure call occurs, calling procedure increment SP and transfers control to the called procedure.

$$SP = SP + x$$

where, x → caller record size (size of activation records)

, when the control returns to the caller, the ... [illegible]
[pr]ocedure decrements SP and de-allocate the activation
[r]ecord of the called procedure.

$$SP = SP - x$$

* **Initialization of stack**:-

→ At first, the procedure initializes the stack by setting SP to the start of the stack area in memory.

→ The code is,

```
MOV #stackStart, SP    //initialize the stack
code for the first procedure
                       // terminate
HALT
```

* **Implementation of call statement**:- (code for procedure call)

→ The procedure call sequence increments SP with the record size, saves the return address and transfers control to the called procedure.

→ The code for procedure call is,

```
ADD #caller.recordsize, SP
MOV #here+16, *SP     //save return address
GOTO callee.code_area
```

where, caller.recordsize - size of the activation reco[rd]

\#here+16 - address of the instruction follo[wing] the GOTO.

Scanned by TapScanner

* Implementation of Return statement:- (code for return)

→ The return sequence consists of 2 parts.

i) Called procedure side.

- The called procedure transfers control to the return address of the caller.

(ie) GOTO *0(SP) /* return to caller */

where,

0(SP) - address of first word in the activation record.

*0(SP) - return address saved at 0(SP).

ii) Caller side:-

- The return sequence restores the SP value

SUB #caller.recordsize, SP

↳ This part of the return sequence, which decrements SP, thereby restoring SP to its previous value. (beginning of the activation record of the caller)

(ie) after the subtraction, SP points to the beginning of the activation record.

* Example:-

consider the three-address statements for the procedure s, p.

```
/* code for s */
action1
call q
action2
halt
/* code for p */
action3
return
```

/* code for q */

action$_4$

call p

action$_5$

call q

action$_6$

call q

→ Consider the sizes of the activation records at compile time for procedures,

s - ssize

p - psize

q - qsize.

→ Let us assume the code for the procedure starts at address

s : 100

p : 200

q : 300 & stack starts at address 600.

The target code for the program is as follows,

// code for procedure s

```
100 : MOV #600, SP        // initialize the stack
108 : ACTION1
128 : ADD #ssize, SP      // call sequence begins
136 : MOV #152, *SP       // push return address
144 : GOTO 300            // call q
152 : SUB #ssize, SP      // restore SP
160 : ACTION2
180 : HALT
      ...
```

// code for procedure p.

```
200 : ACTION3
220 : GOTO *0(SP)         // return
      ...
```

// code for procedure q

```
300 : ACTION4
320 : ADD #qsize, SP
328 : MOV #344, *SP      // push return address
336 : GOTO 200           // call p
344 : SUB #qsize, SP
352 : ACTION5
372 : ADD #qsize, SP
380 : MOV #396, *SP      // push return address
388 : GOTO 300           // call q
396 : SUB #qsize, SP
404 : ACTION6
424 : ADD #qsize, SP
432 : MOV #448, *SP      // push return address
440 : GOTO 300           // call q
448 : SUB #qsize, SP
456 : GOTO *0(SP)        // return
...
600 :                    // stack starts here.
```

Basic Blocks and Flow Graphs:

→ The graphical representation of the three-address statements, called flow graph, which is used for understanding the code generation algorithms.

→ In flow graph, the nodes represent computations and edges represent flow of control.

Scanned by TapScanner

DAG - Efficient I Flow (Inprint

UNIT - 5

CODE OPTIMIZATION

[Introduction - The principle sources of Optimization - Peephole optimization - Optimization of basic blocks - Loops in flow graphs - Introduction to global data-flow analysis - Code improving transformations]

Introduction

→ The code optimization phase attempts to improve the intermediate code, so that faster running machine code will result

→ This is an optimal phase of a compiler

→ The aim of this phase is to make the program to run faster or take less space or both.

* Compilers that apply code-improving transformations are called optimizing compilers.

→ The optimizing compiler is need to create an effective target code.

Criteria for code-improving transformations:-

- While doing the code-improving transformations the following factors has to be considered

1. A transformation must preserve the meaning of prog.
2. A transformation must, on the average, speedup prog. by a measurable amount.
3. A transformation must be simple and worth the effort.

1.2. Getting better performance (Performance of an optimizing compiler)

→ The performance of the target code depends on the reduction of running time of the program.

↳ This is done by improving the program at all levels from the source level to target level.

→ In order to improve the performance of the compiler, changes can be made both at the user end (source) as well as the compiler end (target).

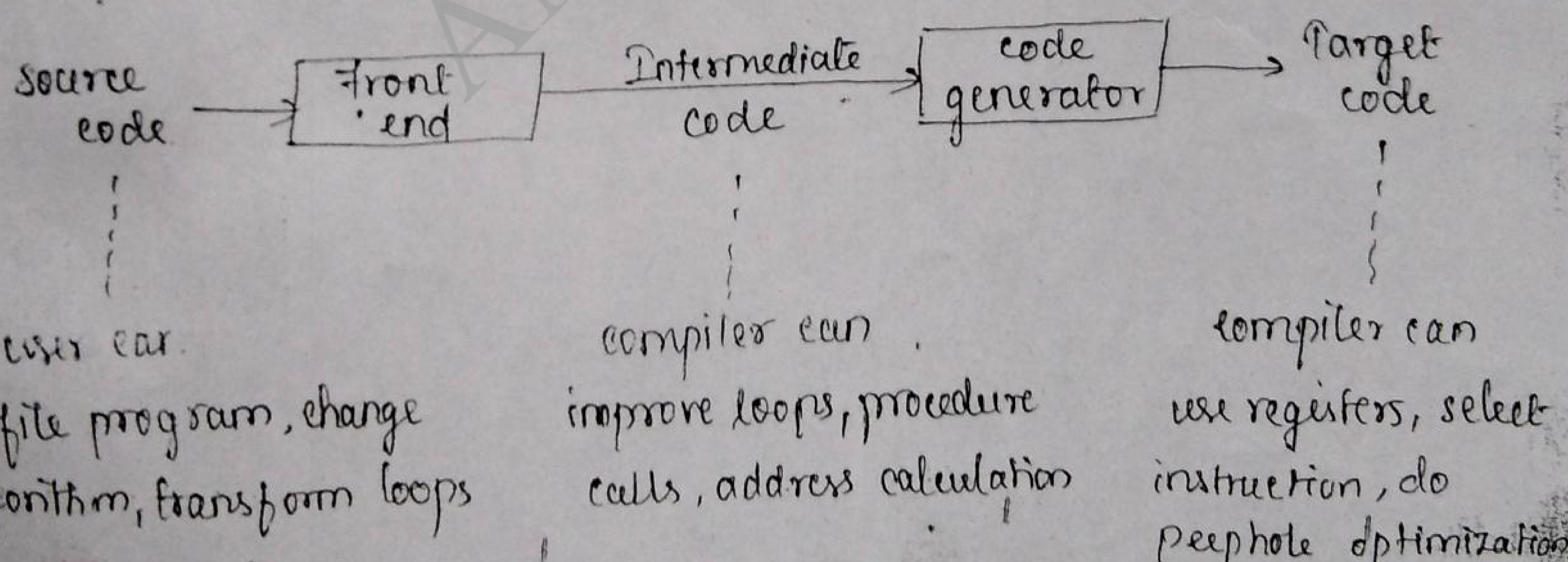


Fig. places for potential improvements by the user and the compiler

- At each level, the available options fall between the two extrem of finding a better algorithm & of implementing a given algorithm that fewer operations are performed.

In the user end, the source language can be given in different forms. (i) A user can,

* profile program
* change algorithm
* transform loops.

→ Some changes can be made at the compiler end. At the level of intermediate language, the compiler can improve,

i) loops
ii) procedure calls
iii) Address calculations.

→ At the level of the target machine, it is the duty of the compiler to use the machine resources. A compiler can,

i) use registers
ii) select instructions
iii) do peephole optimizations. (transformations)

5.1.3 An organization for an optimizing compiler

→ The code improvement phase consists of control-flow & data-flow analysis. The compiler is organized as follows.

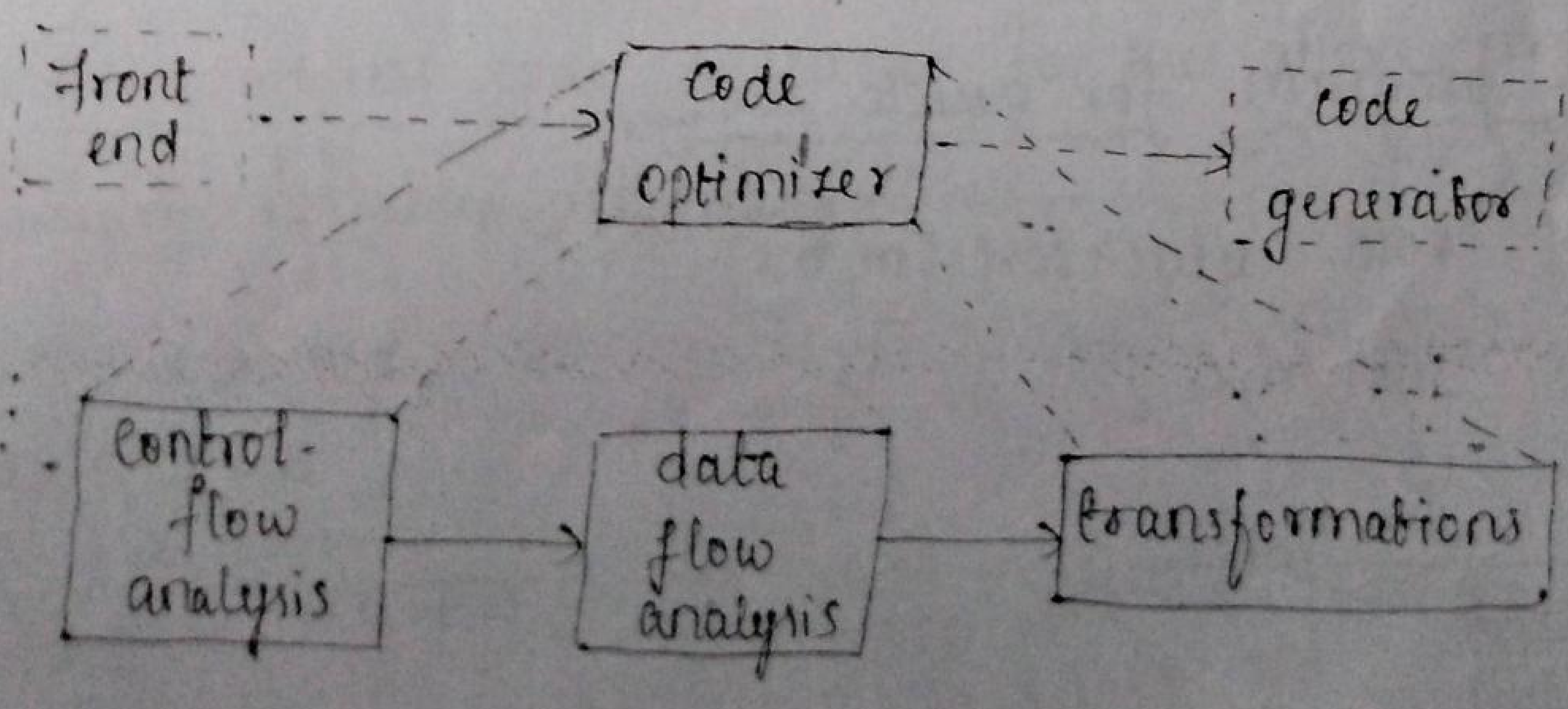


fig Organization of the code optimizer

→ The code generator produces the target code from optimized intermediate code.

→ The organization has the following advantages.

1. The operation needed to implement the high-level constructs are made explicit in the intermediate code so it is (easy) possible to optimize them.

2. The intermediate code can be independent of the target machine, so the optimizer doesnot have to change much, if the code generator is replaced by one for a different machine.

* In the code optimizer, programs are represented by flow graphs, in which edges indicate the flow of control and nodes represent the basic block.

* Example:

- Consider the following C program for Quicksort and construct three-address code sequence & flow graph and also optimize the code by using principle sources of optimization.

i) C-program for quick sort.

```
void quicksort (m, n)
int m, n;
{
    int i, j;
    int v, x;
```

```
if (n<=m) return;
i=m-1;
j=n;
v=a[n];
while(1)
{
    do
        i=i+1;
        while (a[i]<v);
    do
        j=j-1;
        while (a[j]>v);
    if (i>=j) break;
    x=a[i];
    a[i]=a[j];
    a[j]=x;
}
x=a[i];
a[i]=a[n];
a[n]=x;
quicksort (m,j);
quicksort (i+1, n);
}
```

) The three-address code sequence for the above program with temporary variables are as follows,

(1) i := m-1

(2) j := n

(3) t_1 := 4 * n

(4) v := a[t_1]

(5) i := i+1

(6) t_2 := 4 * i

(7) $t_3 := a[t_2]$

(8) if $t_3 < v$ goto (5)

(9) $j := j - 1$

(10) $t_4 := 4 * j$

(11) $t_5 := a[t_4]$

(12) if $t_5 > v$ goto (9)

(13) if $i >= j$ goto (9)

(14) $t_6 := 4 * i$

(15) $x := a[t_6]$

(16) $t_7 := 4 * i$

(17) $t_8 := 4 * j$

(18) $t_9 := a[t_8]$

(19) $a[t_7] := t_9$

(20) $t_{10} := 4 * j$

(21) $a[t_{10}] := x$

(22) goto (5)

(23) $t_{11} := 4 * i$

(24) $x := a[t_{11}]$

(25) $t_{12} := 4 * i$

(26) $t_{13} := 4 * n$

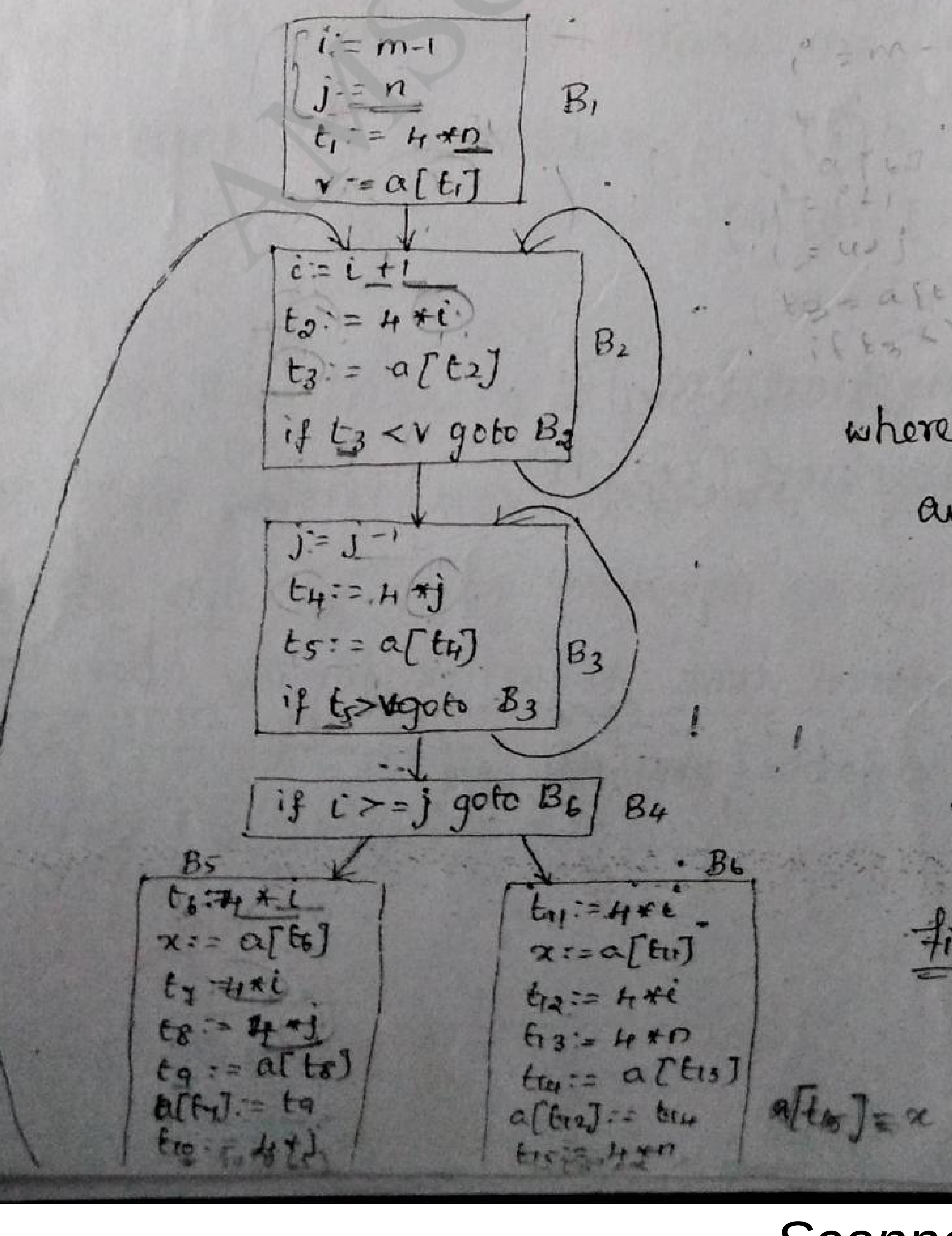
(27) $t_{14} := a[t_{13}]$

(28) $a[t_{12}] := t_{14}$

(29) $t_{15} := 4 * n$

(30) $a[t_{15}] := x$.

iii) Flow graph for the basic block is shown below.



where $B_1, B_2, B_3, B_4, B_5, B_6$ are Basic blocks

fig: flow graph for quicksort algorithm.

