CS8602

Compiler Design

(Anna University - Regulation)

UNIT 1 – INTRODUCTION TO COMPILERS

Topics to be Covered

Translators-Compilation and Interpretation-Language processors -The Phases of Compiler-Errors Encountered in Different Phases-The Grouping of Phases-Compiler Construction Tools - Programming Language basics.

Translators:

A **translator** is a computer program that performs the translation of a program written in a given programming language into a functionally equivalent program in a different computer language, without losing the functional or logical structure of the original code (the "essence" of each program).

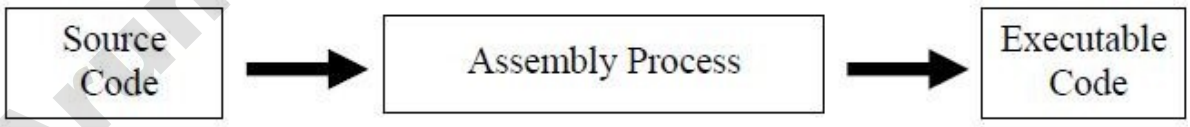
Types of Computer Language Translators:

The widely used translators that translate the code of a computer program into a machine code are:

1. *Assemblers*
2. *Interpreters*
3. *Compilers*

Assembler:

An Assembler converts an assembly program into machine code.



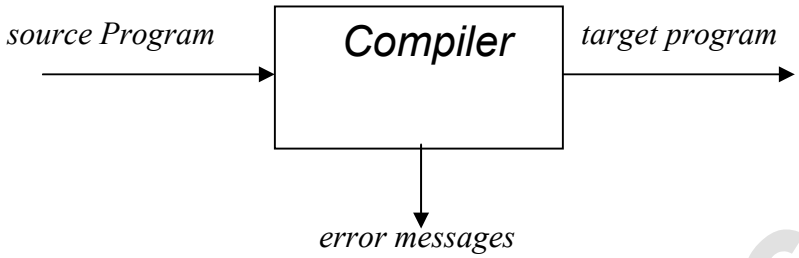
Compilation and Interpretation:

Compilation:

Compilation is the conceptual process of translating source code into a CPU-executable binary target code.

Compiler:

A compiler is a program that reads a program written in one language – the *source* language – and translates it into an equivalent program in another language – the *target language.*



As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

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

Advantages of Compiler:

1. Fast in execution
2. The object/executable code produced by a compiler can be distributed or executed without having to have the compiler present.
3. The object program can be used whenever required without the need to of recompilation.

Disadvantages of Compiler:

1. Debugging a program is much harder. Therefore not so good at finding errors.
2. When an error is found, the whole program has to be re-compiled.

History of Compiler:

- Until 1952 most of the programs were written in assembly language
- In 1952 Grace Hopper writes the first compiler for the A-0 programming language
- Between 1957 – 58 John Backus writes the first Fortran compiler. Optimization of the code was the integral component of the compiler.

Applications of Compiler Technology:

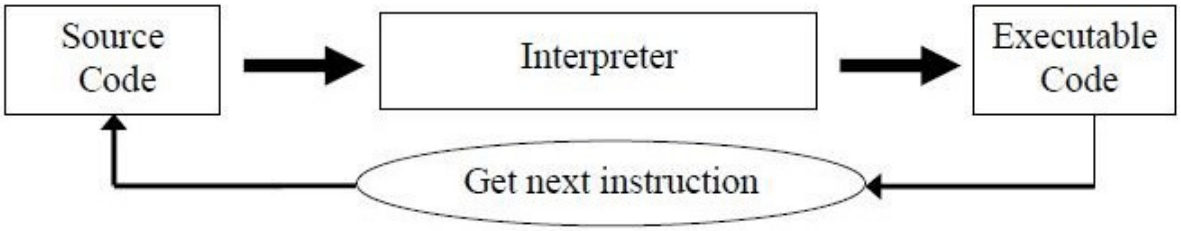
- Implementation of High Level Programming Languages
- Optimizations for Computer Architectures (both *parallelism and memory hierarchies* improve the potential performance of a machine, but they must be harnessed effectively by the compiler to deliver real performance of an application)
- Design of a new computer architecture
- Program Translations (Program Translation techniques are: Binary Translation, Hardware Synthesis, Database Query Interpreters, Compiled Simulation)
- Software Productivity Tools (Ex. Structure editors, type checking, bound checking, memory management tools, etc)

Interpretation:

Interpretation is the conceptual process of translating a high level source code into executable code.

Interpreter:

An Interpreter is also a program that translates high-level source code into executable code. However the difference between a compiler and an interpreter is that **an interpreter translates one line at a time and then executes it**: no object code is produced, and so the program has to be interpreted each time it is to be run. If the program performs a section code 1000 times, then the section is translated into machine code 1000 times since each line is interpreted and then executed.



Advantages of an Interpreter:

1. Good at locating errors in programs
2. Debugging is easier since the interpreter stops when it encounters an error.
3. If an error is deducted there is no need to retranslate the whole program

Disadvantages of an Interpreter:

1. Rather slow
2. No object code is produced, so a translation has to be done every time the program is running.
3. For the program to run, the Interpreter must be present

Difference between Compiler and Interpreter:

S.No.	***Compiler***	***Interpreter***
1.	Compiler works on the complete program at once. It takes the entire program as input.	Interpreter Program works line by line. It takes one statement at a time as input.
2.	Compiler generates intermediate code, called the object code or machine code.	Interpreter does not generate intermediate object code or machine code.
3.	Compiler executes conditional control statements (like if-else and switch-case) and logical constructs faster than interpreter.	Interpreter executes conditional control statements at a much slower speed.
4.	Compiled program take more memory because the entire object code has to reside in memory.	Interpreter does not generate intermediate object code. As a result, interpreted programs are more memory efficient.

S.No.	*Compiler*	*Interpreter*
5.	Compile once and run any time. Compiled program does not need to be compiled every time.	Interpreted programs are interpreted line by line every time they are executed.
6.	Errors are reported after the entire program is checked for syntactical and other errors.	Error is reported as soon as the first error is encountered. Rest of the program will be checked until the existing error is removed.
7.	A compiled language is more difficult to debug.	Debugging is easy because interpreter stops and report errors as it encounters them.
8.	Compiler does not allow a program to run until it is completely error-free.	Interpreter runs the program from the first line and stops execution only if it encounters an error.
9.	Compiled languages are more efficient but difficult to debug.	Interpreted languages are less efficient but easier to debug.
10.	*Examples:* *C, C++, COBOL*	*Examples:* *BASIC, VISUAL BASIC, Python, Ruby, PERL, MATLAB, Lisp*

Hybrid Compiler:

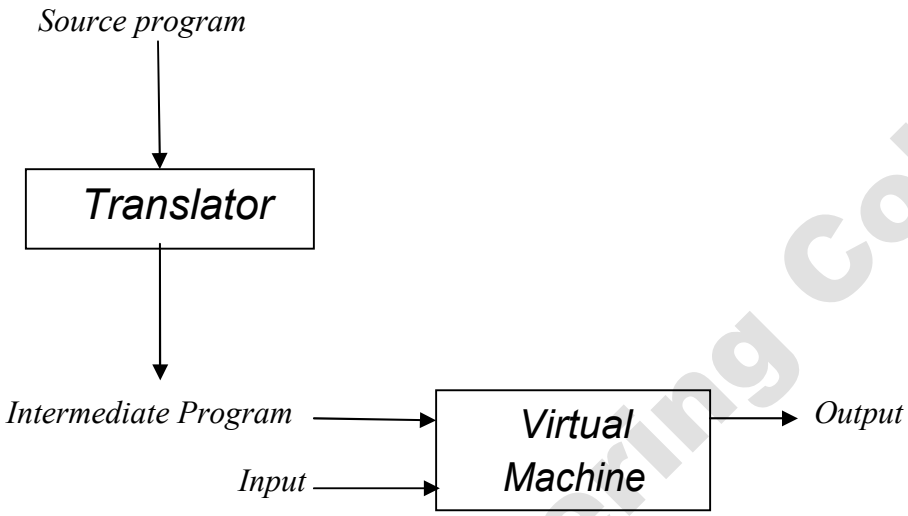
Hybrid compiler is a compiler which translates a human readable source code to an intermediate byte code for later interpretation. So these languages do have both features of a compiler and an interpreter. These types of compilers are commonly known as Just In-time Compilers (JIT).

Example of a Hybrid Compiler:

Java is one good example for these types of compilers. Java language processors combine compilation and interpretation. A Java Source program may be first compiled into an intermediate form called *byte codes.* The byte codes are then interpreted by a virtual machine.

A benefit of this arrangement is that the byte codes compiled on one machine can be interpreted on another machine, perhaps across a network.

In order to achieve faster processing of inputs to outputs, some Java compilers called *just-in-time* compilers, translate the byte codes into machine language immediately before they run the intermediate program to process the input.



Compilers are not only used to translate a source language into the assembly or machine language but also used in other places.

Example:

1. ***Text Formatters:*** A text formatter takes input that is stream of characters, most of which is text, some of which includes commands to indicate paragraphs, figures, or mathematical structures like subscripts and superscripts.
2. ***Silicon compilers:*** A silicon compiler has a source language that is similar or identical to a conventional programming language. The variable of the language represent logical signals (0 or 1) or groups of signals in a switching circuit. The output is a circuit design in an appropriate language.
3. ***Query Interpreters:*** A query interpreter translates a predicate containing relational and Boolean operators into commands to search a database for records satisfying that predicate.

Language Processors:

A language processor is a program that processes the programs written in programming language (source language). A part of a language processor is a language translator, which translates the program from the source language into machine code, assembly language or other language.

An integrated software developmental environment includes many different kinds of language processors. They are:

1. Pre Processor
2. Compiler
3. Assembler
4. Linker
5. Loader

1. Pre Processor

The Pre Processor is the system software which is used to process the source program before fed into the compiler. They may perform the following functions:

1. *Macro Processing:* A preprocessor may allow a user to define macros that are shorthand for longer constructs.
2. *File Inclusion:* A preprocessor may include header files into the program text. For example, the C pre-processor causes the contents of the file <global.h> to replace the statement #include <global.h> when it processes a file containing this statement.
3. *Rational Preprocessors:* These processors provides the user with built-in macros for constructs like while-statements or if-statements etc.,
4. *Language Extensions:* It provides features similar to built-in macros. For example, the language Equel is a database query language embedded in C.

2. Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole

source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

3. Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

4. Linker

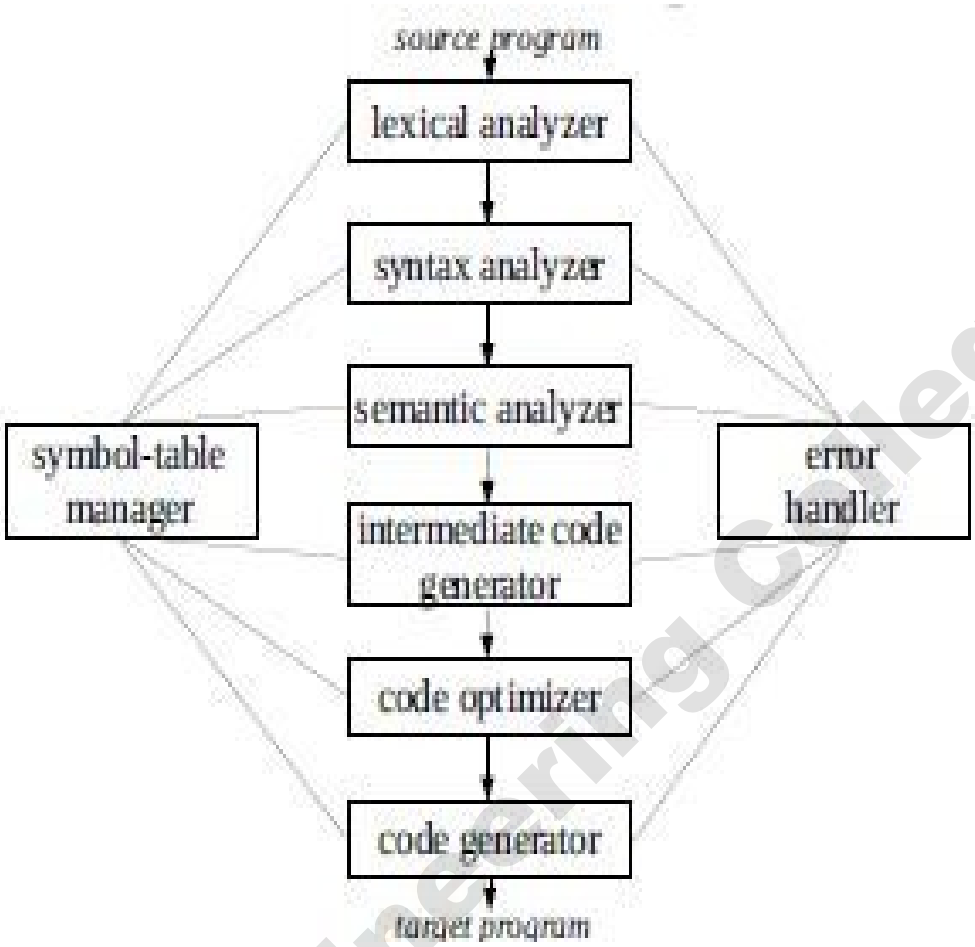
Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

5. Loader

Loader is a part of operating system and is responsible for loading executable files into memory and executes them. It calculates the size of a program *instructions and data* and creates memory space for it. It initializes various registers to initiate execution.

Phases of Compiler:

A compiler operates in phases, each of which transforms the source program from one representation to another.



The Analysis – Synthesis Model of Compilation:

There are two parts to compilation:

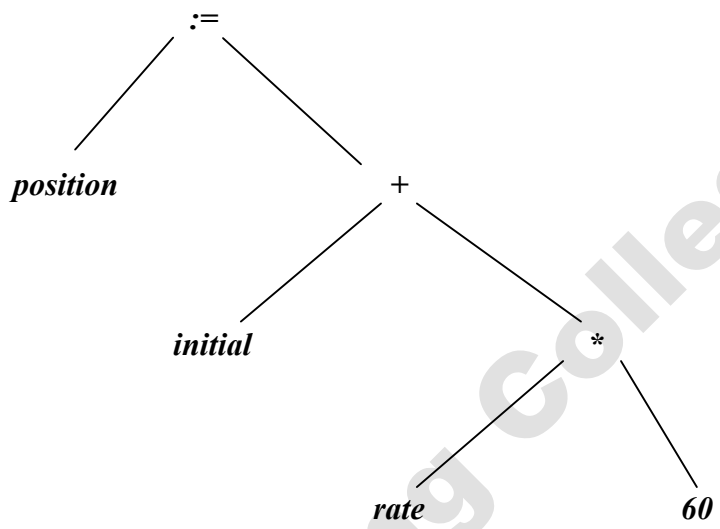
- Analysis and
- Synthesis

1. Analysis:

The first three phases forms the bulk of the analysis portion of a compiler. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a syntax tree, in which each node represents an operation and the children of a node represent the arguments of the operation.

Example:

*Syntax tree for position := initial + rate * 60*



2. Synthesis Part:

The synthesis part constructs the desired target program from the intermediate representation. This part requires most specialized techniques.

<u>The Analysis Phase:</u>

Lexical Analysis: The lexical analysis phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically sequence of characters, such as identifier, a keyword (if, while, etc), a punctuation character, or a multi-character operator work like :=. The character sequence forming a token is called the *lexeme* for the token.

Certain tokens will be augmented by a "lexical value". Ex. When an identifier ***rate*** is found, the lexical analyzer generates the token ***id*** and also enters ***rate*** into the symbol table, if it is not already exist. The lexical value associated with this ***id*** then points to the symbol-table entry for ***rate.***

*Example: position := initial + rate * 60*

Tokens:

1. *position, initial and rate* - ***id***
2. *:=, + and * are* ***signs***
3. *60 is a* ***number***

Thus the lexical analyzer will give the output as:

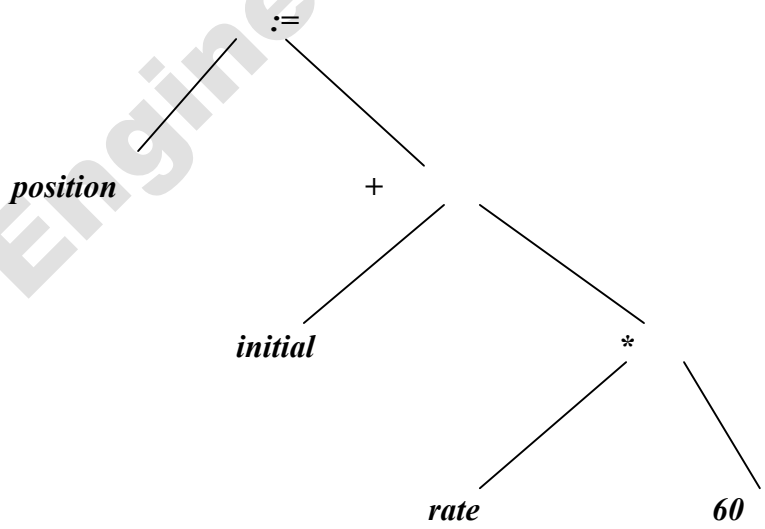
$Id_1 := id_2 + id_3 * 60$

Syntax Analysis:

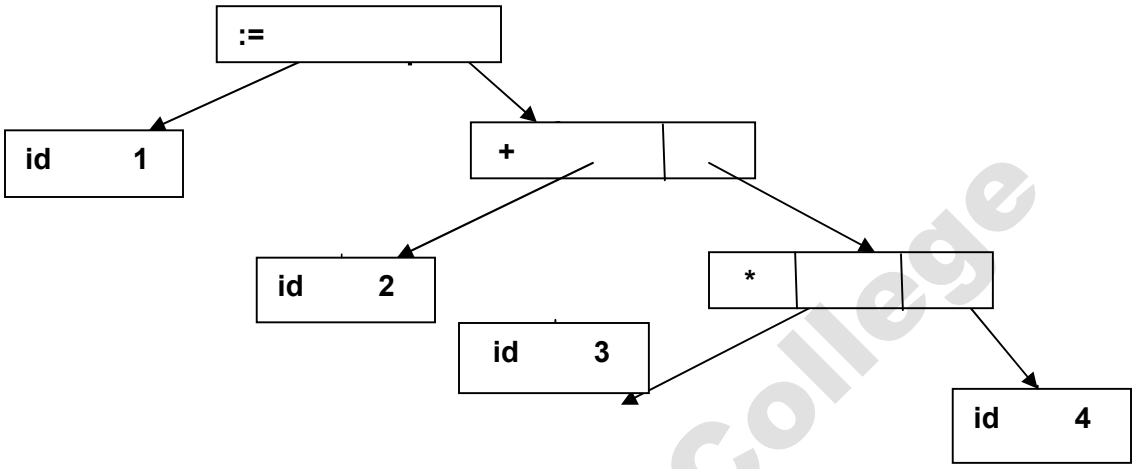
The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree *or syntax tree*. In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

It imposes a hierarchical structure of the token stream in the form of parse tree or syntax tree. The syntax tree can be represented by using suitable data structure.

*Example: position := initial + rate * 60*



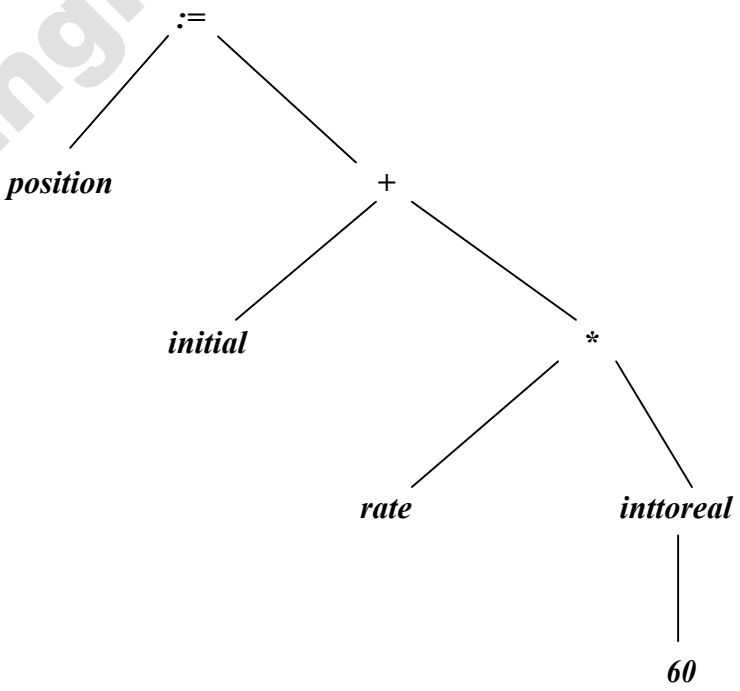
Data structure of the above tree:



Semantic Analysis:

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

This analysis inserts a conversion from integer to real in the above syntax tree.



Synthesis Phase:

Intermediate Code Generation:

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Intermediate code have two properties: easy to produce and easy to translate into the target program. An intermediate code representation can have many forms. One of the form is *three-address code,* which is like the assembly language for a machine in which every memory location can act like a register and *three-address code* have at most three operands.

Example: The output of the semantic analysis can be represented in the following intermediate form:

temp1 := inttoreal (60)

*temp2 := id3 * temp1*

temp3 := id2 + temp2

id1 := temp3

Code Optimization:

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources *CPU*, *memory*. In the following example the natural algorithm is used for optimizing the code.

Example:

The output of intermediate code can be optimized as:

*temp1 := id3 * 60.0*

id1 := id2 + temp1

The compiler that do most code optimization are called *"optimizing compilers"*.

Code Generation:

This is the final phase of the compiler which generates the target code, consisting normally of relocatable machine code or assembly code. Variables are assigned to the registers.

Example:

The output of above optimized code can be generated as:

MOVF id3, R2

MULF #60.0, R2

MOVF id2, R1

ADDF R2, R1

MOVF R1, id3

The first and the second operands of each instruction specify a source and destination respectively. The F in each instruction denotes the floating point numbers. The # signifies that 60.0 is to be treated as constant.

Activities of Compiler:

Symbol table manager and error handler are the other two activities in the compiler which is also referred as phases. These two activities interact with all the six phases of a compiler.

Symbol Table Manager:

The symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.

The attributes of the identifiers may provide the information about the storage allocated for an identifier, its type, its scope (where in the program it is valid), and in the case of procedure

names the attributes provide information about the number and types of its arguments, the method of passing each argument (eg. by reference), and the type returned, if any.

The symbol table allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. Attributes of the identifiers cannot be determined during lexical analysis phase. But it can be determined during the syntax and semantic analysis phases. The other phase like code generators uses the symbol table to retrieve the details about the identifiers.

