COMPILER DESIGN

CSE

V SEMESTER

Prepared by:
Dr. K Rajendra Prasad
Ms. E Uma Shankari
Ms. B Ramya Sree
Ms. K Sharanya

The course will able the students to	
I	Apply the principles in the theory of computation to the various stages in the design of compilers.
II	Demonstrate the phases of the compilation process and able to describe the purpose and operation of each phase.
III	Analyze problems related to the stages in the translation process.
IV	Exercise and reinforce prior programming knowledge with a non-trivial programming project to construct a compiler.

COURSE OUTCOMES (COs):

The course will able the students to	
CO1	Understand the various phases of compiler and design the lexical analyzer
CO2	Explore the similarities and differences among various parsing techniques and grammar transformation techniques
CO3	Analyze and implement syntax directed translations schemes and intermediate code generation
CO4	Describe the concepts of type checking and analyze runtime allocation strategies
CO5	Demonstrate the algorithms to perform code optimization and code generation.

The course will able the students to	
CLO1	Define the phases of a typical compiler, including the front and backend.
CLO2	Recognize the underlying formal models such as finite state automata, push-down automata and their connection to language definition through regular expressions and grammars
CLO3	Identify tokens of a typical high-level programming language; define regular expressions for tokens and design and implement a lexical analyzer using a typical scanner generator.
CLO4	Explain the role of a parser in a compiler and relate the yield of a parse tree to a grammar derivation.

The course will able the students to	
CLO5	Apply an algorithm for a top-down or a bottom-up parser construction; construct a parser for a given context-free grammar.
CLO6	Demonstrate Lex tool to create a lexical analyzer and Yacc tool to create a parser.
CLO7	Understand syntax directed translation schemes for a given context free grammar.
CLO8	Implement the static semantic checking and type checking using syntax directed definition (SDD) and syntax directed translation (SDT).

The course will able the students to	
CLO9	Understand the need of intermediate code generation phase in compilers.
CLO10	Write intermediate code for statements like assignment, conditional, loops and functions in high level language.
CLO11	Explain the role of a semantic analyzer and type checking; create a syntax-directed definition and an annotated parse tree; describe the purpose of a syntax tree.
CLO12	Design syntax directed translation schemes for a given context free grammar.

The course will able the students to	
CLO13	Explain the role of different types of runtime environments and memory organization for implementation of programming languages.
CLO14	Differentiate static vs. dynamic storage allocation and the usage of activation records to manage program modules and their data.
CLO15	Understand the roleof symbol table data structure in the construction of compiler.
CLO16	Learn the code optimization techniques to improve the performance of a program in terms of speed & space

The course will able the students to	
CLO17	Implement the global optimization using data flow analysis such as basic blocks and DAG.
CLO18	Understand the code generation techniques to generate target code.
CLO19	Design and implement a small compiler using a software engineering approach.
CLO20	Apply the optimization techniques to intermediate code and generate machine code

INTRODUCTION TO COMPILERS
AND PARSING

The course will able the students to	
CLO1	Define the phases of a typical compiler, including the front and backend.
CLO2	Recognize the underlying formal models such as finite state automata, push-down automata and their connection to language definition through regular expressions and grammars.
CLO3	Identify tokens of a typical high-level programming language; define regular expressions for tokens and design and implement a lexical analyzer using a typical scanner generator.
CLO4	Explain the role of a parser in a compiler and relate the yield of a parse tree to a grammar derivation

Preprocessor:

A preprocessor produce input to compilers. They may perform the following functions

- Macro processing:

- File inclusion
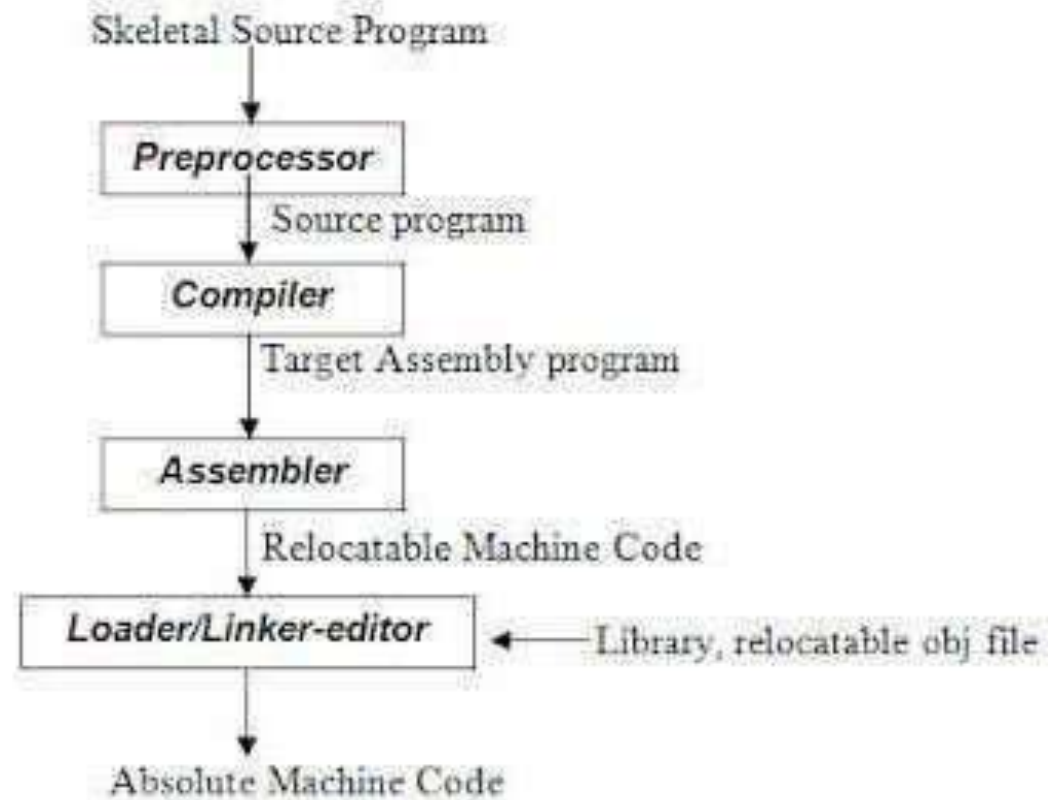
- Rational preprocessor

- Language Extensions

Skeletal Source Program

↓

Preprocessor

↓ Source program

Compiler

↓ Target Assembly program

Assembler

↓ Relocatable Machine Code

Loader/Linker-editor ← Library, relocatable obj file

↓

Absolute Machine Code

Fig 1.1 Language –processing System

Definition of Compiler

Compiler:

- Compiler is a translator program that translates a program written in(HLL)the source program and translates it into an equivalent program in (MLL) the target program.

- Executing a program written in HLL programming language is basically of two parts.

- the source program must first be compiled translated into a object program.

- Then the results object program is loaded into a memory executed.

14
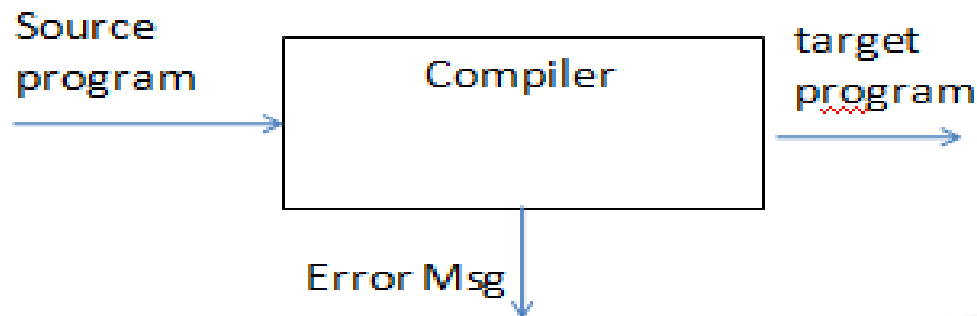
Assembler:

- Programmers found it difficult to write or read programs in machine language.

- Programs known as assembler were written to automate the translation of assembly language in to machine language.

Interpreter:

- An interpreter is a program that appears to execute a source program as if it were machine language

Interpreter:

Advantages:

- Modification of user program can be easily made and implemented
  as execution proceeds.

- Type of object that denotes various may change dynamically.

- Debugging a program and finding errors is simplified task for a program used for interpretation.

- The interpreter for the language makes it machine independent.

Disadvantages:

- The execution of the program is slower.

- Memory consumption is more.

linker : A linker combines one or more object files and possible some library code into either some executable, some library or a list of error messages.

Loader : A loader is a program that places programs into memory and prepares them for execution."

- It would be more efficient if subroutines could be translated into object
  form the loader could "relocate" directly behind the user's program.

- The task of adjusting programs o they may be placed in arbitrary core locations is called relocation.

- A compiler operates in phases.
  A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.

 There are two phases of compilation.

- Analysis (Machine Independent/Language Dependent)

- Synthesis (Machine Dependent/Language independent)
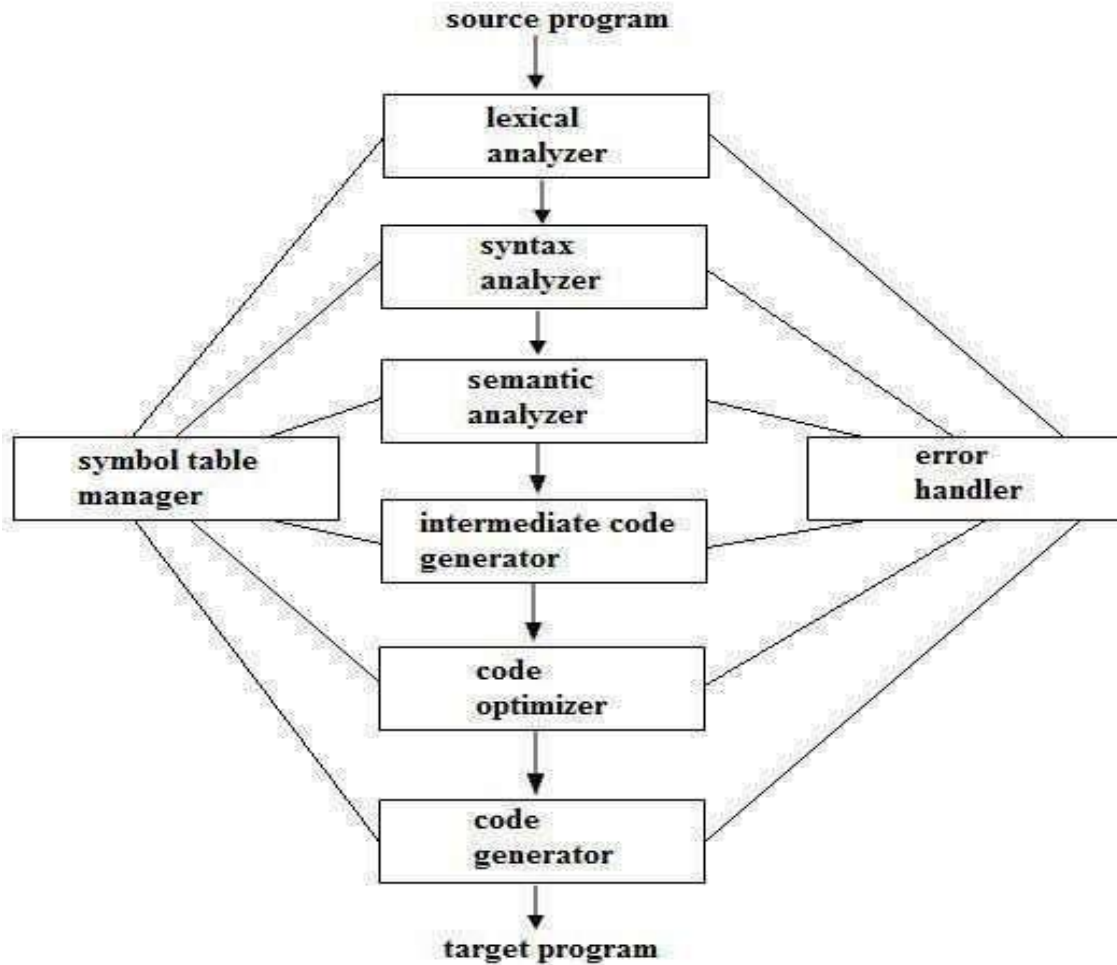
Fig 1.5 Phases of a compiler

Lexical Analysis:

- Lexical Analysis is also called as scanner

- It reads the source program one character at a time.

- carving the source program into a sequence of automatic units called tokens.

Syntax Analysis:

- The second stage of translation is called syntax analysis or parsing.

- In this phase expressions, statements, declarations etc... are identified by using the results of lexical analysis.

- Syntax analysis is aided by using techniques based on formal grammar of the programming language

Intermediate Code Generations:

- An intermediate representation of the final machine language code is produced.

- This phase bridges the analysis and synthesis phases of translation.

Code Optimization:

- This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:

- The last phase of translation is code generation.

- A number of optimizations to Reduce the length of machine language program are carried out during this phase.

- The output of the code generator is the machine language program of the specified computer

Table Management:

- This is the portion to keep the names used by the program and records essential information about each.

- The data structure used to record this information called a Symbol Table.

Error Handlers:

- It is invoked when a flaw error in the source program is detected.

- The output of LA is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser.

- The SA groups the tokens together into syntactic structure called as expression.

- Expression may further be combined to form statements.