Principle Sources of Optimization

→ There are several methods for transforming the program code into optimized code.

→ The transformation may be of 2 types.

i) Local – The transformations can be performed by looking only at the statements in the basic block

ii) Global – The transformations which are performed at more than one basic block is called glo[bal]

→ Many transformations can be performed at both the local and global values. Local transformations are usually performed f

* There are number of ways in which a compiler can improve a program without changing the function it computes

1) Function-preserving transformations

2) Loop optimization.

5.2.1. Function-preserving transformations:-

→ In this transformation, the compiler improves the program without changing the function it computes.

→ Most frequently used function-preserving transformation are as follows, (ie)

i) Common subexpression elimination

ii) Copy propagation

iii) Dead-code elimination

constant folding.

i) Common Subexpression elimination:

→ An occurrence of an expression E is called as com… subexpression, if E was previously computed and the values of variables in E has not been changed till the next computation.

→ For eg:

Consider the basic block B5 & B6 in the flow graph of quick sort.

* In the flow-graph block B5 has common sub-expressions like the computation of 4*i and 4*j by the variables t_7 and t_{10} respectively. This can be modified as follows,

B5

```
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
```

Before

⟹

B5

```
t6 := 4*i
x := a[t6]

t8 := 4*j
t9 := a[t8]
a[t6] := t9

a[t8] := x
goto B2
```

After

→ In this local transformation, the above block B5, the statement t_6, t_7, t_8 & t_{10} have the common subexpression 4*i & 4*j respectively. The variable t_7 & t_{10} have been eliminated.

(ie) t_7 is replaced by t_6 & t_{10} is replaced by t_8

likewise perform the same in block B6.

B6 (Before)

```
t11 := 4*i
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x
```

⟹

B6 (After)

```
t11 := 4*i
x := a[t11]
t13 := 4*n
t14 := a[t13]
a[t11] := t14
a[t13] := x
```

→ Since we have eliminated the common subexpression within the basic block, so it is called local transformation.

* Now let us perform the both local and global common subexpression elimination for the quicksort flow graph.

(ie)

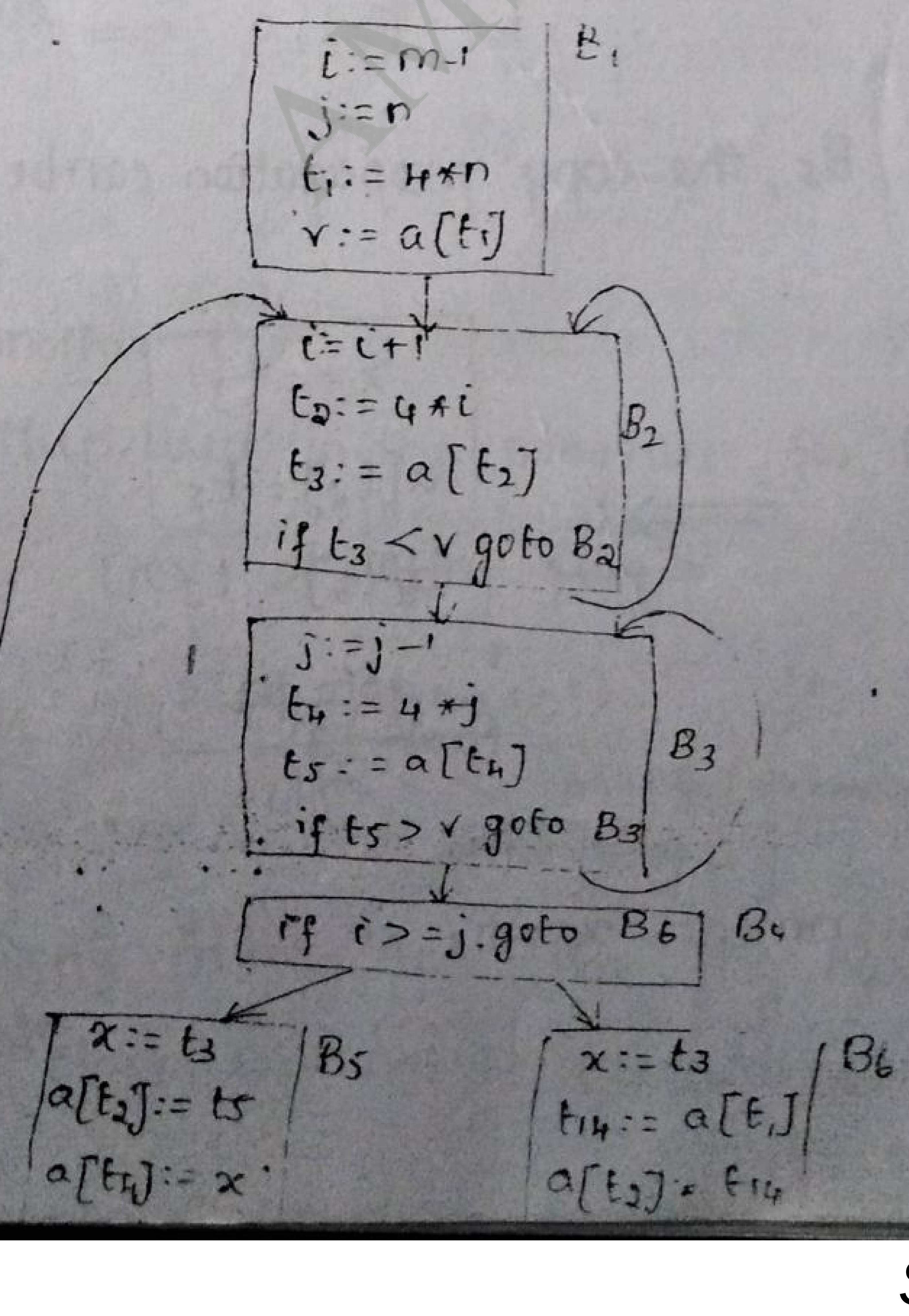


Fig: flow graph for quick sort after eliminating global common subexpression

→ In this flow graph, $x := a[t_6]$ can be replaced by $x := t_3$ where t_3 is holding $a[t_2]$.

ii) <u>Copy propagation</u>:-

→ An assignment of the form, $f := g$ followed by any assignments of 'f' will be replaced by 'g'. This method is called as '<u>copy propagation</u>' or '<u>copy statements</u>' or '<u>copies</u>.'

→ The process of the copy propagation transformation is to use 'g' for 'f', wherever possible after the copy statement $f := g$.

→ This will <u>improve</u> the unnecessary assignment on variable.

(ie)

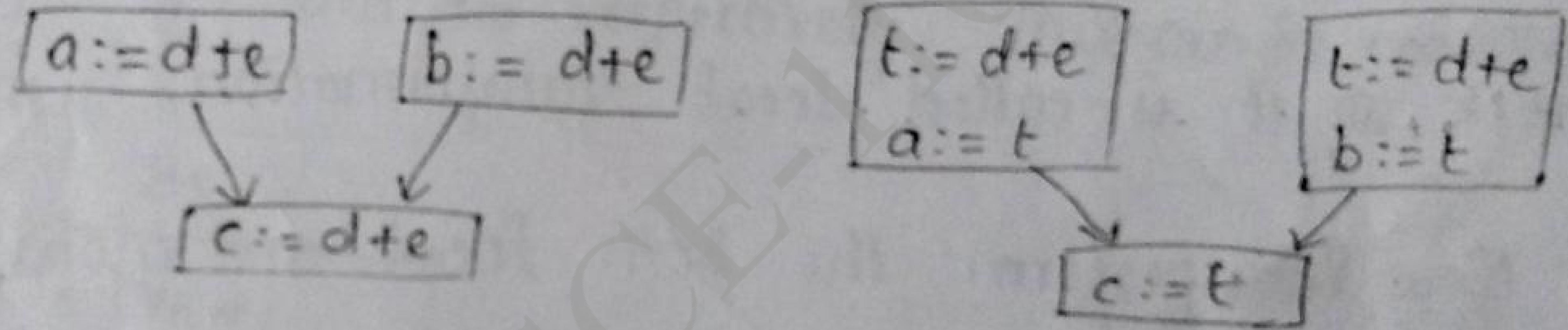
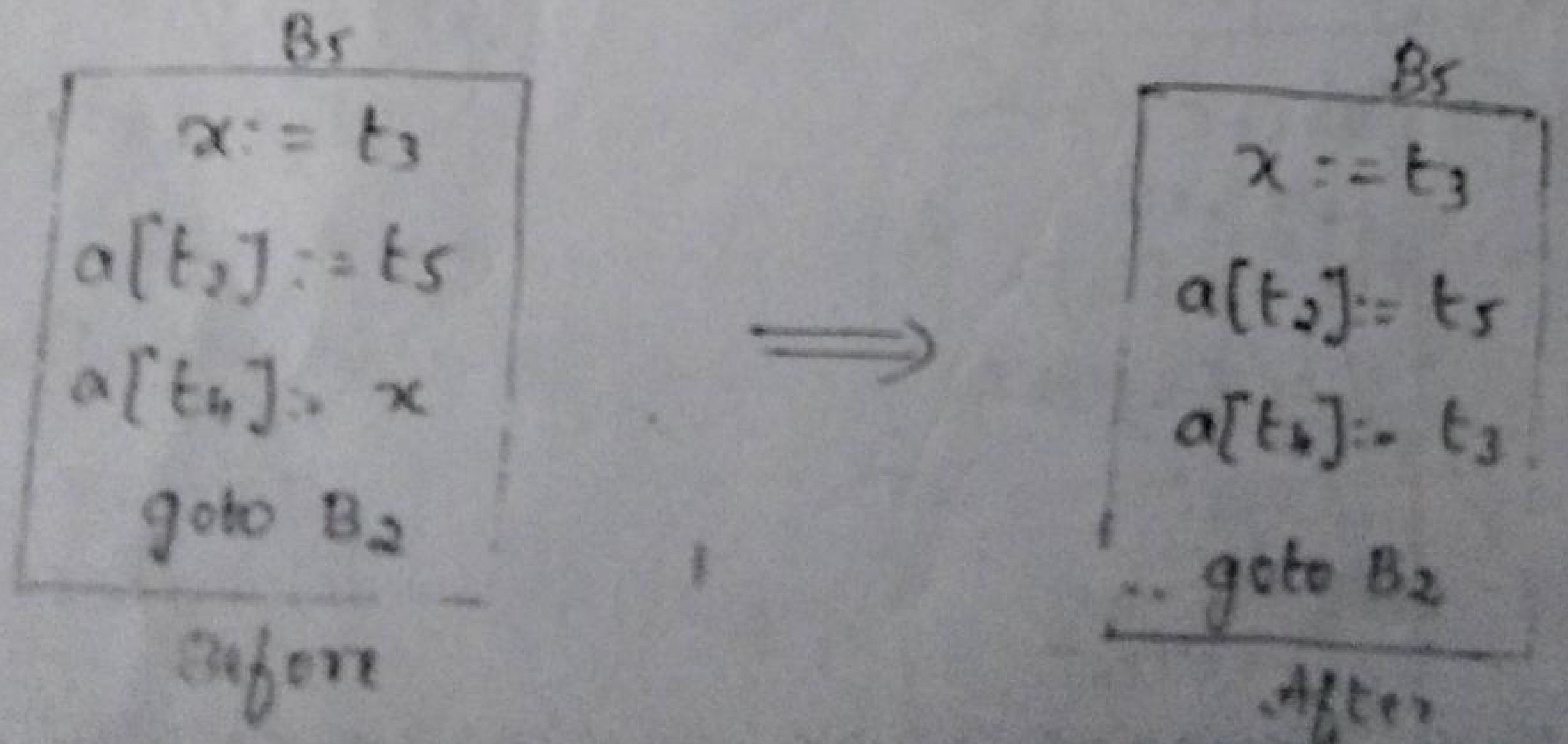


Fig: copies introduced during common subexpression elimination.

× For eg:

In block B_5, the copy propagation can be applied as follows,



∴ Here the assignment $x := t_3$ in block B_5 is a copy statement

→ A variable is said to be live at a point in a prog if it's value can be used subsequently, otherwise it is dead at that point

(ie) :– A statement or variable is dead or useless code if that statement computes some values that never get used

→ So we can eliminate those statement that is never used for the rest of the program.

* Eg:

i) The assignment $x := t_3$ in block B_5 & B_6 is a dead code

After eliminating dead code the block B_5 & B_6 becomes,

(ie)
$x := t_3$
$a[t_2] := t_5$
$a[t_4] := x$
goto B_2

⟹

B_5
$a[t_2] := t_5$
$a[t_4] := t_3$
goto B_2

Similarly in B_6
$t_{14} := a[t_1]$
$a[t_2] := t_{14}$
$a[t_1] := t_3$

ii) Another eg:- a variable which is set to false and is then used in the program for true condition.

(ie) # define ~~flag~~ debug 0

...

if (~~flag~~ debug == 1) { printf (debugging...
~~do something~~ }

– By copy propagation, the flag has been set to zero. So the true block will not be executed atleast once, because the code is dead.

iv) Constant folding :-

→ If the value of the variable is identified as constant during compile time, then it is replaced by the constant instead is called as constant folding.

(ie) - The substitution of values for names whose values are constant is known as constant folding.

* For eg:

```
int i=5;                 int i=5;
 ...          ⟹           ...
k = i+j;                 k = 5+j;
 ...                      ...
```


5.2.2 Loop Optimization :-

→ Much of the compiling time is spent in inner loops.

→ The running time of the program may be improved if we decrease the number of instructions inside the loop(inner loop) by moving the code outside the loop.

* There are three techniques for loop optimization.

i) Code motion

ii) Induction - variable elimination

iii) Reduction in strength.

i) Code motion :- It reduce the amount of code in program

→ Code motion moves the code outside the loop.

(ie) → The instructions inside the loop has been moved outside the loop without affecting the logic. This type of movement is ... code motion

→ This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) & places the expression before the loop.