Error Handler: (Error Detection and Reporting)

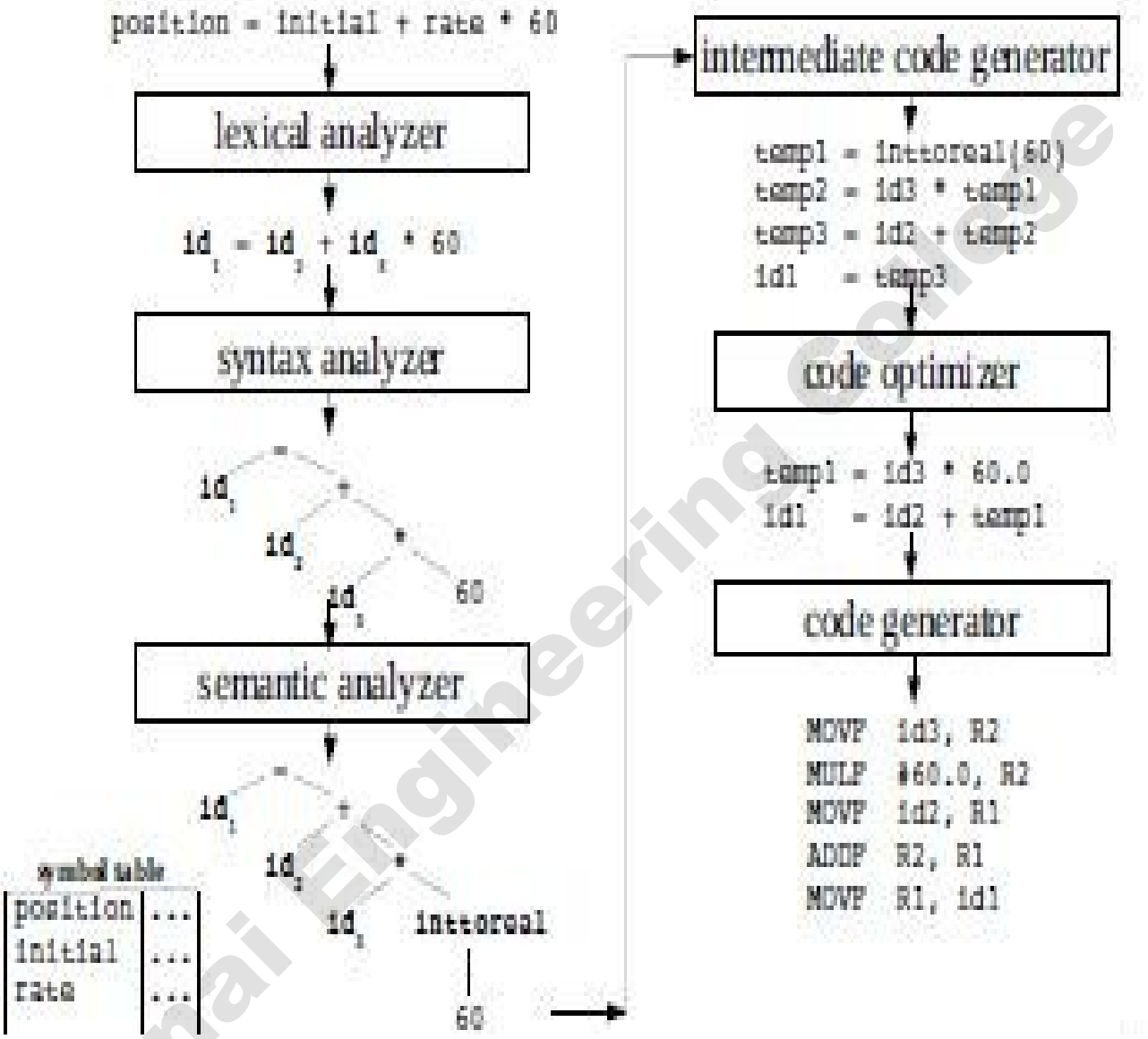
Each phase can encounter errors. After the deduction of an error, a phase must somehow deal with that error, so that the compilation can proceed, allowing further errors in the source program to be detected.

Lexical Analysis Phase: If the characters remaining in the input do not form any token of the language, then the lexical analysis phase detect the error.

Syntax Analysis Phase: The large fraction of errors is handled by syntax and semantic analysis phases. If the token stream violates the structure rules (syntax) of the language, then this phase detects the error.

Semantic Analysis Phase: If the constructs have right syntactic structure but no meaning to the operation involved, then this phase detects the error. Ex. Adding two identifiers, one of which is the name of the array, and the other the name of a procedure.

Translation of statement



Errors Encountered in Different Phases:

Program submitted to a compiler often have errors of various kinds. So, good compiler should be able to detect as many errors as possible in various ways and also recover from them.

Each phase can encounter errors. After the deduction of an error, a phase must somehow deal with that error, so that the compilation can proceed, allowing further errors in the source program to be detected.

Errors during Lexical Analysis:

If the characters remaining in the input do not form any token of the language, then the lexical analysis phase detect the error.

There are relatively few errors which can be detected during lexical analysis.

i. Strange characters

Some programming languages do not use all possible characters, so any strange ones which appear can be reported. However almost any character is allowed within a quoted string.

ii. Long quoted strings (1)

Many programming languages do not allow quoted strings to extend over more than one line; in such cases a missing quote can be detected.

iii. Long quoted strings (2)

If quoted strings can extend over multiple lines then a missing quote can cause quite a lot of text to be 'swallowed up' before an error is detected.

iv. Invalid numbers

A number such as 123.45.67 could be detected as invalid during lexical analysis (provided the language does not allow a full stop to appear immediately after a number). Some compiler writers prefer to treat this as two consecutive numbers 123.45 and .67 as far as lexical analysis is concerned and leave it to the syntax analyser to report an error. Some languages do not allow a number to start with a full stop/decimal point, in which case the lexical analyzer can easily detect this situation.

Error Recovery Actions:

The possible error-recovery actions are:

i) Deleting an extraneous character
ii) Inserting a missing character
iii) Replacing an incorrect character by correct character
iv) Transposing two adjacent characters

For example:

fi (a == 1) ....

Here ***fi*** is a valid identifier. But the open parentheses followed by the identifier may tell ***fi*** is misspelling of the keyword if or an undeclared function identifier.

Errors in Syntax Analysis:

The large fraction of errors is handled by syntax and semantic analysis phases. If the token stream violates the structure rules (syntax) of the language, then this phase detects the error.

The errors detected in this phase 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). Unbalanced parenthesis in expressions is handled

During syntax analysis, the compiler is usually trying to decide what to do next on the basis of expecting one of a small number of tokens. Hence in most cases it is possible to automatically generate a useful error message just by listing the tokens which would be acceptable at that point.

```
Source: A + * B
Error:      | Found '*', expect one of: Identifier, Constant, '('
```

More specific hand-tailored error messages may be needed in cases of bracket mismatch.

```
Source: C := ( A + B * 3 ;
Error:                | Missing ')' or earlier surplus '('
```

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc.. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Errors during Semantic Analysis

Semantic errors are mistakes concerning the meaning of a program construct; they may be either type errors, logical errors or run-time errors:

(i) **Type errors** occur when an operator is applied to an argument of the wrong type, or to the wrong number of arguments.

(ii) **Logical errors** occur when a badly conceived program is executed, for example: while x = y do ... when x and y initially have the same value and the body of loop need not change the value of either x or y.

(iii) **Run-time errors** are errors that can be detected only when the program is executed, for example:

```
var x : real; readln(x); writeln(1/x)
```

which would produce a run time error if the user input 0.

Syntax errors must be detected by a compiler and at least reported to the user (in a helpful way). If possible, the compiler should make the appropriate correction(s). Semantic errors are much harder and sometimes impossible for a computer to detect.

The Grouping of Phases:

Depending on the relationship between phases, the phases are grouped together as front end and a back end.

Front End:

The front end consists of phases that depend primarily on the source language and are largely independent of the target machine. The phases of front end are:

- Lexical Analysis
- Syntactic Analysis
- Creation of the symbol table
- Semantic Analysis
- Generation of the intermediate code
- A part of code optimization
- Error Handling that goes along with the above said phases

Back End:

The back end includes the phases of the compiler that depend on the target machine, and these phases do not depend on the source language, but depend on the intermediate language. The phases of back end are:

- Code Optimization
- Code Generation
- Necessary Symbol table and error handling operations

Categories of Compiler Design:

Based on the grouping of phases there are two types of compiler design is possible:

1. A Single Compiler for different Machine - It is possible to produce a single compiler for the same source language on a different machine by taking the front end of a compiler as common and redo its associated back end.
2. Several Compiler for One Machine – It is possible to produce several compilers for one machine by using a common back end for the different front ends.

Compiler Construction Tools:

In order to atomize the development of compilers some general tools have been created. These tools use specialized languages for specifying and implementing the component. The most successful tool should hide the details of the generation algorithm and produce components which can be easily integrated into the remainder of the compiler. These tools are often referred as *compiler – compilers, compiler – generators, or translator-writing systems.*

Some of the compiler-construction tools are:

Parser generators: Automatically produce syntax analyzers from a grammatical description of a programming language.

Scanner generators: Produce lexical analyzers from a regular-expression description of the tokens of a language.

Syntax-directed translation engines: Produce collections of routines for walking a parse tree and generating intermediate code.

Code-generator generators: Produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

Data-flow analysis engines: Facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

Compiler-construction toolkits: Provide an integrated set of routines for constructing various phases of a compiler.

Programming Language Basics:

The important terminology and distinctions that appear in the programming languages are:

1. The Static / Dynamic Distinction:

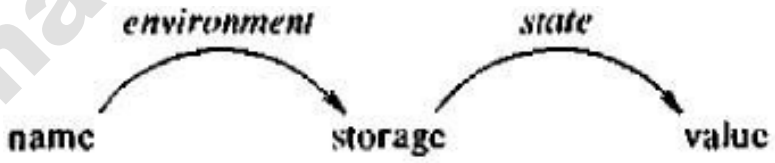
- A programming language can have *static policy* and *dynamic policy*.
- *Static Policy:* The issues that can be decided at compile time by compiler is called *static policy.*
- *Dynamic Policy:* The issues that can be decided at run time of the program is called *dynamic policy.*
- One of the issue decision policy in the language is the scope of declarations.
- *Scope Rules:* The *scope* of a declaration of *x* is the context in which uses of *x* refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program and can be determined by compiler. Otherwise, the language uses *dynamic scope.*
- *Example in Java:*

  public static int x;

  The compiler can determine the location of integer *x* in memory.

2. Environments and States:

The association of names with locations in memory (the *store)* and then with values can be described by two state mappings that change as the program runs.



Two-State Mapping from Names to Values

The *environment* is a mapping from names to locations in the store.

The *state* is a mapping from locations in store to their values. That is, the state maps *l*-values to their corresponding *r*-values, in the terminology of C.

Example:

The storage address 100, associated with variable *pi,* holds *0.* After the assignment *pi := 3.14*, the same storage is associated with *pi*, but the value held there is *3.14.*

3. Static Scope and Block Structure:

Scope Rules: The *scope* of a declaration of *x* is the context in which uses of *x* refer to this declaration. . A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program and can be determined by compiler. Otherwise, the language uses *dynamic scope.*

- *Example in Java:*

  public static int x;

The compiler can determine the location of integer *x* in memory.

The static-scope policy is as follows:

1. A C program consists of a sequence of top-level declarations of variables and functions.
2. Functions may have variable declarations within them, where variables include local variables and parameters. The scope of each such declaration is restricted to the function in which it appears.
3. The scope of a top-level declaration of a name *x* consists of the entire program that follows, with the exception of those statements that lie within a function that also has a declaration of *x.*

Block Structures:

Languages that allow blocks to be nested are said to have *block structure.* A name a: in a nested block *B* is in the scope of a declaration *D* of *x* in an enclosing block if there is no other declaration of *x* in an intervening block.

4. Explicit Access Control:

- Classes and structures introduce a new scope for their members.
- The use of keywords like **public, private,** and **protected,** object oriented languages such as C + + or Java provide explicit control over access to member names in a super class.
- These keywords support *encapsulation* by restricting access.
- Thus,
    - Private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C + + term).
    - Protected names are accessible to subclasses.
    - Public names are accessible from outside the class.

5. Dynamic Scope:

- *Scope Rules:* The *scope* of a declaration of *x* is the context in which uses of *x* refer to this declaration.
- A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program and can be determined by compiler.
- *Example in Java:*

  public static int x;

  The compiler can determine the location of integer *x* in memory.
- The language uses *dynamic scope* if it is not possible to determine the scope of a declaration during compile time.
- *Example in Java:*

  public int x;
- With dynamic scope, as the program runs, the same use of *x* could refer to any of several different declarations of *x.*

6. ***Parameter Passing Mechanism:*** Parameters are passed from a calling procedure to the callee either by value (call by value) or by reference (call by reference). Depending on the procedure call, the actual parameters associated with formal parameters will differ.

Call-By-Value: In *call-by-value,* the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure.

Call-By-Reference:

In *call-by-reference,* the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

Call-By-Name:

A third mechanism — call-by-name — was used in the early programming language Algol **60.** It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct).

When large objects are passed by value, the values passed are really references to the objects themselves, resulting in an effective call-by-reference.

7. ***Aliasing:*** When parameters are (effectively) passed by reference, two formal parameters can refer to the same object, called aliasing. This possibility allows a change in one variable to change another.

UNIT 2 – LEXICAL ANALYSIS

Topics to be Covered

Need and Role of Lexical Analyzer-Lexical Errors-Expressing Tokens by Regular Expressions-Converting Regular Expression to DFA- Minimization of DFA-Language for Specifying Lexical Analyzers-LEX-Design of Lexical Analyzer for a sample Language.

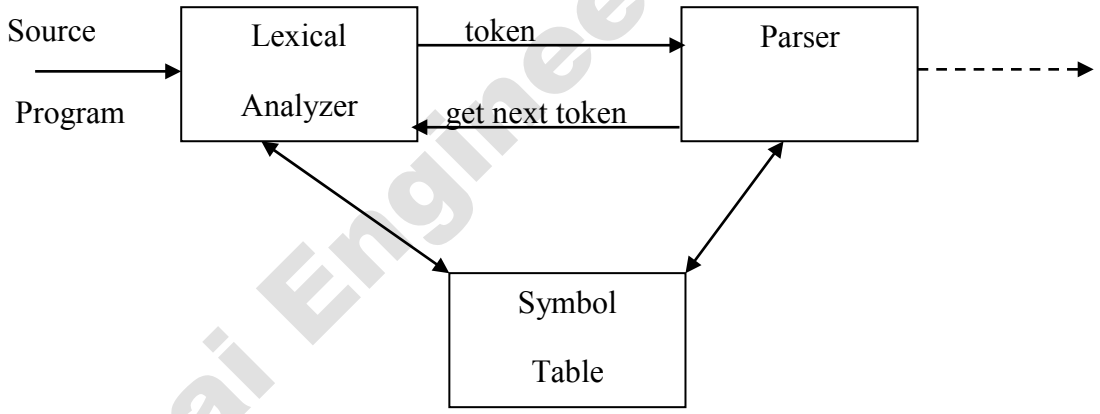
Lexical Analysis

The Role of the Lexical Analyzer

The lexical analyzer is the first phase of a compiler.

Main Task of Lexical Analyzer:

Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.



The above diagram illustrates that the lexical analyzer is a subroutine or a co routine of the parser. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Secondary Tasks of Lexical Analyzer:

Since Lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface.

1. Stripping out from the source program comments and white space in the form of blank, tab and newline characters.
2. Correlating error messages from the compiler with the source program. Example, the lexical analyzer may keep track of the number of newline characters seen, so that a line number can be associated with an error message.

Phases of Lexical Analyzer:

Lexical analyzers are divided into a cascade of two phases:

Scanning – the scanner is responsible for doing simple tasks (Example – Fortran compiler use a scanner to eliminate blanks from the input)

Lexical analysis – the lexical analyzer does the more complex operations.

Issues in Lexical Analysis:

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing:

1. To make the design simpler. The separation of lexical analysis from syntax analysis allows the other phases to be simpler. For example, parsing a document with comments and white spaces is more complex than it is removed in the previous phase itself.
2. To improve the efficiency of the compiler. A separate lexical analyzer allows to construct an efficient processor. A large amount of time is spent in reading the source program and partitioning it into tokens. Specialized buffering techniques speed up the performance.
3. To enhance the compiler portability. Input alphabets and device specific anomalies can be restricted to the lexical analyzer.

Tokens, Patterns and Lexemes:

Token: A token is an atomic unit represents a logically cohesive sequence of characters such as an identifier, a keyword, an operator, constants, literal strings, punctuation symbols such as parentheses, commas and semicolons.

Eg. rate - identifier

+, - - operator

if - keyword

Pattern: A pattern is a rule used to describe lexeme. It is a set of strings in the input for which the same token is produced as output.

Lexeme: A lexeme is a sequence of characters in the source program which is matched by the pattern for a token. i.e. lexemes represents tokens.

Token	Sample Lexemes	Informal Description of Pattern
Const	Const	const
If	If	if
Relation	<, <=, =, < >, >, >=	< or <= or = or < > or > or >=
Id	pi, count, A2	Letter followed by letters and digits
Num	3.1416, 0, 6.02E23	any numeric constant
Literal	"garbage collection"	any characters between " and " except "

Attributes for Tokens:

When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler.

For example, the pattern **relation** matches the operators like <, <=, >, >=, =, < >. It is necessary to identify operator which is matched with the pattern.

The lexical analyzer collects other information about tokens as its attributes. A token has only a single attribute, a pointer to the symbol -table entry in which the information about the token is kept.

For example: The tokens and associated attribute-values for the Fortran statement

X = Y * Z ** 4

are written below as a sequence of pairs:

<**id,** pointer to symbol-table entry for X>

<**assign_op,**>

<**id,** pointer to symbol-table entry for Y>

<**mult_op,**>

<**id,** pointer to symbol-table entry for Z>

<**exp_op,**>

<**num,** integer value 4>

For certain attribute pairs, there is no need for an attribute value.

Eg. <**assign_op,**>

For others, the compiler stores the character string that forms a value in a symbol table.

Lexical Errors:

A lexical analyzer has a very localized view of a source programs.

The possible error-recovery actions are:

i) Deleting an extraneous character
ii) Inserting a missing character
iii) Replacing an incorrect character by correct character
iv) Transposing two adjacent characters

For example:

fi (a == 1) ....

Here ***fi*** is a valid identifier. But the open parentheses followed by the identifier may tell ***fi*** is misspelling of the keyword if or an undeclared function identifier.

INPUT BUFFERING:

Input buffering is a method used to read the source program and to identify the tokens efficiently. There are three general approaches to the implementation of a lexical analyzer.

1. Use a lexical-analyzer generator to produce the lexical analyzer from a regular-expression based specification. In this case, the generator provides routines for reading and buffering the input. Example – Lex Compiler
2. Write the lexical analyzer in a conventional systems-programming language, using the I/O facilities of that language to read the input.
3. Write the lexical analyzer in assembly language and explicitly manage the reading of input.

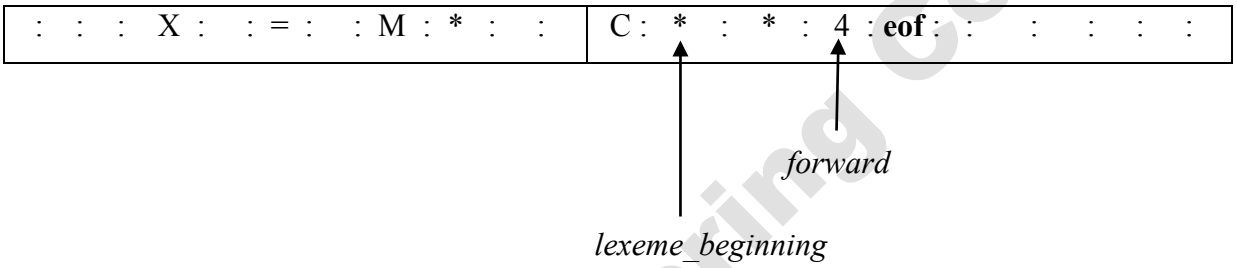
Since the lexical analyzer is the only phase of the compiler that reads the source program character-by-character, it is possible to spend a considerably amount of time in the lexical analysis phase. Thus the speed of lexical analysis is a concern in compiler design.

The following technique uses two-buffer input scheme to identify the tokens. The speed of the lexical .analyzer can be improved by using the sentinels to mark the buffer end.

Buffer Pairs:

The lexical analyzer needs to look-ahead many characters beyond the lexeme for finding the pattern. The lexical analyzer uses a function ungetc( ) to push the look-ahead characters back into the input stream. In order to reduce the amount of overhead required to process an input character, specialized buffering techniques have been developed.

A buffer is divided into N-character halves where N is the number of characters on one disk block. Example 1024 or 4096



Input Buffer with two halves

The processing of buffer pair is as follows:

1. Read N input character into each half of the buffer using one system read command instead of reading each input character
2. If fewer than N characters remain in the input, then eof marker is read into the buffer after the input characters.
3. Two pointers to the input buffer are maintained. Initially both pointers point to the first character of the next lexeme to be found.
    a. Begin pointer points the s tart of the lexeme
    b. The forward pointer is set to the character at its right end
4. Once the lexeme is identified, both pointers are set to the character immediately past the lexeme.

If the forward pointer is reaching the halfway mark, the right half is filled with N new input characters. If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer. The number of tests to be required is very large.

Code to advance forward pointer:

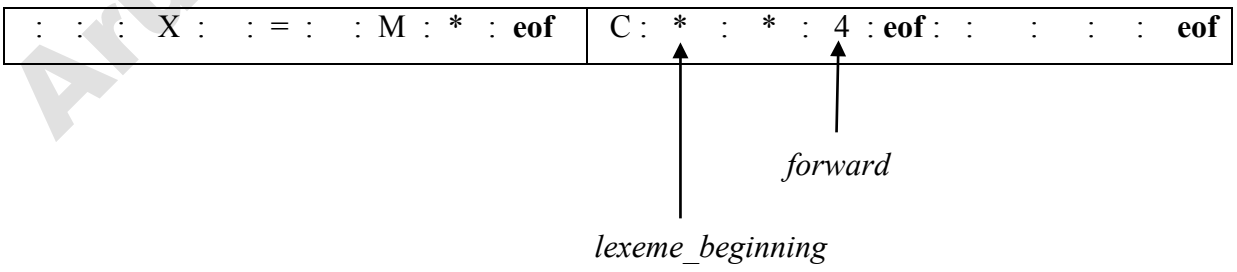
```
if forward at end of first half then begin
      reload second half;
      forward := forward + 1
end

else if forward at end of second half then begin
     reload first half;
     move forward to beginning of first half
end
else
     forward := forward + 1;
```

Sentinels:

In the previous scheme mentioned a check should be made each time when the *forward* pointer is moved that we have not moved off one half of the buffer. i.e. only one **eof** marker at the end.

A sentinel is a special character which is not a part of the source program used to represent the end of file. (eof)

Instead of testing the forward pointer each time by two tests, extend each buffer half to hold a sentinel character at the end and reduce the number of tests to one.



Sentinels at end of each buffer half

For most of the cases, the code performs only one test to see whether forward point to an eof. If it reaches the end of a buffer or the end of the file, then we performs more tests for checking each half and to reload other half of the buffer.

Look ahead code with sentinels:

```
forward := forward + 1;
if forward ↑ eof then begin
    if forward at end of first half then begin
        reload second half;
        forward := forward + 1
    end
    else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
    end
    else /* eof within a buffer signifying end of input */
        terminate lexical analysis
end
```

SPECIFICATION OF TOKENS:

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for set of strings.

Strings and Languages:

Alphabet: An alphabet or character class denotes any finite set of symbols. For example, Letters, Characters, ASCII characters, EBCDIC characters

String: A string over some alphabet is a finite sequence of symbols drawn from that alphabet. For example, 1 0 1 0 1 1 is a string over {0, 1}* , ϵ is a empty string over {0, 1}*

Length of the String : The length of the string 1 0 1 is denoted as | 1 0 1 | = 3 i.e. the number of occurrences of symbol is S.