Example:

position:= initial + rate *60

- Lexical Analysis:

  Tokens   : id1:=id2+id3*60

- Syntax Analysis:
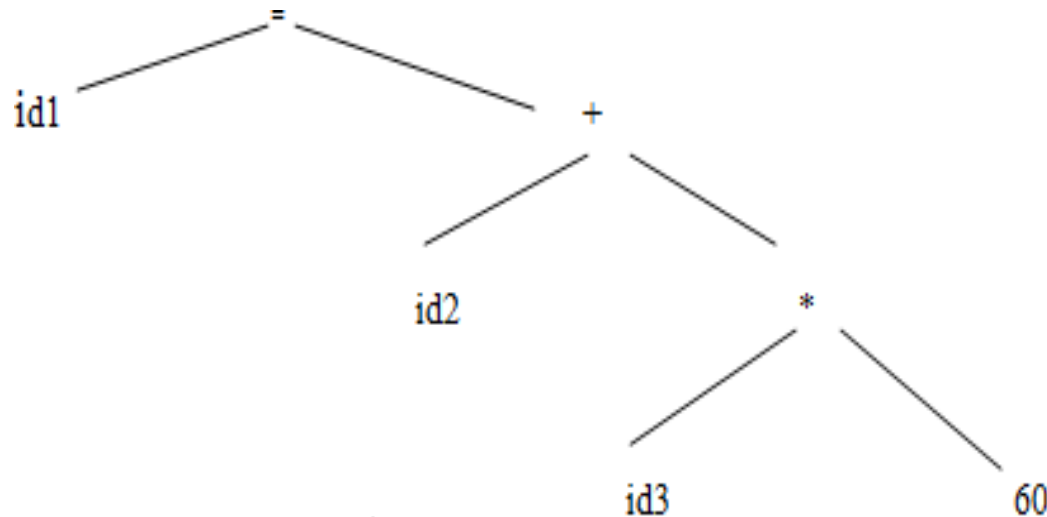
- **Intermediate Code Generation:** temp1:=
  int to   real(60) temp2:=
 id3 * temp1
  temp3:= id2 + temp2
  id1:= temp3

- **Code Optimization:**
  temp1: = id3 *
  60.0   id1:= id2
  +temp1

- **Code Generation:**
  MOVF id3,r1
  MULF $60,r1
  MOVF id2,r2
  ADDF r2,r1
  MOVF r1,id1

Role of Lexical Analyzer:

- The LA is the first phase of a compiler. Lexical analysis is called as linear analysis or scanning.

- In this phase the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

- Upon receiving a get next token command form the parser, the lexical analyzer reads the input character until it can identify the next token
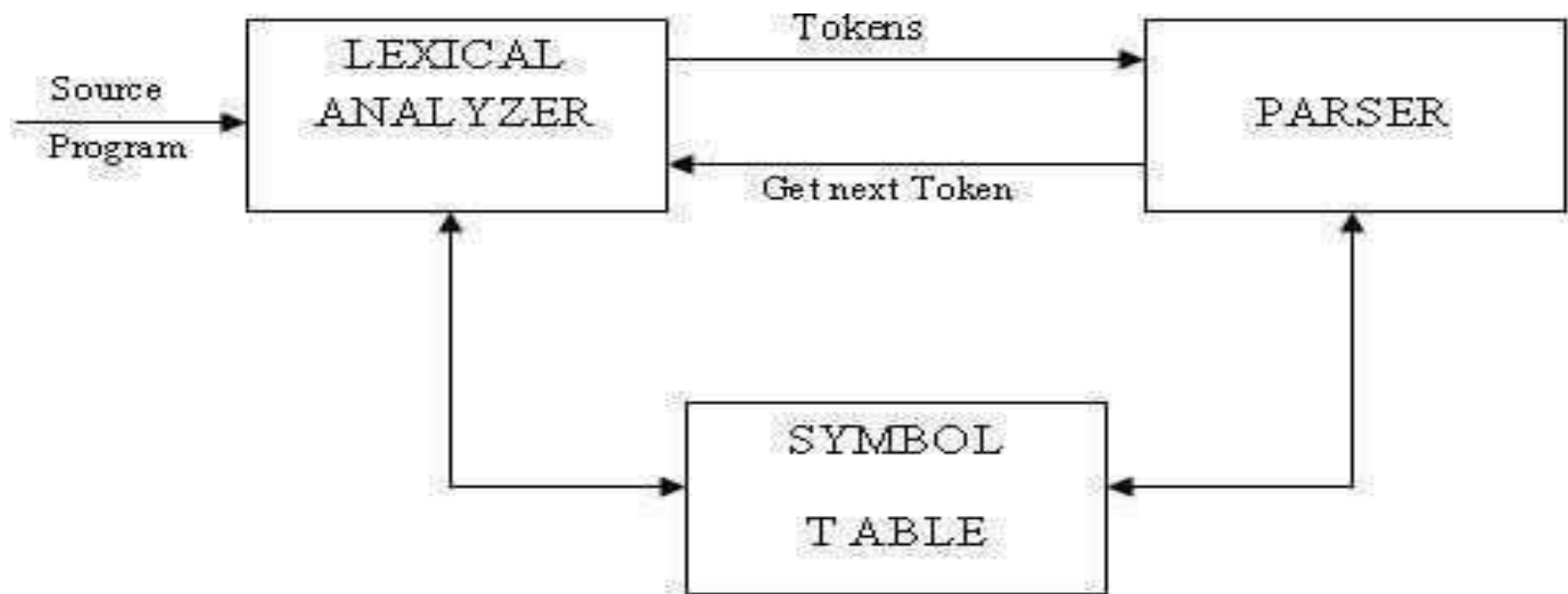
Token, Lexeme, Pattern:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are: Identifiers, keywords, operators, special symbols and Constants

Pattern:

- A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme:

- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Regular Expressions:

There are 3 specifications of tokens

- Strings

- Language

- Regular expression

An alphabet or character class is a finite set of symbols.

A string over an alphabet is a finite sequence of symbols drawn from that alphabet.

A language is any countable set of strings over some fixed alphabet.

Operations on strings

The following string-related terms are commonly used:

- A prefix of string s is any string obtained by removing zero or more
  symbols from the end of strings.

  Example: ban is a prefix of banana.

- A suffix of string s is any string obtained by removing zero or more
  more
  symbols from the beginning of s.

  Example: nana is a suffix of banana.

- A substring of s is obtained by deleting any prefix and any suffix
  from s.

  Example: nan is a substring of banana.

Operations on languages:

The following are the operations that can be applied to languages:

- Union

- Concatenation

- Kleene closure

- Positive closure

Operations on languages:

Let L={0,1} and S={a, b, c}

- Union   : L U S= {0, 1, a, b, c}

- Concatenation: L.S= {0a, 1a, 0b, 1b, 0c, 1c}

- Kleene closure: $L^* = \{\ \varepsilon,\ 0,1,\ 00....\}$

- Positive closure: $L^+ = \{0,\ 1,\ 00....\}$

Rules for Regular Expressions:

- ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.

- Suppose r and s are regular expressions denoting the languages L(r)
  and L(s). Then,
  - (r)|(s) is a regular expression denoting the language L(r) U L(s).
  - (r)(s) is a regular expression denoting the language L(r)L(s).
  - (r)* is a regular expression denoting (L(r))*.
  - (r) is a regular expression denoting L(r).

Description Of Automata:

- An automata has a mechanism to read input from input tape,

- Any language is recognized by some automation, Hence these automation are basically language "acceptors" or "language recognizers".

Types of Finite Automata

- Deterministic Automata

- Non-Deterministic Automata

Deterministic Automata:

A deterministic finite automata has at most one transition from each state

on any input. A DFA is a special case of a NFA in which:

- It has no transitions on input € ,

- Each input symbol has at most one transition from any state

- DFA formally defined by 5 tuple notation M = (Q, ∑, δ, qo, F), where Q is a
  finite „set of states", which is non empty.

- ∑ is „input alphabets", indicates input set.

- qo is an „initial state" and qo is in Q ie, qo, ∑, Q F is a set of Final states",

- δ is a „transmission function" or mapping function, using this function the next state can be determined

- The regular expression is converted into minimized DFA by the following procedure:

 **Regular expression → NFA → DFA → Minimized DFA**

- The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.

Nondeterministic Automata:

A NFA is a mathematical model consists of

- A set of states S.

- A set of input symbols ∑.

- A transition is a move from one state to another.

- A state so that is distinguished as the start (or initial) state

- A set of states F distinguished as accepting (or final) state.

- A number of transition to a single symbol.

- A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, in which the nodes are the states and the labeled edges represent the transition function.

- The transition graph for an NFA that recognizes the language (a|b)*abb is shown
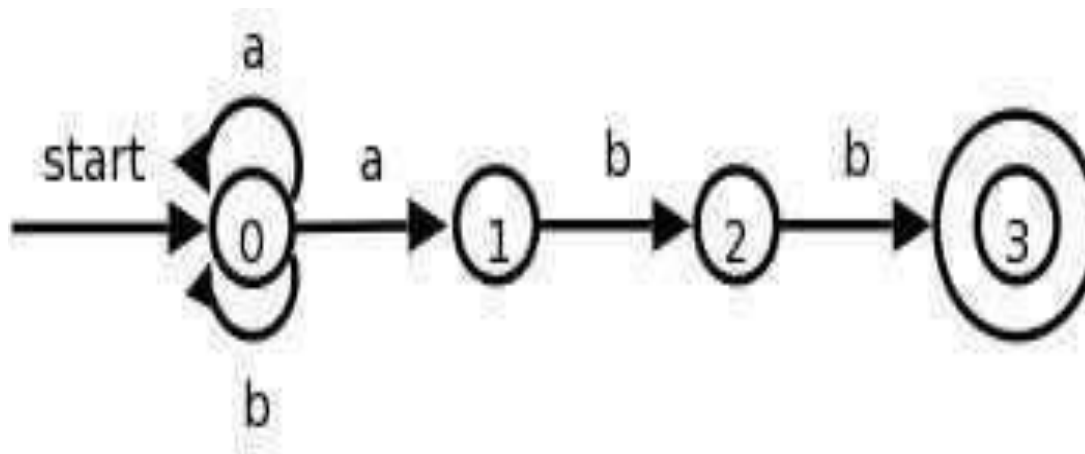
Pass and Phases of Translation:

A compiler can have many phases and passes.

- **Pass**: A pass refers to the traversal of a compiler through the entire program.

- **Phase**: A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage.

- A pass can have more than one phase.

Phases:

Phases are collected into a front end and back end

Frontend:

- The front end consists of those phases, or parts of phase, that depends
  on the source language and is independent of the target machine.

- These normally include lexical and syntactic analysis, the creation of the
  symbol table, semantic analysis, and the generation of intermediate code.

- Code optimization can be done by front end as well.

- The front end also includes the error handling that goes along with each of these phases.

Back end:

- The back end includes those portions of the compiler that depend on the target machine and generally, these portions do not depend on the source language.

Bootstrapping:

- Bootstrapping is the process of writing a compiler in the target programming language which it is intended to complete.

- Applying this technique leads to a self-hosting compiler. A compiler is characterized by three languages:

- Source Language

- Target Language

- Implementation Language
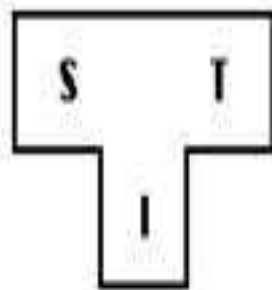
$$^{S}C_{I}^{T}$$

Notation: $^{S}C_{I}^{T}$ represents a compiler for Source *S*, Target *T*, implemented in *I*. The *T-diagram* shown above is also used to depict the same compiler.

- Create $^S C_A^A$, a compiler for a subset, S, of the desired language, L, using language A, which runs on machine A. (Language A may be assembly language.)
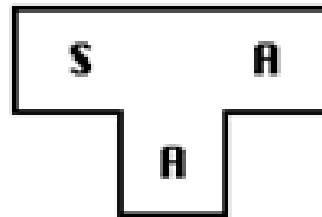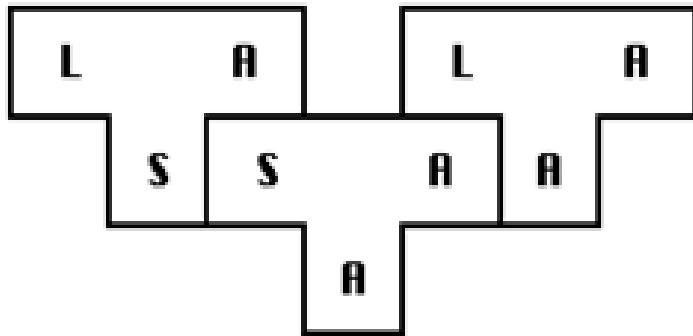

- Create $^L C_S^A$, a compiler for language L written in a subset of L.
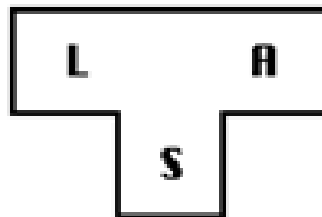
- Compile $^LC^A_S$ using $^SC^A_A$ to obtain $^LC^A_A$, a compiler for language L, which runs on machine A and produces code for machine A.

$$^LC^A_S \rightarrow {}^SC^A_A \rightarrow {}^LC^A_A$$

lexical analyzer with Lex:

- 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 program a.out.

- This a.out is the lexical analyzer that transforms an input stream into a sequence of tokens
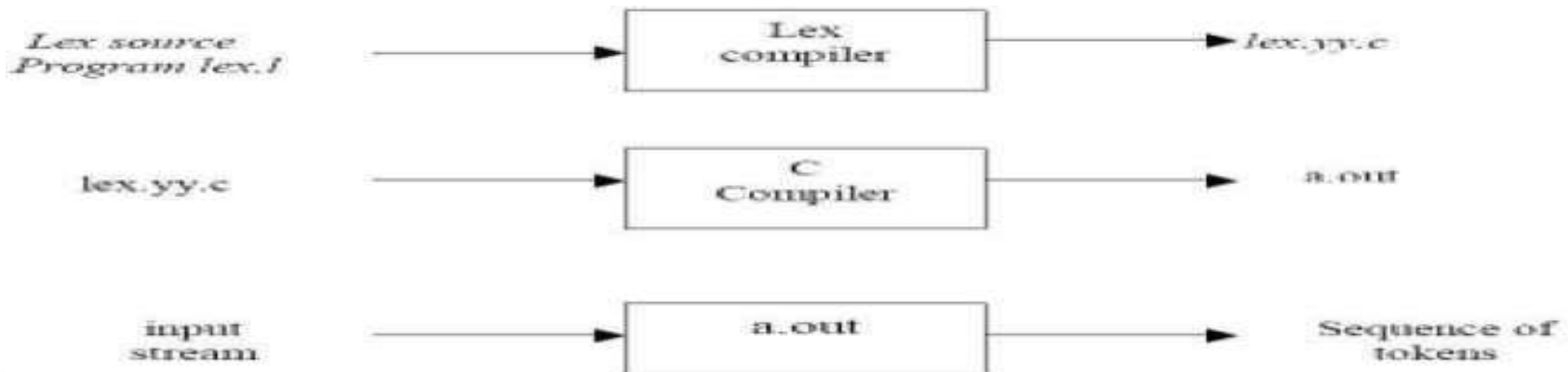
- Lex.l is an a input file written in a language which describes the generation of lexical analyzer.