* for eg:

Eg. i) while (i <= limit − 2)
{
. . . .
}

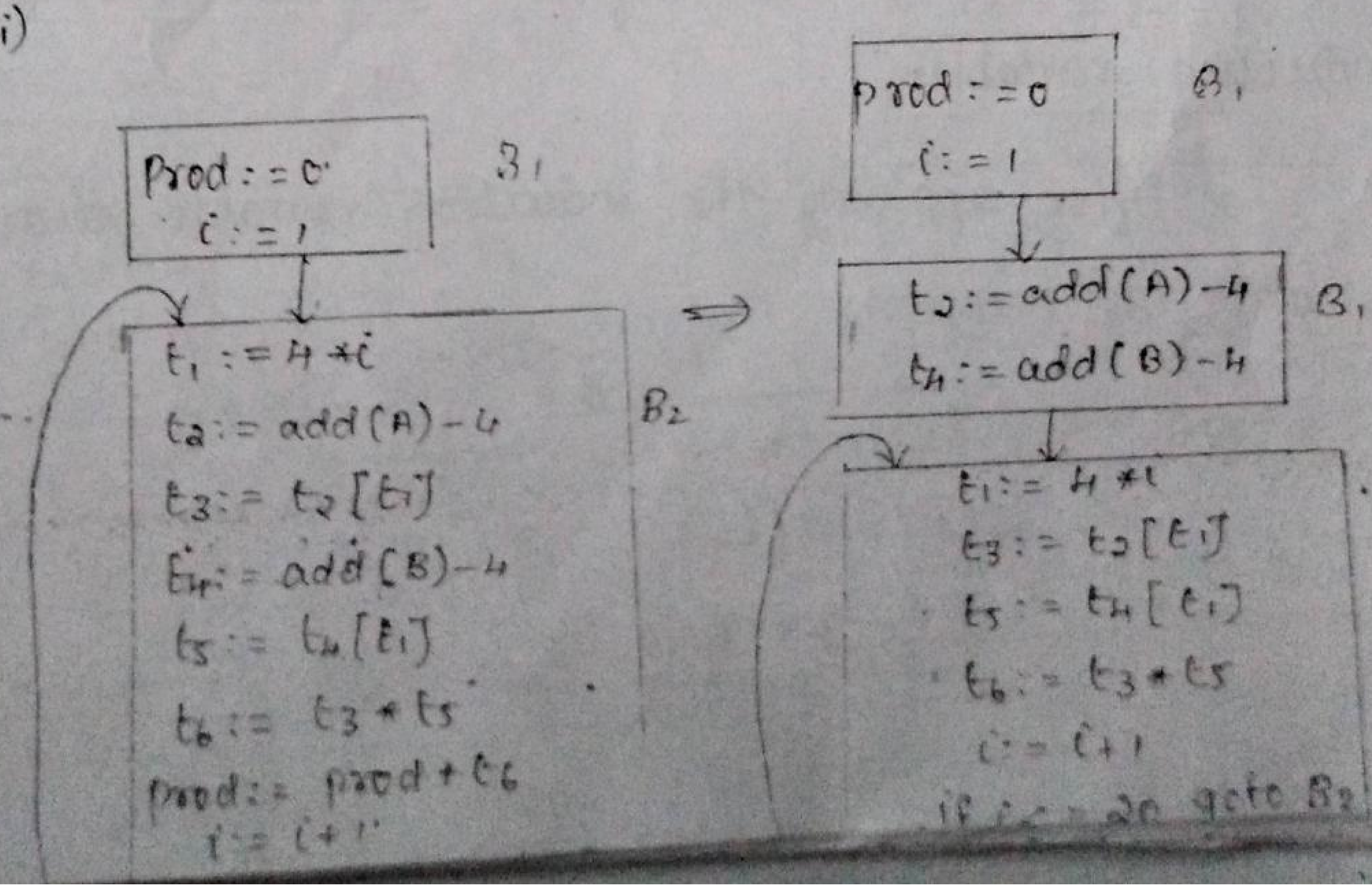
[→ In this while statement, each time the compiler has to compute value of 'limit − 2']

→ Here the evaluation of "limit − 2" is a loop invariant computation. and we can apply code motion.

This code can be changed as.

```
t := limit - 2 ;
while ( i <= t )
{
  . . .
}
```


Eg. ii)



ii) Induction variable elimination:-

→ Two variables are called as induction variables, when there is a change in one variable will correspondingly change the other.

* when there are two or more induction variables in a loop, it is possible to remove it by induction variable elimination.

→ For eg:-
Consider the following block B_3,

```
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
```

B_3 ⇒

- In the above block B_3, note that the values of j and t_4 remains in lock-step, (ie) every time the value of j decrease by 1, that of t_4 decreases by 4

Since $4*j$ is assigned to t_4. Such identifiers are induction variables.

→ After applying the induction variable elimination, the block B_3 becomes,

```
t4 := t4 - 4
t5 := a[t4]
if t5 > v goto B3
```

B_3

Reduction in strength :-

→ This is the way of replacing the most expensive operator by cheap operator

→ Eg:

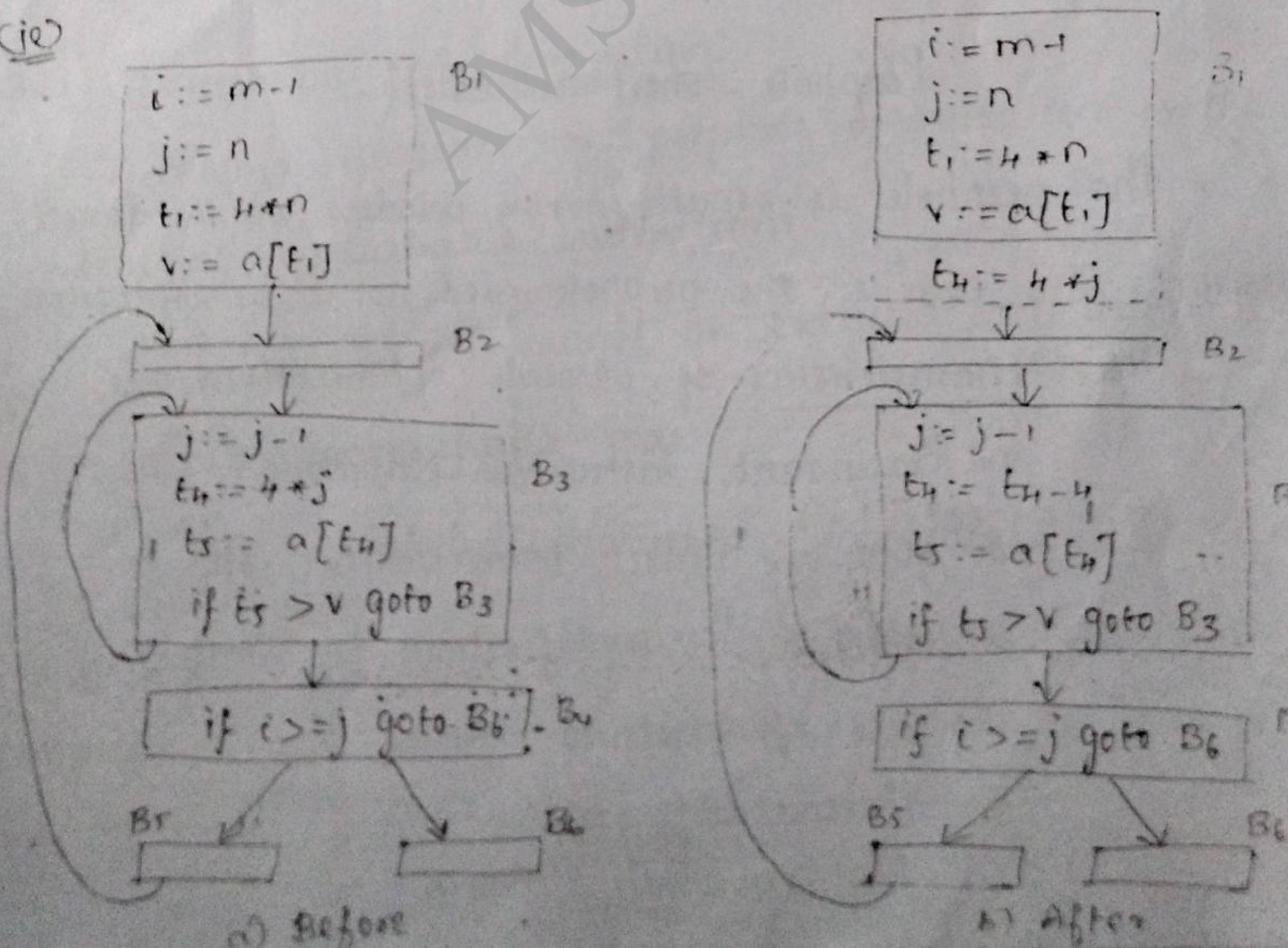
$a ** 2 \Rightarrow a * a$

$a * 2 \Rightarrow a + a$

* Here we consider the block B_3, we cannot get rid of 'j' or 't_4' completely, because 't_4' is used in B_3 and 'j' is used in B_4. The problem is that t_4 doesnot have any value, when we enter the block B_3 for the first time.

— So we place an initialization of t_4 at the end of block B_1 where j is initialized. This is reduction in strength.

(ie)



3 Peephole Optimization.

→ The code generation phase produces the target code of statement-by-statement, which contains redundant instructions and suboptimal constructs.

→ The quality of the target code can be improved by applying optimizing transformations to the target program.

→ A simple but effective technique for locally improving the target code is "peephole optimization".

* Peephole optimization is a method for trying to improve the performance of the target program by examining the short sequence of target instructions & replacing these short instructions by shorter or faster sequence of instructions.

Peephole - short sequence

* The peephole is small, moving window on the target program. The code in the peephole need not to be contiguous.

→ The characteristics of peephole optimizations are,

1. Redundant - instruction elimination
2. flow-of-control optimizations
3. Algebraic simplifications
4. use of machine idioms
5. Unreachable code elimination
6. Reduction in strength.

Redundant - instruction elimination:- (Redundant loads & Stor

→ Here we are going to eliminate the redundant load an Store instruction

→ For eg -

Consider the following instruction sequence,

Mov R_0, a

Mov a, R_0

- In the above sequence, the statement (2) can be deleted, because when (2) is executed, the instruction (1) assure that the value of a is already in register R_0

- we also need to note some conditions, that we couldnot b sure that (1) was always executed immediately before (2) and so we could not remove the (2) instruction.

5.3.2 Unreachable code elimination:-

→ Another function of peephole optimization is the removal of unreachable instructions.

* A code which cannot be executed once during executio is called as unreachable code

* An unlabeled instruction immediately following an unconditional jump may not be executed once. This instruction need to be eliminated or it can be modified into a runnable instruction.

→ Eg:

Consider the code,

```
#define debug 0
---
if (debug)
{
    print debugging information
}
```

- The intermediate representation of the above code is as follows

```
if debug = 1 goto L1
goto L2
L1: print debugging information
L2: ---
```

- Peephole optimization is to eliminate jump over jumps. Thus, no matter what the value of debug. So the three address code can be changed as follows,

```
if debug ≠ 1 goto L2
    print debugging information
L2: ---
```

Now, since debug is set to 0 at the beginning of the program, constant propagation should be replaced by,

```
if 0 ≠ 1 goto L2
    print debugging information
```

Since 0 ≠ 1 is always true, this will become

goto L2

print debugging information

L2:

→ Therefore the statements printing debugging information are unreachable and can be eliminated one at a time.

5.3.3. *Flow-of-control optimizations:*

→ Generally the intermediate code has the following jump statements.

(ie) * jumps to jumps

* jumps to conditional jumps

* Conditional jumps to jumps.

– The above jumps are unnecessary jumps & it can be eliminated in either the intermediate code or the target code as follows,

* Eg:- if we have the jump sequence shown below

i) goto L1

...

L1 : goto L2

– This code sequence can be eliminated as follows:

goto L2

L1 : goto L2

possible to eliminate the statement L_1 : goto L_2, provid-

it is preceded by an unconditional jump:

- Similarly the sequence,

ii)

if a<b goto L_1

....

L_1 : goto L_2

can be replaced by,

if a<b goto L_2

....

L_1 : goto L_2

Finally, suppose there is only one jump to L_1 and L_1 is preceded by an unconditional goto. Then the sequence is,

iii)

goto L_1

...

L_1 : if a<b goto L_2

L_3 :

may be replaced by,

if a<b goto L_2

goto L_3

...

L_3 : ___

5.3.4 Algebraic Simplification:

→ The algebraic identities occur in the three-address c[ode] can be simplified without changing the meaning of the c[ode].

→ There is no end to the amount of algebraic simplification that can be attempted through peephole optimization.

- However, only a few algebraic identities occur frequently enough that it is worth considering implementing th[em].

* For eg: Consider the statement,

$x := x + 0$

$x := x * 1$

- These statements are straight forward and can be eliminated through peephole optimization

5.3.5 Reduction in Strength:

→ This replaces expensive operations by equivalent cheaper operations on the target machine.

(ie) Exponential operator → **
Multiplication operator → *
Additional operator → +

↑ cost

→ Eg:
i) x^2 is replaced by $x * x$
ii) $x * 2$ is replaced by $x + x$
iii) Fixed-point multiplication or division is replaced as a shift operation

5.3.6. Use of machine idioms:

→ Some operations are replaced by hardware instru[ctions] which implements efficiently. Usuage of hardware instruction reduces execution time.

→ Some machines have auto-increment and auto-decrement addressing modes. Use of these modes greatly improves the quality of the code when pushing or popping a stack.

* Eg: $i := i + 1$

- This can be replaced by auto-increment instruction "INC R_1".

→ These are about the peephole optimizations.

.4 Optimization of Basic Blocks

→ The optimization or transformations performed using basic blocks can be easily implemented using directed acyclic graph (DAG).

* The DAG is a pictorial way of representing the three-address code, where internal nodes represent the computations & leaf-node represent the variables.

→ Many of the structure preserving transformations, such as common subexpression elimination, dead-code elimination & algebraic transformations such as reduction in strength can be implemented by constructing a DAG for a basic block.

Common subexpression elimination:-

→ There is a node in the DAG, for each node represent of the initial values of the variables in the basic block and also there is a node n associated with each statements of the block.

→ The children of that node 'n', represent the statements prior to S. The nodes whose values are live on exit are called the output nodes.

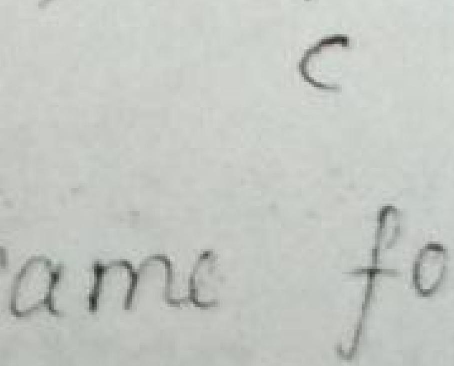
* Here the optimization is performed using common subexpressions. Whenever a new node x is to added, check whether there is any node y with the same children in the same order and also with the same operator. If it is there, then y uses the value of x.