Language: A language denotes any set of strings over some fixed alphabet Σ.

Example Language L = $\{0^n 1^n \mid n > 0\}$

Some common terms associated with parts of a string are as follows:

Let *s* be the string where S = "regular".

TERM	**DEFINITION**
prefix of *s*	A string obtained by removing zero or more trailing symbols of string *s*; eg. *ban* is a prefix of *banana*
suffix of *s*	A string formed by deleting zero or more of the leading symbols of *s;* eg. *nana* is a suffix of *banana*
substring of *s*	A string obtained by deleting a prefix and a suffix from *s; eg. nan* is a substring of *banana.*
proper prefix, suffix or substring of *s*	Any nonempty string *x* that is, respectively, a prefix, suffix, or substring of *s* such that $s \neq x$
subsequence of *s*	Any string formed by deleting zero or more not necessarily contiguous symbols from *s;* eg. *baaa* is a subsequence of *banana.*

Operations on Languages:

There are several important operations that ca be applied to languages. For lexical analysis the following operations are applied:

OPERATION	**DEFINITION**
union of *L* and *M* written *L U M*	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of *L* and *M* written *LM*	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of *L* written *L**	$L^* = \bigcup_{i=0}^{\infty} L^i$ *L** denotes "zero or more concatenations of" *L*
positive closure of *L* written	∞

L^+	$L^+ = \bigcup_{i=1} L^i$ L^+ denotes "one or more concatenations of" L

Example:

Let L = {A, B, . . . , Z, a, b, . . . , z} and

D = {0, 1, . . . , 9}

By applying operators defined above on these languages L and D we get the following new languages:

1. $L \cup D$ is the set of letters and digits
   i.e. $L \cup D$ = {A,B, . . . ,Z, a, b, . . . , z, 0, 1, . . . , 9}
2. LD is the set of strings consisting of a letter followed by a digit
   i.e. LD = {0A, 0B, . . . , 0Z, 0a, 0b, . . . , 0z, 1A, 1B, . . ,1Z, 1a, 1b, . . ,1z,.....}
3. L^4 is the set of all four-letter strings i.e. L^4 = { aBAC, MNop, ...}
4. L^* is the set of all strings of letters, including ε the empty string
   i.e. L^* = { ε, A, B, . . . , Z, a, b, ..., z, AB, BA, aB, ...}
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter
6. D^+ is the set of all strings of one or more digits

Regular Expressions:

A regular expression is built out of simple regular expressions using a set of defining rules. Each regular expression r denotes a language $L(r)$.

Rules that define the regular expressions:

Basis:

i) ε is a regular expression denotes the language { ε }.

ii) If a is a symbol in Σ then a is a regular expression denotes the language { a }

Induction:

iii) Suppose r and s are regular expressions denoting the language $L(r)$ and $L(s)$. Then,
   a. $(r) \mid (s)$ is a regular expression denoting $L(r) \cup L(s)$.
   b. $(r)(s)$ is a regular expression denoting $L(r)\,L(s)$.

c. *(r)** is a regular expression denoting *(L (r))**.

d. *(r)* is a regular expression denoting *L (r).*

A language denoted by a regular expression is said to be a *regular set.*

The precedence and associativity of operators are as follows:

1. the unary operator * has the highest precedence and is left associative.
2. concatenation has the second highest precedence and is left associative.
3. | has the lowest precedence and is left associative.

Unnecessary parentheses can be avoided in the regular expression if the above precedence is adopted. For example the regular expression: *(a) | ((b)* (c))* is equivalent to *a | b*c.*

Example:

Let Σ = {a,b}

1. The regular expression *a | b* denotes the set *{ a, b }*
2. The regular expression *(a | b) (a | b)* denotes *{aa, ab, ba, bb},* the set of all strings of *a's* and *b's* of length two. Another regular expression for this same set is *aa | ab | ba | bb.*
3. The regular expression *a** denotes the set of all strings of zero or more *a's* i.e. *{ε, a, aa, aaa, . . . }*
4. The regular expression *(a | b)** denotes the set of all strings containing zero or more instances of an *a* or *b,* that is, the set of all strings of *a's* and *b's.* An equivalent regular expression for this set is *(a*b*)**
5. The regular expression *a | a*b* denotes the set containing the string *a* and all strings consisting of zero or more *a's* followed by a *b.*

If two regular expressions *r* and *s* denote the same language, then we say *r* and *s* are equivalent and write *r = s.* For example, *(a | b) = (b | a).*

There are number of algebraic laws obeyed by regular expressions and these laws can be used to manipulate regular expressions into equivalent forms.

Let *r, s* and *t* be the regular expression. The following are the algebraic laws for these regular expressions:

AXIOM	DESCRIPTION
$r \mid s = s \mid r$	| is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	| is associative
$(rs) t = r (st)$	Concatenation is associative
$r (s \mid t) = rs \mid rt$ $(s \mid t) r = sr \mid st$	Concatenation distributes over |
$\epsilon r = r$ $r\epsilon = r$	ϵ is the identity element for concatenation
$r^* = (r \mid \epsilon)^*$	relation between * and ϵ
$r^{**} = r^*$	* is idempotent

Regular Definitions:

The regular expressions can be given names and defining regular expressions using these names is called regular definition. If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

. . . . . .

$d_n \rightarrow r_n$

where each d_i is a distinct name, and each r_i is a regular expression over the symbols in Σ U { $d_1, d_2, \ldots\ldots, d_{i-1}$ }, i.e., the basic symbols and the previously defined names.

Example:

1. *Regular Definition for identifiers:*

   letter → A | B | .....| Z | a | b | .....| z

   digit → 0 | 1 | .....| 9

   id → **letter (letter | digit)***

2. *Regular Definition for num:*

digit → 0 | 1 | . . . | 9

digits → **digit digit***

optional_fraction → **. digits** | ϵ

optional_exponent → (E (+ | - | ϵ) **digits**) | ϵ

num → **digits optional_fraction optional_exponent**

Notational Shorthands:

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. *One or more instances(+):* The unary postfix operator + means "one or more instances of". Example – $(r)^+$ - Set of all strings of one or more occurrences of *r*.
2. *Zero or One Instance (?):* The unary postfix operator ? means " zero or one instance of". Example – *(r)?* – One or zero occurrence of *r*.
   The regular definition for num can be written by using unary + and unary ? operator as follows:

   digit → 0 | 1 | . . . | 9

   digits → **digit**$^+$

   optional_fraction → **(. digits) ?**

   optional_exponent → (E (+ | -)? **digits**) ?

   num → **digits optional_fraction optional_exponent**

3. *Character Classes:* The notation where *a, b* and *c* are alphabet symbols denotes the regular expression *a* | *b* | *c*. An abbreviated character class such as [a – z] denotes the regular expression *a* | *b* | . . . | *z*.
   Using character classes the identifiers can be described as strings generated by regular expression: [A – Za – z] [A – Z a – z 0 – 9]*

Recognition of Tokens:

The tokens are recognized by following the grammatical specification of tokens.

Example:

Consider the following grammar fragment:

stmt → **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| ϵ

expr → *term* **relop** *term*

| *term*

term → **id**

| **num**

where the terminals **if, then, else, relop, id** and **num** generate sets of strings given by the following regular definitions:

if → if

then → then

else → else

relop → < | <= | = | < > | > | >=

id → **letter (letter | digit)***

num → **digit**$^+$ (**. digit** $^+$)? (E (+ | **-**)? **digit**$^+$) ?

letter → A | B | . . . | Z | a | b | . . . | z

digit → 0 | 1 | . . . | 9

Regular definition for White Space (ws) is:

delim → blank | tab | newline

ws → delim^{+}

The goal of the lexical analyzer is to isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute value using the table given below:

Regular Expression	Token	Attribute - Value
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	Pointer to table entry
num	num	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
< >	relop	NE
>	relop	GT
>=	relop	GE

Regular Expression Patterns for Tokens

Transition Diagrams:

As an intermediate step in the construction of a lexical analyzer, a stylized flowchart called a *transition diagram.* Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token. It is used to keep track of information about characters that are seen as the forward pointer scans the input.

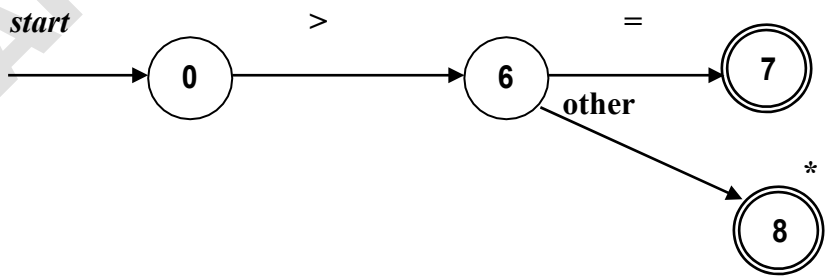
Positions in a transition diagram are drawn as circles and are called *states.* The states are connected by arrows, called *edges.* Edges leaving state *s* have labels indicating the input characters that can next appear after the transition diagram has reached state *s*. The label **other** refers to any character that is not indicated by any of the other edges leaving *s.*

One state is labeled as *start* state; it is the initial state of the transition diagram where control resides when we begin to recognize token. Certain states may have actions that are executed when the flow of control reaches that state. On entering a state we read the next input character. If there is an edge from the current state whose label matches this input character, then we go to the state pointed by the edge. Otherwise, we indicate failure.

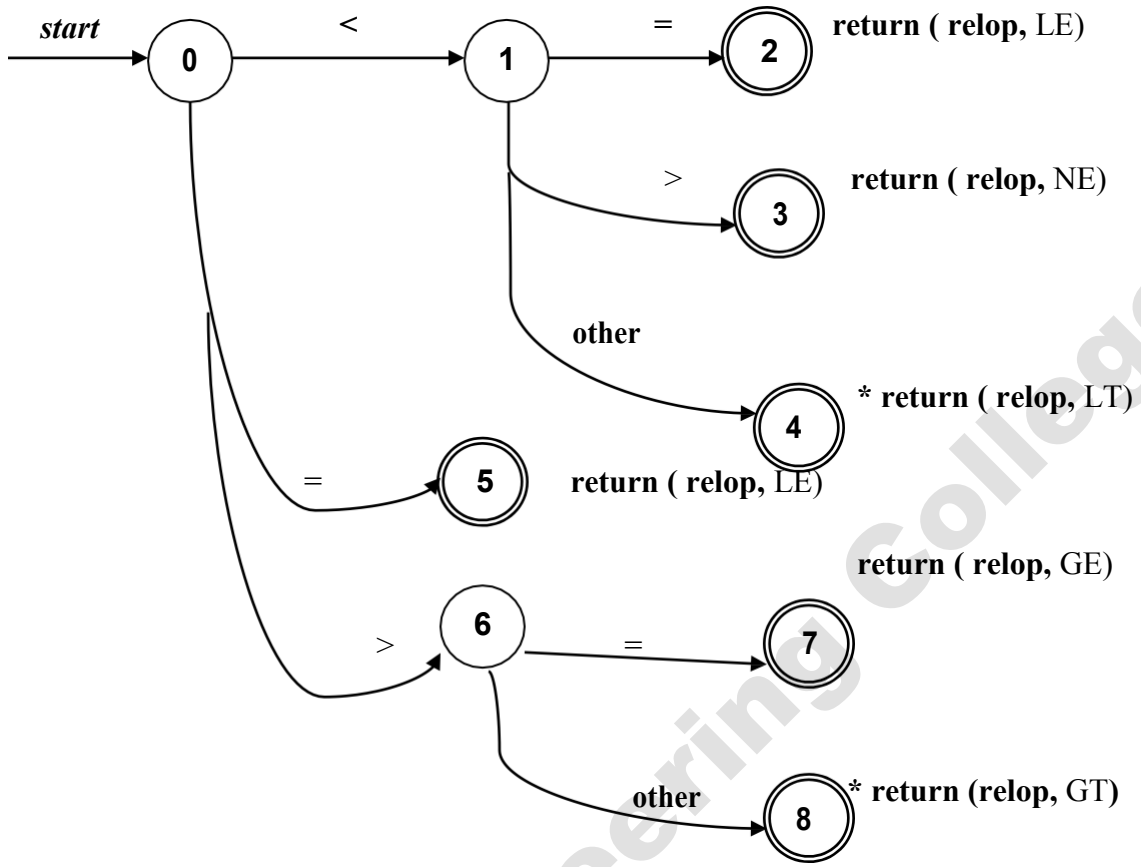
The symbol * is used to indicate states on which the input retraction must take place.

There may be several transition diagram, each specifying a group of tokens. If failure occurs in one transition diagram, then the forward pointer is retracted to where it was in the start state of this diagram, and activate the next transition diagram. Since the lexeme beginning and forward pointers marked the same position in the start state of the diagram, the forward pointer is retracted to the position marked by the lexeme_begining pointer. If failure occurs in all transition diagrams, then a lexical error has been detected and an error-recovery routine is invoked.

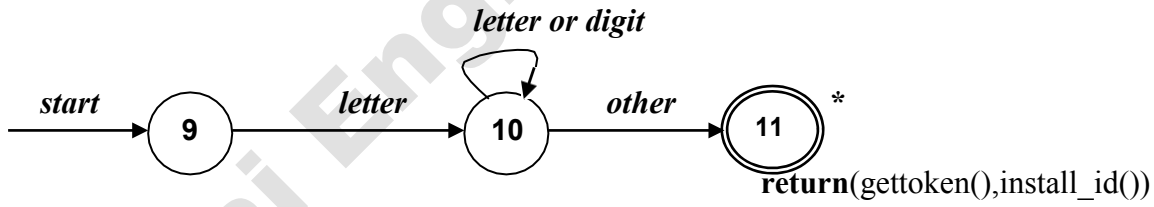
Transition Diagram for >=:



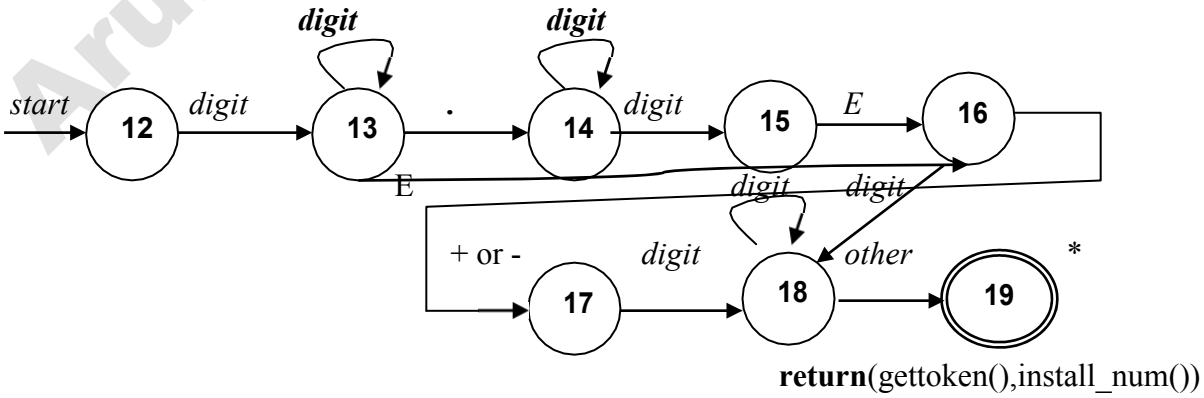
Transition Diagram for Relational Operators:

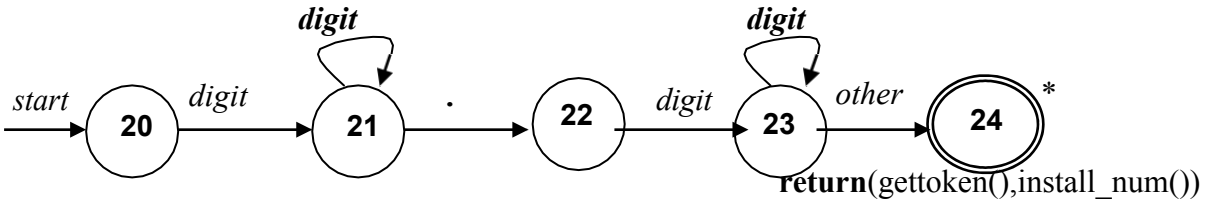


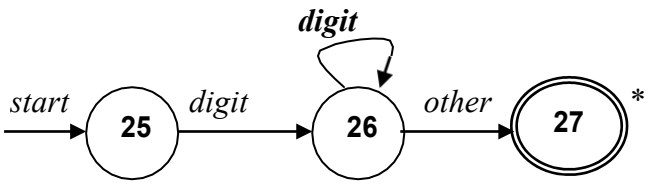
Transition Diagram for identifiers and keywords:



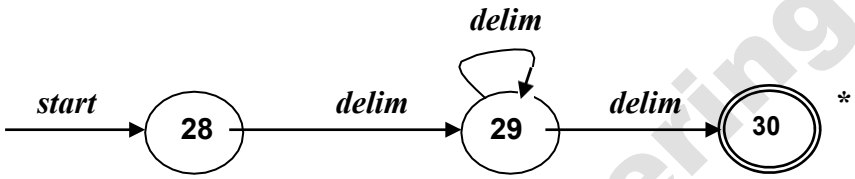
Transition Diagram for Unsigned Numbers in Pascal:







Transition Diagram for white space:



Convert Regular Expression to DFA -

Regular expression is used to represent the language (lexeme) of finite automata (lexical analyzer).

Finite automata

A recognizer for a language is a program that takes as input a string *x* and answers *yes* if *x* is a sentence of the language and *no* otherwise.

A regular expression is compiled into a recognizer by constructing a generalized transition diagram called a Finite Automaton (FA).

Finite automata can be Non-deterministic Finite Automata (NFA) or Deterministic Finite Automata (DFA).

It is given by M = (Q, Σ, *qo,* F, δ*)*.

Where Q - Set of states

Σ - Set of input symbols

qo - Start state

F - set of final states

δ - Transition function (mapping states to input symbol).

$\delta : Q \times \Sigma \rightarrow Q$

• Non-deterministic Finite Automata (NFA)

o More than one transition occurs for any input symbol from a state.

o Transition can occur even on empty string (Ɛ).

• Deterministic Finite Automata (DFA)

o For each state and for each input symbol, exactly one transition occurs from that state.

Regular expression can be converted into DFA by the following methods:

(i) Thompson's subset construction

• Given regular expression is converted into NFA

• Resultant NFA is converted into DFA

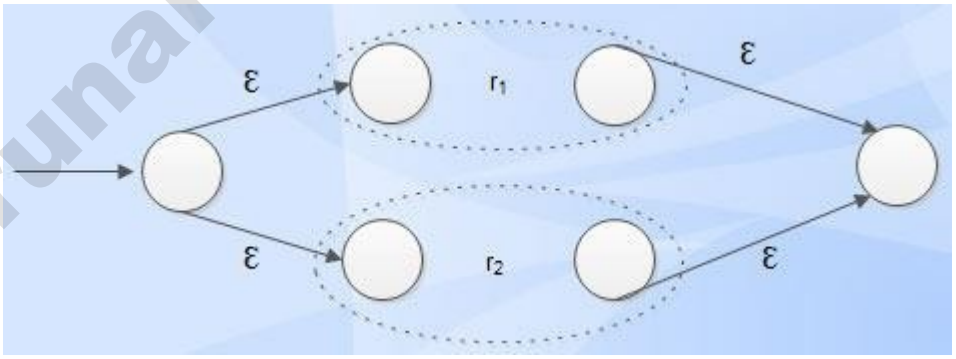
(ii) Direct Method

• In direct method, given regular expression is converted directly into DFA.

Rules for Conversion of Regular Expression to NFA

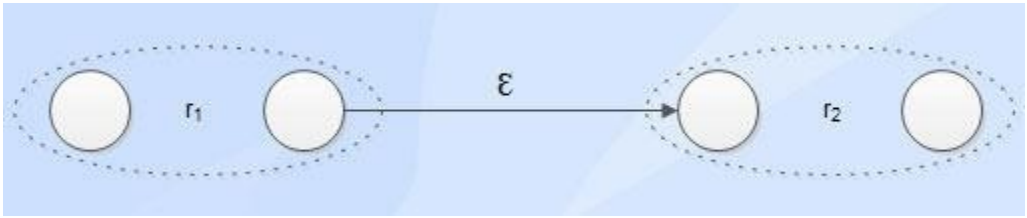
• Union

$$\mathbf{r = r_1 + r_2}$$



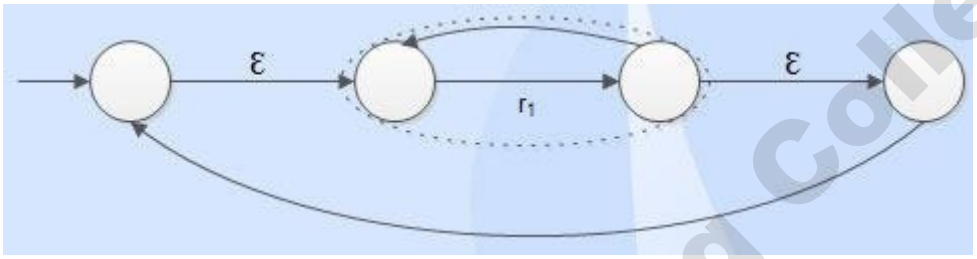
Concatenation

$$\mathbf{r = r_1\, r_2}$$



Closure

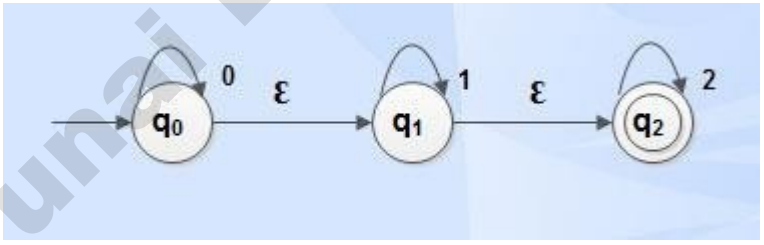
$$r = r_1^*$$



Ɛ –closure

Ɛ - Closure is the set of states that are reachable from the state concerned on taking empty string as input. It describes the path that consumes empty string (Ɛ) to reach some states of NFA.

Example 1

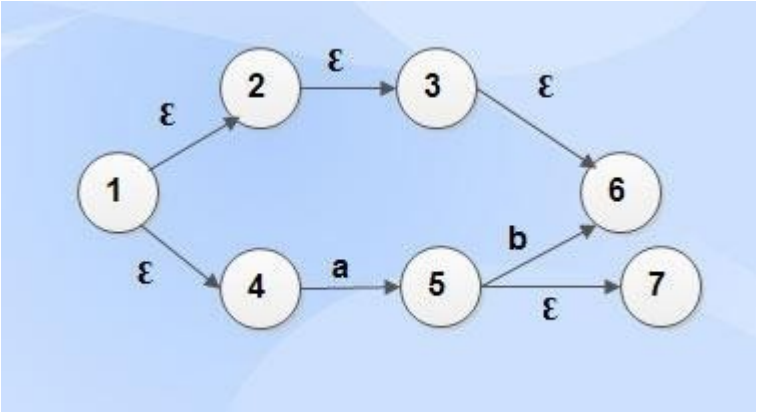


Ɛ -closure(q_0) = *{* q_0, q_1, q_2}

Ɛ –closure(q_1) = {q_1, q_2}

Ɛ -closure(q_2) = { q_0}

Example 2



Ɛ -closure (l) = {l, 2, 3, 4, 6}

Ɛ-closure (2) = {2, 3, 6}

Ɛ-closure (3) = {3, 6}

Ɛ-closure (4) = {4}

Ɛ-closure (5) = {5, 7}

Ɛ -closure (6) = {6}

Ɛ-closure (7) = {7}

Sub-set Construction

• Given regular expression is converted into NFA.