- lex compiler transforms lex.l to a C program known as lex.yy.c.

- Lex.yy.c is compiled by the C compiler to a file called a.out.

- The output of C compiler is the working lexical analyzer which takes
  stream of input characters and produces a stream of tokens.

- yylval is a global variable which is shared by lexical analyzer and
  parser to return the name and an attribute value of token.

- Another tool for lexical analyzer generation is Flex

Lex Specification:

A Lex program consists of three parts:

- definitions

- rules

- user subroutines

Lex Specification:

- **Definitions** include declarations of variables, constants, and regular definitions

- **Rules** are statements of the form p1 {action1} p2 {action2} … pn {action}

  where pi is regular expression and actioni describes what action the lexical

  analyzer should take when pattern pi 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.

Structure of Lex Programs:

Lex program will be in following form

declarations

%%

translation rules

%%

auxiliary functions

Structure of Lex Programs:

- **Declarations :**This section includes declaration of variables, constants and regular definitions.

- **Translation rules:**   It contains regular expressions and segments.

- **Auxiliary functions:** This section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer.

Parsing:

- Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer.
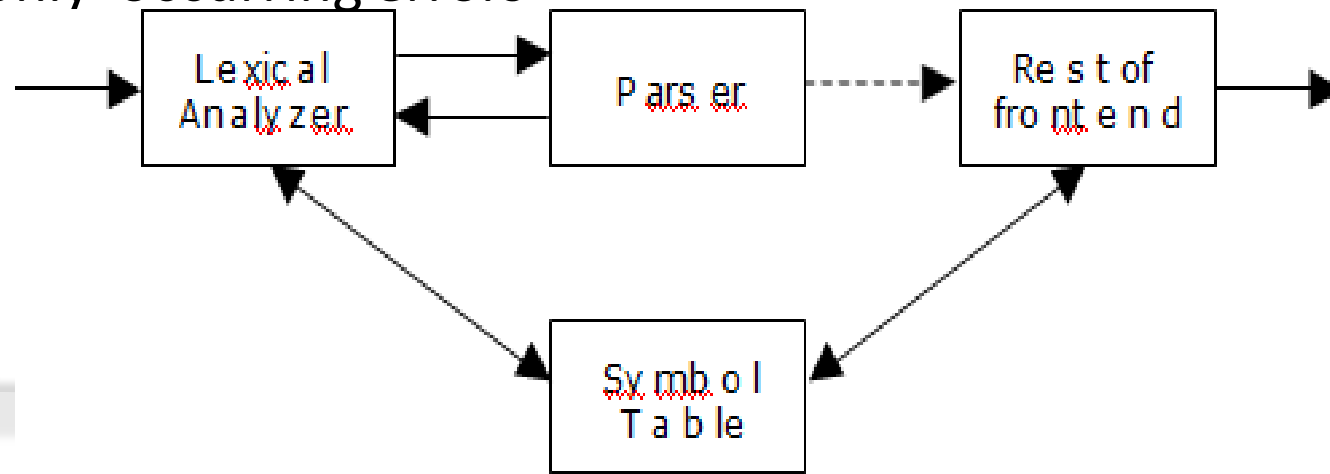
There are two main kinds of parsers

- Top-down

- Bottom-up

- **Top-down:** A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols

- **Bottom-up:** A Bottom-up parser builds the tree starting with the leaves, using productions in reverse to identify strings of symbols that can be grouped together

Role of parser:

- In this process of compilation the parser and lexical analyzer work together. That means, when parser required string of tokens it invokes lexical analyzer .

- In turn, the lexical analyzer supplies tokens to syntax analyzer(parser).

- The parser collects sufficient number of tokens and builds a parse tree.

- By building the parse tree, parser smartly finds the syntactical errors if any. It is also necessary that the parse should recover from commonly occurring errors

```
┌──────────┐      ┌──────────┐        ┌──────────┐
│ Lexical  │ ───> │ Parser   │ -----> │ Rest of  │ ──>
│ Analyzer │ <─── │          │        │ frontend │
└──────────┘      └──────────┘        └──────────┘
         ↖                                ↗
            ┌──────────┐
            │ Symbol   │
            │ Table    │
            └──────────┘
```

67

Context-free Grammars:

Definition:

- Formally, a context-free grammar G is a 4-tuple G = (V, T, P, S), where:
  - V is a finite set of variables (or nonterminals). These describe
    sets of "related" strings.
  - T is a finite set of terminals (i.e., tokens).
  - P is a finite set of productions, each of the form A $\rightarrow \alpha$

- Where A Ɛ V is a variable, and α Ɛ (V ∪ T)* is a sequence of terminals and no terminals.

- S Ɛ V is the start symbol

A context-free grammar has four components:

- A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings.

- A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.

- A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings.

- Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **non-terminals**, called the right side of the production.

Syntax Analyzers:

- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.

- The parser analyzes the source code (token stream) against the production rules to detect any errors in the code.

- The output of this phase is a parse tree
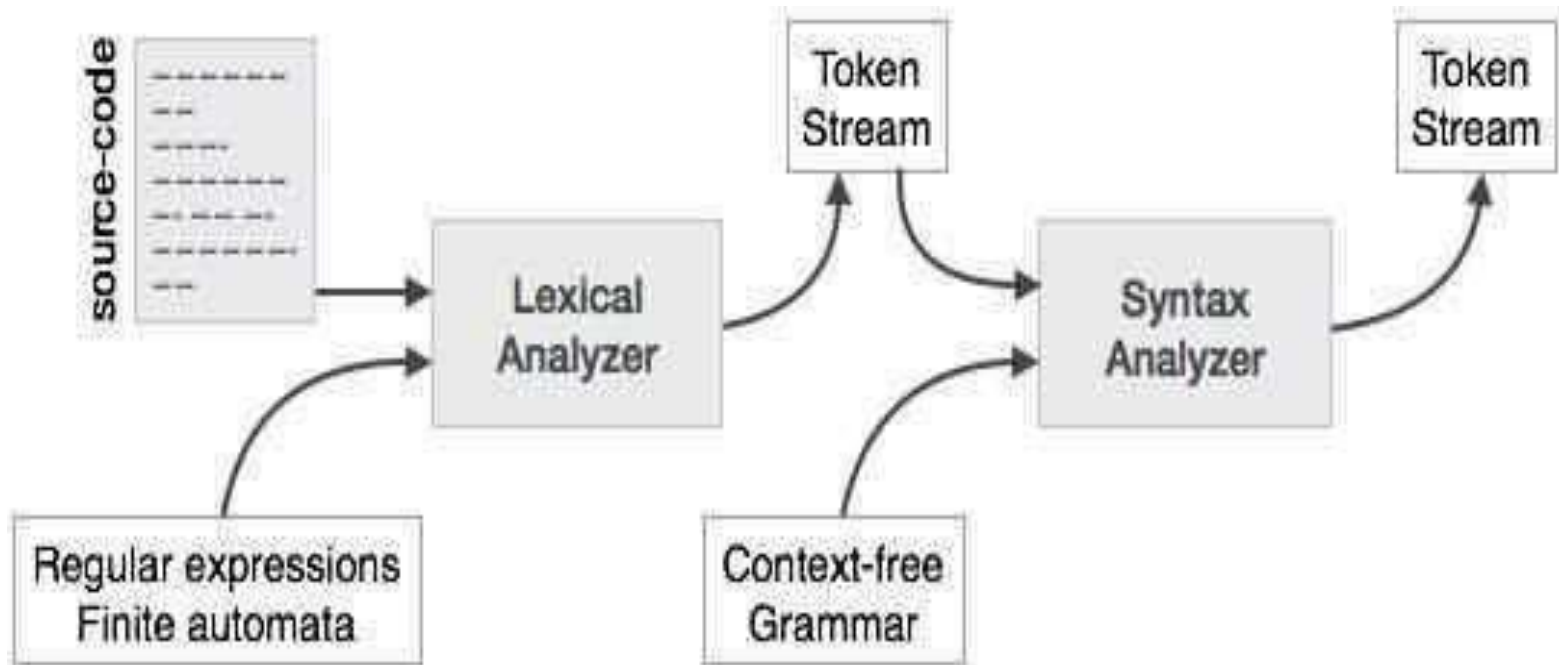
Derivation:

- A derivation is basically a sequence of production rules, in order to get the input string.

- During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

Derivations are two types:

Left-most Derivation:

- If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.

- The sentential form derived by the left-most derivation is called the left-sentential form.

Right-most Derivation:

- If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.

- The sentential form derived from the right-most derivation is called the right-sentential form.

Example:

Production rules:

E $\rightarrow$ E + E

E $\rightarrow$ E * E

E $\rightarrow$ id

Derive the following Input string:

id + id * id

Example:

Left-most Derivation:

E $\rightarrow$ E * E

E $\rightarrow$ E + E * E

E $\rightarrow$ id + E * E E

$\rightarrow$ id + id * E E

$\rightarrow$ id + id * id

Right-most Derivation:

E $\rightarrow$ E + E

E $\rightarrow$ E + E * E

E $\rightarrow$ E + E * id E

$\rightarrow$ E + id * id E

$\rightarrow$ id + id * id

Parse trees:

- A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree.

In a parse tree:

- All leaf nodes are terminals.

- All interior nodes are non-terminals.

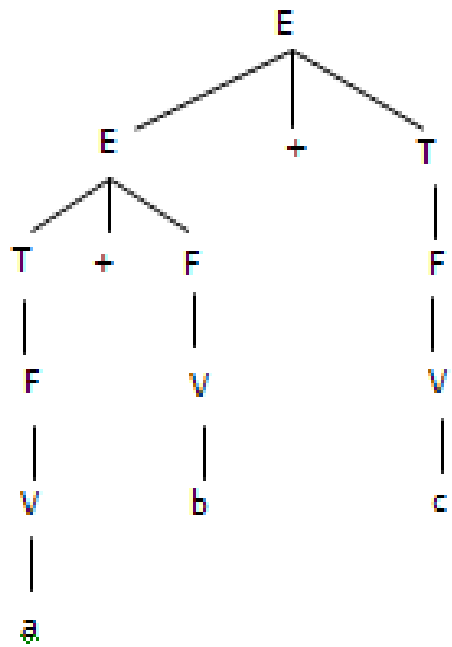- In-order traversal gives original input string.
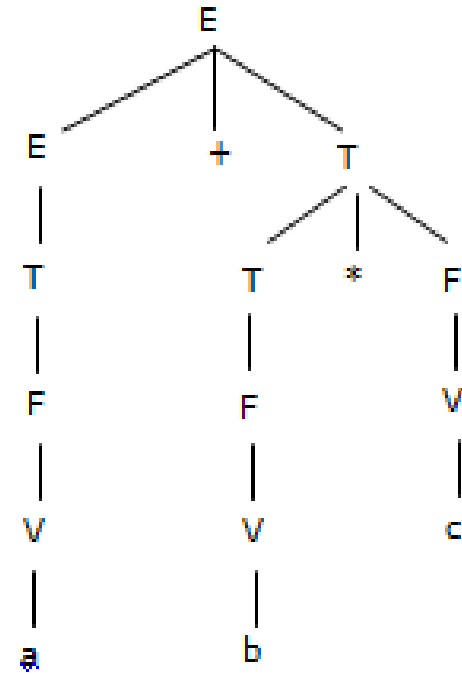
Example:

Let the production P is:

- E→ T | E+T
- T→ F | T*F
- F→ V | (E)
- V→a | b | c |d

Construct the parse tree for the following strings
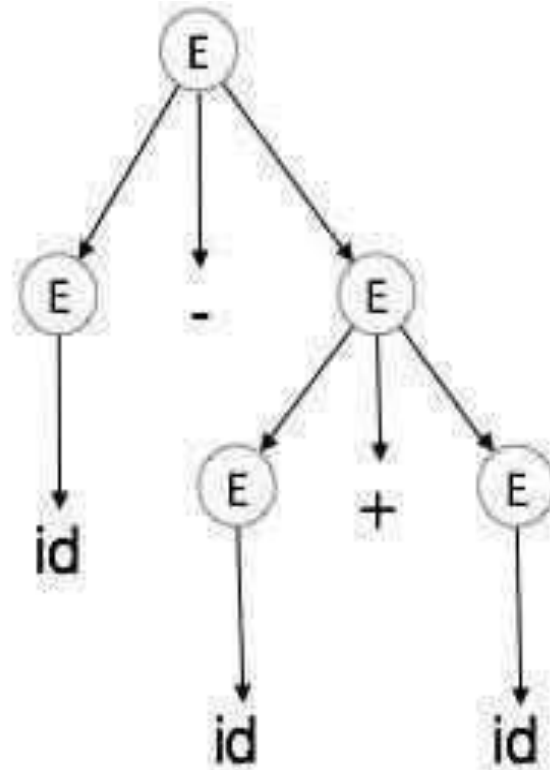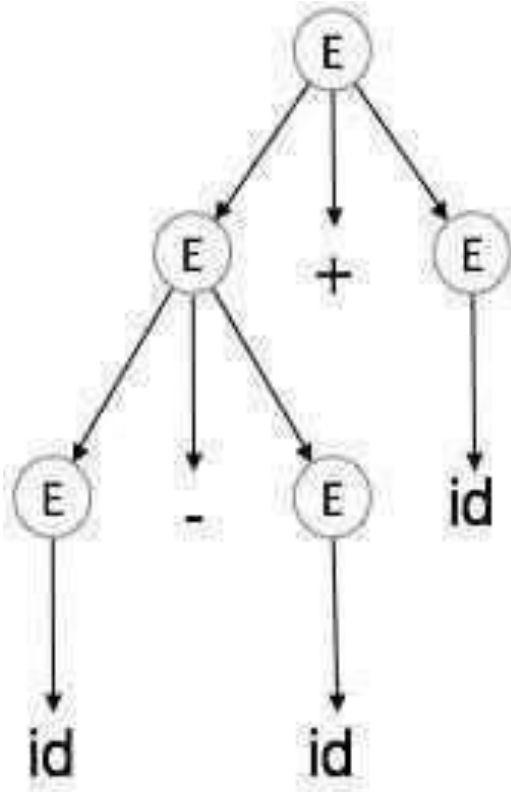
- a * b + c
- a+ b * c

a*b+c

a+b*c

Ambiguity:

- A grammar is ambiguous if there is any sentence with more than one parse tree

- A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

Example:

- E → E + E

- E → E – E

- E → id

- For the string id + id – id, the above grammar generates two parse trees

Left Recursion:

- If there is any non terminal A, such that there is a derivation $A \Rightarrow A\alpha$ for some string a, then the grammar is left recursive.