→ Eg: A DAG for the block,

a := b+c
b := a-d
c := b+c
d := a-d

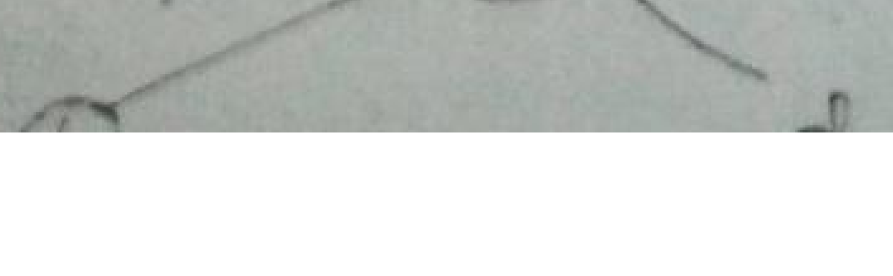
- First create the tree for first statement,

i)



- Repeat the same for the remaining statements

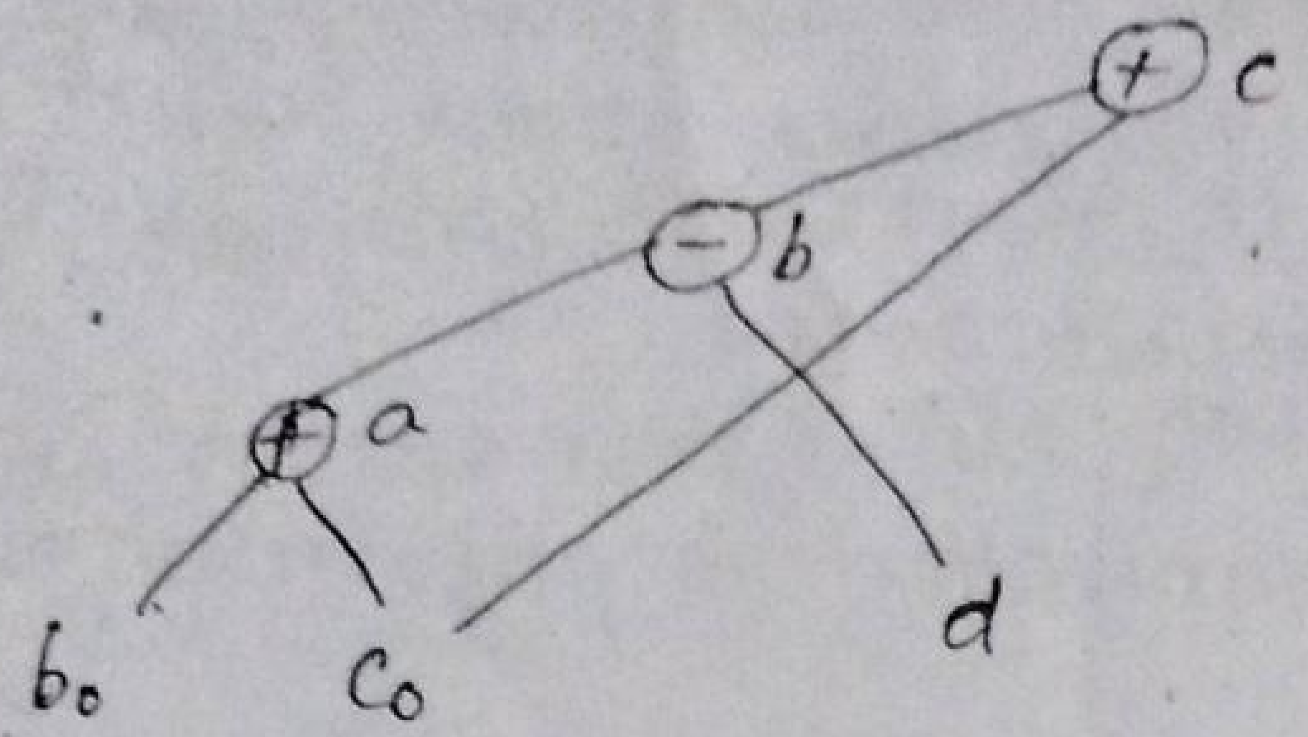
ii)



Since 'a' is already there, use that node and append new tree to it where the operator '-' is labeled as 'b' by making the already available b as b_0.

→ For 3rd statement, do the following.

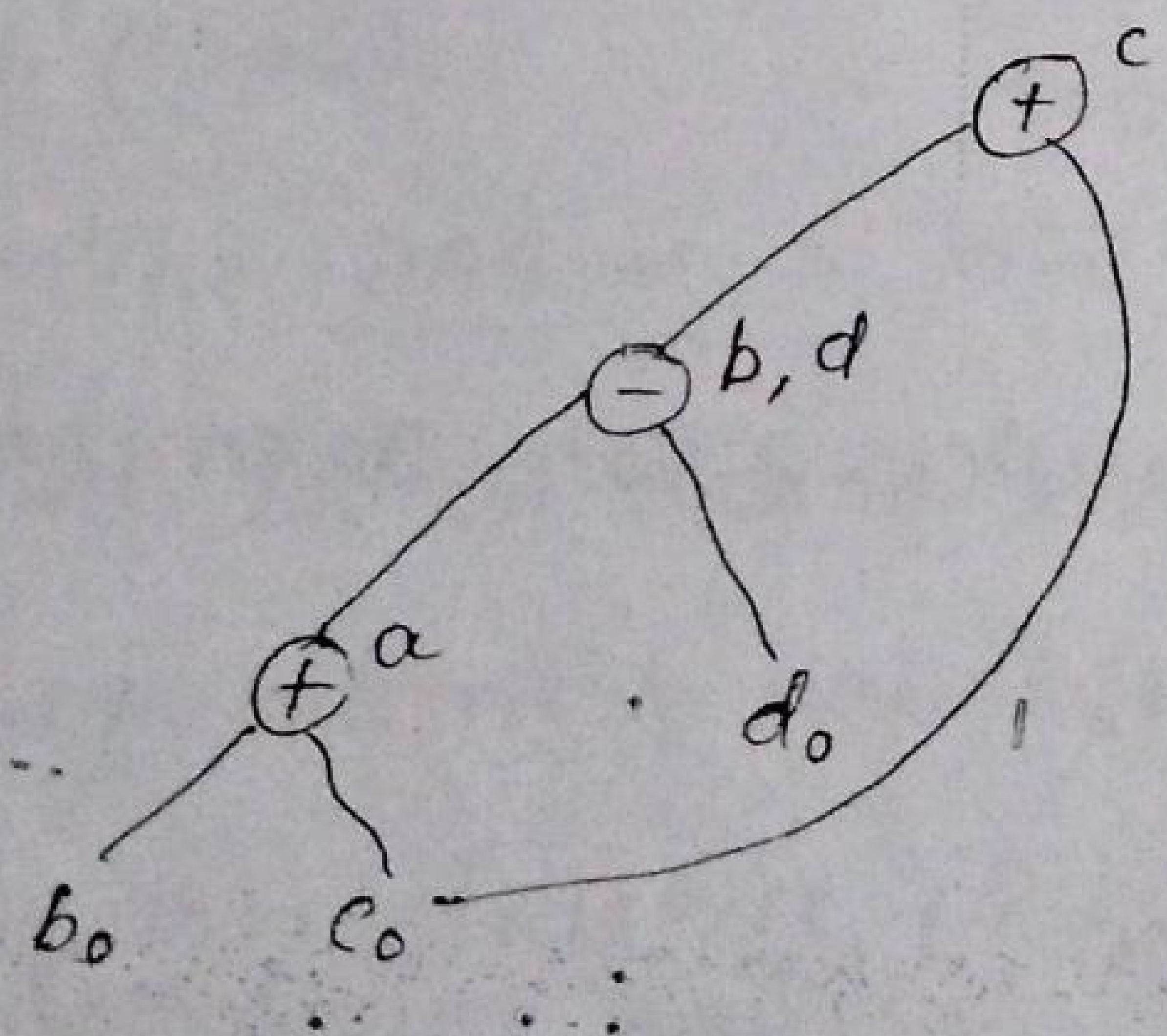
iii)



iv) For the last statement, the operands are a and d with the operator '-', while checking, the same tree is already there with label b.

- It is identified as common subexpression. So no need to create a new tree, just append label d to the node b.

* The dag for the above basic block is,



- From the above DAG, it is understood that there is a block with only three statements. If b is not live on exit, then we have the statements, a := b+c

d := a-d

c := d+c.

- If both b and d are live on exit, then a fourth statement must be used to copy the value from one to the other.

ii) <u>Dead code elimination</u>:

→ In order to <u>delete the dead code</u>,

1) delete the root node (node with no ancestors) from a DAG that has no live variables

2) Do the step(1) repeatedly till it removes all nodes from the DAG.

→ Eg: Now consider the following three-address code sequence

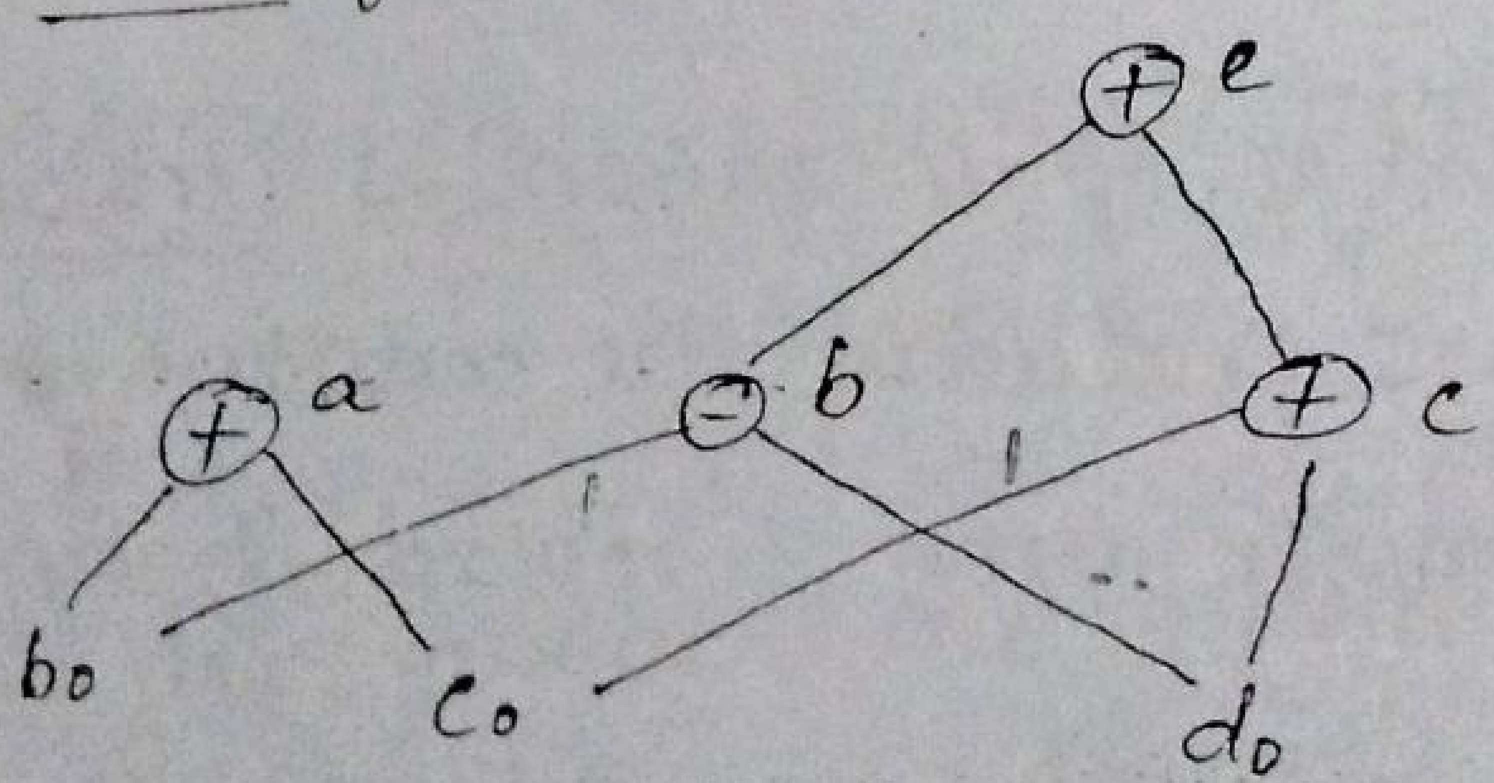
a := b+c

b := b-d

c := c+d

e := b+c

* The <u>DAG</u> for the above sequence is,



- Here if 'a' is a deadcode then delete a node 'a'

5.4.1. Use of algebraic identities:-

→ A simple algebraic transformations for optimiza[tion] is on algebraic identities.

→ Algebraic identities is an important optimization on basic blocks.

i) Some optimizations on algebraic identities are,

$x+0 = 0+x = x$

$x-0 = x$

$x*1 = 1*x = x$

$x/1 = x$

ii) Another class of algebraic optimization that makes reduction in strength

↳ using this replacing a more expensive operator by cheaper ones.

(ie) $x ** 2 = x * x$

$2.0 * x = x + x$

$x / 2 = x * 0.5$

iii) The third class of optimization is constant folding.

- Here we evaluate the constant expressions at compile time & replace it by their values.

* Eg: The expression, pi = 3.14

area = pi * r * r

is replaced as,

area = 3.14 * r * r

While constructing DAG, algebraic transformations such as commutativity and associativity have been applied. When a new node is created, make a check whether such node already exists with left child m & right child n.

iv) Associative laws are also applied to identify the common subexpressions.

Eg: a := b+c
e := c + d + b

- The above code sequence can be written as,

a := b+c
t := c+d
e := t + b

- For DAG, there is no need of temporary variables.

∴ the code can be rewritten as,

a := b+c
e := a + d

v) The conditions "x > y" can also be tested by subtracting the arguments and perform test on the code by the subtraction.

→ These are about the optimization of Basic blocks. //

- some flow graphs are cannot be reducible. That flow graphs are non-reducible.

eg:-

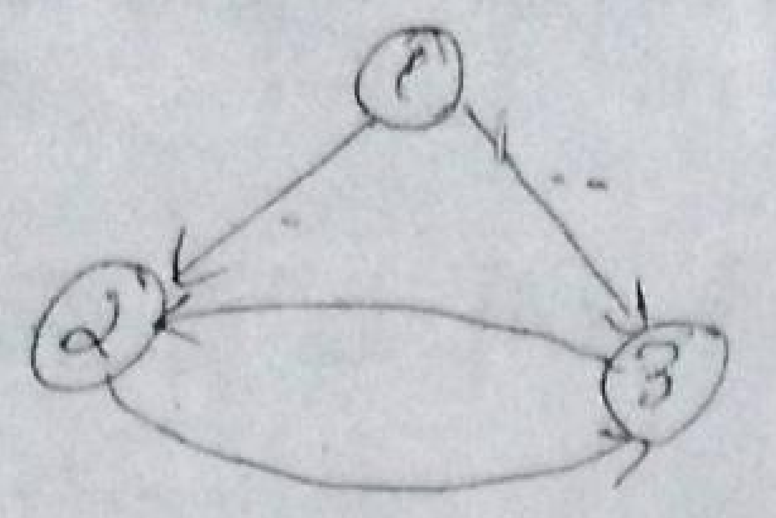


fig: Non-Reducible flow gr

→ These are about the loops in flow graph.

Introduction to Global Data-Flow Analysis

→ Instead of performing the code optimization in a single block, the compiler is designed in such a way, to collect the information about the whole prog. and then distribute that information to each block in a flow graph.

* The process of collecting data-flow information which are useful for the purpose of optimization is called data-flow analysis.

→ The current value of the variable can be identified using data-flow analysis.

→ The data-flow information can be collected by setting up and solving systems of equations that information at various points in a program ... -ible

* The data-flow equation is,

$out[S] = gen[S] \cup (in[S] - kill[S])$

- This equation can be read as

" the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement".

→ This equation is called as data-flow equations

→ The data-flow equations can be solved with 3 factors.

(ie) i) The information about generating and killing depends on the data-flow analysis

ii) Data-flow analysis is affected by the control constructs in a program because data flows along control paths.

iii) Some changes will be there while going through statements like procedure calls, assignments through pointer variables and assignments to array variables

→ while performing global data-flow analysis, there is a change of encountering several points & paths.