• Then, NFA is converted into DFA.

Steps

1. Convert into NFA using above rules for operators (union, concatenation and closure) and precedence.

2. Find Ɛ -closure of all states.

3. Start with epsilon closure of start state of NFA.

4. Apply the input symbols and find its epsilon closure.

Dtran [state, input symbol] = Ɛ -closure (move (state, input symbol))

where Dtran transition function of DFA

5. Analyze the output state to find whether it is a new state.

6. If new state is found, repeat step 4 and step 5 until no more new states are found.

7. Construct the transition table for Dtran function.

8. Draw the transition diagram with start state as the Ɛ -closure (start state of NFA) and final state is the state that contains final state of NFA drawn.

Direct Method

Direct method is used to convert given regular expression directly into DFA.

1. Uses augmented regular expression r#.
2. Important states of NFA correspond to positions in regular expression that hold symbols of the alphabet.
3. Regular expression is represented as syntax tree where interior nodes correspond to operators representing union, concatenation and closure operations.
4. Leaf nodes corresponds to the input symbols
5. Construct DFA directly from a regular expression by computing the functions nullable(n), firstpos(n), lastpos(n) andfollowpos(i) from the syntax tree.
6. nullable (n): Is true for * node and node labeled with Ɛ. For other nodes it is false.
7. firstpos (n): Set of positions at node ti that corresponds to the first symbol of the sub-expression rooted at n.
8. lastpos (n): Set of positions at node ti that corresponds to the last symbol of the sub-expression rooted at n.
9. followpos (i): Set of positions that follows given position by matching the first or last symbol of a string generated by sub-expression of the given regular expression.

Rules for computing nullable, firstpos and lastpos

Node n	**nullable** *(n)*	**firstpos** *(n)*	**lastpos** *(n)*
A leaf labeled Ɛ	True	Ø	Ø
A leaf with position i	False	{i}	{i}
An *or* node $n = c_1 \mid c_2$	Nullable (c_1) or Nullable (c_2)	firstpos (c_1) U firstpos (c_2)	Iastpos (c_1) U Iastpos (c_2)
A cat node $n = c_1c_2$	Nullable (c_1) and Nullable (c_2)	If (Nullable (c_1)) firstpos (c_1) U firstpos (c_2) else firstpos (c_1)	If (Nullable (c_2)) lastpos (c_1) U Iastpos (c_2) else lastpos (c_1)
A star node $n = c_{1*}$	True	firstpos (c_1)	lastpos (c_1)

Computation of followpos

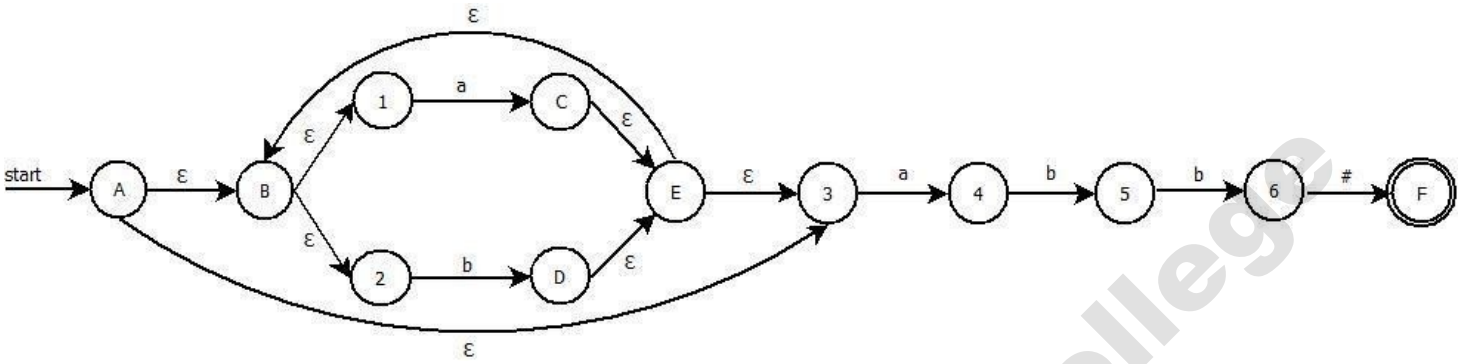
The position of regular expression can follow another in the following ways:

- If *n* is a cat node with left child c_1 and right child c_2, then for every position i in *lastpos*(c_1), all positions in *firstpos*(c_2) are in *followpos(i).*
- For cat node, for each position i in *lastpos* of its *left child,* the *firstpos* of its *right child* will be in f*ollowpos(i).*
- If *n* is a star node and i is a position in *lastpos(n),* then all positions in *firstpos(n)* are in *followpos(i).*
- For star node, the *firstpos* of that node is in *f ollowpos* of all positions in *lastpos* of that node.

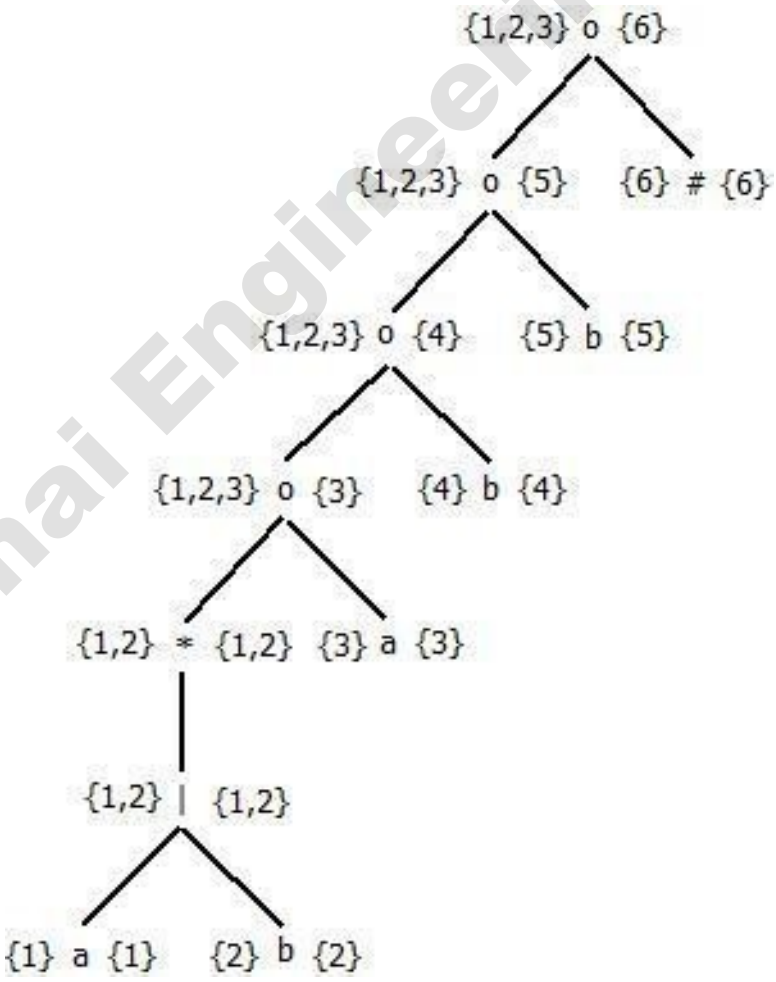
Example:

Thompson's subset construction for

(a+b)*abb



Direct Method for (a+b)*abb #

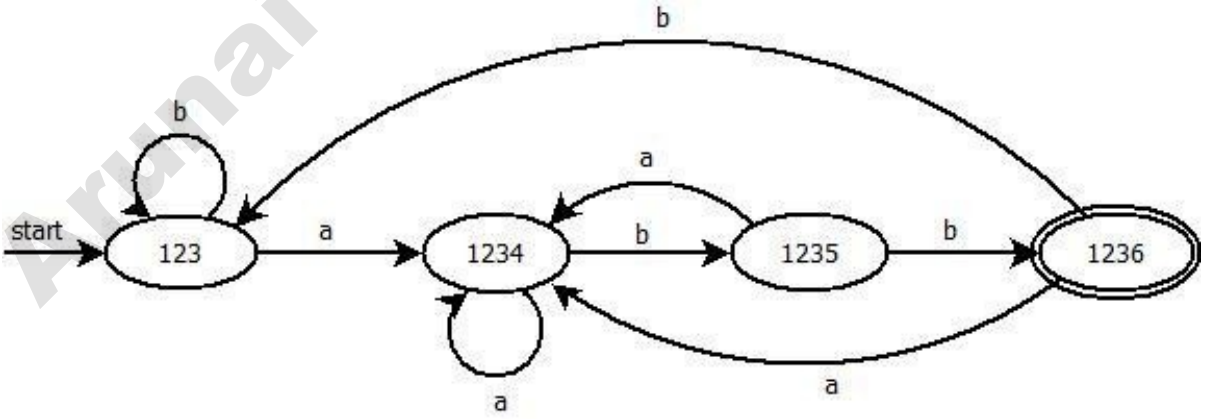


FollowPos

Node n	followpos(n)
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	ø

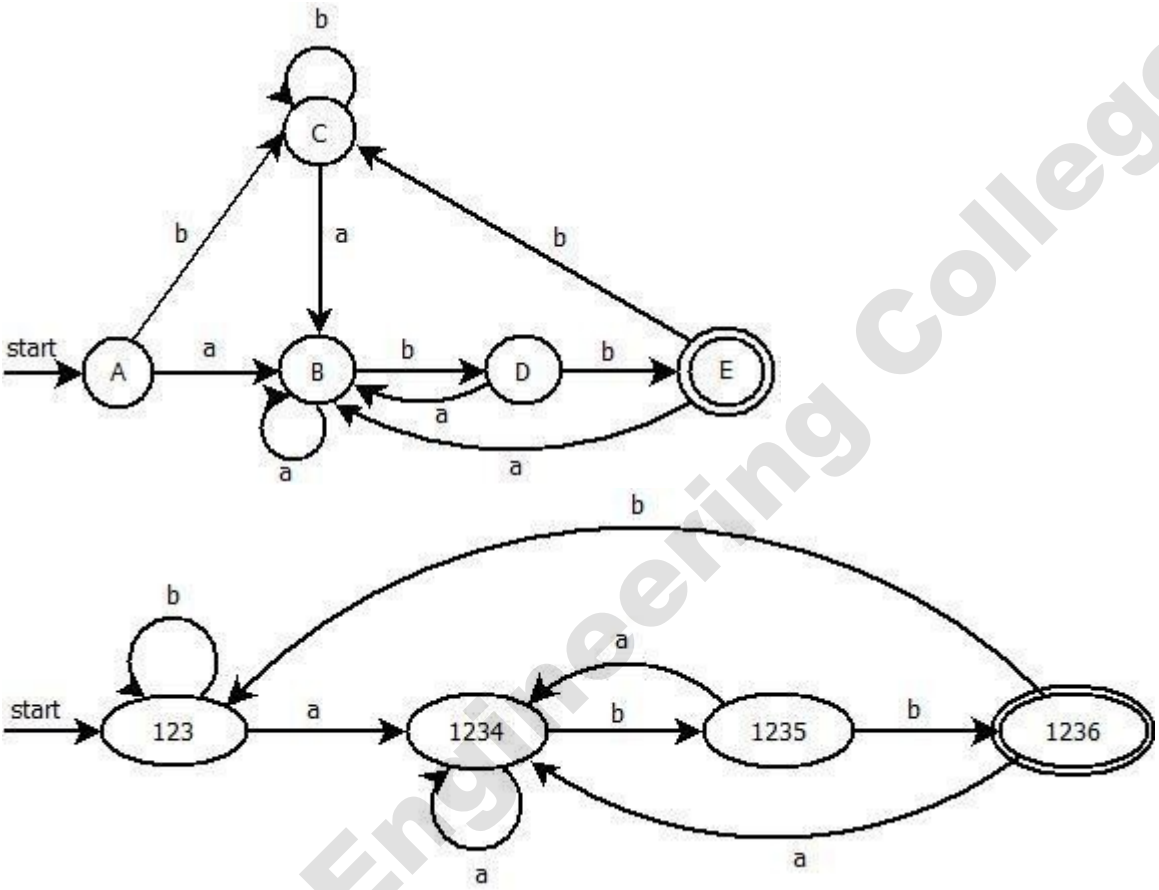
A=firstpos(n_0)={1,2,3}
Dtran[A,a]=
followpos(1) U followpos(3)= {1,2,3,4}=B
Dtran[A,b]=
followpos(2)={1,2,3}=A
Dtran[B,a]=
followpos(1) U followpos(3)=B
Dtran[B,b]=
followpos(2) U followpos(4)={1,2,3,5}=C

. . . .



Minimizing the Number of States of a DFA

Equivalent automata
{A, C}=123
{B}=1234
{D}=1235
{E}=1236
Exists a minimum state DFA



A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

There is a wide range of tools for constructing lexical analyzers.

- LEX
- YACC

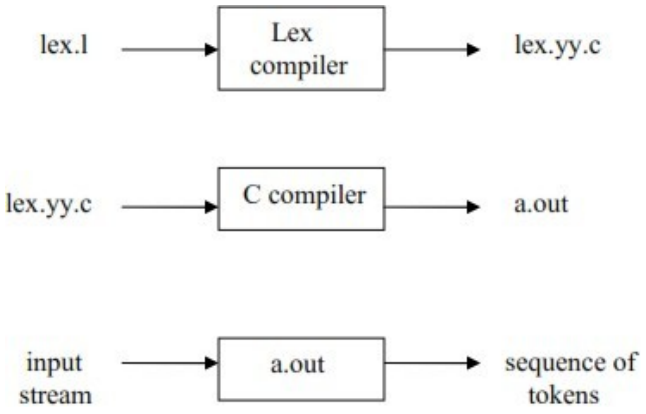
LEX

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.

Creating a lexical analyzer

First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.

Finally, lex.yy.c is run through the C compiler to produce an object progra m a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



Lex Specification

A Lex program consists of three parts:

{ definitions }
%%
{ rules }
%%
{ user subroutines }

- **Definitions** include declarations of variables, constants, and regular definitions
- **Rules** are statements of the form
  p_1 {action$_1$}
  p_2 {action$_2$}
  …
  p_n {action$_n$}
  where p_i is regular expression and action$_i$ describes what action the lexical analyzer should take when pattern p_i matches a lexeme. Actions are written in C code.
- **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

Example:

```
%{
 int v=0,c=0;
%}
%%
[aeiouAEIOU] v++;
[a-zA-Z] c++;
%%
main()
{
printf("ENTER INTPUT : \n");
yylex();
printf("VOWELS=%d\nCONSONANTS=%d\n",v,c);
}
```

UNIT-III

SYNTAX ANALYSIS

Need and Role of the Parser-Context Free Grammars -Top Down Parsing -General Strategies- Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item-Construction of SLR Parsing Table -Introduction to LALR Parser - Error Handling and Recovery in Syntax Analyzer-YACC-Design of a syntax Analyzer for a Sample Language .

SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

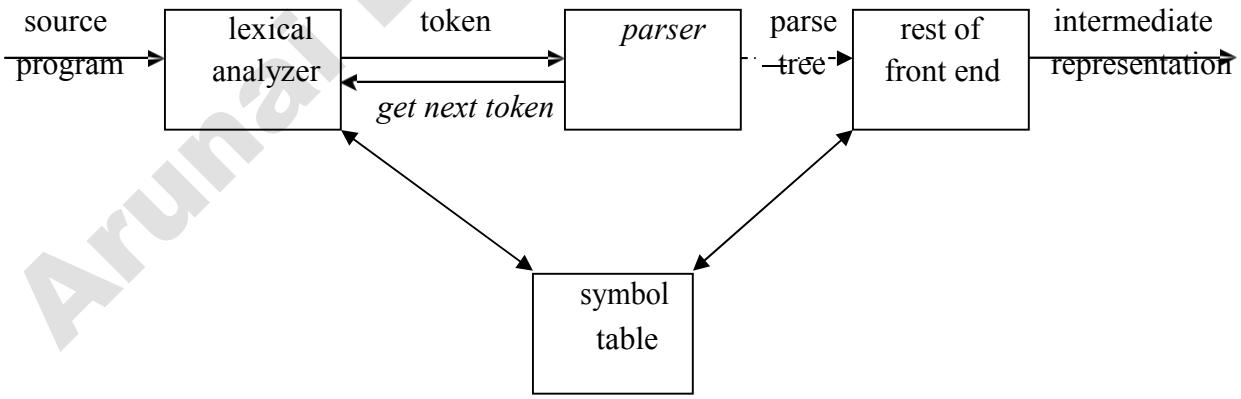
Advantages of grammar for syntactic specification:

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

THE ROLE OF PARSER

The parser or syntactic analyzer obtains a 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 the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Position of parser in compiler model



Functions of the parser:

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

Issues:

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals, start symbol** and **productions.**

Terminals : These are the basic symbols from which strings are formed.

Non-Terminals : These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

Start Symbol : One non-terminal in the grammar is denoted as the "Start-symbol" and the set of strings it denotes is the language defined by the grammar.

Productions : It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

Example of context-free grammar: The following grammar defines simple arithmetic expressions:

expr → *expr op expr*
expr → (*expr*)
expr → - *expr*
expr → **id**
op → +
op → -
op → *
op → /
op → ↑

In this grammar,

- **id** + - * / ↑ ( ) are terminals.
- *expr* , *op* are non-terminals.
- *expr* is the start symbol.
- Each line is a production.

Derivations:

Two basic requirements for a grammar are :
1. To generate a valid string.
2. To recognize a valid string.

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

Example : Consider the following grammar for arithmetic expressions :

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

To generate a valid string - (id+id) from the grammar the steps are

1. $E \rightarrow - E$
2. $E \rightarrow - (E)$
3. $E \rightarrow - (E+E)$
4. $E \rightarrow - (id+E)$
5. $E \rightarrow - (id+id)$

In the above derivation,

- E is the start symbol.
- - (id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

Types of derivations:

The two types of derivation are:

1. Left most derivation
2. Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.

- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

Example:

Given grammar G : E → E+E | E*E | (E) | - E | id

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	**RIGHTMOST DERIVATION** E
→ - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E) E
→ - (id+E)	E → - (E+id) E
→ - (id+id)	E → - (id+id)

- String that appear in leftmost derivation are called **left sentinel forms.**
- String that appear in rightmost derivation are called **right sentinel forms.**

Sentinels:

Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

Yield or frontier of tree:

Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called **yield** or **frontier** of the tree.

Ambiguity:

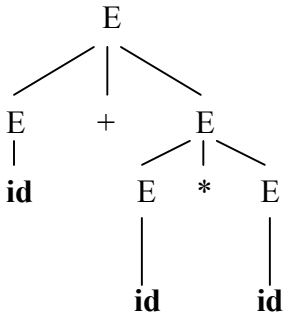
A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

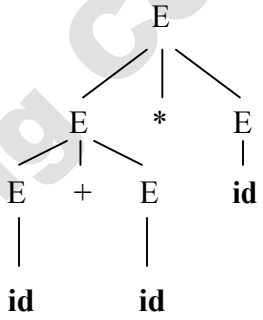
Example : Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

The two corresponding parse trees are :





WRITING A GRAMMAR

There are four categories in writing a grammar :

1. Regular Expression Vs Context Free Grammar
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable.

Regular Expressions vs. Context-Free Grammars:

REGULAR EXPRESSION	CONTEXT-FREE GRAMMAR
It is used to describe the tokens of programming languages.	It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non- terminal.
It is used to check whether the given input is valid or not using **transition diagram.**	It is used to check whether the given input is valid or not using **derivation**.
The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions 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.
- Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

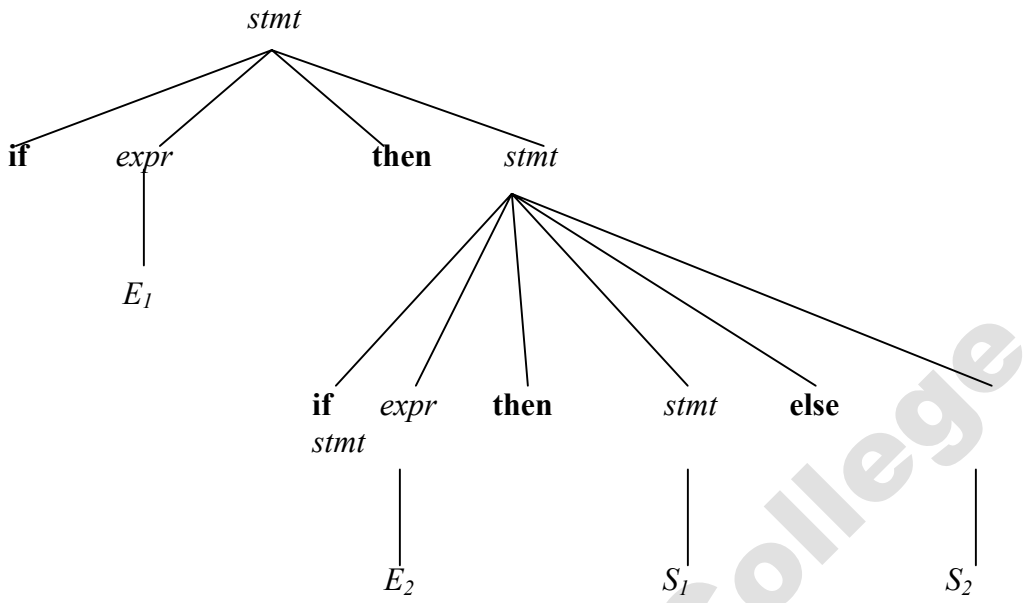
Eliminating ambiguity:

Ambiguity of the grammar that produces more than one parse tree for leftmost or ri ghtmost derivation can be eliminated by re-writing the grammar.

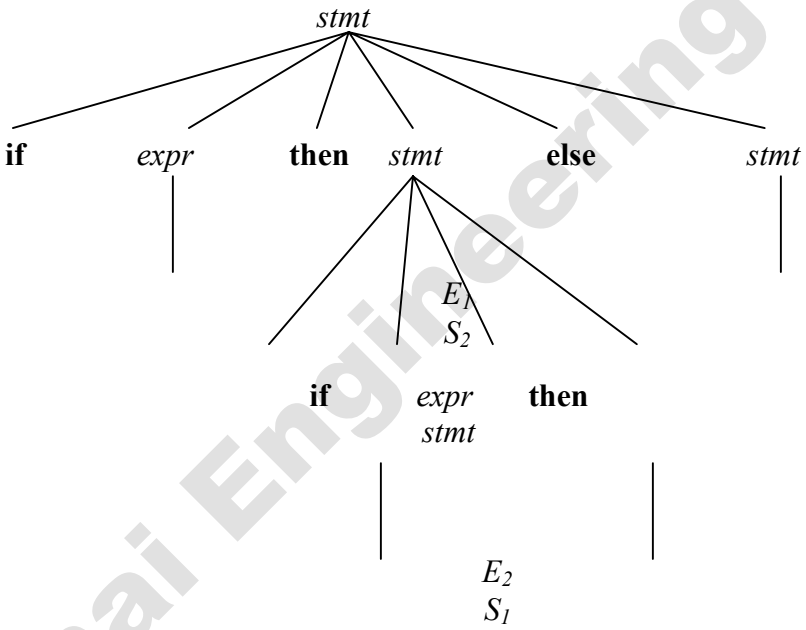
Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

This grammar is ambiguous since the string **if E_1 then if E_2 then S_1 else S_2** has the following two parse trees for leftmost derivation :

1.