- Top down parsers cannot handle left recursive grammars.

- If our expression grammar is left recursive:

- This can lead to non termination in a top-down parser.

- For a top-down parser, any recursion must be right recursion.

- Convert the left recursion to right recursion

Algorithm for eliminating left Recursion:

- Replace the above A productions by the following: $A \Rightarrow$

  $\beta1\ A^I \mid \beta2\ A^I \mid - - - \mid \beta n\ A^I$

  $A^I \Rightarrow \alpha1\ A^I \mid \alpha2\ A^I \mid - - - \mid \alpha m\ A^I \mid \varepsilon$

  Where, A^I is a new non terminal.

Example 1:

Remove the left recursion from the production: A → A α| β

Left recursion eliminate by Applying the transformation yields:

- A → β A^I

- A^I → α A^I |ε

Example 2:

Remove the left recursion from the productions:

E → E + T | T   T → T * F | F

Applying the transformation yields:

E → T E^I

T → F T^I

E^I → T E^I |ε

T^I → F T^I | ε

Left factoring:

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

Algorithm:

- For all A Ɛ non-terminal, find the longest prefix α that occurs in two

  or more right-hand sides of A

        If $\alpha \neq \varepsilon$ then replace all of the A productions,

        A → α βI | α β2 | - - - | α βn | r

With

A → α A^I | r

A^I → βI | β2| - - - | βn | ε

Where, A^I is a new element of non-terminal.

Algorithm:

- Repeat until no common prefixes remain.

- It is easy to remove common prefixes by left factoring, creating new non-terminal

  Let the given grammar:

  A→ab1 | ab2 | ab3

- We can see that, for every production, there is a common prefix & if we choose any production here, it is not confirmed that we will not need to backtrack.

  A →aA'

  A'→ b1 | b2| b3

Example:

- Left factoring is required to eliminate non-determinism of a grammar

  Suppose a grammar, S

  -> abS | aSb

- Here, S is deriving the same terminal a in the production rule (two alternative choices for S), which follows non-determinism.

  S →aS'

  S' → bS | Sb

  Thus, S' can be replaced for bS or Sb

Ambiguity:

- A grammar is said to be ambiguous if it produces more than one parse tree for some sentence.

- An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence
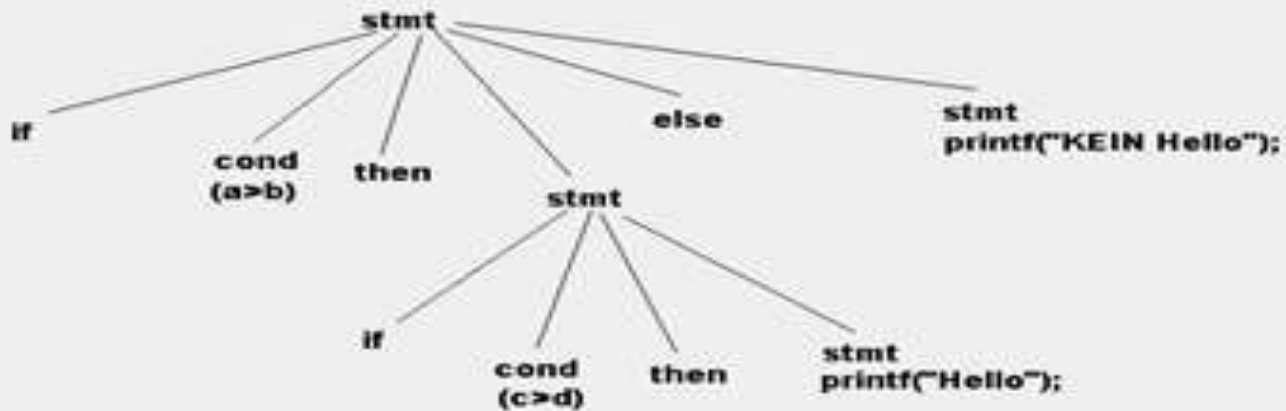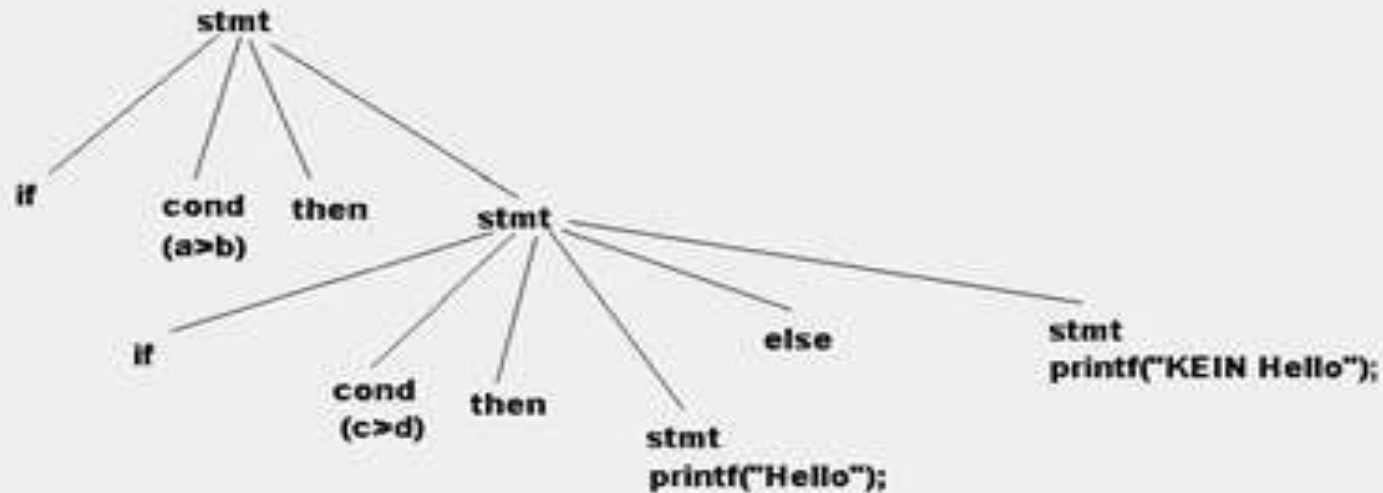
Consider the following 'dangling else' grammar

- stmt → **if** cond **then** stmt

    | **if** cond **then** stmt **else** stmt

    | **other**

Here "**other**" stands for any other statement.

- According to this grammar, the compound conditional statement: if

(a>b) then

    if(c>d) then

         printf("Hello");

    else

         printf("KEIN Hello");

- The above grammar has 2 different parse trees

- The problem here is that in the two different parse trees the 'else' is matched with different 'then'.

- In all programming languages with conditional statements of this form, the first parse tree is preferred .

- The general rule is " Match each else with the closest previous unmatched then".

- The disambiguating rule can be incorporated directly into the grammar.

- For example, we can rewrite the previous grammar as the following unambiguous grammar.

- The idea is that a statement appearing between a then and an else must be "matched" i.e., it must not end with an unmatched then followed by a statement, for the else would then be forced to match with this unmatched then.

- A matched statement is either an if-then-else statement containing no unmatched statement or it is any other kind of unmatched statement.

- **unambiguous grammar** :

stmt→ matched_stmt | unmatched_stmt

matched_stmt→**if** cond **then** matched_ stmt **else** matched_stmt

|**other**

unmatched_stmt→**if** cond **then** stmt

| **if** cond **then** matched_stmt **else** unmatched_stmt

- The grammar generates only one parsing for the given input string and that corresponds to the first tree.

Classes of parsing:

- There are two parsing techniques, these parsing techniques work on the following principle

- The parser scans the input string from left to right and identifiers that the derivation is leftmost or rightmost

Two types of parsing:

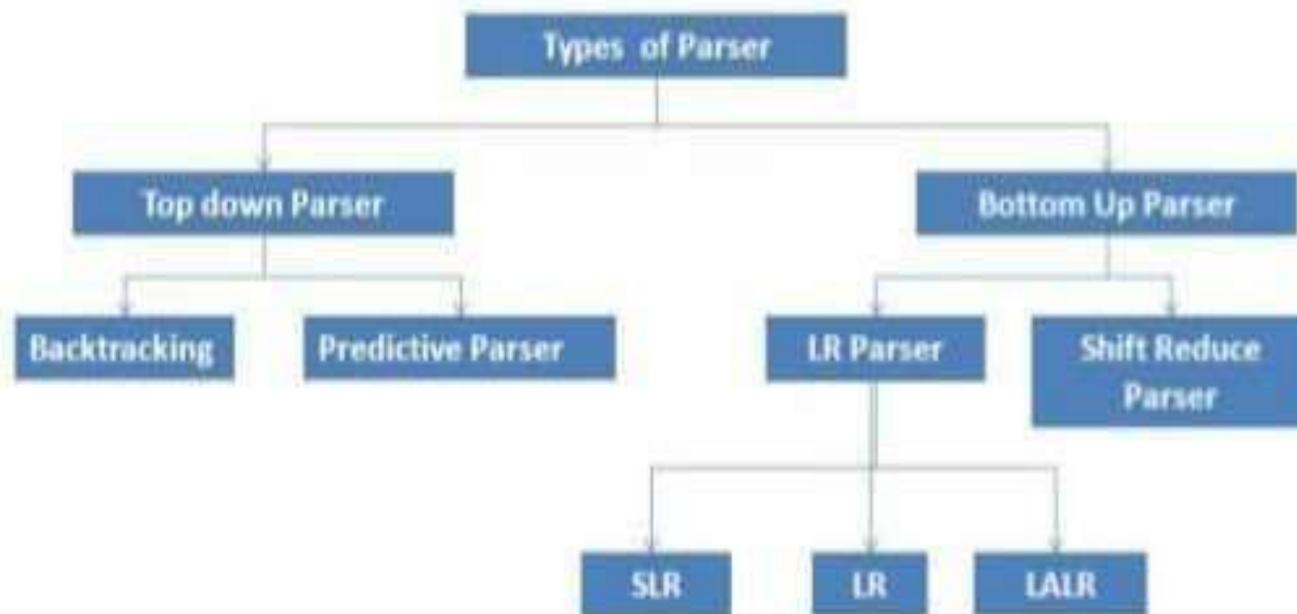- Top down parsing

- Bottom up parsing

Top down parsing:

- When the parse tree can be constructed from root and expanded to leaves, then such type of parse is called top-down parser.

- The name itself tells us that the parse tree can be built from top to bottom.

Bottom up parsing:

- When the parse tree can be constructed from leaves to root, then such type of parse is called as bottom–up parse.

- Thus the parse tree is built in bottom up manner.

Types of Parser

Top-down parsing:

Recursive descent parsing :

- It is a common form of top-down parsing.

- It is called recursive as it uses recursive procedures to process the input.

- Recursive descent parsing suffers from backtracking.

Backtracking :

- It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production.

- This technique may process the input string more than once to determine the right production.
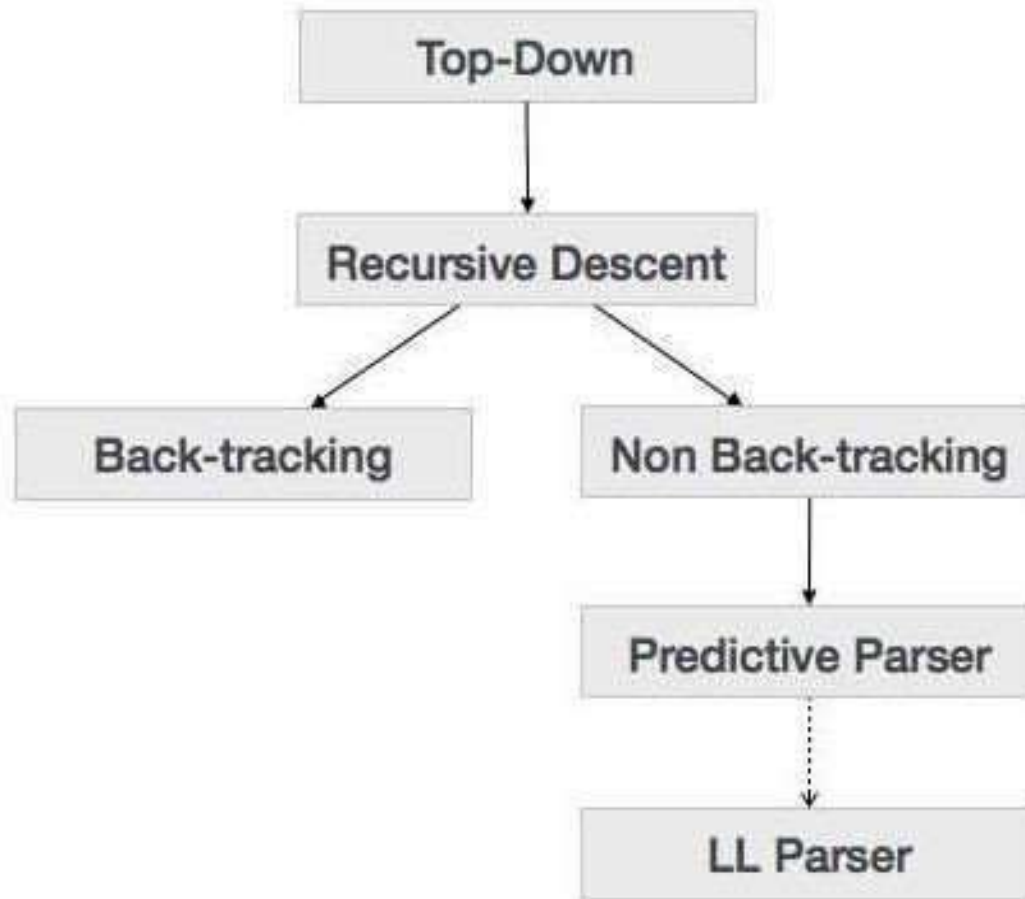
Example:

S → if E then S else S

   | while E do S

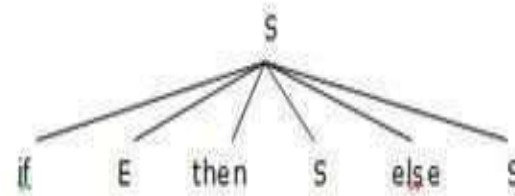 | print E

→ true | False | id

The input token string is:
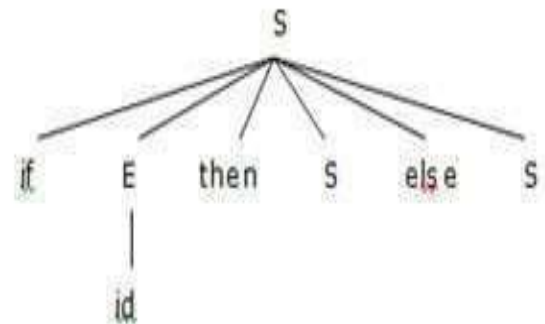
If id then while true do print else print.
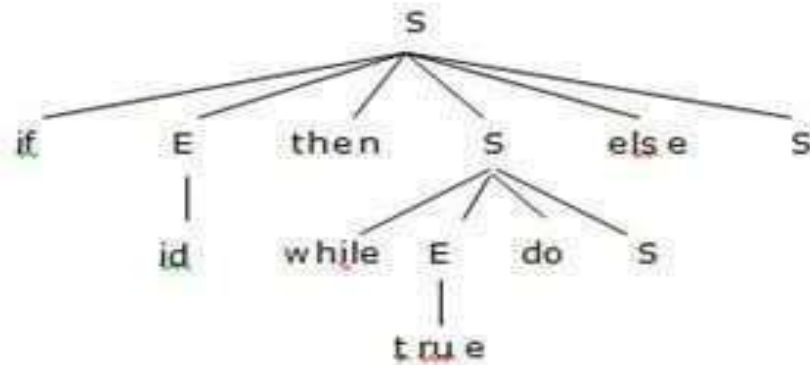

Step1:
  S

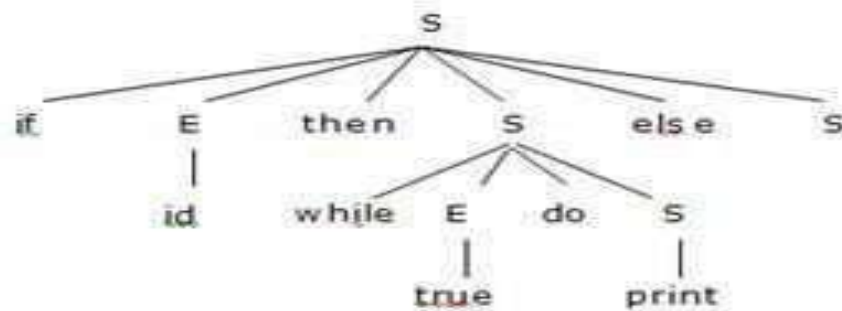Step2:

Step3:

Step4:
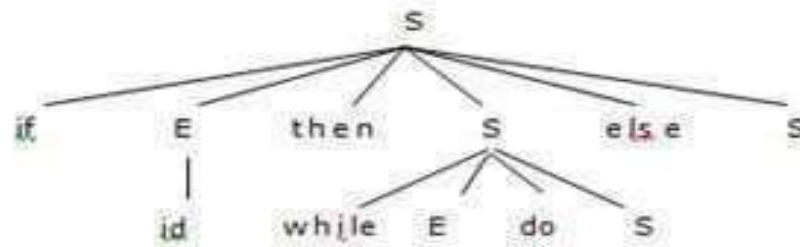
Step5:

Step6:

Step7:
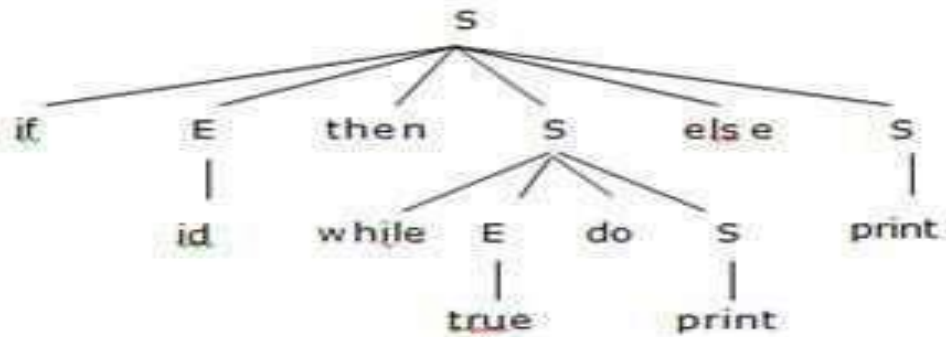
Backtracking:

- Top- down parsers start from the root node (start symbol) and match the
  input string against the production rules to replace them (if matched).

- To understand this, take the following example of CFG

  S → rXd | rZd

  X → oa | ea Z

  → ai

For an input string: read, a top-down parser, will behave like this:

- It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'.

- The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. 'e').

- The parser tries to expand non-terminal 'X' and checks its production from the left (X → oa).

- It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

- Now the parser matches all the input letters in an ordered manner. The string is accepted.

Step1:

Step2:

Step3:
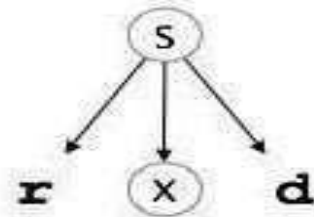
back-tracking
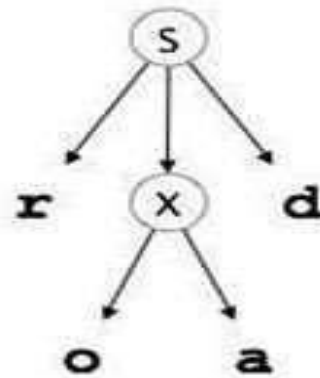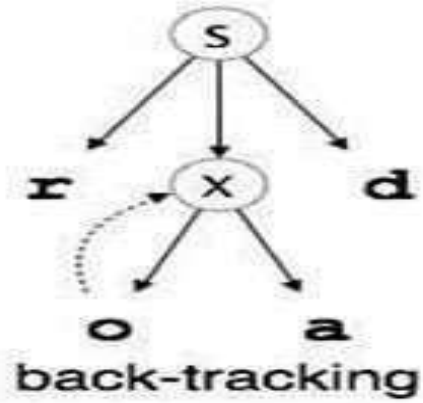
Step4:
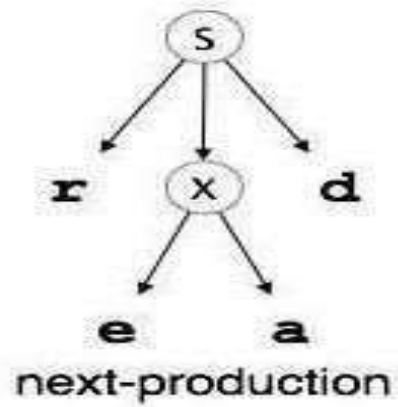
next-production

Recursive-descent parsing:

Recursive descent parser is a top-down parser.

- It requires backtracking to find the correct production to be applied.

- The parsing program consists of a set of procedures, one for each non-terminal.

- Process begins with the procedure for start symbol.

- Start symbol is placed at the root node and on encountering each non- terminal, the procedure concerned is called to expand the non-terminal with its corresponding production.

- Procedure is called recursively until all non-terminals are expanded.

- Successful completion occurs when the scan over entire input string is done. ie., all terminals in the sentence are derived by parse tree

Algorithm for Recursive-descent parsing:

void A()

{

    choose an A-production, A ----> $X_1 X_2 X_3 ... X_k$;

        for (i = 1 to k)

        if (X_i is a non-terminal)

            call procedure X_i ();

        else if (X_i equals the current input

            symbol a) advance the input to

            the next symbol;

        else error;

}

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( );

Recursive procedure:

EPRIME( );

end Procedure

T( ) begin

F( );

TPRIME();

end

Procedure TPRIME( )

begin

If input_symbol='*' then

Recursive procedure:

ADVANCE( );

F( );

TPRIME( );

end Procedure

F( ) begin

If input-symbol='id' then

ADVANCE( );

else if input-symbol='(' then

ADVANCE( );

Recursive procedure:

E( );

else if input-symbol=')' then

ADVANCE( );

end

else

ERROR( );

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

- FIRST

- FOLLOW

- If α is any string of grammar symbols, let FIRST (α) be the set of terminals that begin the strings derived from α. If α=>€,then € is also in FIRST(α).

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

- The set of terminals a such that there exist a derivation of the form S=>αAaβ for some α and β. If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

Rules for first( ):

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

- If X → ε is a production, then add ε to FIRST(X).

- If X is non-terminal and X → aα is a production then add a to FIRST(X).

- If X is non-terminal and X → Y1 Y2…Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1); that is, Y1,….Yi-1=> ε. If ε is in FIRST (Yj) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow( ):

- If S is a start symbol, then FOLLOW(S) contains $.

- If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in follow(B).

- If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Example:

Construct the FIRST and FOLLOW for the grammar:

A → BC | EFGH | H

B → b

C → c | ε

E → e | ε

F →CE

G → g

H →h | ε

Solution:

Finding first () set:

first (H) = first (h) ∪ first (ε) = {h, ε}

first (G) = first (g) = {g}

first (C) = first (c) ∪ first (ε) = {c, ε}

first (E) = first (e) ∪ first (ε) = {e, ε}

first (F) = first (CE) = (first (c) - {ε}) ∪ first (E)

  = (c, ε} {ε}) ∪ {e, ε} = {c, e, ε}

first (B) = first (b)={b}

Solution:

Finding first () set:

first (A) = first (BC) ∪ first (EFGH) ∪ first (H)

$\qquad$ = first (B) ∪ (first (E) − {ε}) ∪ first (FGH) ∪ {h, ε}

$\qquad$ = {b, h, ε} ∪ {e} ∪ (first (F) − {ε}) ∪ first (GH)

$\qquad$ = {b, e, h, ε} ∪ {C, e} ∪ first (G)

$\qquad$ = {b, c, e, h, ε} ∪ {g} = {b, c, e, g, h, ε}

Finding follow() sets:

follow(A) = {$}

follow(B) = first(C) − {ε} ∪ follow(A) = {C, $}

follow(G) = first(H) − {ε} ∪ follow(A)

   ={h, ε} − {ε} ∪ {$} = {h, $}

follow(H) = follow(A) = {$} follow(F) =

first(GH) − {ε} = {g}

follow(E) = first(FGH) m- {ε} ∪ follow(F)

   = ((first(F) − {ε}) ∪ first(GH)) − {ε} ∪ follow(F)

   = {c, e} ∪ {g} ∪ {g} = {c, e, g}

   follow(C) = follow(A) ∪ first (E) − {ε} ∪ follow (F)

      ={$} ∪ {e, ε} ∪ {g} = {e, g, $}

Predictive parser:

- 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

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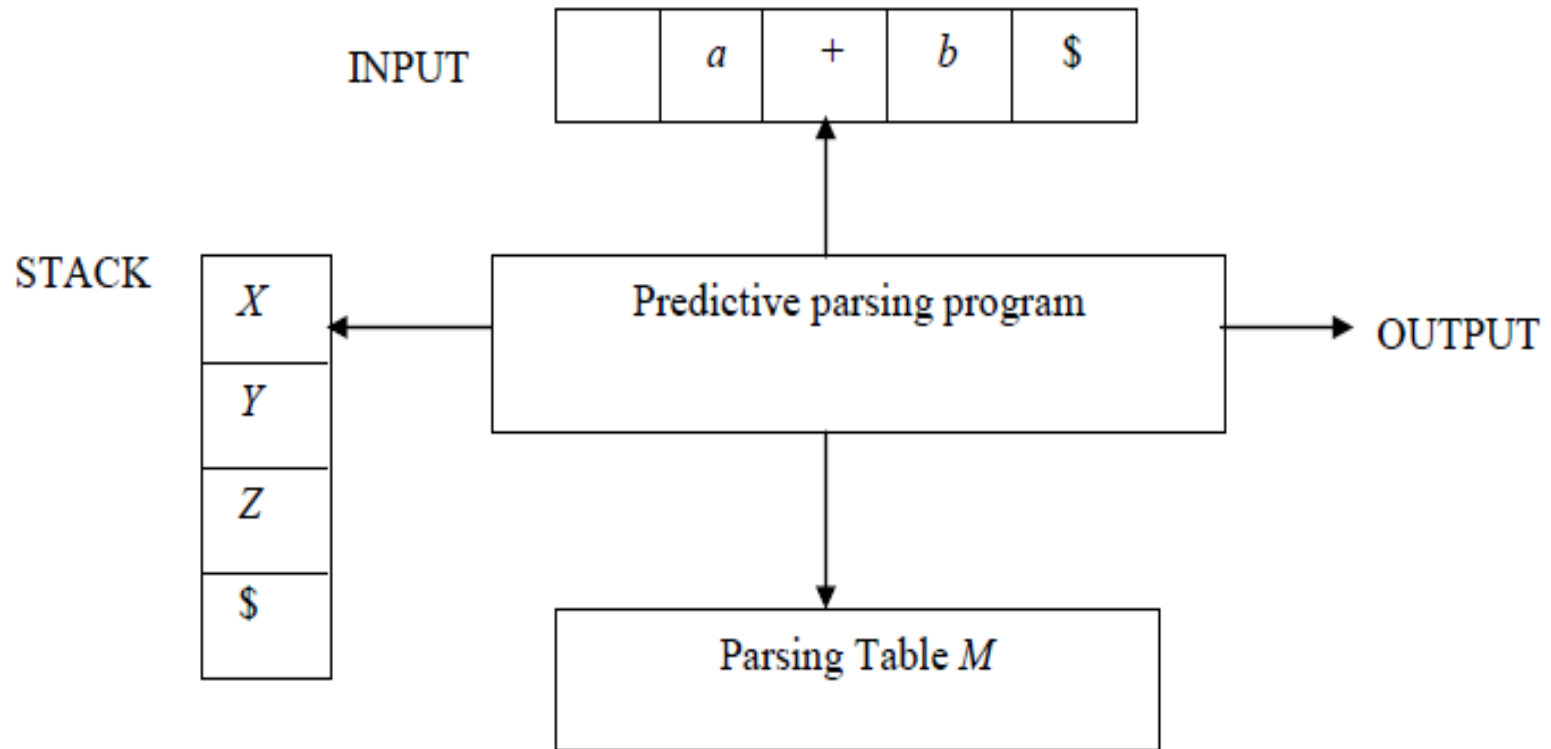
Input buffer:

- 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

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

- For each production A → α of the grammar, do steps 2 and 3.