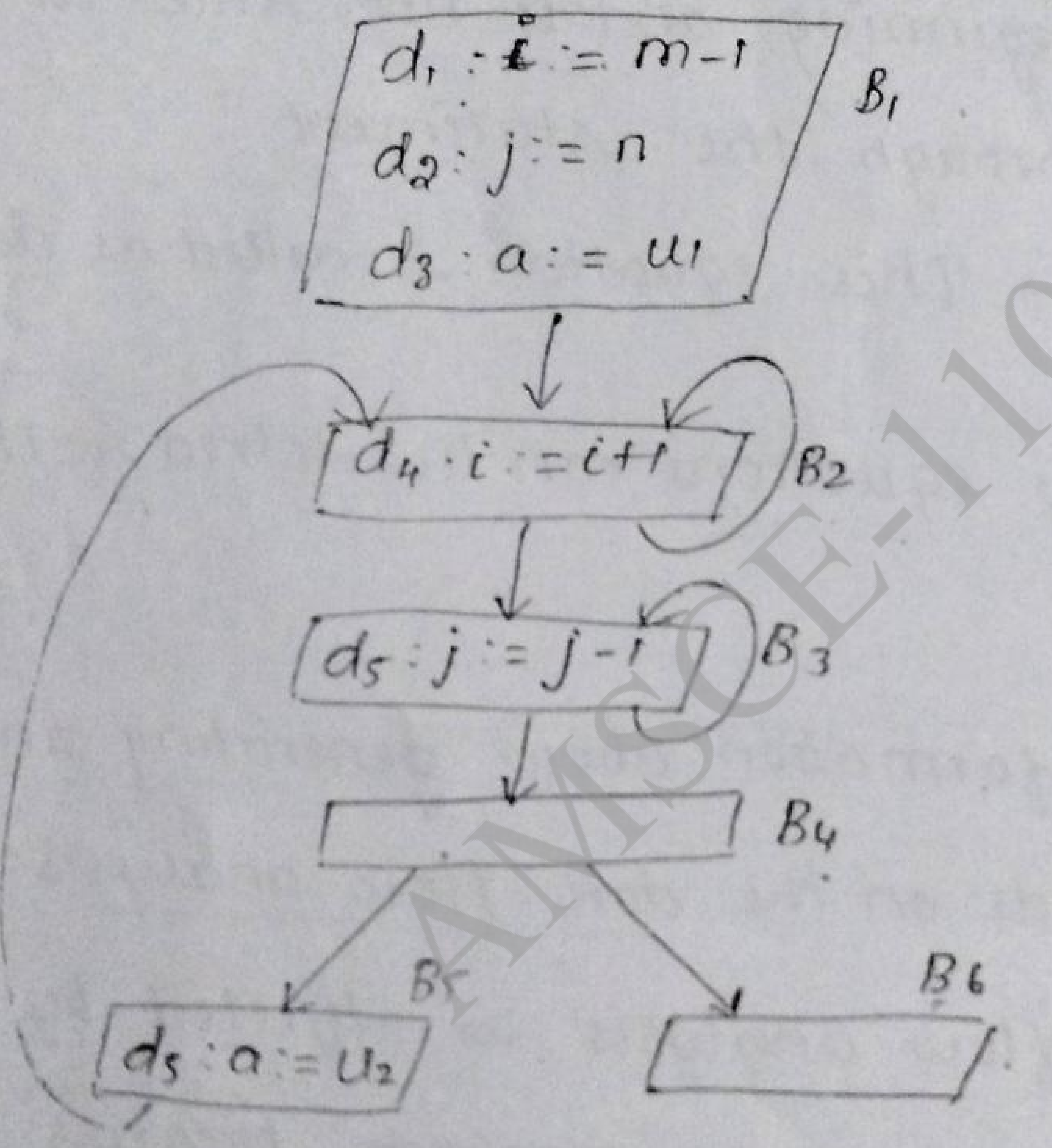
5.6.1. Points and paths:

→ A point is the instance at which the state of a variable is defined.

→ In a basic block, a point is defined between two adjacent statements and also before the first statement and after the last.

---

* Eg: Consider the flow graph,



d_1, d_2, d_3, d_4, d_5 → definition

fig: A flow graph

– Here the block B_1 contains four points: one before any of the assignments (ie. start of the assignment), and one after each of three assignments (ie. one after $i := m-1$ and one after $j := n$, and another one after $a := u_1$).

→ The global view can be identified by considering all the points in all the blocks.

A path is defined as a way from P_1 to P_n is a sequence of points $p_1, p_2, \ldots p_n$ such that for each i between 1 and n-1, either

i) P_i is the point immediately preceding a statement and P_{i+1} is the point immediately following that statement in the same block, or

ii) P_i is the end of some block and P_{i+1} is the beginning of a successor block.

5.6.2. Reaching Definitions

→ A definition of a variable x is a statement that assigns or may assign a value to x.

* Eg: $x := 4$

→ The definitions can be classified into two types.

i) Ambiguous definition

ii) Unambiguous definition

i) Ambiguous definition:

→ ambiguous definitions are defining the same values in more variables. note 2

→ It can be defined as follows,

* A call for a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure

✓ Eg:

- A parameter x passed from the main procedure is also defined formally in the called procedure with some other name.

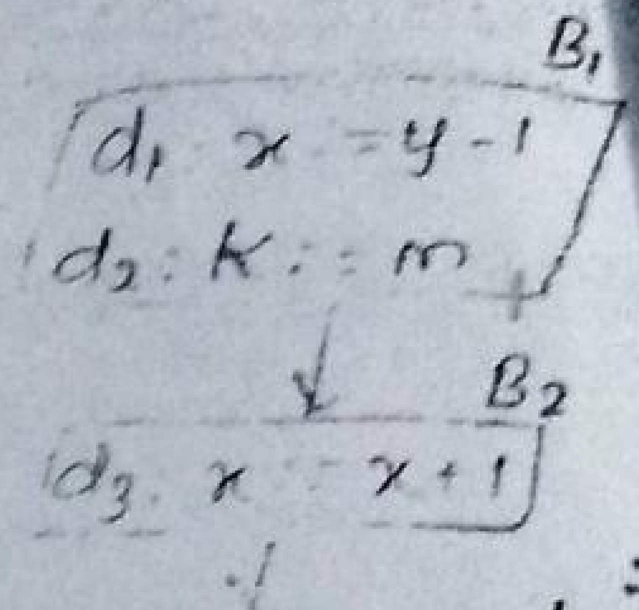


Fig: Flowgraph

(ie) The definition of $x := y-1$ and $k := m$ in block B_1 reach the beginning of block B_2. But the definition of $x := x+1$ in block B_2 kills the definition of x in block B_1.

* An assignment statement through a pointer that could refer to x.

✓ Eg:

the assignment $*q := y$ is a definition of x if it is possible that q points to x.

→ A definition 'd' reaches a point p if there is a path from the point immediately following 'd' to 'p' such that d is not killed along that path.

5.6.3 Data-flow analysis of structured programs:-

→ Flow graphs for control-flow constructs such as do-while statements have a useful property.

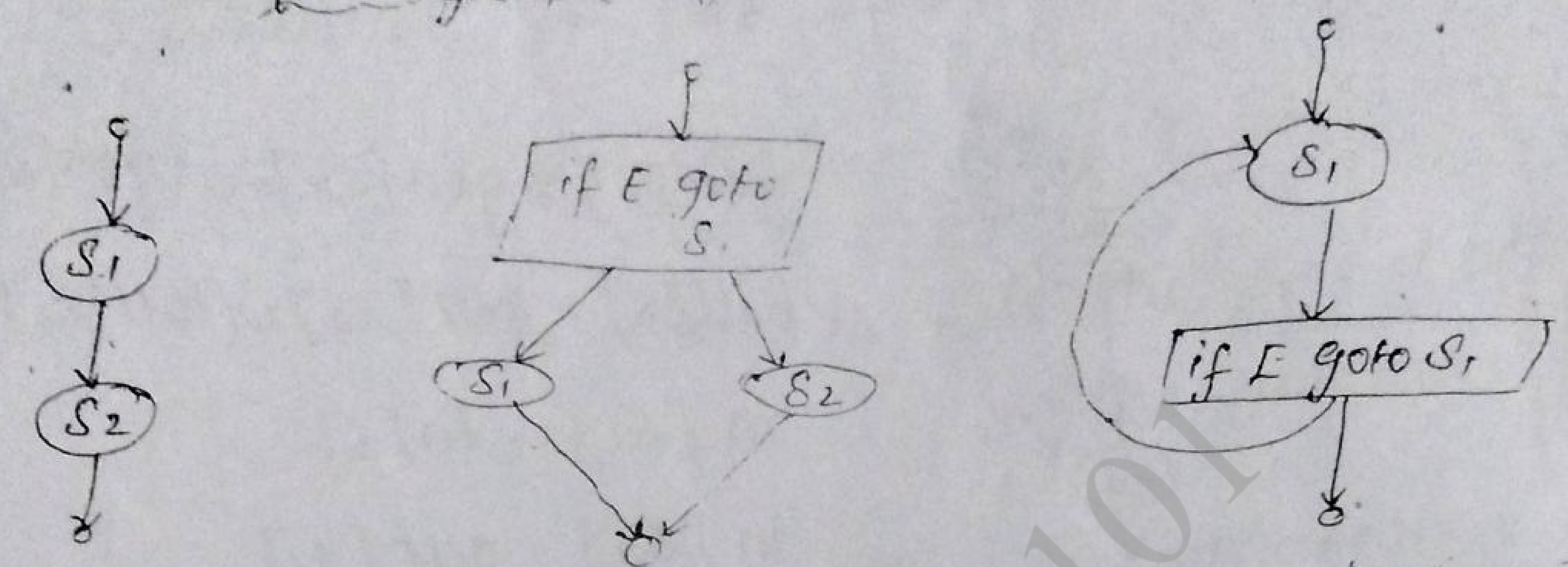
(ie) There is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over.

- Some of the structured flow-control constructs exploit this property with the following syntax.

(i.e.) $S \rightarrow id := E \mid S ; S \mid$ if E then S else $S \mid$ do S while E

$E \rightarrow id + id \mid id$

- The flowgraph for the above statements,



$S_1 ; S_2$ if E then S_1 else S_2 do S_1 while E

Fig Some structured control construct

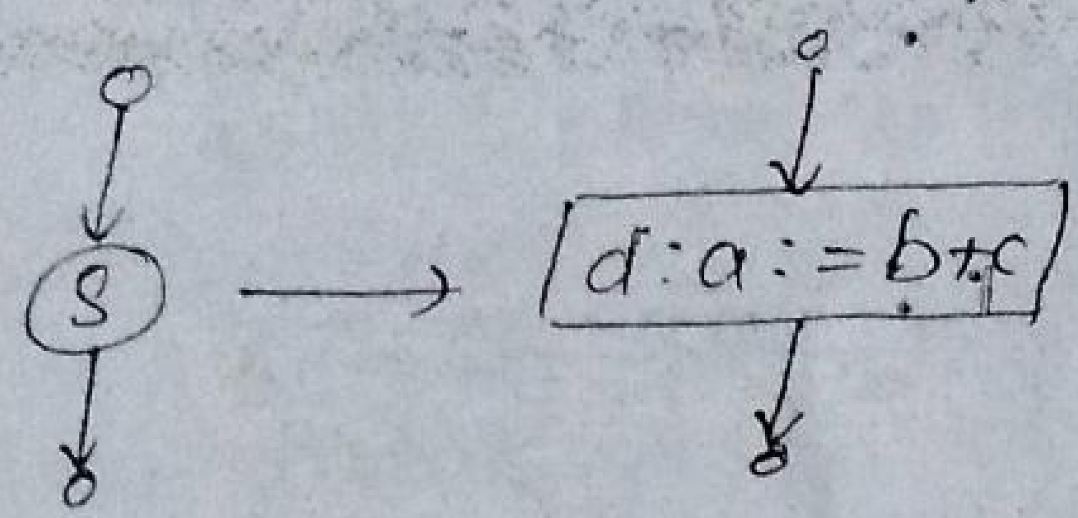
→ A set of nodes N in a portion of a flow graph that includes a header, which dominates all other nodes in the portion is called a region.

* Dummy blocks with no statements are indicated by open circles.

* The beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points respectively of the statements

→ The data-flow equations for the above control and flow constructs are as follows.

i)

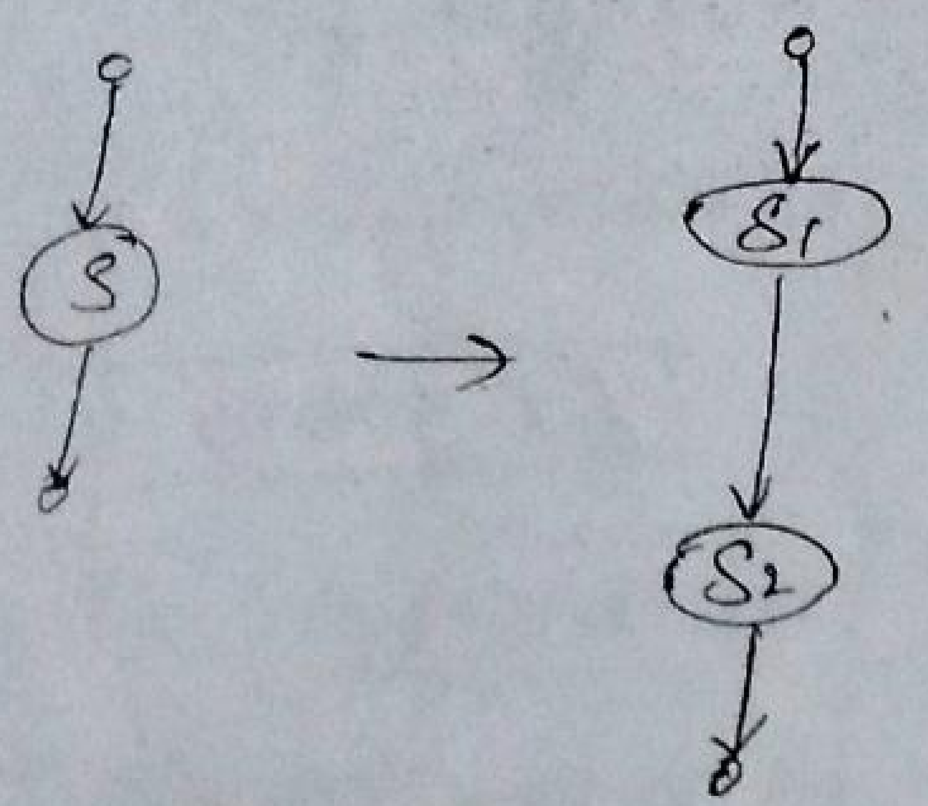


$gen[S] = \{d\}$

$kill[S] = D_a - \{d\}$

$out[S] = gen[S] \cup (in[S] - kill[S])$

ii)



$gen[S] = gen[S_2] \cup (gen[S_1] - kill[S_2])$

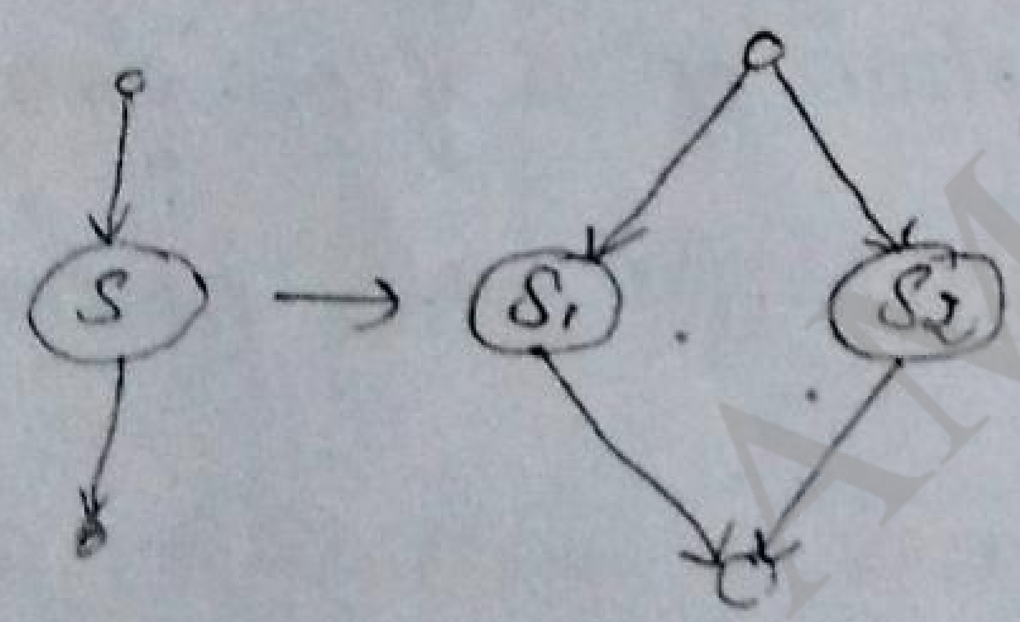
$kill[S] = kill[S_2] \cup (kill[S_1] - gen[S_2])$