2.



To eliminate ambiguity, the following grammar may be used:

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 said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα | β it can be replaced with a sequence of two productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

without changing the set of strings derivable from A.

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.
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 $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$
            where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j-productions;
    end
    eliminate the immediate left recursion among the A_i-productions
end

Example : Consider the following grammar for arithmetic expressions: E

→ E+T | T

T → T*F | F F

→ (E) | id

First eliminate the left recursion for E as

E → TE'

E' → +TE' | ε

Then eliminate for T as

T → FT' T'→

*FT' | ε

Thus the obtained grammar after eliminating left recursion is

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Left factoring:

Left factoring is a grammar 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 can rewrite the A-productions to defer the decision until we have seen enough of the input to makethe right choice.

If there is any production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, it can be rewritten as

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Consider the grammar , G : S → iEtS | iEtSeS | a
E → b

Left factored, this grammar becomes

S → iEtSS' | a
S' → eS | ε
E → b

PARSING

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

Types of parsing:

1. Top down parsing
2. Bottom up parsing

- Top–down parsing : A parser can start with the start symbol and try to transform it to the input string.
  Example : LL Parsers.
- Bottom–up parsing : A parser can start with input and attempt to rewrite it into the start symbol.
  Example : LR Parsers.

TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING

- Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.**
- This parsing method may involve **backtracking**, that is, making repeated scans of the input.

Example for backtracking :

Consider the grammar G : S → cAd

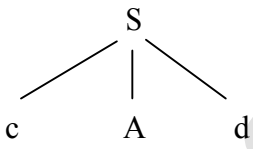
A → ab | a

and the input string w=cad.

The parse tree can be constructed using the following top-down approach :

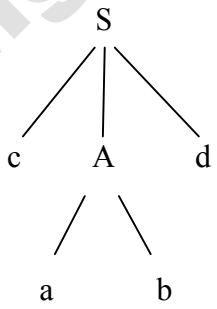
Step1:

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 second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



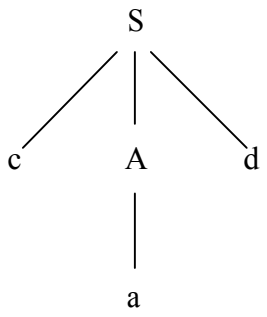
Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'**.** But the third leaf of tree is b which does not match with the input symbol **d.**

Hence discard the chosen production and reset the pointer to second position. This is called **backtracking.**

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Example for recursive decent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating the left-recursion the grammar becomes,

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Now we can write the procedure for grammar as follows:

Recursive procedure:

```
Procedure E()
begin
        T( );
        EPRIME( );
end
```

```
Procedure EPRIME( )
begin
        If input_symbol='+' then
        ADVANCE( );
        T( );
        EPRIME( );
end

Procedure T( )
begin
        F( );
        TPRIME( );
end

Procedure TPRIME( )
begin
        If input_symbol='*' then
        ADVANCE( );
        F( );
        TPRIME( );
end

Procedure F( )
begin
        If input-symbol='id' then
        ADVANCE( );
        else if input-symbol='(' then
        ADVANCE( );
        E( );
        else if input-symbol=')' then
        ADVANCE( );
end

else ERROR( );
```

Stack implementation:

To recognize input **id+id*id :**

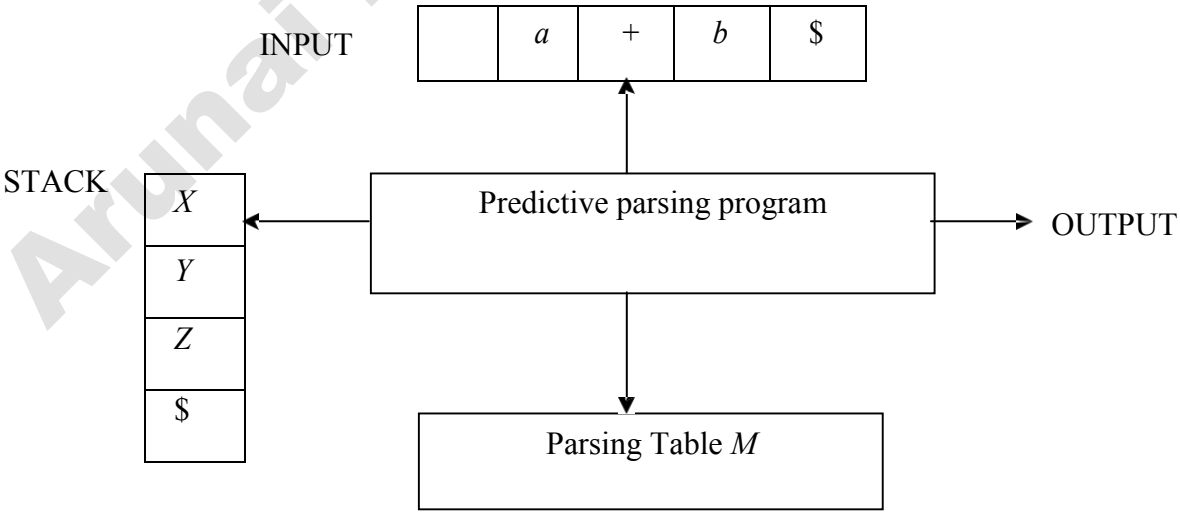
PROCEDURE	INPUT STRING
E( )	**id**+id*id
T( )	**id**+id*id
F( )	**id**+id*id
ADVANCE( )	id+id*id

TPRIME( )	id+id*id
EPRIME( )	id+id*id
ADVANCE( )	id+**id***id
T( )	id+**id***id
F( )	id+**id***id
ADVANCE( )	id+id***id
TPRIME( )	id+id***id
ADVANCE( )	id+id***id**
F( )	id+id***id**
ADVANCE( )	id+id***id**
TPRIME( )	id+id***id**

2. PREDICTIVE PARSING

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Non-recursive predictive parser



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

Parsing table:

It is a two-dimensional array $M[A, a]$, where **'*A*'** is a non-terminal and **'*a*'** is a terminal.

Predictive parsing program:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If $X = a =$ $, the parser halts and announces successful completion of parsing.
2. If $X = a \neq$ $, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal , the program consults entry $M[X, a]$ of the parsing table M. This entry will either be an X-production of the grammar or an error entry.
   If $M[X, a] = \{X \rightarrow UVW\}$,the parser replaces X on top of the stack by WVU.
   If $M[X, a]$ = **error**, the parser calls an error recovery routine.

Algorithm for nonrecursive predictive parsing:

Input : A string w and a parsing table M for grammar G.

Output : If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

Method : Initially, the parser has $\$S$ on the stack with S, the start symbol of G on top, and $w\$$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is as follows:

```
set ip to point to the first symbol of w$;
repeat
        let X be the top stack symbol and a the symbol pointed to by ip;
        if X is a terminal or $ then
            if X = a then
                pop X from the stack and advance ip
            else error()
        else            /* X is a non-terminal */
            if M[X, a] = X → Y1Y2 … Yk then begin
```

pop X from the stack;
push Y_k, Y_{k-1}, … ,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 = $ /* stack is empty */

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST

2. FOLLOW

Rules for first( ):

1. If X is terminal, then FIRST(X) is {X}.

2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).

3. If X is non-terminal and X → $a\alpha$ is a production then add a to FIRST(X).

4. If X is 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}); that is, $Y_1,\ldots Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST($Y_j$) for all j=1,2,..,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).

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

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 → α 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 be **error**.

Example:

Consider the following grammar :

E → E+T | T
T → T*F | F
F → (E) | id

After eliminating left-recursion the grammar is

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

First( ) :

FIRST(E) = { (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

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

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

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

Predictive parsing table :

NON-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)		

Stack implementation:

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

LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

S → iEtS | iEtSeS | a
E → b

After eliminating left factoring, we have

S → iEtSS' | a
S'→ eS | ε
E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) = { i, a }

FIRST(S') = {e, ε }

FIRST(E) = { b}

FOLLOW(S) = { $,e }

FOLLOW(S') = {

$,e }

FOLLOW(E) =

{t}

Parsing table:

NON-TERMINAL	A	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

Since there are more than one production, the grammar is not LL(1) grammar.

Actions performed in predictive parsing:

1. Shift
2. Reduce
3. Accept
4. Error

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.

BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.

A general type of bottom-up parser is a **shift-reduce parser**.

SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:
Consider the
grammar: S →
aABe
A → Abc | b
B → d
The sentence to be recognized is **abbcde.**

REDUCTION (LEFTMOST)		**RIGHTMOST DERIVATION**
a**b**bcde	(A → b)	**S** → aA**B**e
a**Abc**de	(A → Abc)	→ a**A**de
aA**d**e	(B → d)	→ a**A**bcde
aABe	(S → aABe)	→ abbcde
S		

The reductions trace out the right-most derivation in reverse.

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 left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

E → E+E
E → E*E
E → (E)
E → id

And the input string $id_1+id_2*id_3$

The rightmost derivation is :

E → **E+E**
→ E+**E*E**
→ E+E***id**$_3$
→ E+**id**$_2$*id$_3$
→ **id**$_1$+id$_2$*id$_3$

In the above derivation the underlined substrings are called **handles.**

Handle pruning:

A rightmost derivation in reverse can be obtained by "**handle pruning**".
(i.e.) if *w* is a sentence or string of the grammar at hand, then $w = \gamma_n$, where γ_n is the n^{th} right-sentinel form of some rightmost derivation.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
$	id$_1$+id$_2$*id$_3$ $	shift
$ id$_1$	+id$_2$*id$_3$ $	reduce by E→id
$ E	+id$_2$*id$_3$ $	shift
$ E+	id$_2$*id$_3$ $	shift
$ E+id$_2$	*id$_3$ $	reduce by E→id
$ E+E	*id$_3$ $	shift
$ E+E*	id3 $	shift
$ E+E*id3	$	reduce by E→id
$ E+E*E	$	reduce by E→ E *E
$ E+E	$	reduce by E→ E+E
$ E	$	accept

Actions in shift-reduce parser:

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.

2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

1. Shift-reduce conflict:

Example:

Consider the grammar:

E→E+E | E*E | id and input id+id*id

Stack	Input	Action	Stack	Input	Action
$ E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift
$ E	*id $	Shift	$E+E*	id $	Shift
$ E*	id $	Shift	$E+E*id	$	Reduce by E→id
$ E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$ E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$ E			$E		

2. Reduce-reduce conflict:

Consider the grammar:

M → R+R | R+c |
R R → c
and input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c $	Shift	$	c+c $	Shift
$ c	+c $	Reduce by R→c	$ c	+c $	Reduce by R→c
$ R	+c $	Shift	$ R	+c $	Shift
$ R+	c $	Shift	$ R+	c $	Shift
$ R+c	$	Reduce by R→c	$ R+c	$	Reduce by M→R+c
$ R+R	$	Reduce by M→R+R	$ M	$	
$ M	$				

Viable prefixes:

- α is a viable prefix of the grammar if there is *w* such that α*w* is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

OPERATOR-PRECEDENCE PARSING

An efficient way of constructing shift-reduce parser is called operator-precedence parsing.

Operator precedence parser can be constructed from a grammar called Operator-grammar. These grammars have the property that no production on right side is ε or has two adjacent non - terminals.

Example:

Consider the grammar:

E → EAE | (E) | -E | id
A → + | - | * | / | ↑

Since the right side EAE has three consecutive non-terminals, the grammar can be written as follows:

E → E+E | E-E | E*E | E/E | E↑E | -E | id

Operator precedence relations:

There are three disjoint precedence relations namely

$<\cdot$	-	less than
=	-	equal to
$\cdot>$	-	greater than

The relations give the following meaning:

$a <\cdot b$ – a yields precedence to b
a = b – a has the same precedence as b
$a \cdot > b$ – a takes precedence over b

Rules for binary operations:

1. If operator θ_1 has higher precedence than operator θ_2, then make
   $\theta_1 \cdot > \theta_2$ and $\theta_2 <\cdot \theta_1$

2. If operators θ_1 and θ_2, are of equal precedence, then make
   $\theta_1 \cdot > \theta_2$ and $\theta_2 \cdot > \theta_1$ if operators are left associative
   $\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$ if right associative

3. Make the following for all operators θ:
   $\theta <\cdot$ id , id $\cdot >$ θ
   $\theta <\cdot$ (, ($<\cdot$ θ
) $\cdot >$ θ , θ $\cdot >$)
   θ $\cdot >$ $, $ $<\cdot$ θ

Also make
(=) , ($<\cdot$ (,) $\cdot>$) , ($<\cdot$ id , id $\cdot>$) , \$ $<\cdot$ id , id $\cdot>$ \$, \$ $<\cdot$ (,) $\cdot>$ \$

Example:

Operator-precedence relations for the grammar

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id is given in the following table assuming

1. ↑ is of highest precedence and right-associative
2. * and / are of next higher precedence and left-associative, and
3. + and - are of lowest precedence and left-associative

Note that the **blanks** in the table denote error entries.

TABLE : Operator-precedence relations

	+	-	*	/	↑	id	(	)	$
+	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
-	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
*	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
/	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
↑	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
id	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$			$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	=	
)	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$			$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$		

Operator precedence parsing algorithm:

Input : An input string *w* and a table of precedence relations.
Output : If *w* is well formed, a *skeletal* parse tree **,**with a placeholder non-terminal E labeling all interior nodes; otherwise, an error indication.
Method : Initially the stack contains \$ and the input buffer the string *w* \$. To parse, we execute the following program :

(1) Set *ip* to point to the first symbol of *w*\$;
(2) repeat forever
(3) if \$ is on top of the stack and *ip* points to \$ **then**
(4) return
else begin
(5) let *a* be the topmost terminal symbol on the stack
and let *b* be the symbol pointed to by *ip;*
(6) if $a <\cdot b$ or $a = b$ **then begin**
(7) push *b* onto the stack;
(8) advance *ip* to the next input symbol;
end;

```
(9)     else if a ·> b then              /*reduce*/
(10)        repeat
(11)            pop the stack
(12)        until the top stack terminal is related by <·
                to the terminal most recentlypopped
(13)    else error( )
        end
```

Stack implementation of operator precedence parsing:

Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.

The initial configuration of an operator precedence parsing is

STACK	INPUT
$	w $

where w is the input string to be parsed.

Example:

Consider the grammar E → E+E | E-E | E*E | E/E | E↑E | (E) | id. Input string is **id+id*id** .The implementation is as follows:

STACK	INPUT	COMMENT
$	<· id+id*id $	shift id
$ id	·> +id*id $	pop the top of the stack id
$	<· +id*id $	shift +
$ +	<· id*id $	shift id
$ +id	·> *id $	pop id
$ +	<· *id $	shift *
$ + *	<· id $	shift id
$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

Advantages of operator precedence parsing:

1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar , the grammar can be ignored. The grammar is not referred anymore during implementation.

Disadvantages of operator precedence parsing:

1. It is hard to handle tokens like the minus sign (-) which has two different precedence.
2. Only a small class of grammar can be parsed using operator-precedence parser.

LR PARSERS

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(*k*) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the '*k*' for the number of input symbols. When '*k*' is omitted, it is assumed to be 1.

Advantages of LR parsing:

- ✓ It recognizes virtually all programming language constructs for which CFG can be written.
- ✓ It is an efficient non-backtracking shift-reduce parsing method.
- ✓ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- ✓ It detects a syntactic error as soon as possible.

Drawbacks of LR method:

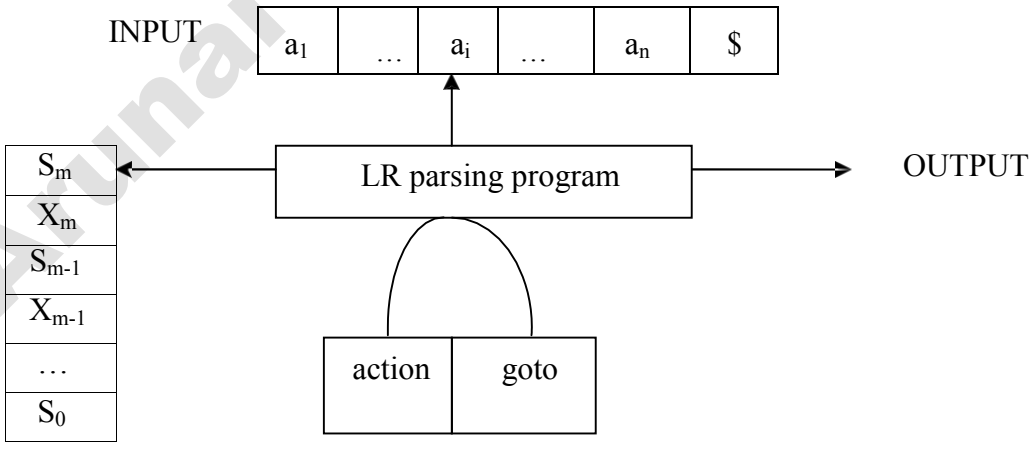
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR
   - Easiest to implement, least powerful.
2. CLR- Canonical LR
   - Most powerful, most expensive.
3. LALR- Look-Ahead LR
   - Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:



STACK

It consists of : an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*).

- The driver program is the same for all LR parser.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines s_m, the state currently on top of stack, and a_i, the current input symbol. It then consults *action*$[s_m,a_i]$ in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production $A \rightarrow \beta$,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

LR Parsing algorithm:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and *w*\$ in the input buffer. The parser then executes the following program :

```
set ip to point to the first input symbol of w$;
repeat forever begin
    let s be the state on top of the stack and
        a the symbol pointed to by ip;
    if action[s, a] = shift s' then begin
        push a then s' on top of the stack;
        advance ip to the next input symbol
    end
    else if action[s, a] = reduce A→β then begin
        pop 2* | β | symbols 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→ β
    end
    else if action[s, a] = accept then
        return
    else error( )
end
```

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 set of items and X is grammar symbol.

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. For example, production A → XYZ yields the four items :

A → **.** XYZ
A → X **.** YZ
A → XY **.** Z
A → XYZ **.**

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 two rules:

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such that [A→ α . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

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

<u>Algorithm for construction of SLR parsing table:</u>

Input : An augmented grammar G'
Output : The SLR parsing table functions *action* and *goto* for G'
Method :

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 functions for state *i* are determined as follows:
   (a) If [A→α·*a*β] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If [A→α·] is in I_i , then set *action*[*i*,*a*] to "reduce A→α" for all *a* in FOLLOW(A).
   (c) If [S'→S.] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If *goto*(I_i,A) = I_j, then *goto*[i,A] = *j*.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→.S].

Example for SLR parsing:

Construct SLR parsing for the following grammar :

G : E → E + T | T
T → T * F | F
F → (E) | id

The given grammar is :

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

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

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

Step 2 : Find 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 (I_4 , id)
I_5 : F → id **.**

GOTO (I_0 , T)
I_2 : E → T .
T → T . * F

GOTO (I_0 , F)
I_3 : T → F .

GOTO (I_0 , ( )
I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_0 , id)
I_5 : F → id .

GOTO (I_1 , +)
I_6 : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_2 , *)
I_7 : T → T * . F
F → . (E)
F → . id

GOTO (I_4 , E)
I_8 : 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_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)

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

GOTO (I_4 , ( )

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

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

Blank entries are error entries.

Stack implementation:

Check whether the input **id + id * id** is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ; **shift**
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ; **reduce** by F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ; **reduce** by T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ; **reduce** by E → T
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ; **shift**
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ; **shift**
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ; **reduce** by F → id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ; **reduce** by T → F
0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ; **shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ; **shift**
0 E 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ; **reduce** by F → id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ; **reduce** by T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ; **reduce** by E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) = **accept**

Building LR(1) itemsets, LR(1) and LALR parse tables

A, S, X: non-terminals
x,y, a, ß: string of terminals and/or non-terminals
C: one terminal or one non-terminal
Start: [S --> . w , $] is the item associated with the start state.
Read: Starting a new state (reading on one terminal or non-terminal, C) comes from
[A --> x.Cy , w] then new state includes [A --> xC.y , w] .
Complete: if [A --> x . X a , u] is an item, then completing on X gives the item(s) [X --> .ß , z] where z ϵ FIRST(au)
Consider the augmented grammar G':

0. S' --> S$
1. S --> CC
2. C --> eC
3. C --> d

LR(1) Itemsets

State	Item	Notes
I_0	S' --> .S$, $	complete on S; read on S goes to state 1
	S --> .CC , $	complete on C; FIRST($$) is $; read on C goes to state 2
	C --> .eC , e|d	FIRST(C$) is e|d ; read on 'e' goes to state 3
	C --> .d , e|d	FIRST(C$) is e|d ; read on 'd' goes to state 4
I_1	S' --> S.$, $	accept
I_2	S --> C.C , $	read on C goes to state 5
	C --> .eC , $	FIRST(λ$) is $; read on 'e' goes to state 6
	C --> .d , $	FIRST(λ$) is $; read on 'd' goes to state 7
I_3	C --> e.C , e|d	read on C goes to 8
	C --> .eC , e|d	FIRST(λ(e|d)) is e|d; read on 'e' is to state 3 again
	C --> .d , e|d	FIRST(λ(e|d)) is e|d; read on 'd' is to state 4 again
I_4	C --> d. , e|d	reduce on rule 3
I_5	S --> CC. , $	reduce on rule 1
I_6	C --> e.C , $	read on C goes to 9
	C --> .eC , $	FIRST(λ$) is $; read on 'e' is to state 6 again
	C --> .d , $	FIRST(λ$) is $; read on 'd' is to state 7 again
I_7	C --> d. , $	reduce on rule 3
I_8	S --> eC. , e|d	reduce on rule 2
I_9	S --> eC. , $	reduce on rule 2

The SLR parse table:

	e	d	$		S	C
0	s3	s4			1	2
1			accept			
2	s3	s4				5
3	s3	s4				6
4	r3	r3	r3			
5			r1			
6	r2	r2	r2			

The LR(1) parse table (same as before, except when you do a reduce – items with dot at end – instead of using the whole FOLLOW set, only use symbols after the comma):

	e	d	$		S	C

To create LALR table, merge states by their core sets (for state use either just first number, or use all the original state numbers to create a unique longer number):

	e	d	$		S	C
0	s36	s47			1	2
1			accept			
2	s36	s47				5
36	s36	s47				89
47	r3	r3	r3			
5			r1			
89	r2	r2	r2			

To create LALR table, merge states by their core sets (for state use either just first number, or use all the original state numbers to create a unique longer number):

	e	d	$		S	C
0	s36	s47			1	2
1			accept			
2	s36	s47				5
36	s36	s47				89
47	r3	r3	r3			
5			r1			
89	r2	r2	r2			

Syntax error handling :

Programs can contain errors at many different levels. For example :
1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Functions of error handler :

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

Error recovery strategies :

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Panic mode recovery:

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

Phrase level recovery:

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

Error productions:

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

Global correction:

Given an incorrect input string x and grammar G, certain algorithms 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.