- For each terminal a in FIRST(α), add A → α to M[A, a].

- 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 → α to M[A, $].

- Make each undefined entry of M be error.

Example :

Construct a predictive parsing table for the given grammar

Consider the following grammar :

E → E+T | T

T → T*F | F

F → (E) | id

Step 1:

- Suppose if the given grammar is left Recursive then convert the given grammar (and) into non-left Recursive grammar (as it goes to infinite loop). ⬚

- After eliminating left-recursion the grammar is

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Step 2:

- Find the FIRST(X) and FOLLOW(X) for all the variables.

- The variables are: {E, E^I, T, T^I, F} Terminals are: {+, *, (,), id} and $

Computation of FIRST() sets:

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

FIRST (T^I) = FIRST (*FT^I) U FIRST (ε) = {*, ε}

FIRST (T) = FIRST (FT^I) = FIRST (F) = {(, id}

FIRST (E^I) = FIRST (+TE^I) U FIRST (ε) = {+, ε}

FIRST (E) = FIRST (TE^I) = FIRST (T) = {(, id}

Computation of FOLLOW() sets:

FOLLOW (E) = {$} U FIRST ( )) = {$,)}

FOLLOW (E^I) = FOLLOW (E) = {$,)}

FOLLOW (T) = (FIRST (E^I) - {ε}) U FOLLOW (E) U FOLLOW (E^I)

$\qquad$ = {+,), $}

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

FOLLOW (F) = (FIRST (T^I) - {ε}) U FOLLOW (T) U FOLLOW (T^I)

$\qquad$ = {*, +,), $}

Step 3:
Construction of parsing table:

Terminals / Variables	+	*	(	)	id	$
E			$E \rightarrow TE^I$		$E \rightarrow TE^I$	
E^I	$E^I \rightarrow +TE^I$			$E^I \rightarrow \epsilon$		$E^I \rightarrow \epsilon$
T			$T \rightarrow FT^I$		$T \rightarrow FT^I$	
T^I	$T^I \rightarrow \epsilon$	$T^I \rightarrow *FT$		$T^I \rightarrow \epsilon$		$T^I \rightarrow \epsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

Step 4:

Moves made by predictive parser on the input id + id * id is:

STACK	INPUT	REMARK
		S
$ E	id + id * id $	E and id are not identical; so see E on id in parse table, the production is E→TE^I; pop E, push E^I and T i.e., move in reverse order.
$ E^I T	id + id * id $	See T on id the production is T → F T^I ; Pop T, push T^I and F; Proceed until both are identical.
$ E^I T^I F	id + id * id $	F → id
$ E^I T^I id	id + id * id $	Identical; pop id and remove id from input symbol.
$ E^I T^I	+id * id $	See T^I on +; T^I → ∈ so, pop T^I

$\$ E^I$	$+ id * id \$$	See E^I on $+$; $E^I \to +T\,E^I$; push E^I, $+$ and T
$\$ E^I T +$	$+ id * id \$$	Identical; pop $+$ and remove $+$ from input symbol.
$\$ E^I T$	$id * id \$$	
$\$ E^I T^I F$	$id * id \$$	$T \to F\,T^I$
$\$ E^I T^I id$	$id * id \$$	$F \to id$
$\$ E^I T^I$	$* id \$$	
$\$ E^I T^I F *$	$* id \$$	$T^I \to * F\,T^I$

$E^I T^I F$	id $	
$E^I T^I$ id	id $	$F \rightarrow id$
$E^I T^I$	$	$T^I \rightarrow \epsilon$
E^I	$	$E^I \rightarrow \epsilon$
$	$	Accept.

LL(1) grammar

LL(1) grammars:

- A grammar is LL(1) if it is possible to choose the next production by looking at only the next token in the input string.

- The first "L" in LL(1) stands for scanning the input from left to right.

- The second "L" stands for producing a leftmost derivation.

- The "1" stands for using one input symbol of look a head at each step in making parsing action decisions.

- No LL (1) grammar can be ambiguous or left recursive.

LL(1) grammars:

- If there were no multiple entries in the Recursive decent parser table, the given grammar is LL (1).

- If the grammar G is ambiguous, left recursive then the recursive decent table will have atleast one multiply defined entry.

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^1		
S^1			S^1→eS S^1→ε			S^1→ε
E		E→b				

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

Example:

- Construct a predictive parsing table for the given grammar to check whether given grammar is LL(1) or Not.

- Consider the following grammar:

    S → AC$

    C → c | ε

    A → aBCd | BQ | ε

    B → bB | d   Q → q

Example:

Solution:

Finding the first () sets:

First (Q) = {q}

First (B) = {b, d}

First (C) = {c, ε}

First (A) = First (aBCd) ∪ First (BQ) ∪ First (ε)

        = {a} ∪ First (B) ∪ First (d) ∪{ε}

        = {a} ∪First (bB) ∪ First (d) ∪ {ε}

        = {a} ∪ {b} ∪ {d} ∪ {ε}

        = {a, b, d, ε}

Example:

Solution:

First (S) = First (AC$)

$\qquad$ = (First (A) − {ε}) ∪ (First (C) − {ε}) ∪ First (ε)

$\qquad$ = ({a, b, d, ε} − {ε}) ∪ ({c, ε} − {ε}) ∪ {ε}

$\qquad$ = {a, b, d, c, ε}

Example:

Solution:

Finding Follow () sets:

Follow (S) = {$}

Follow (A) = (First (C) − {ε}) ∪ First ($) = ({c, ε} − {ε}) ∪ {$}

Follow (A) = {c, $}

Follow (B) = (First (C) − {ε}) ∪ First (d) ∪ First (Q)

$\qquad$ = {c} ∪ {d} ∪{q}

$\qquad$ = {c, d, q}

Follow (C) = (First ($) ∪ First (d) = {d, $}

Follow (Q) = (First (A) = {c, $}

The parsing table for this grammar is:

Non-terminal	a	b	c	D	q	$
S	S→AC$	S→AC$	S→AC$	S→AC$		S→AC$
A	A→aBCd	A→BQ	A→ε	A→BQ		A→ε
B		B→bB		B→d		
C			C→c	C→ε		C→ε
Q					Q→q	

Since there was only single entry in each column, so the grammar is LL(1) grammar.

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, —Compilers–Principles, Techniques and Tools‖, Pearson Education, Low Price Edition, 2004

BOTTOM-UP PARSING

The course will able the students to	
CLO4	Explain the role of a parser in a compiler and relate the yield of a parse tree to a grammar derivation.
CLO5	Apply an algorithm for a top-down or a bottom-up parser construction; construct a parser for a given context-free grammar.
CLO6	Demonstrate Lex tool to create a lexical analyzer and Yacc tool to create a parser.

Handle:

- Always making progress by replacing a substring with LHS of a matching production will not lead to the goal/start symbol.

- A Handle of a string is a substring that matches the right side of a production

- If the grammar is unambiguous, every right sentential form has exactly one handle.

- More formally, A handle is a production $A \rightarrow \beta$ and a position in the current right-sentential form $\alpha\beta\omega$ such that:

    $S \Rightarrow \alpha A\omega \Rightarrow \alpha/\beta\omega$

- Conflicts Type
  - shift-reduce
  - reduce-reduce

- Shift-reduce and reduce-reduce conflicts are caused by
  - The limitations of the LR parsing method (even when the grammar is unambiguous)
    - Ambiguity of the grammar

Grammar
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow \mathbf{id}$

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce $E \rightarrow \mathbf{id}$
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce $E \rightarrow \mathbf{id}$
$E+E	*id$	**shift (or reduce?)**
$E+E*	id$	shift
$E+E*id	$	reduce $E \rightarrow \mathbf{id}$
$E+E*E	$	reduce $E \rightarrow E * E$
$E+E	$	reduce $E \rightarrow E + E$
$E	$	accept

How to resolve conflicts?

Find handles to be reduced

Ambiguous grammar:

$S \rightarrow$ **if** E **then** S
    | **if** E **then** S **else** S
    | **other**

Resolve in favor of shift, so **else** matches closest **if**

Stack	Input	Action
$\$...$	$...\$$	...
$\$...$ if E then S	else...$\$$	shift or reduce?

Shift **else** to **if** E **then** S or
Reduce **if** E **then** S

Grammar

$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$

Stack	Input	Action
$	aa$	shift
$a	a$	reduce $A \rightarrow \mathbf{a}$ <u>or</u> $B \rightarrow \mathbf{a}$?

Resolve in favor
of reduce $A \rightarrow \mathbf{a}$,
otherwise we're stuck!

Stack	Input	Action
$	Id - num * id	Shift
$ id	- num * id	Reduce factor → id
$ Factor	- num * id	Reduce Term → Factor
$ Term	- num * id	Reduce Expr → Term
$ Expr	- num * id	Shift
$ Expr -	num * id	Shift
$ Expr – num	* id	Reduce Factor → num
$ Expr – Factor	* id	Reduce Term → Factor
$ Expr – Term	* id	Shift
$ Expr – Term *	id	Shift
$ Expr – Term * id	-	Reduce Factor → id
$ Expr – Term & Factor	-	Reduce Term → Term * Factor
$ Expr – Term	-	Reduce Expr → Expr – Term
$ Expr	-	Reduce Goal → Expr
$ Goal	-	Accept

- shift until the top of the stack is the right end of a handle
- Find the left end of the handle & reduce.

Procedure:

1.Shift until top of stack is the right end of a handle.

2.   Find the left end of the handle and reduce.

•Dangling-else problem:

•stmt→if expr then stmt/if expr then stmt/other then example string is: if E_1 then if E_2 then S_1 else S_2 has two parse trees (ambiguity) and so this grammar is not of LR(k) type.

- LR parser are table-driven
  - Much like the non recursive LL parsers.
- The reasons of the using the LR parsing
  - An LR-parsing method is the most general non backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other.
  - LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written

- An LR parser makes shift-reduce decisions by maintaining states to keep track.

- An item of a grammar G is a production of G with a dot at some position of the body.

  - Example

    $A \rightarrow X \, Y \, Z$

    $A \rightarrow \, \cdot \, X \, Y \, Z$

$$A \rightarrow X \cdot Y \, Z$$

stack   next derivations
with input strings

Figure 4.35: Model of an LR parser

- An LR parser makes shift-reduce decisions by maintaining states to keep track.
- An item of a grammar G is a production of G with a dot at some position of the body.

  - Example

    $A \rightarrow X\,Y\,Z$

    $A \rightarrow \cdot\,X\,Y\,Z$

    $A \rightarrow X \cdot Y\,Z$

    $A \rightarrow X\,Y \cdot$

    $Z\;A \rightarrow X\,Y\,Z$

    ·

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

Create *closure*(*I*) by the two
   rules: add every item in *I* to
   closure(*I*)
   If $A \rightarrow \alpha \cdot B\beta$ is in *closure*(*I*) and $B \rightarrow \gamma$ is a production, then add the item *B*
   $\rightarrow \cdot \gamma$ to *closure*(*I*). Apply this rule untill no more new items can be added
   to *closure*(*I*).

Divide all the sets of items into two
   Kernel items
      initial item S' $\rightarrow$ · S, and all items whose dots are not at the left end.
   Nonkernel items
      All items with their dots at the left end, except for S' $\rightarrow$ · S

- The grammar G

$$E' \rightarrow E$$
$$E \rightarrow E + T \qquad | \qquad T$$
$$T \rightarrow T * F \qquad\qquad |$$
$$F \quad F \quad \rightarrow (E) | \qquad\qquad \textbf{id}$$

- Let $I = \{ E' \rightarrow \cdot E \}$, then

$$closure(I) \quad = \qquad \{$$
$$E' \rightarrow \cdot E$$
$$E \rightarrow \cdot E + T \quad E \rightarrow \cdot T$$
$$T \rightarrow \cdot T * F$$
$$T \rightarrow \cdot F$$
$$F \rightarrow \cdot (E)$$
$$F \rightarrow \cdot \textbf{id} \quad \}$$

- Function *Goto*(*I, X*)

  - I is a set of items

  - X is a grammar symbol

  - *Goto*(*I, X*) is defined to be the closure of the set of all items [*A* $\rightarrow$ α *X* ·β] such that [*A* $\rightarrow$ α · *X*β] is in *I*.

  - Goto function is used to define the transitions in the LR(0) automation for a grammar.