$in[S_1] = in[S]$

$in[S_2] = out[S_1]$

$out[S] = out[S_2]$

iii)



$gen[S] = gen[S_1] \cup gen[S_2]$

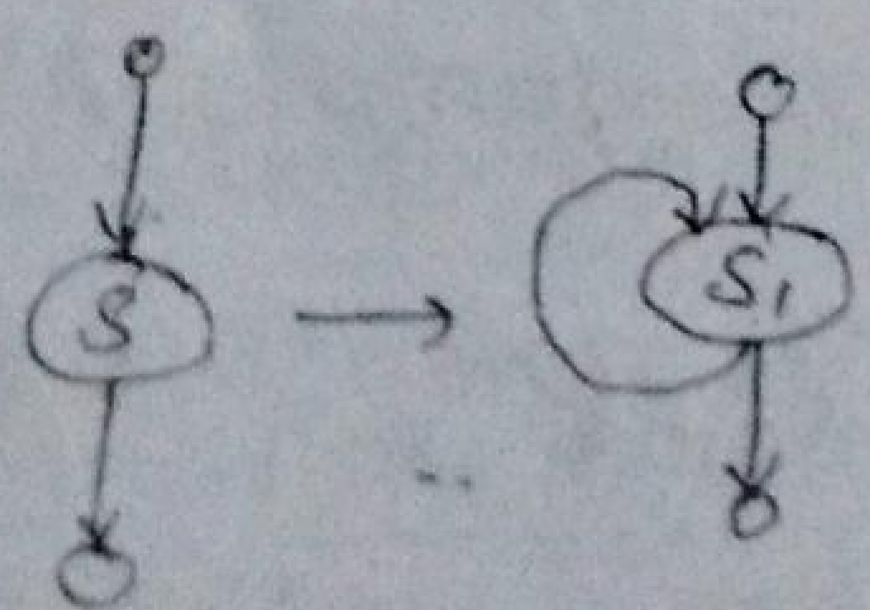
$kill[S] = kill[S_1] \cap kill[S_2]$

$in[S_1] = in[S]$

$in[S_2] = in[S]$

$out[S] = out[S_1] \cup out[S_2]$

iv)



$gen[S] = gen[S_1]$

$kill[S] = kill[S_1]$

$in[S_1] = in[S] \cup gen[S_1]$

$out[S] = out[S_1]$

Fig: Data-flow equation for reaching definition

2. For the

→ The sets gen[S] and kill[S] are synthesized attribute and they are computed bottom up from smallest to large.

* For the definition d: a = b+c; it reaches the end of the statement

(ie) gen[S] = {d}

- Likewise, 'd' kills all other definitions of a

(ie) kill [S] = Da - {d}

↳ where Da → is the set of all definition in the program for variable

5.6.4 Estimation of Data-flow Information

→ Any graph theoretic path in the flow diagram is called as an execution path.

(ie) Execution path is executed when the program is run with atleast one possible input.

→ When we compare the computed 'gen' with the "true gen", we can find that 'true gen' is always a subset of the computed gen. The "true kill" is always a superset of computed kill.

* For eg:

- if the expression E in "do S while E" statement can never be false, then we can never get out of the loop

- Thus the true gen is ϕ and every definition killed by the loop. In the case of reaching definition we normally use the definition information to infer that the value of a variable 'x' at a point is limited to some small number of possibilities.

5.6.5. Computation of in and out :-

→ Data-flow problems can be solved by computing the synthesized attributes "gen" and "kill".

→ There are other kinds of data-flow information such as reaching definition, where we need to compute inherited attributes.

* in[S] is the inherited attribute, and
* out[S] is a synthesized attribute depending on in.

✓ in[S] be the set of definitions reaching the beginning of S.

✓ out[S] is the set of definitions that reach the end of S with following the path outside S.

✓ gen[S] is the set of definitions that reach the end of S without following path outside S.

→ After computing gen[S] and kill[S] bottom-up for all the statements 'S', we may compute 'in' and 'out'

...ing at the statement having that

Representing the complete program.

(ix) $in[S_0] = \phi$, if S_0 is the complete program.

- that is, no definitions reach the beginning of the program.

* For eg:- the data-flow eqn.

$out[S] = gen[S] \cup (in[S] - kill[S])$

- means that, a definition reaches the end of S if either it is generated by S or it reaches the beginning of the statement & is not killed by the statement

5.6.6 Dealing with loops

→ Consider the following loop.

- In this case, we cannot simply use in[S] as in[S₁] because the definitions inside [S₁] that reach the end of S₁ are able to follow the arc back to the beginning of S₁ & these definitions S₁ that reach the end of S₁ also in $in[S_1]$. So we have,

$$in[S_1] = in[S] \cup out[S_1]$$

Similarly we have,

$out[S] = out[S_1]$

– The equations that define a recurrence for $in[S_1]$ and $out[S_1]$ is,

$$I = J \cup O$$
$$O = G \cup (I - K)$$

where,
$I = in[S_1]$
$O = out[S_1]$
$G = gen[S_1]$
$K = kill[S_1]$
$J = in[S_1]$

5.6.7. Representation of Sets:-

→ The set of definitions such as gen[S] and kill[S] can be represented using bit vectors.

* The bit-vector representation for sets also allows the set operations to be effectively implemented.

* We assign a number to each definition in the flow graph. Then the bit vector representing the set of definitions will have 1 in position 'i' if and only if the definition number 'i' is in the set

– The number of a definition statement can be taken as the index of the statement in an array ... pointers to statements.

2. For the instruction ST

by "logical or" and "logical and".

→ The difference A-B of sets A and B can be implemented by taking the compliment of B & then using "logical and" to compute $A \wedge \neg B$, [A intersect Negation B'].

5.6.8. Use-Definition chains:-

→ It is easy to store the reaching definition information as "Use-definition chains" or "Ud-chains" which are lists for each use of a variable, of all the definitions that reach that use.

* If a use of variable 'a' in block B is preceded by no unambiguous definition of 'a' then the Ud-chain for that use of 'a' is the set of definitions of 'a' within in[B] that are definitions of a.

→ These are about the Global Data-flow analysis

5.7. Code Improving Transformations:-

→ The code improving transformations rely on data-flow information. For improving the code, we consider two transformation.

(ie) 1. Function preserving transformation

2. Loop optimization.

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements
a : = b [i] and **a [i] : = b**

Statements	Code Generated	Cost
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments
a : = *p and ***p : = a**

Statements	Code Generated	Cost
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */

x : = y +z
if x < 0 goto z
	MOV y, R_0
	ADD z, R_0
	MOV R_0,x
	CJ< z

THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
  1. Leaves are labeled by unique identifiers, either variable names or constants.
  2. Interior nodes are labeled by an operator symbol.
  3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node n will be node(y).

Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached

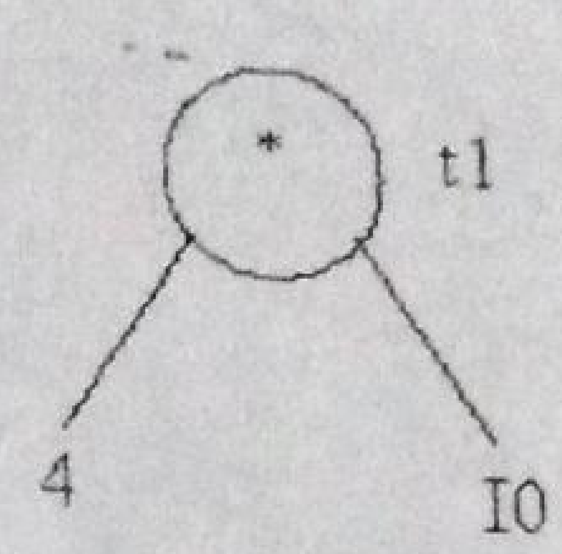
identifiers for the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements:

1. t_1 := 4* i
2. t_2 := a[t_1]
3. t_3 := 4* i
4. t_4 := b[t_3]
5. t_5 := t_2*t_4
6. t_6 := prod+t_5
7. prod := t_6
8. t_7 := i+1
9. i := t_7
10. if i<=20 goto (1)

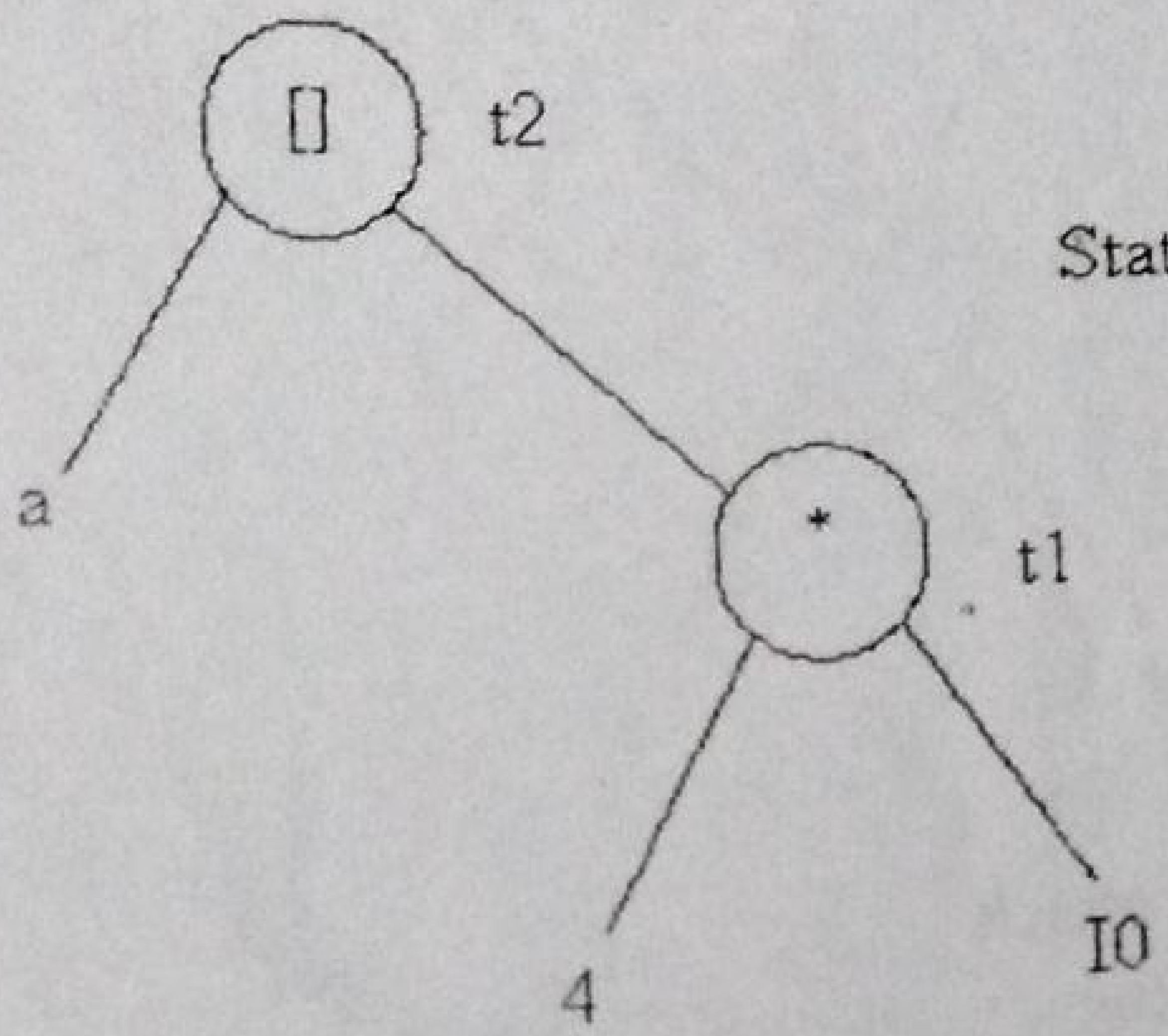
Stages in DAG Construction

(a)



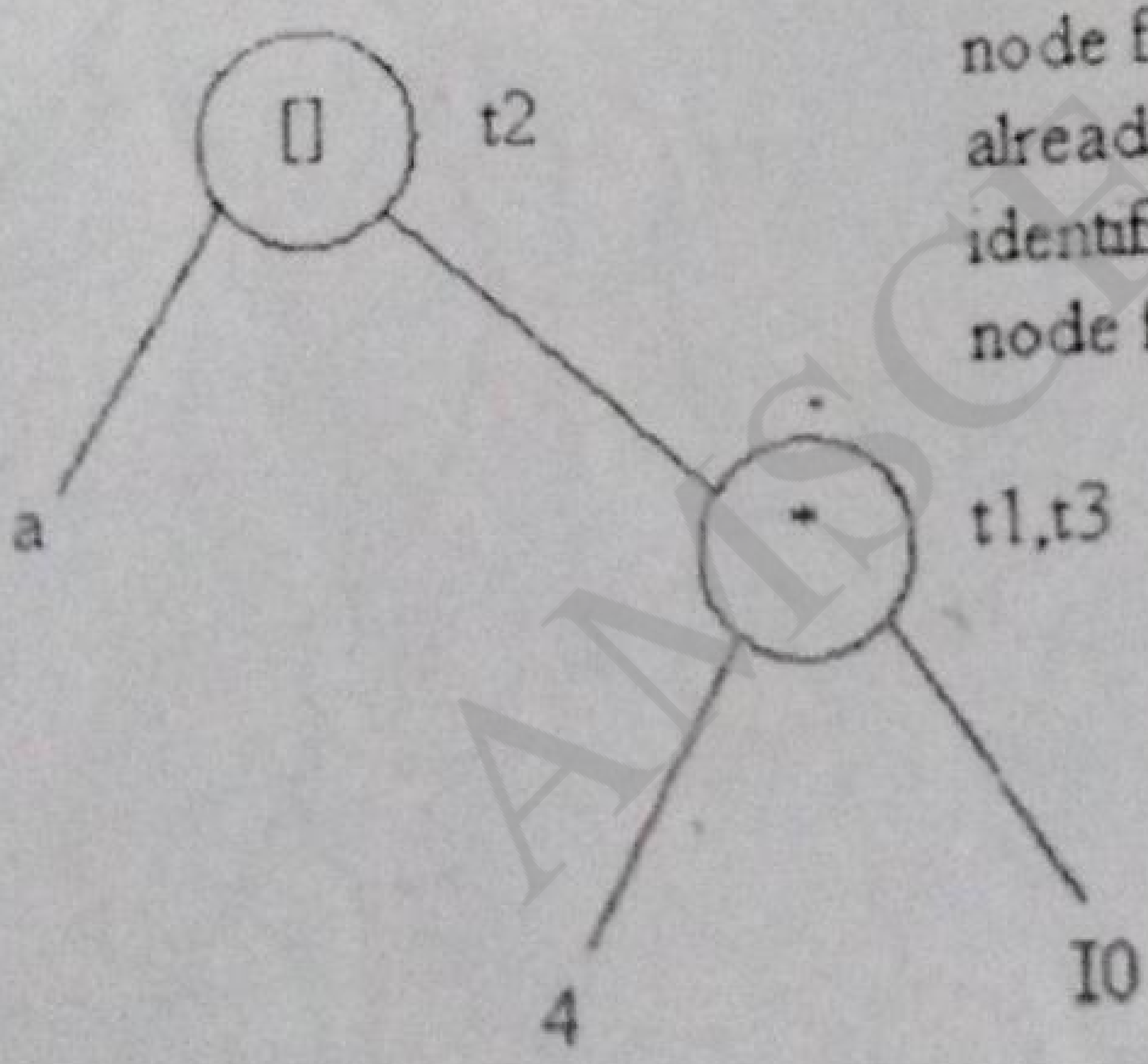
Statement (1)