YACC-Design of a syntax Analyzer for a Sample Language

- Yacc is a tool for constructing parsers.
- It reads a specification file that codifies the grammar of a language and generates a parsing routine.
- Yacc specification describes a CFG, that can be used to generate a parser.
- Elements of a CFG:
  1. Terminals: tokens and literal characters,
  2. Variables (nonterminals): syntactical elements,
  3. Production rules, and
  4. Start rule.

Skeleton of a yacc specification (.y file)

```
translate.y
%{
< C global variables,prototypes,
    comments >
%}
[DEFINITION SECTION]
%%
[PRODUCTION RULES SECTION]
%%

< C auxiliary subroutines>
```

```
y.tab.c is generated after running

  This part will be embedded into
 y.yab.c

contains token declarations.
 Tokens are recognized in lexer.

define how to "understand" the
 input language, and what actions
 to take for each "sentence".

Any user code. For example, a
 main function to call the parser
 function yyparse()
```

Example:

A -> Bc is written in yacc as a: b 'c';

Format of a yacc specification file:

declarations

%%

grammarrules and associatedactions

Declarations:

To define tokens and their characteristics

%token:	declare names of tokens
%left:	define left-associative operators
%right:	define right-associative operators
%nonassoc:	define operators that may not associate with themselves
%type:	declare the type of variables
%union:	declare multiple data types for semantic values
%start:	declare the start symbol (default is the first variable in rules)
%prec:	assign precedence to a rule

```
%{
      C declarations     directly copied to the resulting C program
%}                                  (E.g., variables, types, macros…)
```

Eg:Yacc program to recognize L = {$a^n b^n$ | *n* >=0}.

```
%{
#include<stdio.h>
int valid=1;
%}
%token A B
%%
str:S'\n' {return 0;}
S:A S B
 |
 ;
%%
main()
{
   printf("Enter the string:\n");
   yyparse();
   if(valid==1)
   printf("\nvalid string");
}
```

UNIT IV- SYNTAX DIRECTEDTRANSLATION & RUN TIME ENVIRONMENT

Syntax directed Definitions-Construction of Syntax Tree-Bottom-up Evaluation of S-Attribute Definitions- Design of predictive translator - Type Systems-Specification of a simple type checker-Equivalence of Type Expressions-Type Conversions. RUN-TIME ENVIRONMENT: Source Language Issues-Storage Organization-Storage Allocation-Parameter Passing-Symbol Tables-Dynamic Storage Allocation-Storage Allocation in FORTAN.

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*.

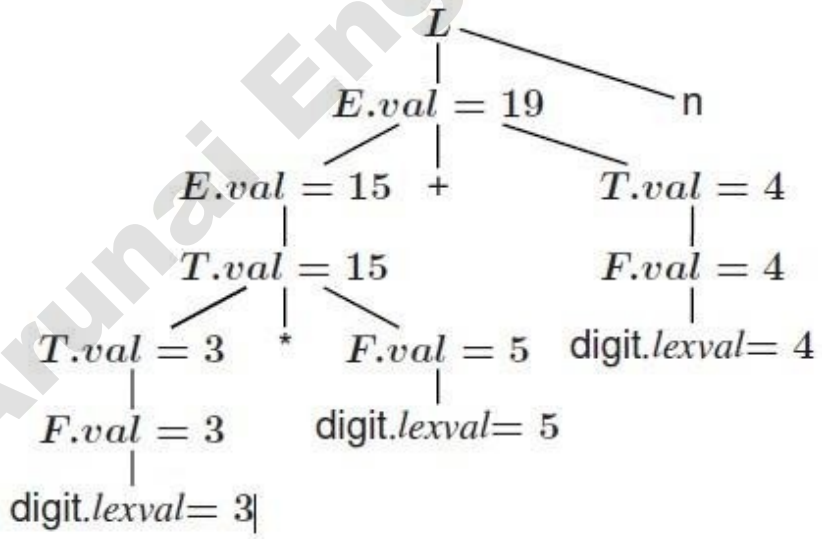
PRODUCTION	SEMANTIC RULE
$L \rightarrow E\text{n}$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \text{digit}$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

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 bya 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).

SDD For Simple Type Declarations

Production	Semantic Rules
1) $D \rightarrow T\ L$	*L.inh* = *T.type*
2) $T \rightarrow$ **int**	*T.type* = integer
3) $T \rightarrow$ **float**	*T.type* = float
4) $L \rightarrow L_1$, **id**	L_1.*inh* = *L.inh* *addType* (**id**.*entry*, *L.inh*)
5) $L \rightarrow$ **id**	*addType* (**id**.*entry*, *L.inh*)

CONSTRUCTION OF SYNTAX TREE

- SDDs are useful for is construction of syntax trees. A syntax tree is a condensed form of parse tree.
- Syntax trees are useful for representing programming language constructs like expressions and statements.
- They help compiler design by decoupling parsing from translation.
- Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
  e.g. a syntax-tree node representing an expression E1 + E2 has label + and two children representing the sub expressions E1 and E2
- Each node is implemented by objects with suitable number of fields; each object will have an op field that is the label of the node with additional fields as follows:
  i) If the node is a leaf, an additional field holds the lexical value for the leaf . This is created by function Leaf(op, val)
  ii) If the node is an interior node, there are as many fields as the node has children in the syntax tree. This is created by function Node(op, c1, c2,...,ck) .
- Example: The S-attributed definition in figure below constructs syntax trees for a simple expression grammar involving only the binary operators + and -. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute node, which represents a node of the syntax
  tree.

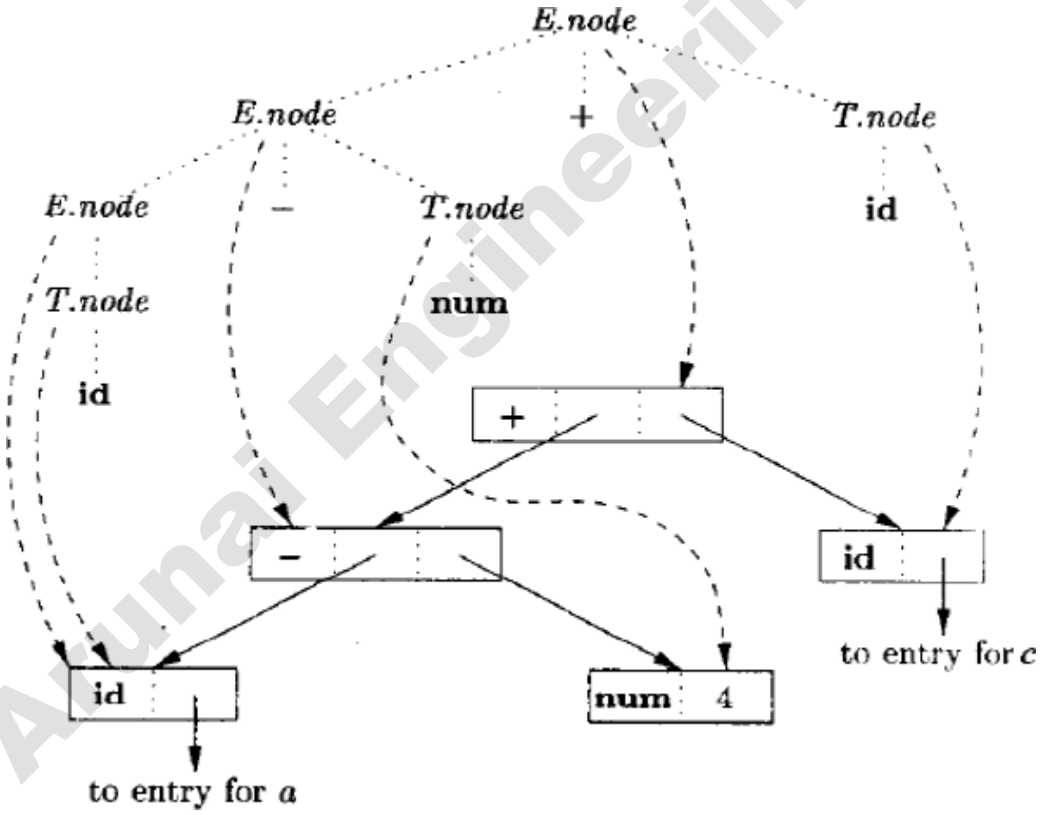
Production	Semantic Rules
1) $E \rightarrow E_1 + T$	*E.node* = **new** *Node* ('+', E_1.*node*, *T.node*)
2) $E \rightarrow E_1 - T$	*E.node* = **new** *Node* ('–', E_1.*node*, *T.node*)
3) $E \rightarrow T$	*E.node* = *T.node*
4) $T \rightarrow (E)$	*E.node* = *T.node*
5) $T \rightarrow$ **id**	*T.node*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

Steps in the construction of the syntax tree for a-4+c

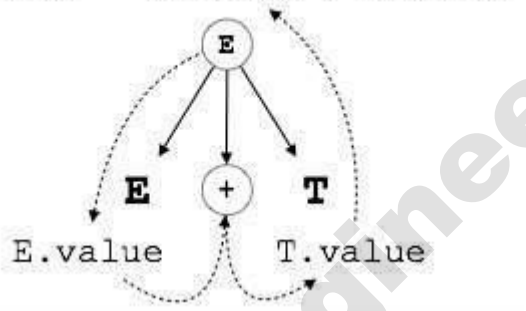
If the rules are evaluated during a post order traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown below ends with p5 pointing to the root of the constructed syntax tree.

1) p_1 = **new** *Leaf* (**id**, *entry-a*);
2) p_2 = **new** *Leaf* (**num**, 4);
3) p_3 = **new** *Node* ('−', p_1, p_2);
4) p_4 = **new** *Leaf* (**id**, *entry-c*);
5) p_5 = **new** *Node* ('+', p_3, p_4);

Syntax tree for a-4+c using the above SDD is shown below.



Bottom-up Evaluation of S-Attribute Definitions

- Syntax-directed definition with only **synthesized** attributes is called S-attributed
- Use LR Parser
- Implementation:
- Stack to hold info about subtrees that have been parsed
- A SDD is a context free grammar with attributes and rules
- Attributes are associated with grammar symbols and rules with productions
- Attributes may be of many kinds: numbers, types, table references, strings, etc.
- Synthesized attributes
    - A synthesized attribute at node N is defined only in terms of attribute values of children of N and at N it
- If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These `E.value = E.value + T.value` s that have their semantic actions written
- 

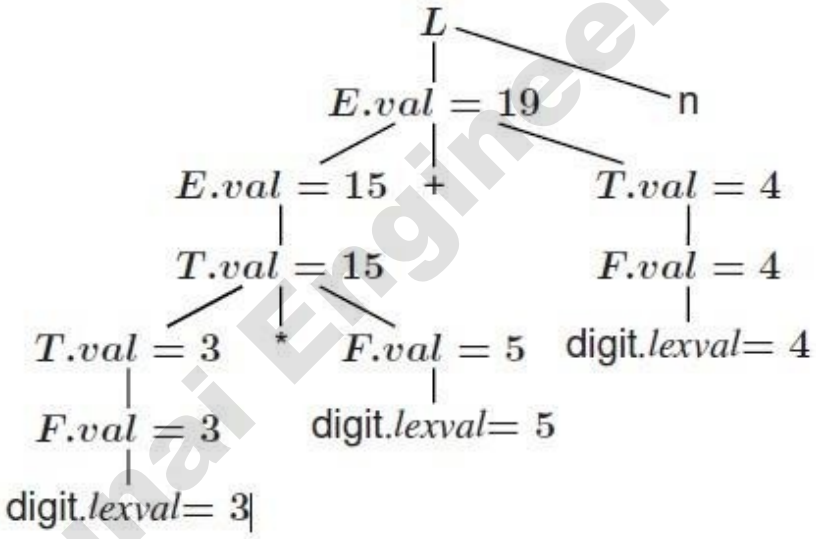
- As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

Syntax Directed Definitions: An Example

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

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

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



Input	Stack	Attribute	Production Used
3 * 5 + 4 $	-	-	
* 5 + 4 $	3	3	
* 5 + 4 $	F	3	F ---> digit
* 5 + 4 $	T	3	T ---> F
5 + 4 $	T *	3	
+ 4 $	T * 5	3 * 5	
+ 4 $	T * F	3 * 5	F -→ digit
+ 4 $	T	15	T ---> T * F
+ 4 $	E	15	E ---> T
4 $	E +	15	
$	E + 4	15 + 4	
$	E + F	15 + 4	F ---> digit
$	E + T	15 4	T ---> F
$	E	19	E ---> E + T
	E	19	
	L	19	L ---> E $

TYPE CHECKING

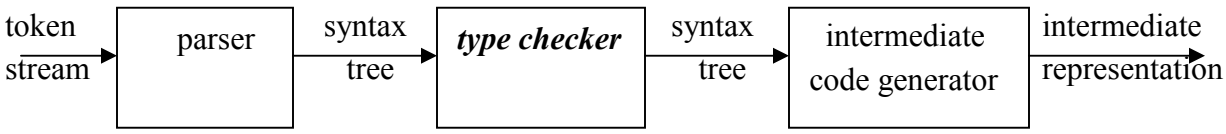
A compiler must check that the source program follows both syntactic and semantic conventions of the source language.
This checking, called *static checking,* detects and reports

programming errors. Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

2. **Flow-of-control checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as break, does not exist in switchstatement.

Position of type checker



- A ***type checker*** verifies that the type of a construct matches that expected by its context. For example : arithmetic operator *mod* in Pascal requires integer operands, so a type checker verifies that the operands of *mod* have type integer.
- Type information gathered by a type checker may be needed when code is generated.

TYPE SYSTEMS

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example : " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

- The type of a language construct will be denoted by a "type expression."
- A type expression is either a basic type or is formed by applying an operator called a ***type constructor*** to other type expressions.
- The sets of basic types and constructors depend on the language to be

checked.

The following are the definitions of type expressions:

1. Basic types such as *boolean, char, integer, real* are type expressions.

   A special basic type, *type_error* , will signal an error during type checking; *void* denoting "the absence of a value" allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression. Constructors include:
   Arrays : If T is a type expression then *array* (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

Products : If T1 and T2 are type expressions, then their Cartesian product T1 X T2 is a type expression.

Records : The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types.

For example:

```
type row = record
                address: integer;
                lexeme: array[1..15] of char
            end;
    var table: array[1...101] of row;
```

declares the type name *row* representing the type expression ***record((address X integer) X (lexeme X array(1..15,char)))*** and the variable *table* to be an array of records of this type.

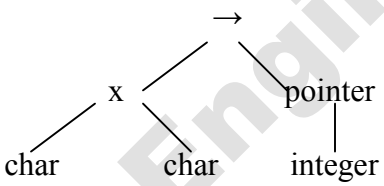
Pointers : If T is a type expression, then *pointer*(T) is a type expression denoting the type "pointer to an object of type T".

For example, ***var p: ↑ row*** declares variable p to have type *pointer*(row).

Functions : A function in programming languages maps a *domain type D* to a *range type R*. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.

Tree representation for char x char → pointer (integer)



Type systems

- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system. It is specified in a syntax-directedmanner.
- Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

- Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic.
- Any check can be done dynamically, if the target code carries the type of an element

along with the value of that element.

Sound type system

A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type_error* to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

- Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of theinput.
- Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

SPECIFICATION OF A SIMPLE TYPE CHECKER

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

A Simple Language

Consider the following grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E
↑

Translation scheme:

P → D ; E
D → D ; D
D → id : T { *addtype* (id.*entry* , T.*type*) }
T → char { T.*type* : = char }
T → integer { T.*type* : = integer }
T → ↑ T1 { T.*type* : = pointer(T1.*type*)
}
T → array [num] of T1 { T.*type* : = array (1… num.val , T1.*type*) }

In the above language,
→ There are two basic types : char and integer ;

→ *type_error* is used to signal errors;
→ the prefix operator ↑ builds a pointer type. Example , ↑ **integer** leads to the type expression ***pointer (integer)***.

Type checking of expressions

In the following rules, the attribute *type* for E gives the type expression assigned to the expression generated by E.

1. E → **literal** { E.*type* : = *char* }
   E → **num** { E.*type* : = *integer* }
   Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2. E → **id** { E.*type* : = *lookup* (**id**.*entry*) }
   lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E1 **mod** E2 { *E.type* : = **if** *E1. type = integer* **and**
E2. type = integer **then** *integer*
else *type_error* }
The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type_error*.

4. E → E1 [E2] { *E.type* : = if *E2.type = integer* **and**
E1.type = array(s,t) **then** *t*
else *type_error* }
In an array reference E1 [E2] , the index expression E2 must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of E1.

5. E → E1 ↑ { *E.type* : = **if** *E1.type = pointer (t)* **then** *t*
else *type_error* }

The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type *t* of the object pointed to by the pointer E.

Type checking of statements

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type_error* is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:
S → **id** : = E { S.*type* : = **if id**.*type* = E.*type* **then** *void*
else *type_error* }

2. Conditional statement:
S → if E **then** S1 { S.*type* : = **if** E.*type* = *boolean* **then** S1.*type*
else *type_error* }

3. While statement:

S → while E do S1 { S.*type* : = **if** E.*type* = *boolean* **then** S1.*type*
else *type_error* }

4. Sequence of statements:

S → S1 ; S2 { S.*type* : = **if** S1.*type* = *void* and
S1.*type* = *void* **then** *void*
else *type_error* }

Type checking of functions

The rule for checking the type of a function application is :

E → E1 (E2) { E.*type* : = **if** E2.*type* = *s* **and**
E1.*type* = *s* → *t* **then** *t*
else *type_error* }

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

SOURCE LANGUAGE ISSUES

Procedures:

A *procedure definition* is a declaration that associates an identifier with a statement. The identifier is the *procedure name*, and the statement is the *procedure body*.

For example, the following is the definition of procedure named *readarray* :

```
procedure readarray;
var i : integer;
begin
    for i : = 1 to 9 do read(a[i])
end;
```

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

Activation trees:

An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

Control stack:

- A *control stack* is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.
- The contents of the control stack are related to paths to the root of the activation tree. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a name.
Declarations may be explicit, such as:

var i : integer ;

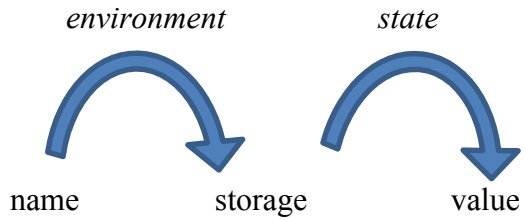
or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.

The portion of the program to which a declaration applies is called the ***scope*** of that declaration.

Binding of names:

Even if each name is declared once in a program, the same name may denote different data objects at run time. "Data object" corresponds to a storage location that holds values.

The term *environment* refers to a function that maps a name to a storage location.
The term *state* refers to a function that maps a storage location to the value held there.

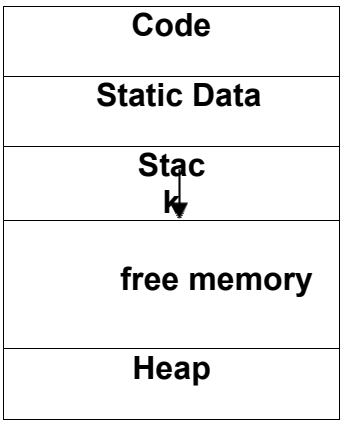


When an *environment* associates storage location *s* with a name *x*, we say that *x* is *bound* to *s*. This association is referred to as a *binding* of *x*.

STORAGE ORGANISATION

- The executing target program runs inits own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughoutmemory.

Typical subdivision of run-time memory:



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored inconsecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressingconstraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytesunused.
- This unused space due to alignment considerations is referred to aspadding.
- The size of some program objects may be known at run time and may be placed inan area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack.*
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.

- Temporary values such as those arising from the evaluation ofexpressions.
- Local data belonging to the procedure whose activation record thisis.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure butfound elsewhere.
- A control link pointing to the activation record of thecaller.

- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greaterefficiency.

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

STATIC ALLOCATION

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

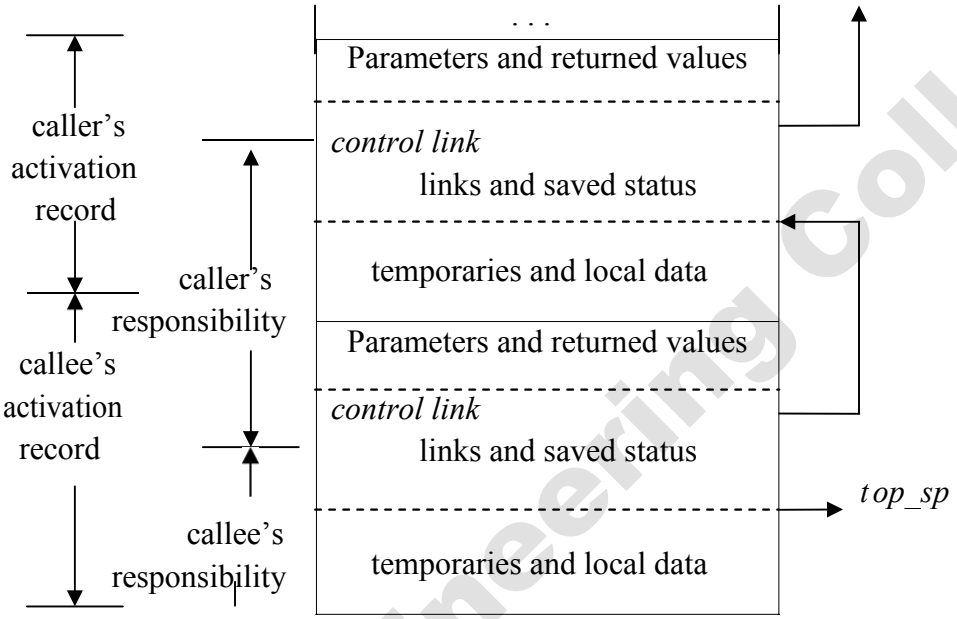
STACK ALLOCATION OF SPACE

- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

Calling sequences:

- Procedures called are implemented in what is called as calling sequence, whichconsists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
  - Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.

- Fixed length items are generally placed in the middle. Such items typicallyinclude the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of thearray.
- We must locate the top-of-stack pointer judiciously. A common approach is tohave it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.