$I = \{$

$E' \rightarrow E \cdot$

$E \rightarrow E \cdot + T \}$

$Goto \ (I, +) = \{$

$E \rightarrow E + \cdot \ T$

$T \rightarrow \cdot \ T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \ \mathbf{id}$

$\quad \}$

The grammar G

$$E' \rightarrow E$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (\ E \ ) \mid \text{id}$$

1. The grammar is augmented with a new start symbol S' and production $S' \rightarrow S$

2. Initially, set $C = closure(\{[S' \rightarrow \bullet S]\})$ (this is the start state of the DFA)

3. For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$
    such that
    $GOTO(I, X) \notin C$ and $goto(I, X) \neq \varnothing$, add the set of items $GOTO(I, X)$ to C

4. Repeat 3 until no more sets can be added to C

- SLR (Simple LR): a simple extension of LR(0) shift-reduce parsing

- SLR eliminates some conflicts by populating the parsing table with reductions $A \rightarrow \alpha$ on symbols in FOLLOW(A)

- Function *Goto*(I, X)

  - I is a set of items

  - X is a grammar symbol

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

  - Goto function is used to define the transitions in the LR(0) automation for a grammar.

$S \to E$
$E \to \text{id} + E$
$E \to \text{id}$

$\Rightarrow$

State I_0:
$S \to \cdot E$
$E \to \cdot \text{id} + E$
$E \to \cdot \text{id}$

$goto(I_0, \text{id})$

State I_2:
$E \to \text{id} \cdot + E$
$E \to \text{id} \cdot$

$goto(I_3, +)$

Shift on **+**

FOLLOW(E)={$}
thus reduce on $

- Reductions do not fill entire rows
- Otherwise the same as LR(0)

1. $S \rightarrow E$
2. $E \rightarrow id + E$
3. $E \rightarrow id$

	id	+	$	E
0	s2			1
1			acc	
2		s3	r3	
3	s2			4
4			r2	

Shift on +

FOLLOW(E)={$}
thus reduce on $

State I_0:
$C' \rightarrow \cdot C$
$C \rightarrow \cdot A\,B$
$A \rightarrow \cdot \mathbf{a}$

State I_1:
$C' \rightarrow C\cdot$

State I_2:
$C \rightarrow A\cdot B$
$B \rightarrow \cdot \mathbf{a}$

State I_3:
$A \rightarrow \mathbf{a}\cdot$

State I_4:
$C \rightarrow A\,B\cdot$

State I_5:
$B \rightarrow \mathbf{a}\cdot$

	a	$	C	A	B
0	s3		1	2	
1		acc			
2	s5				4
3	r2				
4		r1			
5		r3			

Grammar:
0. $C' \rightarrow C$
1. $C \rightarrow A\,B$
2. $A \rightarrow \mathbf{a}$
3. $B \rightarrow \mathbf{a}$

Augmented grammar:

0. $C' \to C$
1. $C \to A\ B$
2. $A \to \mathbf{a}$
3. $B \to \mathbf{a}$

$I_0 = closure(\{[C' \to \bullet C]\})$
$I_1 = goto(I_0, C) = closure(\{[C' \to C\bullet]\})$
…

State I_1:
$C' \to C\bullet$ final

State I_4:
$C \to A\ B\bullet$

$goto(I_0, C)$

start → **State I_0:**
$C' \to \bullet C$
$C \to \bullet A\ B$
$A \to \bullet \mathbf{a}$

$goto(I_0, A)$ → **State I_2:**
$C \to A\bullet B$
$B \to \bullet \mathbf{a}$

$goto(I_2, B)$

$goto(I_2, \mathbf{a})$ → **State I_5:**
$B \to \mathbf{a}\bullet$

$goto(I_0, \mathbf{a})$ → **State I_3:**
$A \to \mathbf{a}\bullet$

Example SLR Parsing Table

State I_0:
$C' \rightarrow \cdot C$
$C \rightarrow \cdot A\,B$
$A \rightarrow \cdot \mathbf{a}$

State I_1:
$C' \rightarrow C \cdot$

State I_2:
$C \rightarrow A \cdot B$
$B \rightarrow \cdot \mathbf{a}$

State I_3:
$A \rightarrow \mathbf{a} \cdot$

State I_4:
$C \rightarrow A\,B \cdot$

State I_5:
$B \rightarrow \mathbf{a} \cdot$

	a	$	C	A	B
0	s3		1	2	
1		acc			
2	s5				4
3	r2				
4		r1			
5		r3			

Grammar:
0. $C' \rightarrow C$
1. $C \rightarrow A\,B$
2. $A \rightarrow \mathbf{a}$
3. $B \rightarrow \mathbf{a}$

- Augment the grammar with $S' \rightarrow S$

- Construct the set $C=\{I_0, I_1, ..., I_n\}$ of LR(1) items

- State i of the parser is constructed from I_i
  - If $[A \rightarrow \alpha \bullet a\beta, b] \in I_i$ and $goto(I_i, a) = I_j$ then set $action[i,a]$=shift j
  - If $[A \rightarrow \alpha \bullet, a] \in I_i$ then set $action[i,a]$=reduce A$\rightarrow \alpha$ (apply only if $A \neq S'$)
  - If $[S' \rightarrow S\bullet, \$]$ is in I_i then set $action[i,\$]$=accept

- If $goto(I_i, A) = I_j$ then set $goto[i,A]=j$

- Repeat 3 until no more entries added

- The initial state i is the I_i holding item $[S' \rightarrow \bullet S, \$]$

Given $S \rightarrow CC$, C
$\rightarrow cC/d$.

1. Number the grammar productions:

- 1. $S \rightarrow CC$
- 2. $C \rightarrow cC$
- 3. $C \rightarrow d$

2. The Augmented grammar is:

- **$S^I \rightarrow S$**
- $S \rightarrow CC$
- $C \rightarrow cC$
- $C \rightarrow d$.

- Constructing the sets of LR(1) items:
- We begin with:
- $S^I \rightarrow .S,\$$ begin with look-a-head (LAH) as $.
- Function closure tells us to add [B$\rightarrow$.r,b] for each production B$\rightarrow$r and terminal b in FIRST (βa).
- Now $\beta \rightarrow$r must be S$\rightarrow$CC, and since β is $\in$ and a is $, b may only be $. Thus,
- **S$\rightarrow$.CC,$**
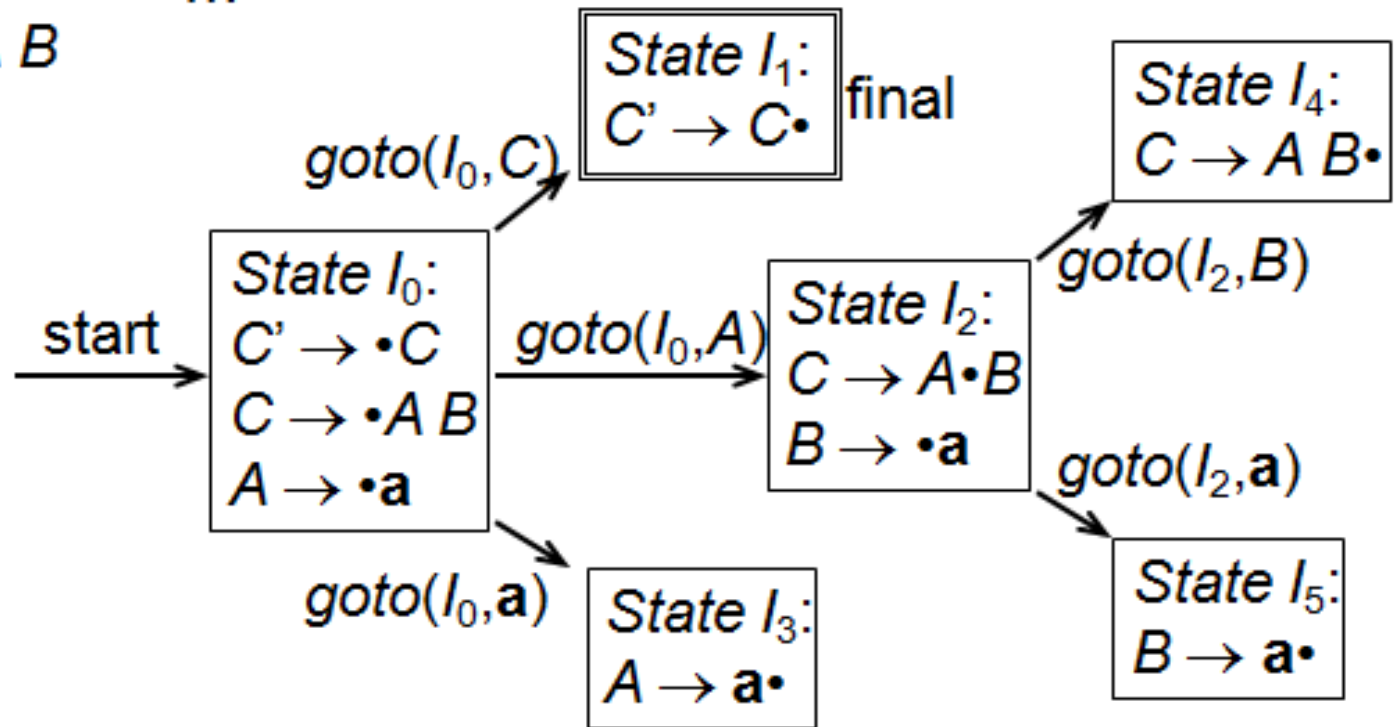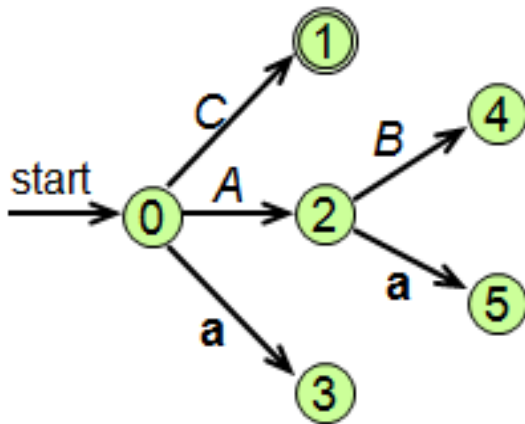
- **S→.CC,$**

FIRST (C$) = FIRST ©

FIRST© = {c,d}

- I$_0$:   S^l→.S,$

          S→.CC,$

          C→.cC,c/

          d

          C→.d.c/d

start computing goto (I$_0$,X) for various non-terminals Goto (I$_0$,S)

Goto (I$_0$,C) ..so on

Example

state	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

The grammar G

$S' \rightarrow S$

$S \rightarrow C\,C$

$C \rightarrow \mathbf{c}\,C \mid \quad \mathbf{d}$

- Unambiguous LR(1) grammar:

$$S \rightarrow L = R \mid R$$
$$L \rightarrow * R \mid \textbf{id}$$
$$R \rightarrow L$$

- Augment with $S' \rightarrow S$

- LR(1) items (next slide)

$I_0:[S' \rightarrow \bullet S, \$]$ $goto(I_0,S)=I_1$

$[S \rightarrow \bullet L=R, \$]$ $goto(I_0,L)=I_2$

$[S \rightarrow \bullet R, \$]$ $goto(I_0,R)=I_3$

$[L \rightarrow \bullet *R, =/\$]$ $goto(I_0,*)=I_4$

$[L \rightarrow \bullet \mathbf{id}, =/\$]$ $goto(I_0,\mathbf{id})=I_5$

$[R \rightarrow \bullet L, \$]$ $goto(I_0,L)=I_2$

$I_1:[S' \rightarrow S\bullet, \$]$

$I_2:[S \rightarrow L\bullet=R, \$]$ $goto(I_0,=)=I_6$

$[R \rightarrow L\bullet, \$]$

$I_3:[S \rightarrow R\bullet, \$]$

$I_4:[L \rightarrow *\bullet R, =/\$]$ $goto(I_4,R)=I_7$

$[R \rightarrow \bullet L, =/\$]$ $goto(I_4,L)=I_8$

$[L \rightarrow \bullet *R, =/\$]$ $goto(I_4,*)=I_4$

$[L \rightarrow \bullet \mathbf{id}, =/\$]$ $goto(I_4,\mathbf{id})=I_5$

$I_5:[L \rightarrow \mathbf{id}\bullet, =/\$]$

$I_6{:}[S \rightarrow L={\bullet}R, \$]$ $\underline{goto}(I_6,R)=I_9$
$[R \rightarrow {\bullet}L, \$]$ $\underline{goto}(I_6,L)=I_{10}$
$[L \rightarrow {\bullet}*R, \$]$ $\underline{goto}(I_6,*)=I_{11}$
$[L \rightarrow {\bullet}\textbf{id}, \$]$ $\underline{goto}(I_6,\textbf{id})=I_{12}$

$I_7{:}[L \rightarrow *R{\bullet},=/\$]$

$I_8{:}[R \rightarrow L{\bullet}, =/\$]$

$I_9{:} [S \rightarrow L=R{\bullet}, \$]$

$I_{10}{:}[R \rightarrow L{\bullet}, \$]$

$I_{11}{:}[L \rightarrow *{\bullet}R, \$]$ $\underline{goto}(I_{11},R)=I_{13}$
$[R \rightarrow {\bullet}L, \$]$ $\underline{goto}(I_{11},L)=I_{10}$
$[L \rightarrow {\bullet}*R, \$]$ $\underline{goto}(I_{11},*)=I_{11}$
$[L \rightarrow {\bullet}\textbf{id}, \$]$ $\underline{goto}(I_{11},\textbf{id})=I_{12}$

$I_{12}{:}[L \rightarrow \textbf{id}{\bullet}, \$]$

$I_{13}{:}[L \rightarrow *R{\bullet}, \$]$

Grammar:
1. S' → S
2. S → L = R
3. S → R
4. L → * R
5. L → **id**
6. R → L

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r6			
3				r3			
4	s5	s4				8	7
5			r5	r5			
6	s12	s11				10	4
7			r4	r4			
8			r6	r6			
9				r2			
10				r6			
11	s12	s11				10	13
12				r5			
13				r4			