(b)



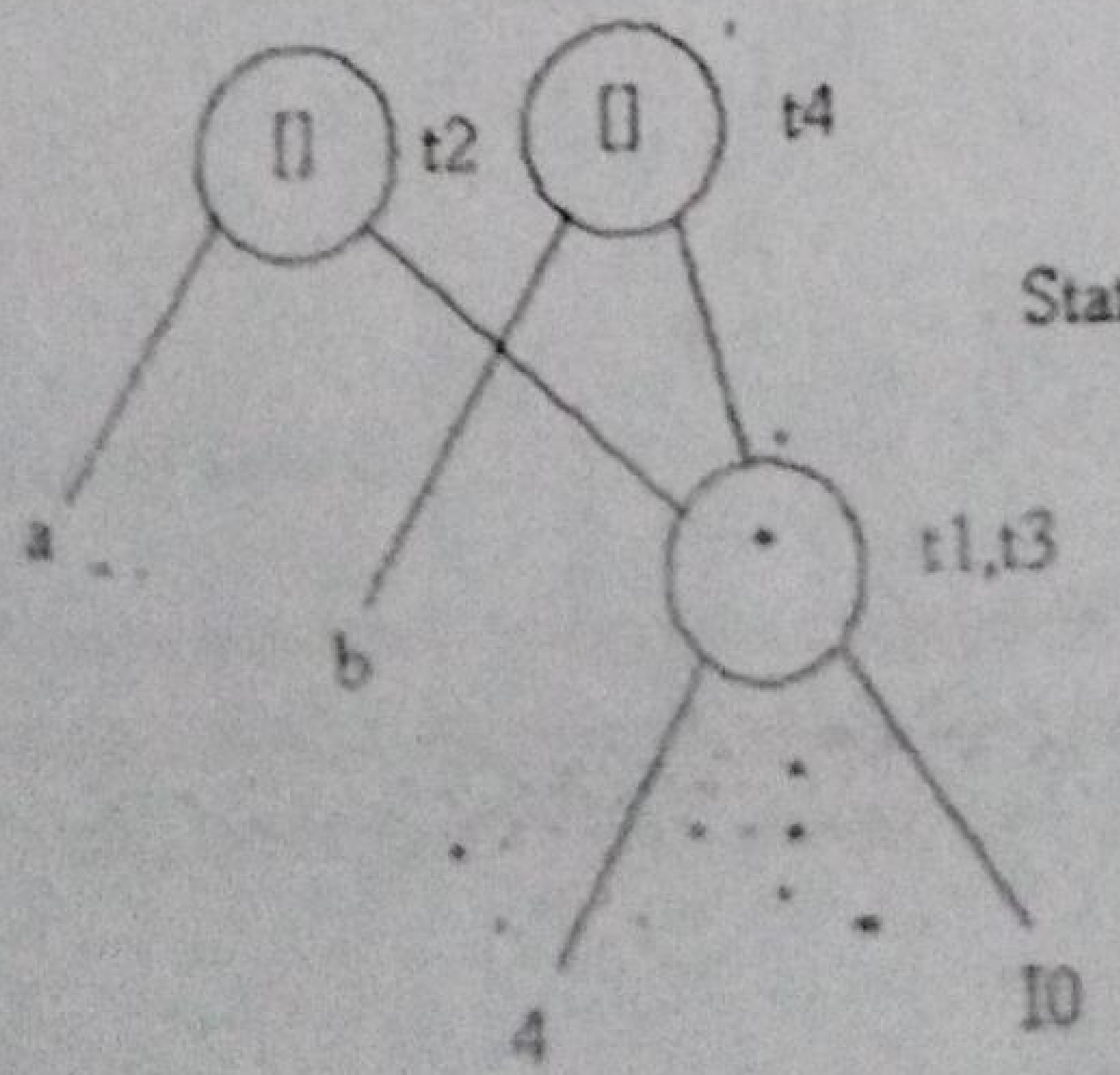
Statement (2)

(c)



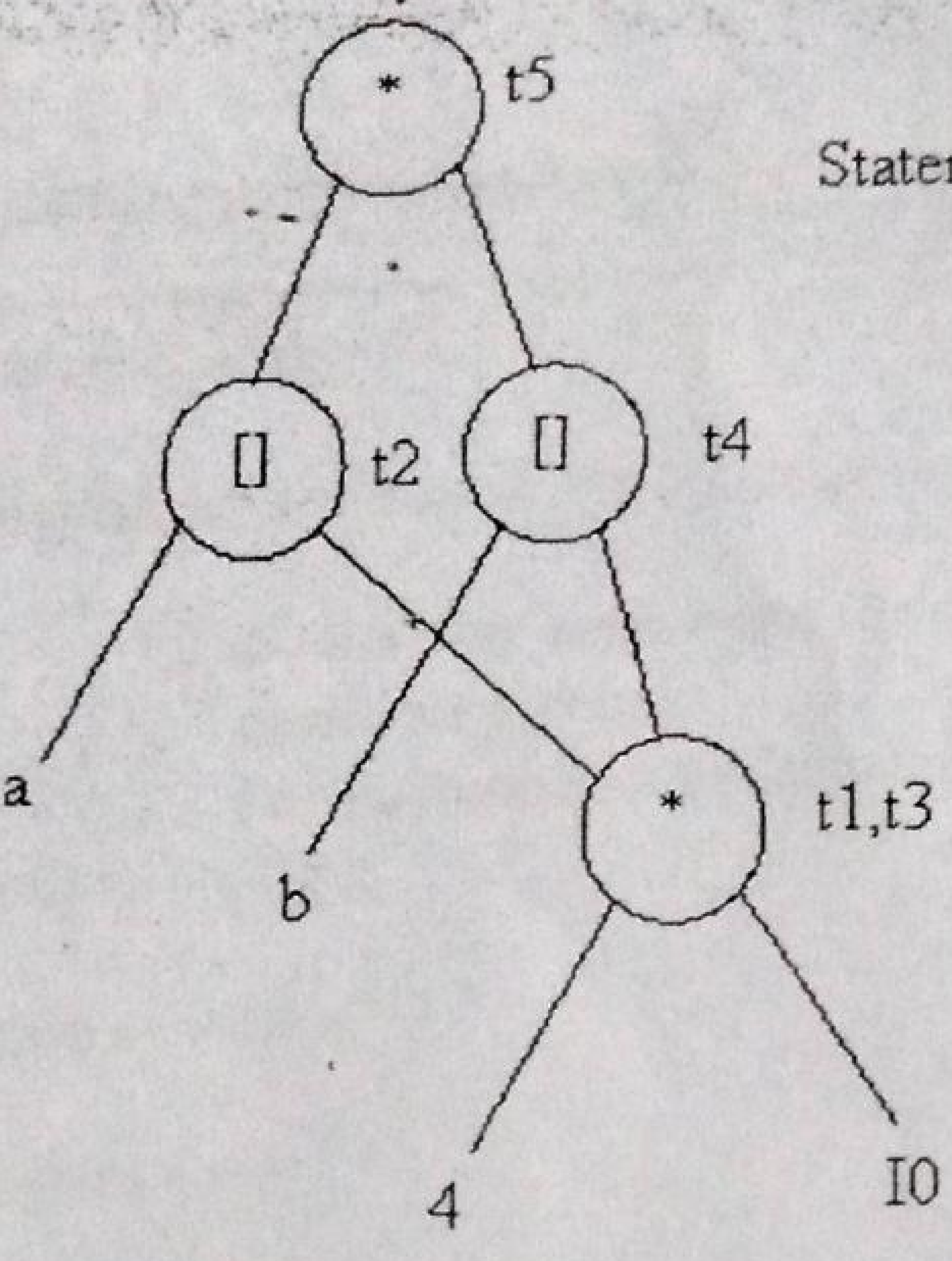
node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

(d)



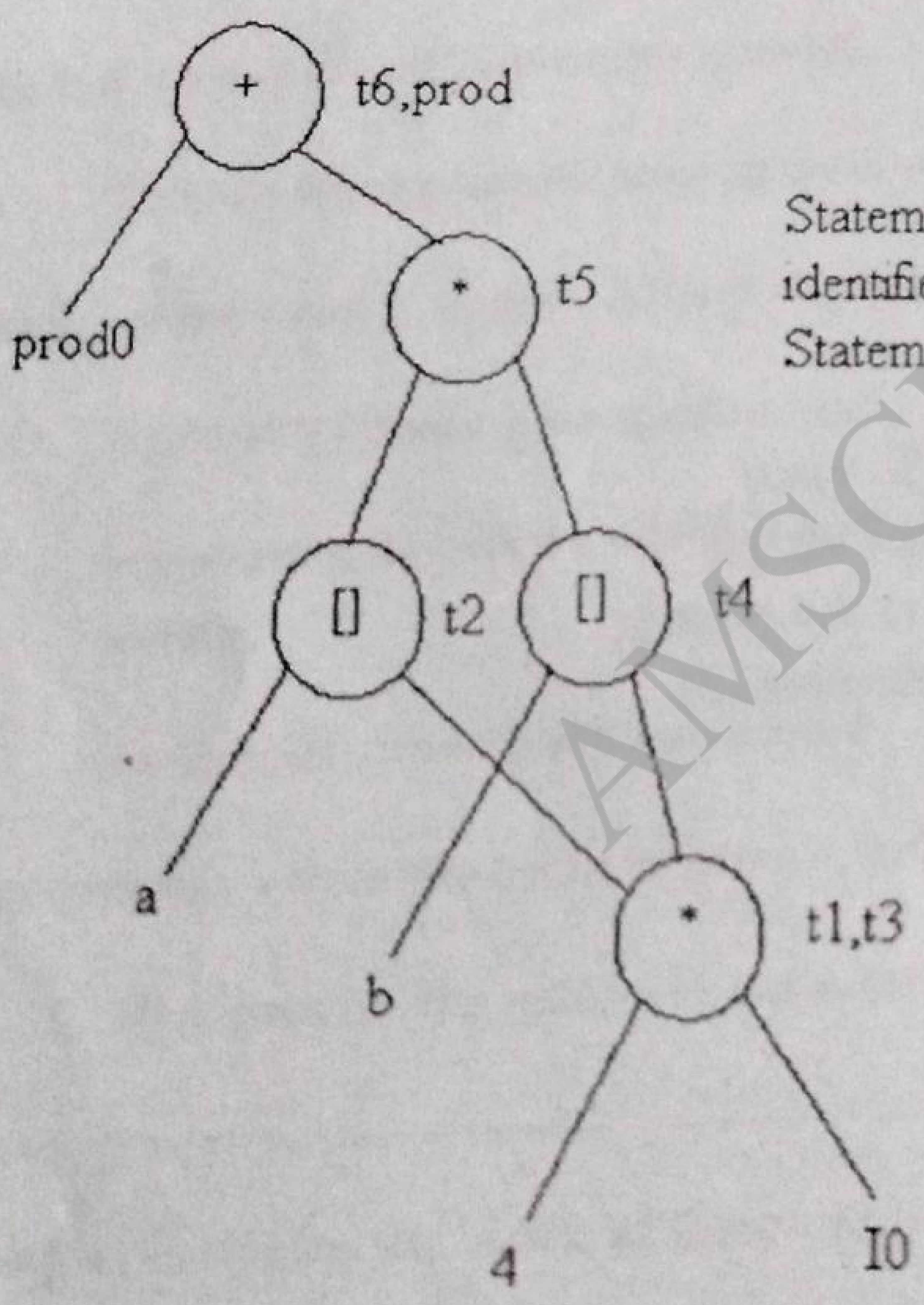
Statement (4)

(e)



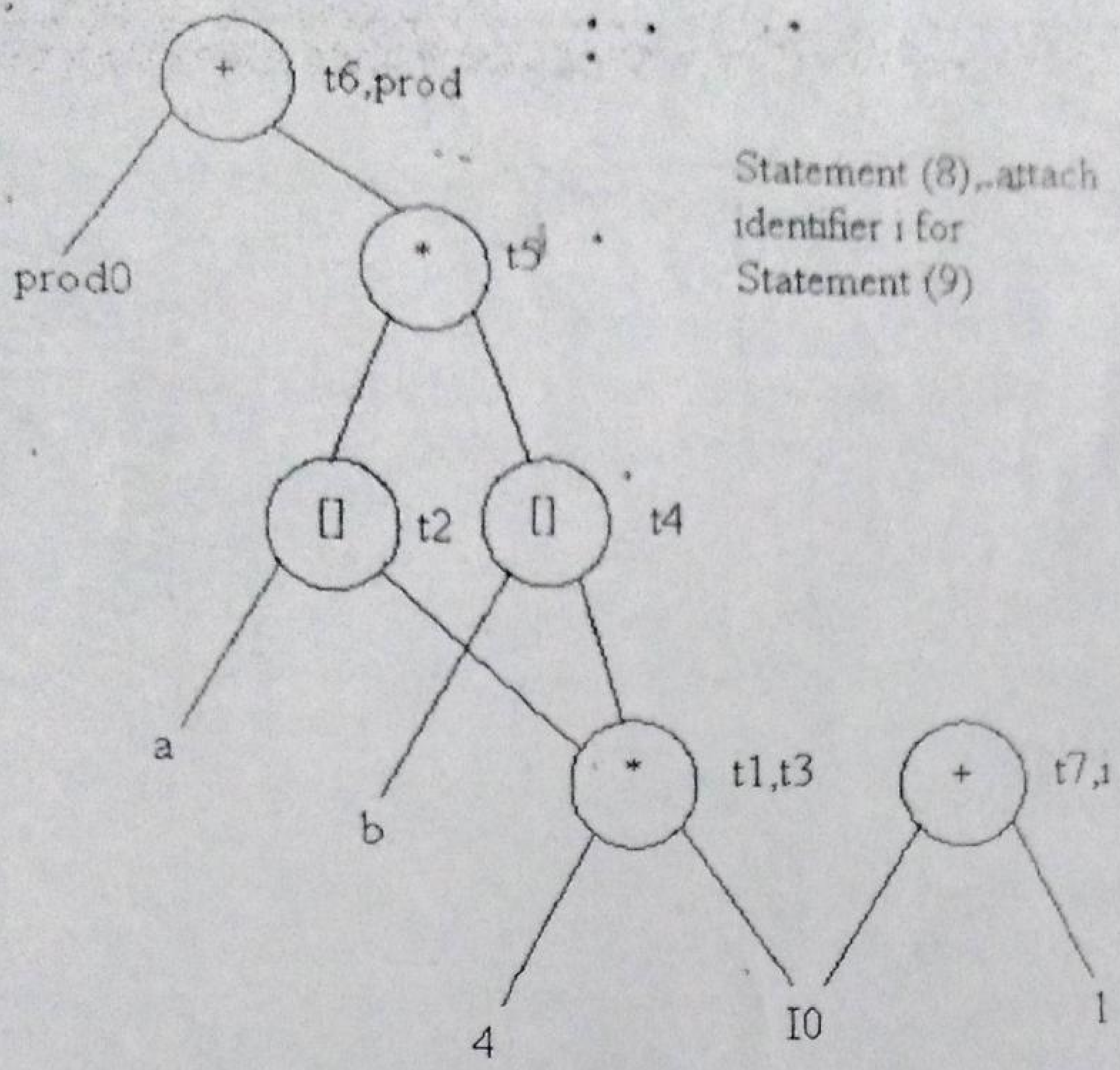
Statement (5)