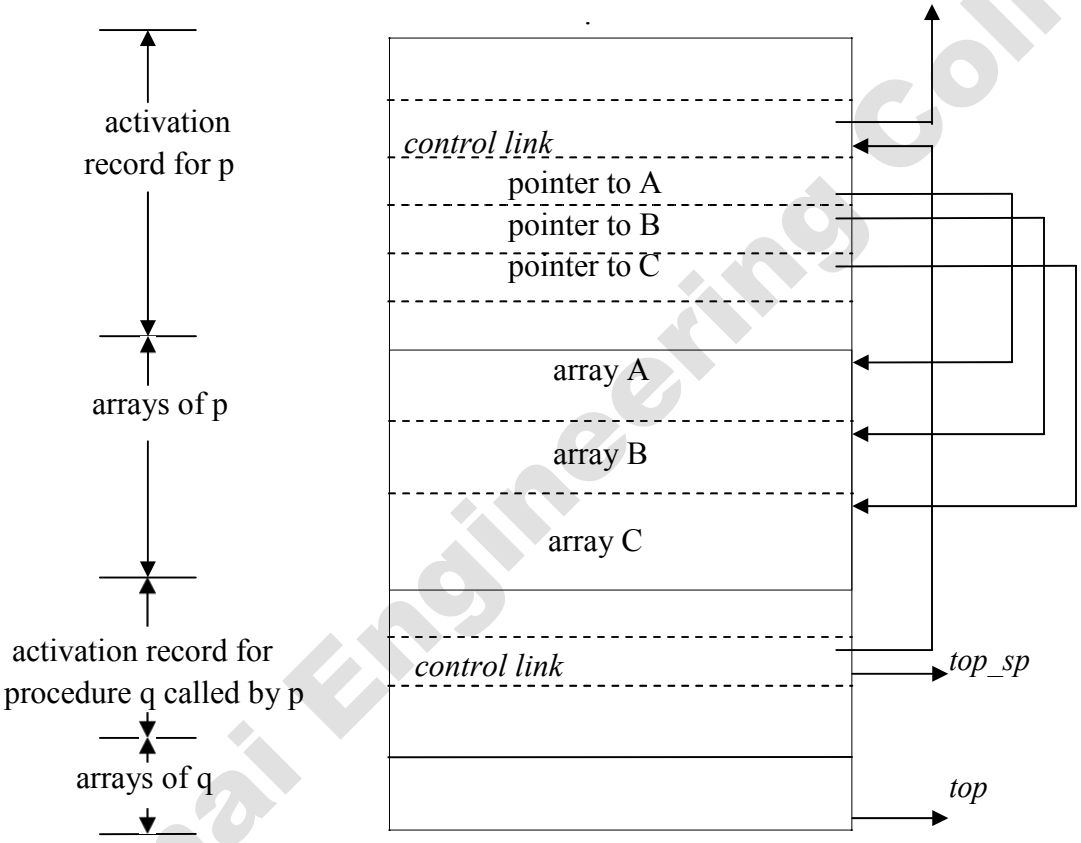


Division of tasks between caller and callee

- The calling sequence and its division between caller and callee are as follows.
  - The caller evaluates the actual parameters.
  - The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
  - The callee saves the register values and other status information.
  - The callee initializes its local data and begins execution.
- A suitable, corresponding return sequence is:
  - The callee places the return value next to the parameters.
  - Using the information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
  - Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp*; the caller therefore may use thatvalue.

Variable length data on stack:

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedurecalled and have a size that depends on the parameters of the call.



Access to dynamically allocated arrays

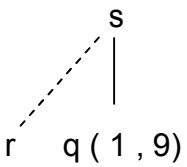
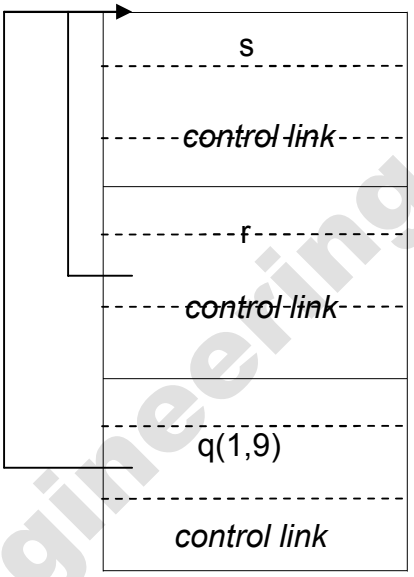
- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers, *top* and *top-sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top-sp* is used to find local, fixed-length fields of the top activationrecord.
- The code to reposition *top* and *top-sp* can be generated at compile time, in terms ofsizes that will become known at run time.

HEAP ALLOCATION

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the	Activation records in the heap	Remarks
		Retained activation record for r

- The record for an activation of procedure r is retained when the activationends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

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

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-valueof 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.

SYMBOL TABLE

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

- A symbol table may serve the following purposes depending upon the language in hand:
- To store the names of all entities in a structured form at one place.•
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the• source code are semantically correct.
- To determine the scope of a name (scope resolution).

Implementation

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways: \

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Operations

A symbol table, either linear or hash, should provide the following operations.

insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example: int a; should be processed by the compiler as:

insert(a, int);

Lookup()

lookup() operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following:

lookup(symbol)

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

Scope Management

A compiler maintains two types of symbol tables: a global symbol table which can be accessed by all the procedures and scope symbol tables that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . . int value=10;

 void pro_one()

{

 int one_1;
```

```
 int one_2;
{
\ int one_3; |_ inner scope 1 int one_4; |
} / int one_5;
 {
\ int one_6; |_ inner scope 2 int one_7; |
} / }
void pro_two()
 {
int two_1; int two_2;
{ \ int two_3; |_ inner scope 3 int two_4; | }
 / int two_5; } . . .
```

The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first a symbol will be searched in the current scope, i.e., current symbol table,
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or the global symbol table has been searched for the name.

UNIT V - CODE OPTIMIZATION AND CODE GENERATION

Topics to be Covered

Principal Sources of Optimization-DAG- Optimization of Basic Blocks-Global Data Flow Analysis-

Efficient Data Flow Algorithms-Issues in Design of a Code Generator - A Simple Code Generator

Algorithm.

INTRODUCTION

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

- Optimizations are classified into two categories. They are
  Machine independentoptimizations:
  Machine dependant optimizations:

Machine independent optimizations:

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

Machine dependant optimizations:

Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

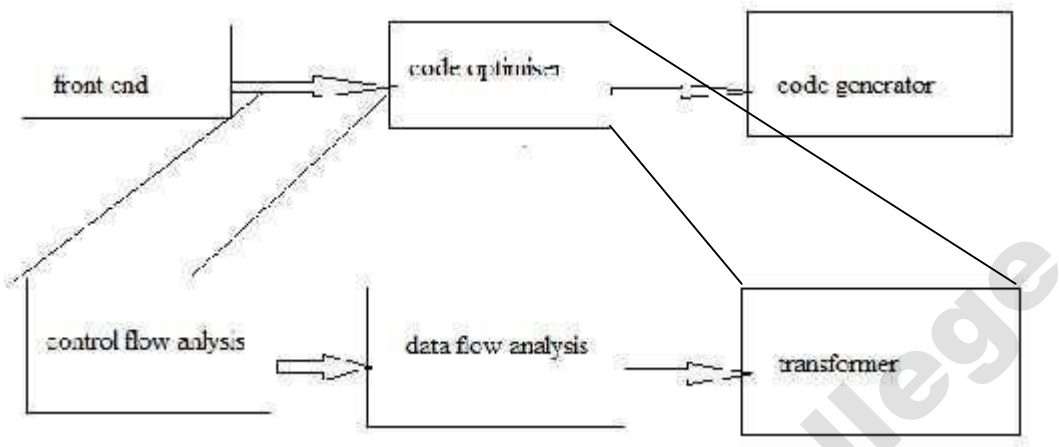
- ✓ Simply stated, the best program transformations are those that yield the most benefit for the least effort.

- ✓ The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the programdoes.

- ✓ A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "optimization" may slow down a program slightly.

- ✓ The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. "Peephole" transformations of this kind are simple enough and beneficial enough to be included in any compiler.

Organization for an OptimizingCompiler:



- Flow analysis is a fundamental prerequisite for many important types of code improvement.

  Generally control flow analysis precedes data flow analysis.

  Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as

  control flow graph

  Call graph

  Data flow analysis (DFA) is the process of ascerting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

PRINCIPAL SOURCES OF OPTIMISATION

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.

Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

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

- The transformations

✓ Common sub expression elimination,
✓ Copy propagation,
✓ Dead-code elimination, and
✓ Constant folding

are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

➢ **Common Sub expressionselimination:**

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

For example

t_1: = 4*i
t_2: = a [t1]
t_3: = 4*j
t_4: = 4*i
t_5: = n
t_6: = b [t_4] +t_5

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

t_1: = 4*i
t_2: = a [t_1]
t_3: = 4*j
t_5: = n
t_6: = b [t_1] +t_5

The common sub expression t_4: =4*i is eliminated as its computation is already in t_1. And value of i is not been changed from definition to use.

➢ **Copy Propagation:**

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

For example:

```
x=Pi;
……
A=x*r*r;
```

The optimization using copy propagation can be done as follows:

```
A=Pi*r*r;
```

Here the variable x is eliminated

➢ **Dead-Code Eliminations:**

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute

values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.
Example:

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

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

➢ **Constant folding**:

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
- One advantage of copy propagation is that it often turns the copy statement into dead code.

✓ For example,
a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation.

➢ **Loop Optimizations:**

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Three techniques are importantfor loop optimization:

✓ code motion, which moves code outside a loop;
✓ Induction-variable elimination, which we apply to replace variables from inner loop.
✓ Reduction in strength, which replaces and expensive operation by a cheaper one, suchas a multiplication by an addition.

➢ **Code Motion:**

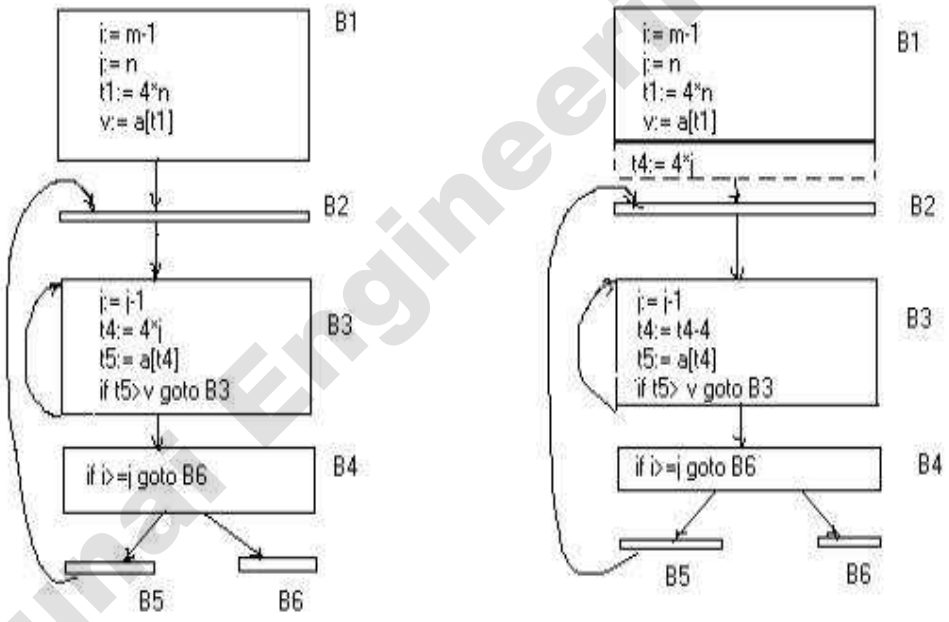
- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:
while (i <= limit-2) /* statement does not change limit*/

Code motion will result in the equivalent of

t= limit-2;
while (i<=t) /* statement does not change limit or t */

➢ **Induction Variables :**

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.
  Example:
  As the relationship t_4:=4*j surely holds after such an assignment to t_4 in Fig. and t_4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t_4:= 4*j-4 must hold. We may therefore replace the assignment t_4:= 4*j by t_4:= t_4-4. The only problem is that t_4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t_4=4*j on entry to the block B3, we place an initializations of t_4 at the end of the block where j itself is



before **after**

initialized, shown by the dashed addition to block B1 in second Fig.

- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

➢ **Reduction In Strength:**

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
  For example, x is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

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.

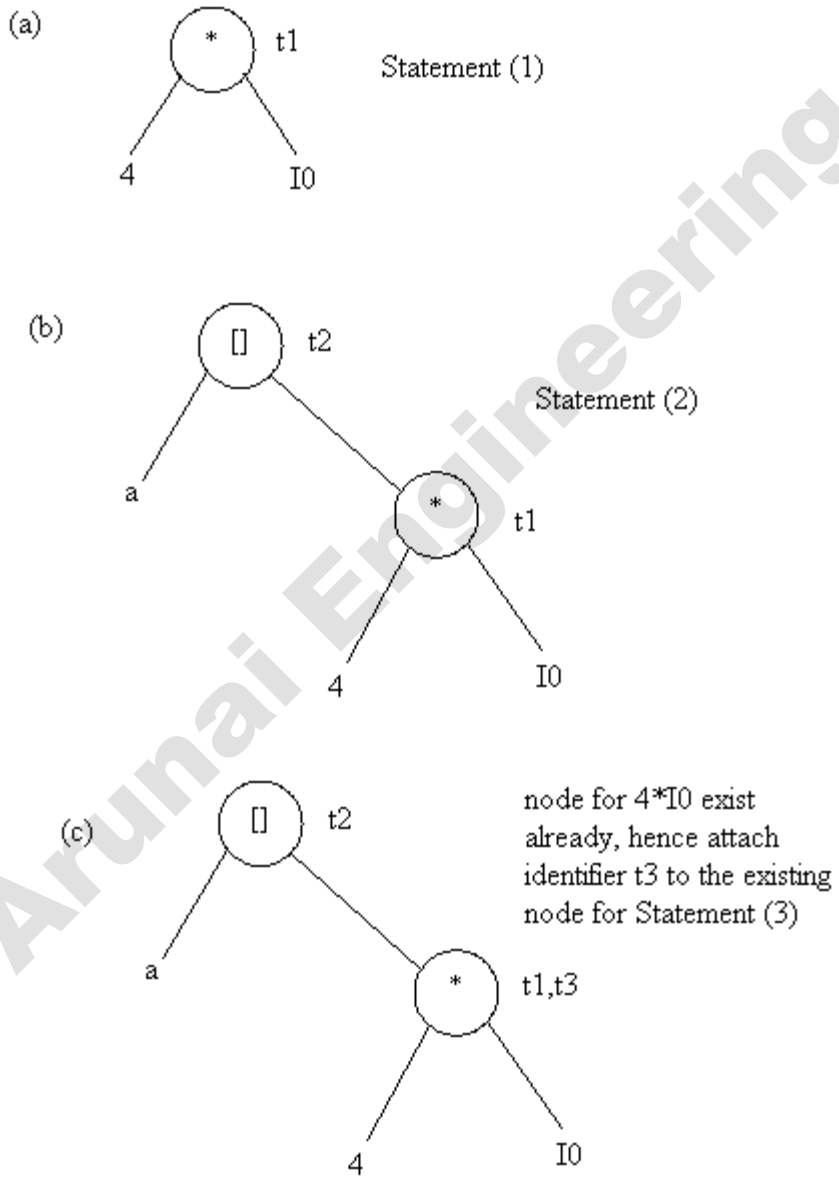
For case(iii), node n will be node(y).

Step 3: Delete x from the list of identifiers for node(x). Append x to the list of

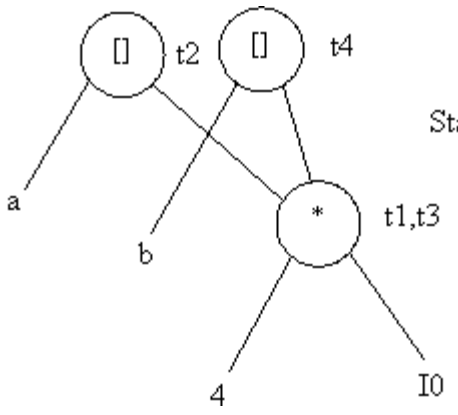
Example: Consider the block of three- address statements:

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

Stages in DAG Construction

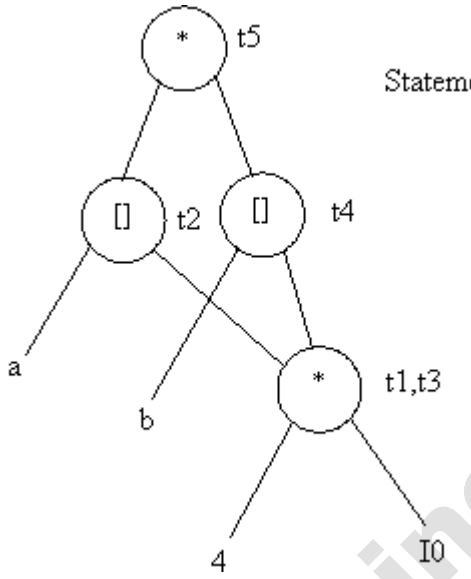


(d)



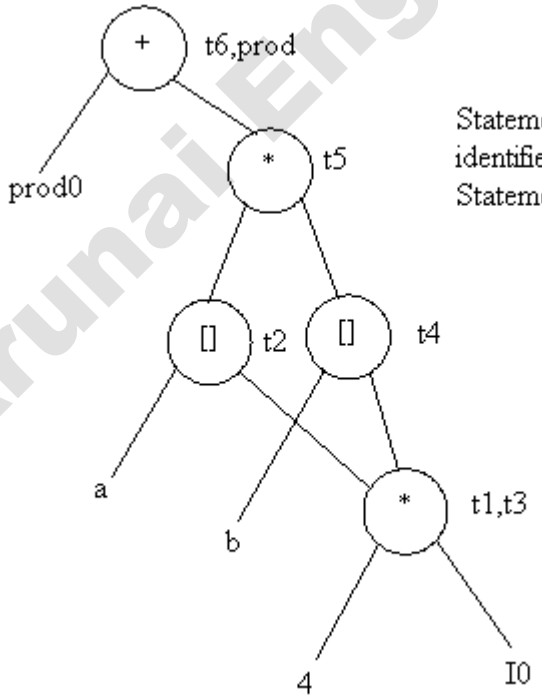
Statement (4)

(e)



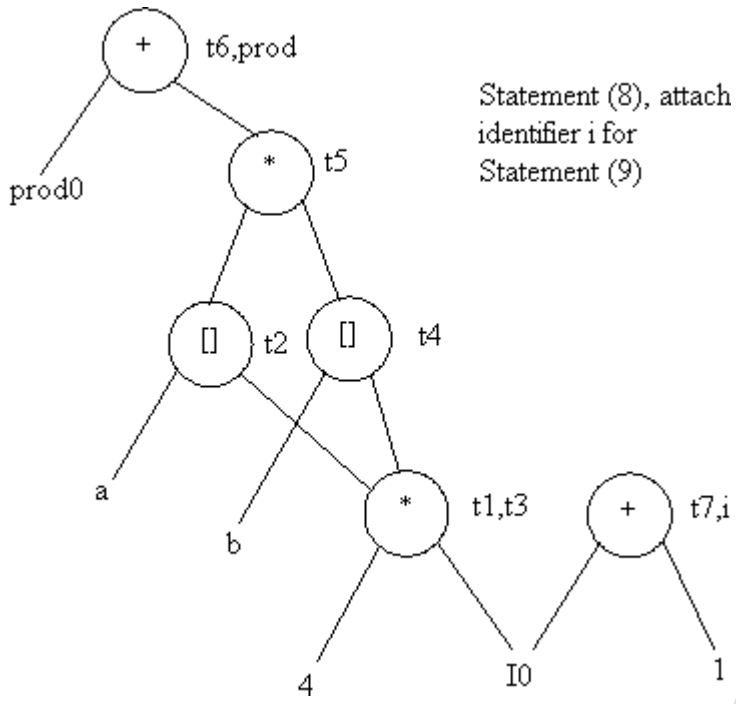
Statement (5)

(f)

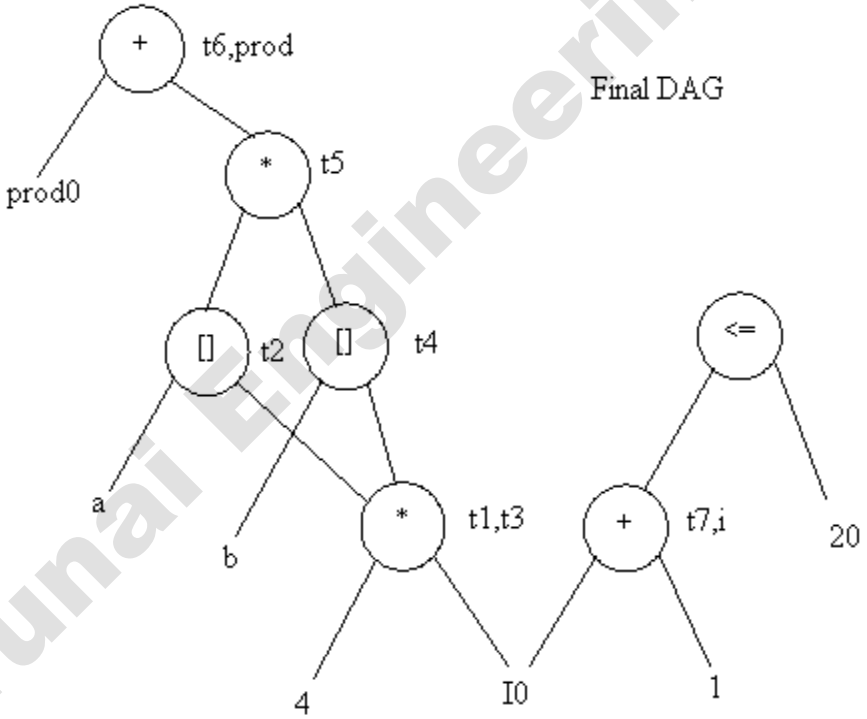


Statement (6), attach identifier prod for Statement (7)

(g)



(h)



Application of DAGs:

1. We can automatically detect commonsub 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.

GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$t_4 := t_1 - t_3$

Generated code sequence for basic block:

MOV a , R_0
ADD b , R_0
MOV c , R_1
ADD d , R_1
MOV R_0 , t_1
MOV e , R_0
SUB R_1 , R_0
MOV t_1 , R_1
SUB R_0 , R_1
MOV R_1 , t_4

Rearranged basic block:

Now t1 occurs immediately before t4.

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Revised code sequence:

MOV c , R_0
ADD d , R_0
MOV a , R_0
SUB R_0 , R_1
MOV a , R_0
ADD b , R_0
SUB R_1 , R_0
MOV R_0 , t_4

In this order, two instructions **MOV R_0 , t_1** and **MOV t_1 , R_1** have been saved.

A Heuristic ordering for Dags

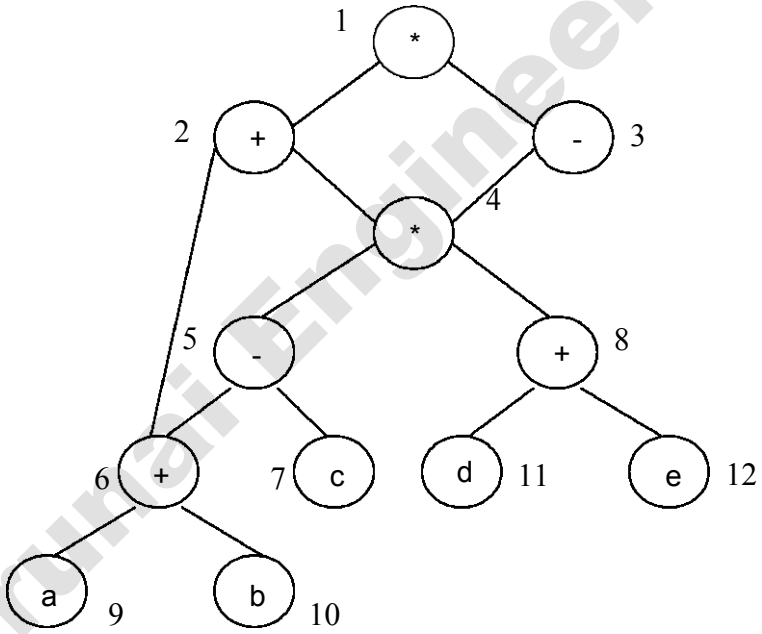
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

The algorithm shown below produces the ordering in reverse.

Algorithm:

```
1)  while unlisted interior nodesremain do begin
2)      select an unlisted node n, all of whose parents have been listed;
3)      list n;
4)      while the leftmost child m of n has no unlisted parents and is not a leaf do
            begin
5)              list m;
6)              n : = m
            end
        end
```

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

$t_8 := d + e$ $t_6 := a$
$+ b$ $t_5 := t_6 - c$ t_4
$:= t_5 * t_8$ $t_3 :=$ t4
$- e$ $t_2 := t_6 + t_4$ t_1
$:= t_2 * t_3$

This will yield an optimal code for the DAG on machine whatever be the number of registers.

OPTIMIZATION OF BASIC BLOCKS
There are two types of basic block optimizations. They are :

- ✓ Structure-Preserving Transformations
- ✓ Algebraic Transformations

Structure-Preserving Transformations:
The primary Structure-Preserving Transformation on basic blocks are:

- ✓ Common sub-expressionelimination
- ✓ Dead code elimination
- ✓ Renaming of temporaryvariables
- ✓ Interchange of two independentadjacent statements.

➢ **Common sub-expressionelimination:**
Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it"s referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

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

The 2nd and 4th statements compute the same expression: b+c and a-d
Basic block can be transformed to

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

➢ **Dead code elimination:**

It‟s possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of constructio n or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

➢ **Renaming of temporary variables:**

- A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.
- In this we can transform a basic block to its equivalent block called normal-form block.

➢ **Interchange of two independent adjacent statements:**

- Two statements

  t_1:=b+c
  t_2:=x+y

  can be interchanged or reordered in its computation in the basic block when value of t_1 does not affect the value of t_2.

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
- The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

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

  the following intermediate code may be generated:

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

- Example:

  x:=x+0 can be removed

  x:=y**2 can be replaced by a cheaper statement x:=y*y

- The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

LOOPS IN FLOW GRAPH

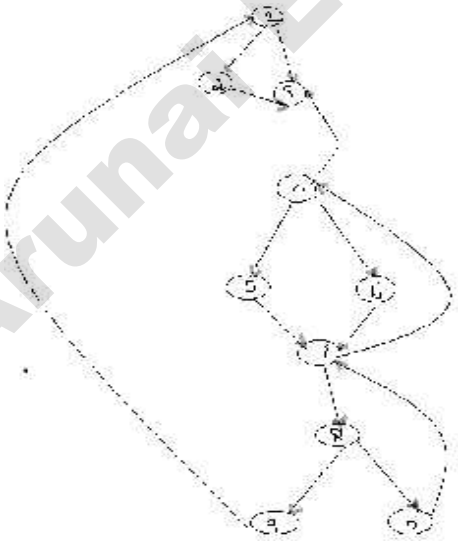
A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

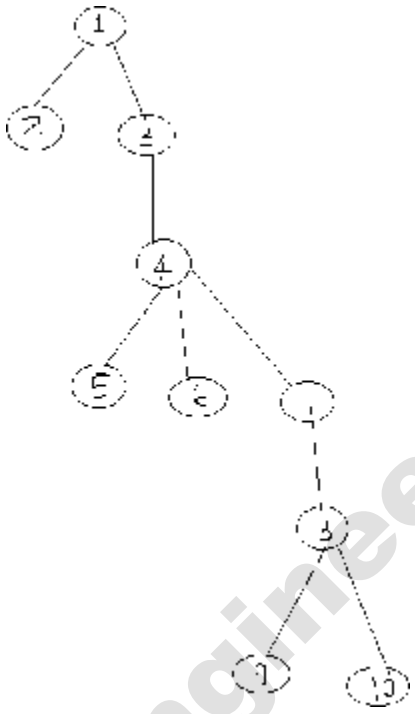
In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by *d dom n*. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

*In the flow graph below,
*Initial node,node1 dominates every node.
*node 2 dominates itself
*node 3 dominates all but 1 and 2.
*node 4 dominates all but 1,2 and 3.
*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.
*node 7 dominates 7,8 ,9 and 10.
*node 8 dominates 8,9 and 10.
*node 9 and 10 dominates only themselves.



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n.
- In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.



D(1)={1} D(2)={1,2}

D(3)={1,3}

D(4)={1,3,4}

D(5)={1,3,4,5}

D(6)={1,3,4,6}

D(7)={1,3,4,7}

D(8)={1,3,4,7,8}

D(9)={1,3,4,7,8,9}

D(10)={1,3,4,7,8,10}

Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.

- The properties of loops are

  ✓ A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
  ✓ There must be at least one way to iterate the loop(i.e.)at least one path back to the header.

- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.

  ✓ Example:

  In the above graph,

  7 → 4 4 DOM 7
  10 →7 7 DOM 10
  4 → 3
  8 → 3
  9 →1

- The above edges will form loop in flow graph.
- Given a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

Algorithm: Constructing the natural loop of a back edge.

Input: A flow graph G and a back edge n→d.

Output: The set loop consisting of all nodes in the natural loop n→d.

Method: Beginning with node n, we consider each node m*d that we know is in loop, to make sure that m‟s predecessors are also placed in loop. Each node in loop, except for d, is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.

Procedure insert(m);
if m is not in *loop* **then begin**
    loop := *loop* U {m};
    push *m* onto *stack*
end;

stack : = empty;

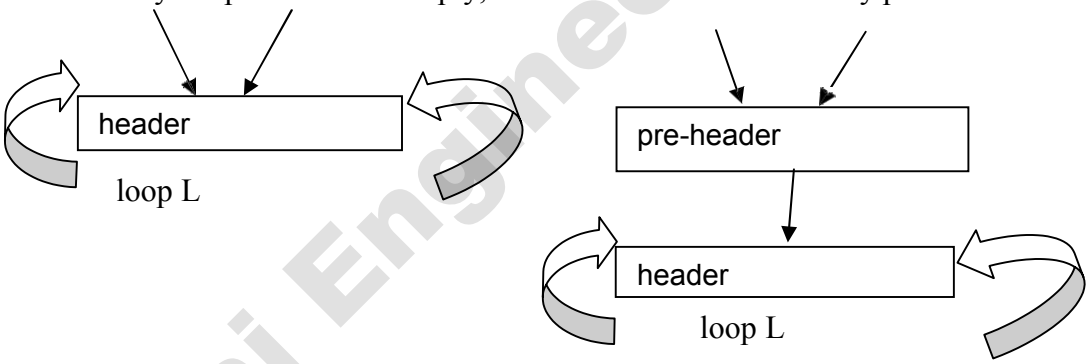
loop : = {*d*};
insert(*n*);
while *stack* is not empty **do begin**
pop *m*, the first element of *stack*, off *stack*;
for each predecessor *p* of *m* **do** *insert(p)*
end

Inner loop:

- If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a singleloop.

Pre-Headers:

- Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheater.
- The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.
- Edges from inside loop L to the header are not changed.
- Initially the pre-header is empty, but transformations on L may place statements in it.



(a) Before **(b) After**

Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.
- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

- The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

- ***Definition***:
  A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, *forward* edges and *back* edges, with the following properties.

✓ The forward edges from an acyclic graph in which every node can be reached from initial node of G.

✓ The back edges consist only of edges where heads dominate theirs tails.

✓ Example: The above flow graph isreducible.

- If we know the relation DOM for a flow graph, we can find and remove all the back edges.

- The remaining edges are forwardedges.

- If the forward edges form an acyclic graph, then we can say the flow graph reducible.

- In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.

- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the targetprogram.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- We shall give the following examples of program transformations that are characteristic of peephole optimizations:

  ✓ Redundant-instructions elimination
  ✓ Flow-of-control optimizations
  ✓ Algebraic simplifications
  ✓ Use of machine idioms
  ✓ Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R_0,a

(2) MOV a,R_0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R_0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

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

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

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

```
If debug =1 goto L2
goto L2
L1: print debugging information
L2 .....(a)
```

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

```
If debug ≠1 goto L2
Print debugging information
L2: .....(b)
```

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

```
If debug ≠0 goto L2
Print debugging information
L2: .....(c)
```

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

Flows-Of-Control Optimizations:

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

```
goto L1
….
L1: gotoL2 by the sequence
goto L2
….
L1: goto L2
```

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

```
if a < b goto L1
….
L1: goto L2 can be replaced by
If a < b goto L2
….
L1: goto L2
```

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

```
Then the sequence
goto L1
……..
L1: if a < b goto L2
L3:.....(1)
```

- May be replaced by

```
If a < b goto L2 goto L3
…….
L3:.....(2)
```

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

Algebraic Simplification:

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

```
x := x+0 Or
x := x * 1
```

- Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

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

- For example, x is invariably cheaper to implement as x*x than as a call to an exponentiation routine.

Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X*X$

Use of Machine Idioms:

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
- The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like

  i :=i+1.

  i:=i+1 → i++ i:=i-1 →

  i- -

INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.

- A compiler could take advantage of "reaching definitions" , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.

- Data-flow information can be collected by setting up and solving systems of equations of the form :

  out [S] = gen [S] U (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."

- The details of how data-flow equations are set and solved depend on three factors.

✓ The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms ofout[s].

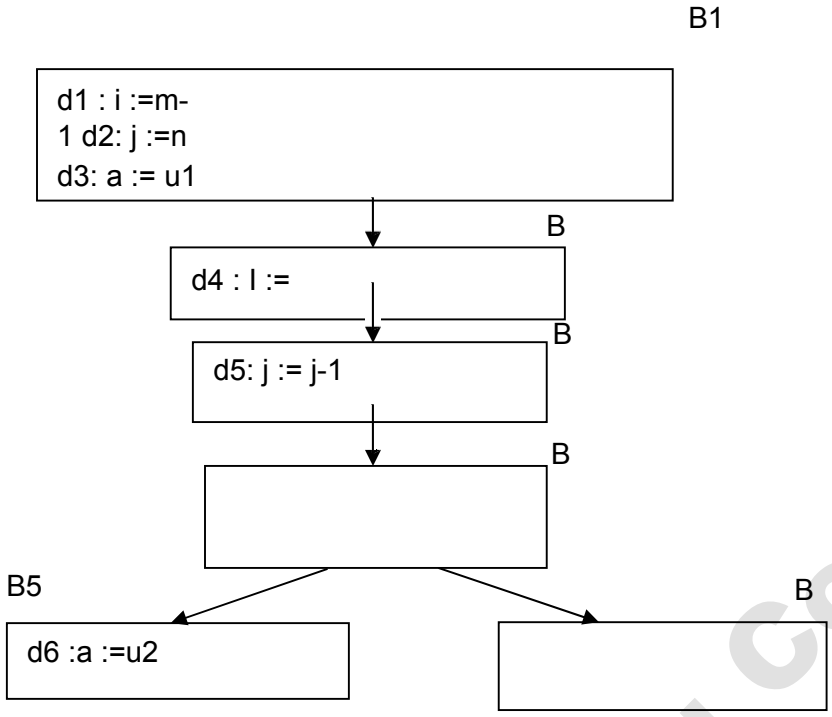
✓ Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

✓ There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:

- Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the

assignments and one after each of the three assignments.



- Now let us take a global view and consider all the points in all the blocks. A path 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

✓ 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

✓ P_i is the end of some block and p_{i+1} is the beginning of a successor block.

Reaching definitions:

- A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x.

- These statements certainly define a value for x, and they are referred to as **unambiguous** definitions of x. There are certain kinds of statements that may define a value for x; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:

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

✓ An assignment through a pointer that could refer to x. For example, the assignment *q: = y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of everyvariable.

- We say 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. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

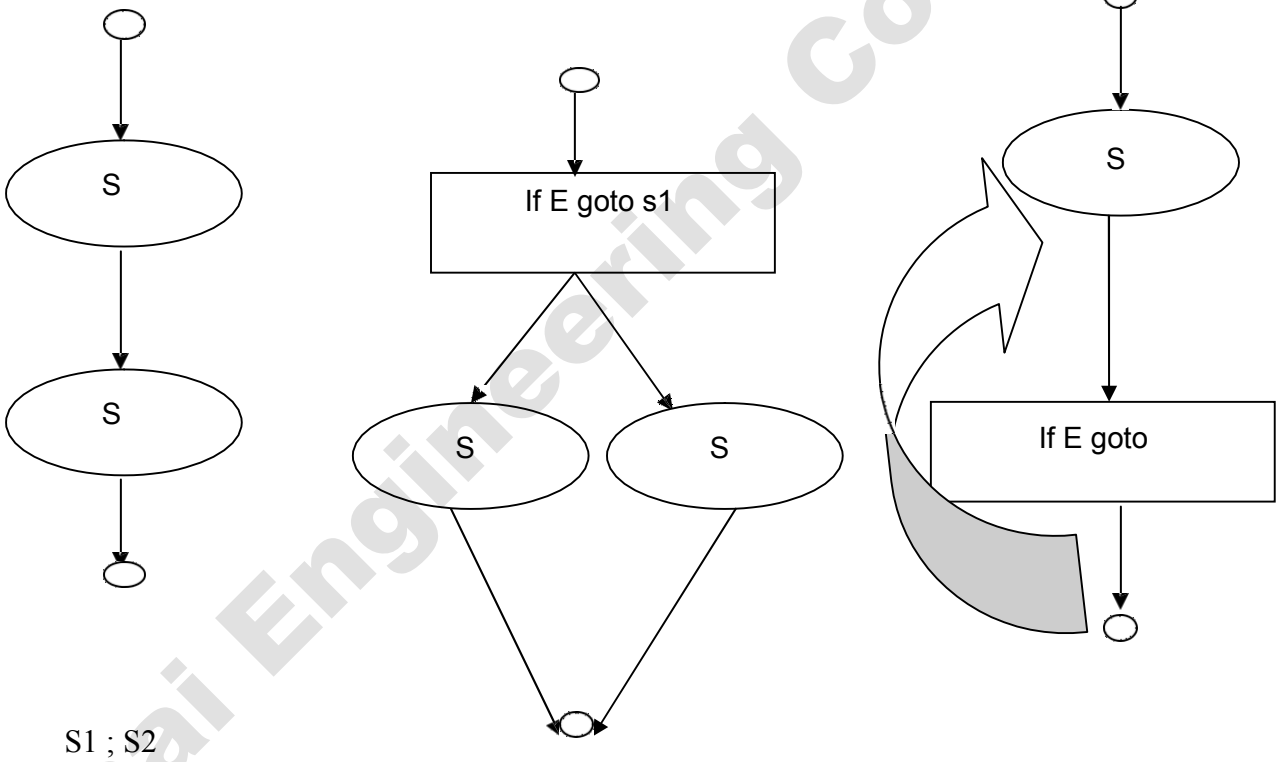
Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: 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. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

  S ⟶ id: = E| S; S | if E then S else S | do S while E

  E ⟶ id + id| id

- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restrictedforms.



S1 ; S2

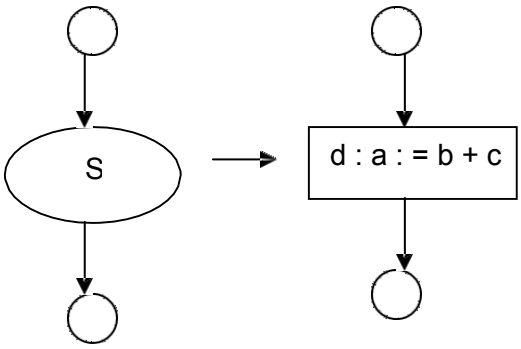
IF E then S1 else S2

do S1 while E

- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

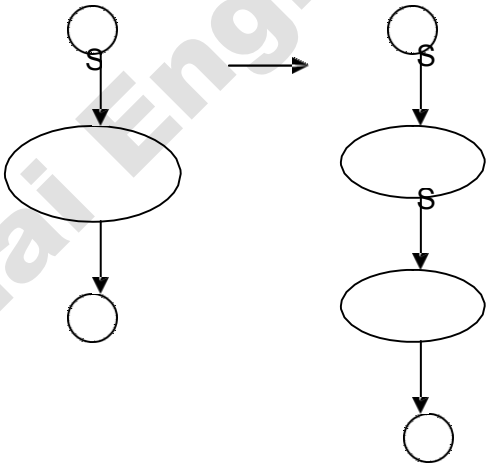
- We say that the beginning points of the dummy blocks at the entry and exit of a statement"s region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.
- **gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.**
- Consider the following data-flow equations for reaching definitions :

i)



gen [S] = { d }
kill [S] = D_a – { d }
out [S] = gen [S] U (in[S] – kill[S])

- Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus
  Gen[S]={d}
- On the other hand, d "kills" all other definitions of a, so we write
  Kill[S] = D_a – {d}
  Where, D_a is the set of all definitions in the program for variable a. ii)



gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])
Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

in [S_1] = in [S]
in [S_2] = out [S_1]
out [S] = out [S_2]

- Under what circumstances is definition d generated by $S=S_1; S_2$? First of all, if it is generated by S_2, then it is surely generated by S. if d is generated by S_1, it will reach the end of S provided it is not killed by S_2. Thus, we write
  $gen[S]=gen[S_2] \cup (gen[S_1]-kill[S_2]$
- Similar reasoning applies to the killing of a definition, so we have
  $Kill[S] = kill[S_2] \cup (kill[S_1] - gen[S_2])$

Conservative estimation of data-flow information:

- There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are "uninterpreted"; that is, there exists inputs to the program that make their branches go either way.

- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.

- When we compare the computed gen with the "true" gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.

- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.

- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.

- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.

- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in.

S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

- The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S.

- Assuming we know in[S] we compute out by equation, that

  is Out[S] = gen[S] U (in[S] - kill[S])

- Considering cascade of two statements S_1; S_2, as in the second case. We start by observing in[S_1]=in[S]. Then, we recursively compute out[S_1], which gives us in[S_2], since a definition reaches the beginning of S_2 if and only if it reaches the end of S_1. Now we can compute out[S_2],and this set is equal to out[S].

- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S_1 or S_2 exactly when it reaches the beginning of S.

  In[S_1] = in[S_2] = in[S]

- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

  Out[S]=out[S_1] U out[S_2]

Representation of sets:

- Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is inthe set.

- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.

- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A

.

Local reaching definitions:

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.

- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

- It is often convenient 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 reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions of a.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.

- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.

- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.

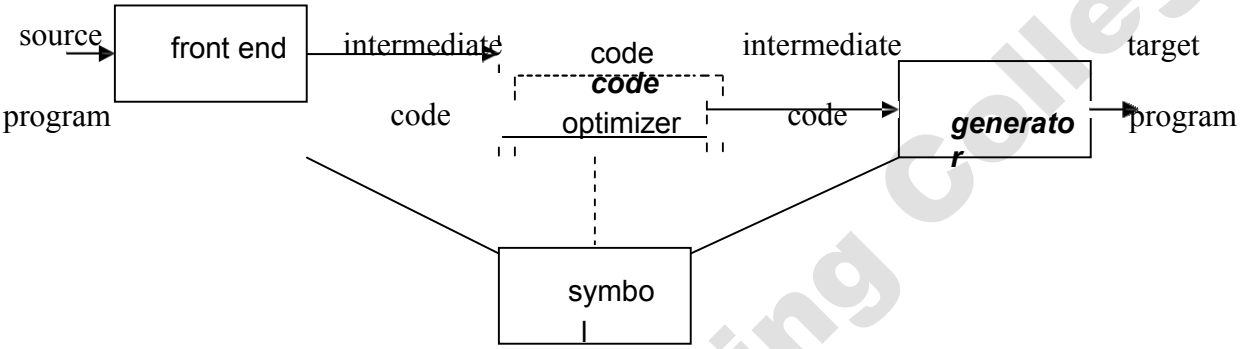
- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval -based methods

- However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

Position of code generator



ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
- Intermediate representationcan be :
  a. Linear representation such as postfix notation
  b. Three address representation such asquadruples
  c. Virtual machine representation such as stack machine code
  d. Graphical representationssuch as syntax trees and dags.
- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to beerror-free.

2. Target program:

- The output of the code generator is the target program. The output may be :
  a. Absolute machinelanguage
     - It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language
   - It allows subprograms to be compiled separately.

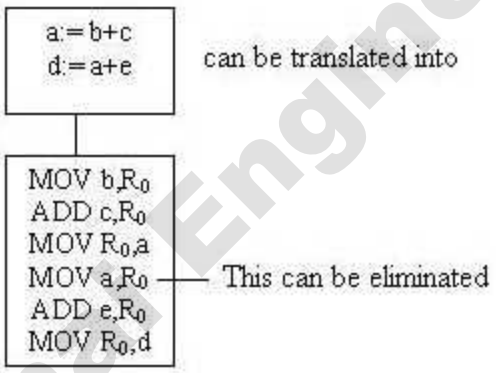
c. Assembly language
   - Code generation is made easier.

3. Memory management:

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions. For example,
  j : **goto** *i* generates jump instruction as follows :
  - if $i < j$, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.
  - if $i > j$, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

4. Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.
- The former statement can be translated into the latter statement as shown below:



5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two subproblems :
  - ***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected.

- ***Register assignment*** – the specific register that a variable will reside in is picked

- Certain machine requires even-odd *register pairs* for some operands and results. For example , consider the division instruction of the form :

  D x, y

  where, x – dividend even register in even/odd register pair
  y – divisor
  even register holds the remainder
  odd register holds the quotient

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- The target computer is a byte-addressable machine with 4 bytes to a word.
- It has *n* general-purpose registers, R_0, R_1, . . . , R_{n-1}.
- It has two-address instructions of the form:

  op source, destination

  where, *op* is an op-code, and *source* and *destination* are data fields.

- It has the following op-codes :

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

- The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

Address modes with their assembly-language forms

MODE	FORM	*ADDRESS*	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	*c*(R)	*c*+*contents*(R)	1
indirect register	*R	*contents* (R)	0
indirect indexed	**c*(R)	*contents*(*c*+ *contents*(R))	1
literal	#*c*	*c*	1

- For example : MOV R_0, M stores contents of Register R_0 into memory location M ; MOV 4(R_0), M stores the value *contents*(4+*contents*(R_0)) into M.

Instruction costs :

- Instruction cost = 1+cost for source and destination addressmodes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.
- Address modes involving memory location or literal have cost one.
- Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.
  For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.
- The three-address statement **a : = b + c** can be implemented by many different instruction sequences :

  i) MOV b, R_0
  ADD c, R_0 cost = 6
  MOV R_0, a

  ii) MOV b, a
  ADD c, a cost = 6

  iii) Assuming R_0, R_1 and R_2 contain the addresses of a, b, and c :
  MOV *R_1, *R_0
  ADD *R_2, *R_0 cost = 2

- In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.

A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three- address statements and effectivelyuses registers to store operands of the statements.

- For example: consider the three-address statement **a := b+c**
  It can have the following sequence of codes:

  ADD R_j, R_i Cost = 1 // if R_i contains b and R_j contains c

  (or)

  ADD c, R_i Cost = 2 // if c is in a memory location

  (or)

  MOV c, R_j Cost = 3 // move c from memory to Rjand add

  ADD R_j, R_i

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z**,** perform the following actions:

2. Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.

3. Consult the address descriptor for y to determine y‟, the current location of y. Prefer the register for y‟ if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y' , L** to place a copy of y in L.

4. Generate the instruction **OP z' , L** where z‟ is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

5. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

- The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

  t : = a – b
  u : = a – c
  v : = t + u
  d : = v + u

with d live at the end.
Code sequence for the example is:

Statements	**Code Generated**	**Register descriptor**	**Address descriptor**
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

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