(f)



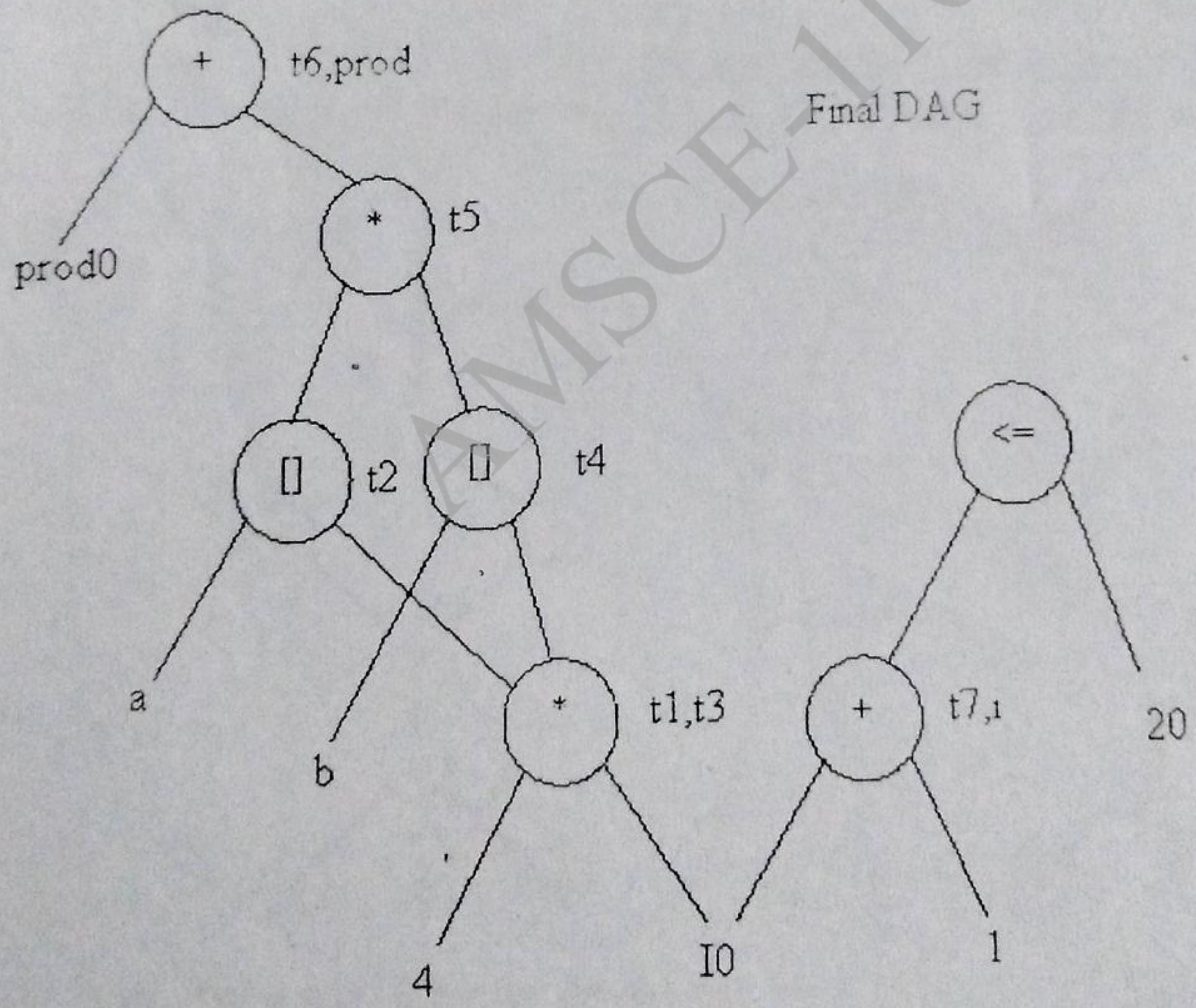
Statement (6), attach identifier prod for Statement (7)

(g)



Statement (8), attach identifier i for Statement (9)

(h)



Final DAG

Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

.5 EFFICIENT DATA FLOW ALGORITHMS

Data-flow analysis speed can be increased by the following two algorithms

1. Depth-First Ordering in iterative Algorithms:
2. Structure-based Data-Flow Analysis.

The first is an application of depth-first ordering to reduce the number of 'passes that the iterative algorithm takes, and the second uses intervals or the $T_1$and T_2 transformations to generalize the syntax-directed approach.

Depth-First Ordering in iterative Algorithms

- Reaching definitions. Available expressions, or live variables, any event of significance at a node will be propagated to that node along an acyclic path.
- Iterative algorithms can be used to track their acyclic nature.
- If a definition d is in in[B] then there is some acyclic path from the block containing d to B such that d is in the **in**'s and **out**'s all along that path.
- If an expression **x+y** is not available at the entrance to block B, then there is some acyclic path that demonstrates that fact; either the path is from the initial node and includes no statement that kills or generates **x+y**, or the path is from a block that kills **x+y** and along the path there is no subsequent generation of **x+y**.
- For live variables. if x is live on exit from block B, then there is an acyclic path from B to a use of x, along with there are no definitions of x.
- If a use of **x** is reached from the end of block B along a path with a cycle, we can eliminate that cycle to find a shorter path along which the use of x is still reached from B.

Procedure

1. First visit the root node of the tree. Eg. (1)
2. If no root node present, then visit the first right hand side node. Eg. (1)
3. After reaching depth visit the missed node by visiting their parent node.

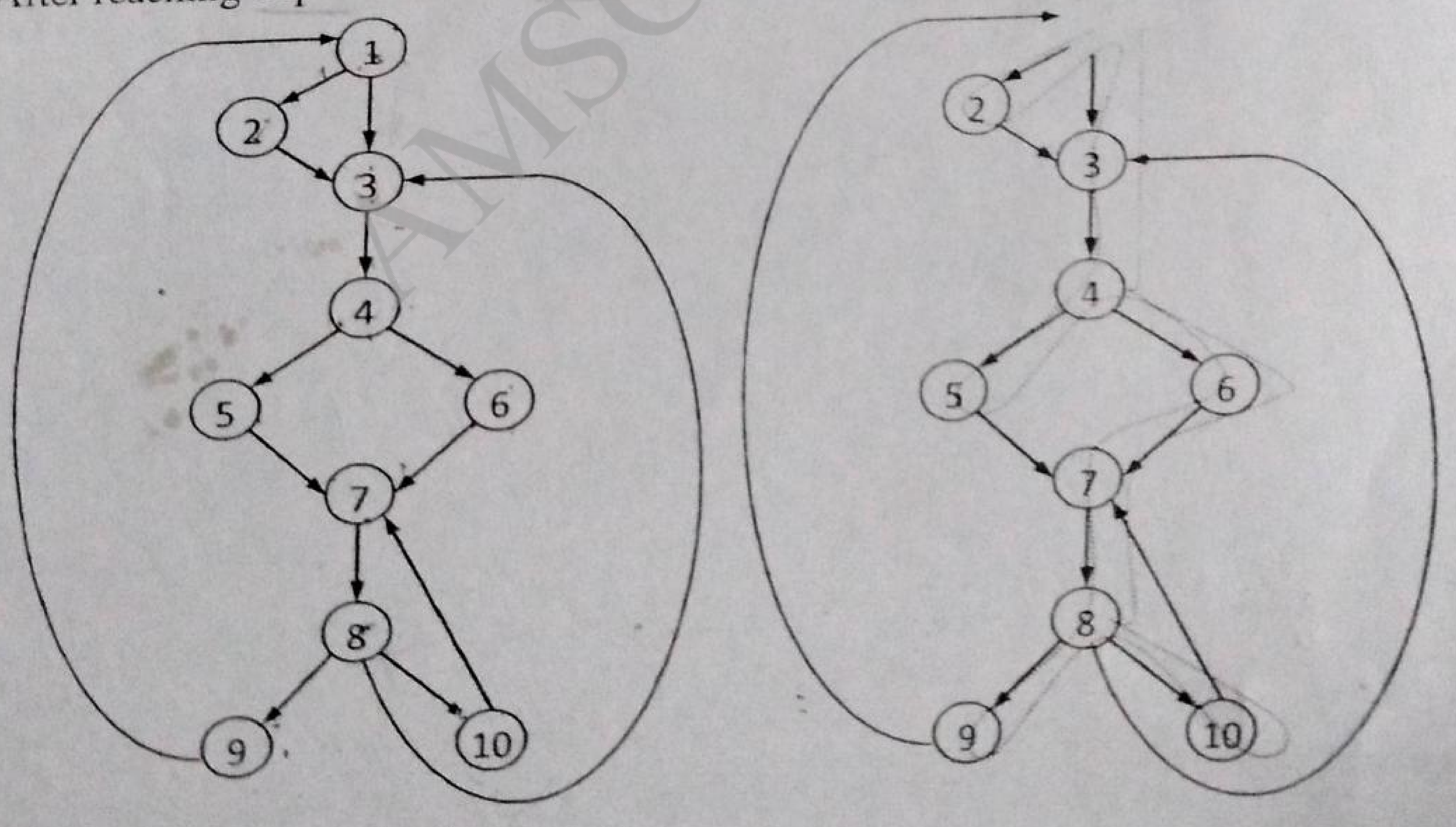


Figure 5.18: Depth first traversal for the given tree.

The order of visiting the edges in the above tree is:

1 → 3 → 4 → 6 → 7 → 8 → 10 → 8 → 9 → 8 → 7 → 6 → 4 → 5 → 4 → 3 → 1 → 2 → 1

Steps:

- After node 4, there is confusion, either 5 or 6, we considered 6.
- After visiting node 10, back tract to 8 to visit 9.
- The definition d from Out[1] will reach In[3] and Out[3] will reach In[4] and so on.

Structure-based Data-Flow Analysis

We can implement data-flow algorithms that visit nodes no more times than the interval depth of the flow graph. The ideas exposed here apply to syntax-directed data-flow algorithms For all sorts of structured control statements.

This algorithm focus on multiple exists in the blocks.

- **Gen** $_{R,B}$ indicates the definition that was generated in the region R of the basic block B.
- **Kill** $_{R,B}$ indicates the definition that was killed in the region R of the basic block B.
- The transfer function **Trans** $_{R,B}$ **(S)** of definition set S is set of definitions reach the end of block B by traveling along paths wholly within R.
- The definitions reaching the end of block B fall into two classes.
  1. Those that are generated within R and propagate to the end of B independent of S.
  2. Those that are not generated in R, but that also are not killed along some path from the header of R to the end of B, and therefore are in **Trans** $_{R,B}$ **(S)** if and only if they are in S.

Thus, we may write trans in the form:

$$\textbf{Trans}_{R,B}(S) = \textbf{Gen}_{R,B} \cup (S - \textbf{Kill}_{R,B})$$

Case 1:

If the transformation does not alter any definition I the basic block B, then the transfer function of region R, is same as the transfer function of Block B.

$$\textbf{Gen}_{B,B} = \textbf{Gen[B]}$$
$$\textbf{kill}_{B,B} = \textbf{Kill[B]}$$

Case 2:

The region R is formed when R_1 consumes R_2. There are no edges from R_2 to R_1. Header of R is the header of R_1. The R_2 does not affect the transfer function of R_1.

$$\textbf{Gen}_{R,B} = \textbf{Gen}_{R1,B}$$
$$\textbf{kill}_{R,B} = \textbf{kill}_{R1,B} \quad \text{for all B in } R_1.$$

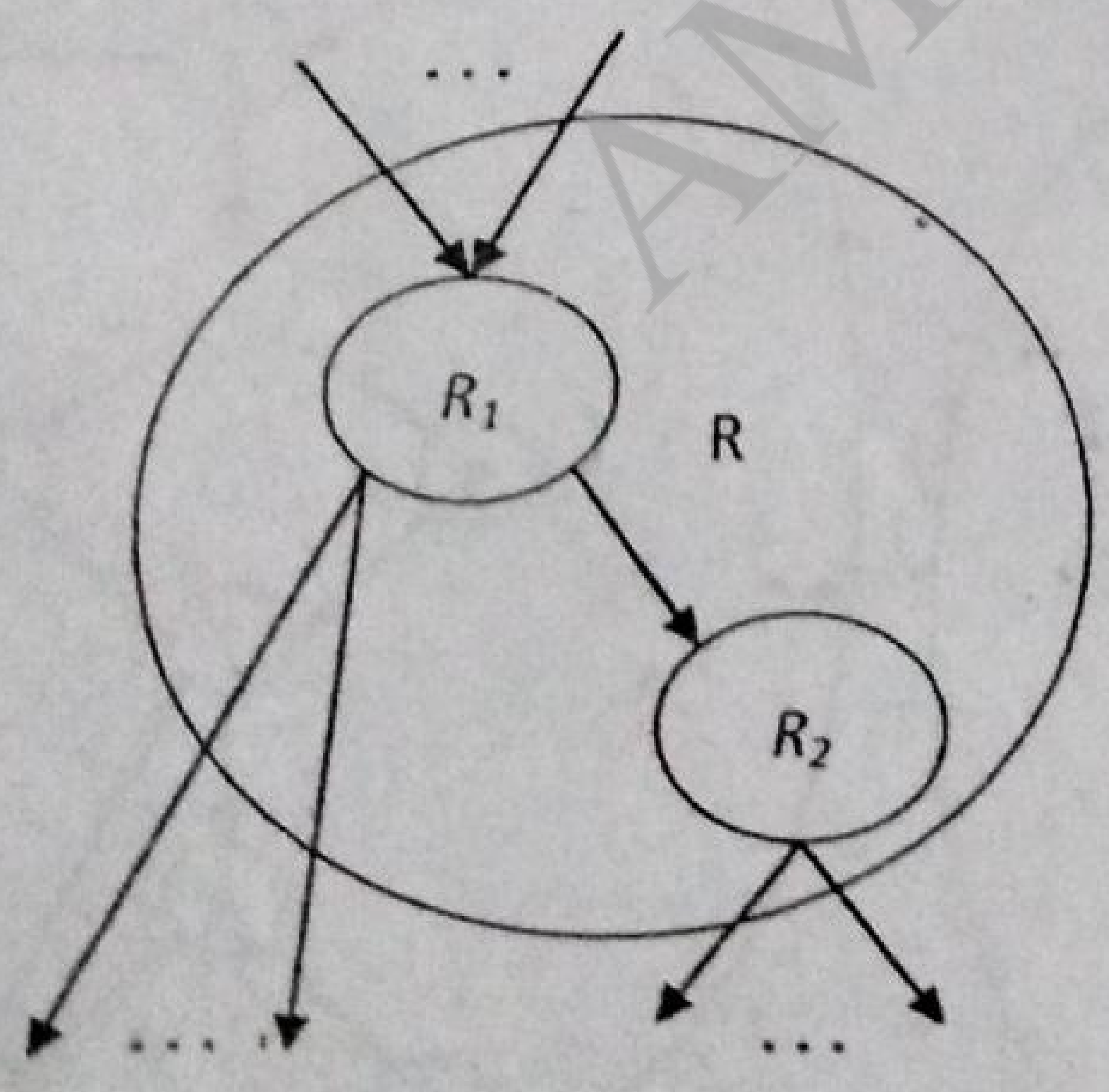


Figure 5.19: Region building by T_2

For B in R_2, a definition can reach the end of B if any of the following conditions hold:

1. The definition is generated within R_2.
2. The definition is generated within R_1 reaches the end of some predecessor of the header of R_2, and is not killed going from the header of R_2 to B.
3. The definition is in the set S available at the header of R_1, not killed going to some predecessor of the header of R_2, and not killed going from the header of R_2 to B.