LALR

- LookAhead LR

- Try to merge states in LR(1) automata

- When the core items in two LR(1) states are the same

- merge them

state	ACTION			GOTO	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

state	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

- Construct sets of LR(1) items

- Combine LR(1) sets with sets of items that share the same first part

- **Another LALR example**

  Given grammar is

  $S \rightarrow CC$

  $C \rightarrow cC \quad C \rightarrow d$

The LALR parsing diagram and table:

Item sets:

I_0: $S' \rightarrow \bullet S, \$$; $S \rightarrow \bullet CC, \$$; $C \rightarrow \bullet cC, c/d$; $C \rightarrow \bullet d, c/d$

I_1: $S \rightarrow S \bullet, \$$

I_2: $S \rightarrow C \bullet C, \$$; $C \rightarrow \bullet cC, \$$; $C \rightarrow \bullet d, \$$

I_3: $C \rightarrow c \bullet C, c/d /\$$; $C \rightarrow \bullet cC, c/d /\$$; $C \rightarrow \bullet d, c/d /\$$

I_4: $C \rightarrow cC \bullet, c/d /\$$

I_5: $C \rightarrow d \bullet, c/d /\$$

I_6: $S \rightarrow CC \bullet, \$$

	c	d	$	S	C
0	3	5		1	2
1			Acc		
2	3	5			6
3	3	5			4
4	C→cC	C→cC	C→cC		
5	C→d	C→d	C→d		
6			S→CC		
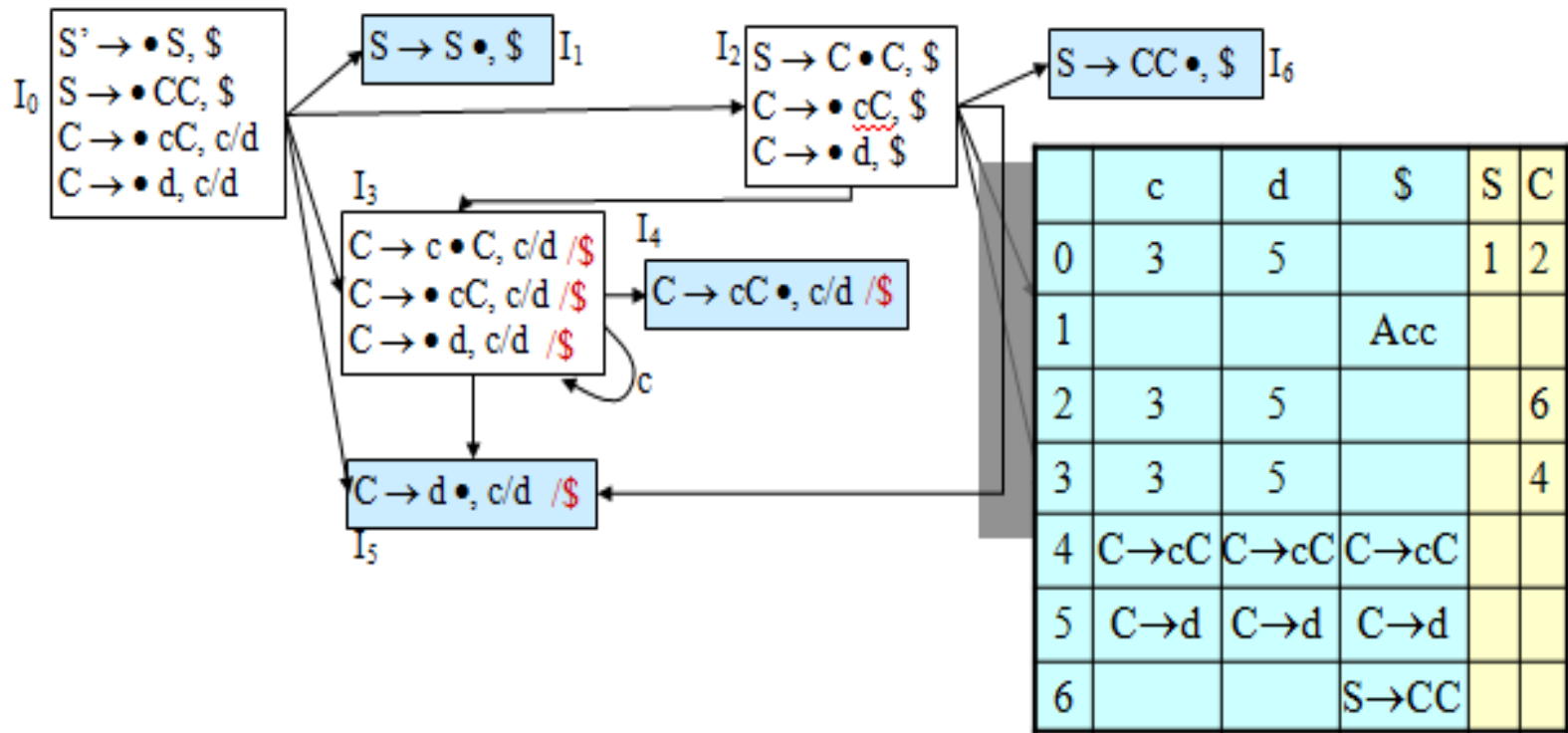
An LR parser can use any one of the following two techniques for error recovery:
- Panic mode
- Phrase level

Panic mode recovery:

It involves the following steps:

•Scan down the stack until a state 'a' with goto on a particular non terminal 'B' is found (by removing states from the stack)

•Zero or more input symbols are discarded until a symbol 'b' is found that can follow 'B'.

•Then the parser stacks the stage goto(s, B) and resumes parsing

Phrase Level Recovery:

It involves the following steps:

•Deciding on programmer errors, basing on the language, that call error routines in the parser table

•Designing appropriate error routines carefully that can modify the top of the stack and/or some symbols on input in a way suitable for error entries in the table.

Ambiguous Grammars

Ambiguous grammars provide a shorter and more natural specification for certain constructs when compared to the equivalent unambiguous grammars.

One can isolate a construct for optimization purposes using ambiguous grammars

One can incorporate a special construct by adding new productions to an existing ambiguous grammar

Ambiguous grammar can be handled by bottom up parser.

Creating LR Parsers for Ambiguous Grammars:

Panic mode recovery:

It involves the following steps:

•Scan down the stack until a state 'a' with goto on a particular non terminal 'B' is found (by removing states from the stack)

•Zero or more input symbols are discarded until a symbol 'b' is found that can follow 'B'.

•Then the parser stacks the stage goto(s, B) and resumes parsing

Phrase Level Recovery:

It involves the following steps:

•Deciding on programmer errors, basing on the language, that call error routines in the parser table

•Designing appropriate error routines carefully that can modify the top of the stack and/or some symbols on input in a way suitable for error entries in the table.

YACC is a automatic tool that generates the parser program.

YACC stands for Yet Another Compiler Compiler. This program is available in UNIX OS.

The construction of LR parser requires lot of work for parsing the input string. Hence, the process must involve automation to achieve efficiency in parsing an input.

Basically YACC is a LALR parser generator that reports conflicts or uncertainties (if at all present) in the form of error messages The typical YACC translator can be represented as shown in the image

YACC Specification

Parts of YACC Specification

The YACC specification file consists of three parts

Declaration section: In this section, ordinary C declarations are inserted and grammar tokens are declared. The tokens should be declared between %{ and %}

Translation rule section:It includes the production rules of context free grammar with corresponding actions

- The specification file comprising these sections can be written as:

%{

 / * declaration section */

%}

/* Translation rule section */

%%

/* Required C functions */

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, —Compilers–Principles, Techniques and Tools‖, Pearson Education, Low Price Edition, 2004

SYNTAX-DIRECTED TRANSLATION AND INTERMEDIATE CODE GENERATION

Running Outcomes

| The course will able the students to	
CLO7	Understand syntax directed translation schemes for a given context free grammar.
CLO8	Implement the static semantic checking and type checking using syntax directed definition (SDD) and syntax directed translation (SDT).
CLO9	Write intermediate code for statements like assignment, conditional, loops and functions in high level language.
CLO10	Explain the role of a semantic analyzer and type checking; create a syntax-directed definition and an annotated parse tree; describe the purpose of a syntax tree.

The course will able the students to

|	
CLO11	Design syntax directed translation schemes for a given context free grammar.

An intermediate code form of source program is an internal form of a program created by the compiler while translating the program created by the compiler while translating the program from a high – level language to assembly code(or)object code(machine code).

An intermediate source form represents a more attractive form of target code than does assembly.

An optimizing Compiler performs optimizations on the intermediate source form and produces an object module.

$$Analysis \quad + \quad syntheses \quad = \quad translation$$

Creates an   generate target code
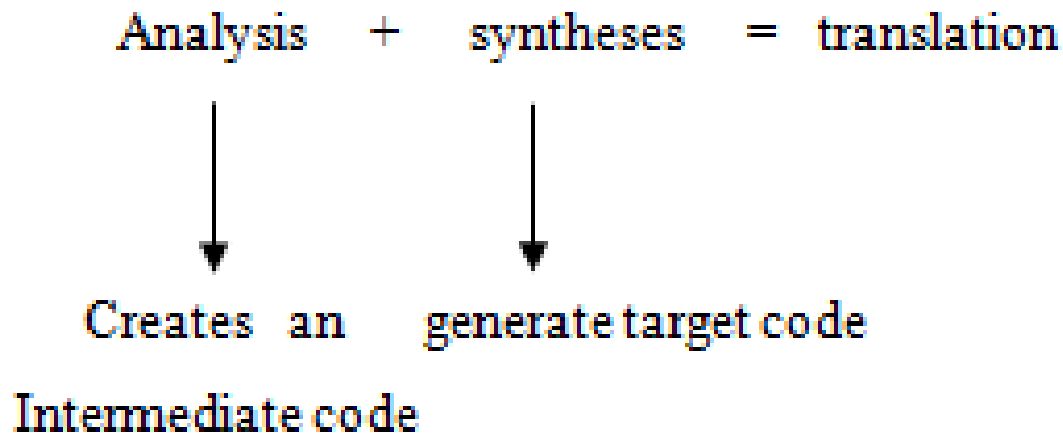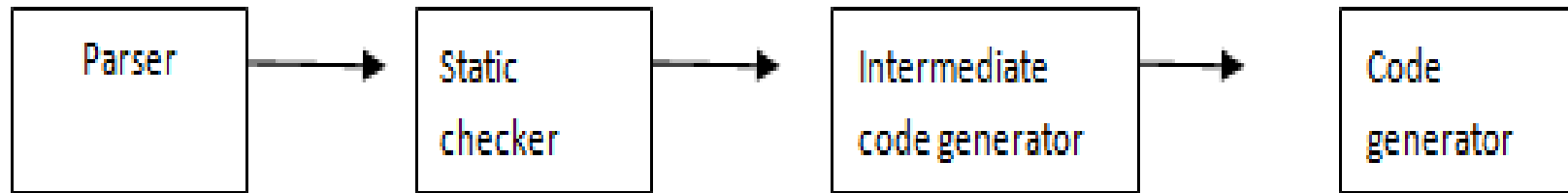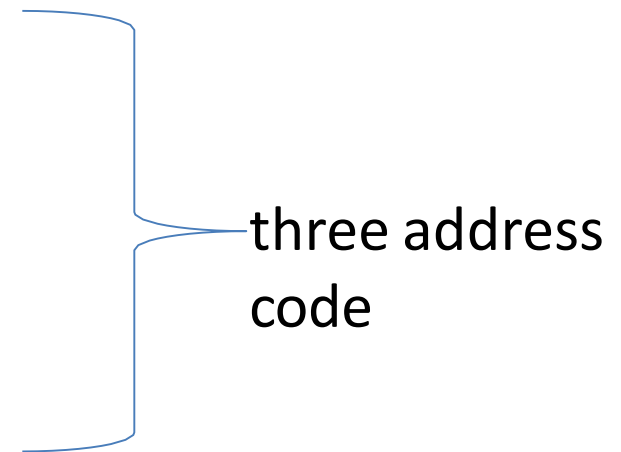
Intermediate code

Position of intermediate code generator

various intermediate code forms
are:

- Polish notation

- Abstract syntax trees(or)syntax trees

- Quadruples

- Triples

- Indirect triples

- Abstract machine code(or)pseudocopde

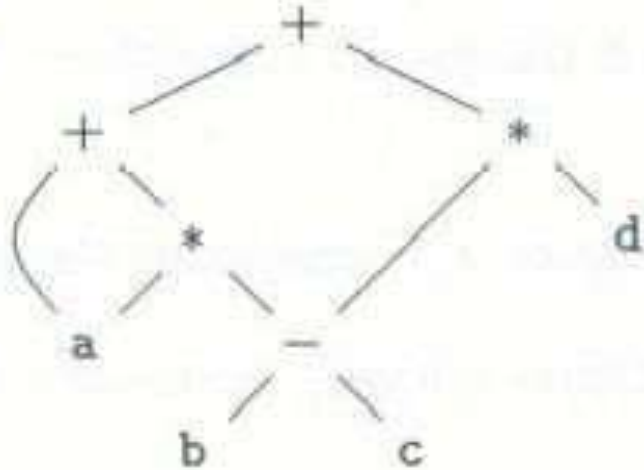three address code

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.

$$x + y*z \rightarrow t1 = y * z$$
$$t2 = x + t1$$

(a) DAG

$$t_1 = b - c$$
$$t_2 = a * t_1$$
$$t_3 = a + t_2$$
$$t_4 = t_1 * d$$
$$t_5 = t_3 + t_4$$

(b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

- Three-address instructions can be implemented as objects or as record with fields for the operator and operands.

- Three such representations
– Quadruple, triples, and indirect triples

A q**uadruple** (or quad) has four fields: op, arg1, arg2, and result.

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

$$t_1 = minus\ c$$
$$t_2 = b * t_1$$
$$t_3 = minus\ c$$
$$t_4 = b * t_3$$
$$t_5 = t_2 + t_4$$
$$a = t_5$$

(a) Three-address code

(b) Quadruples

- A triple has only three fields: op, arg1, and arg2

- Using triples, we refer to the result of an operation x op y by its position, rather by an explicit temporary name
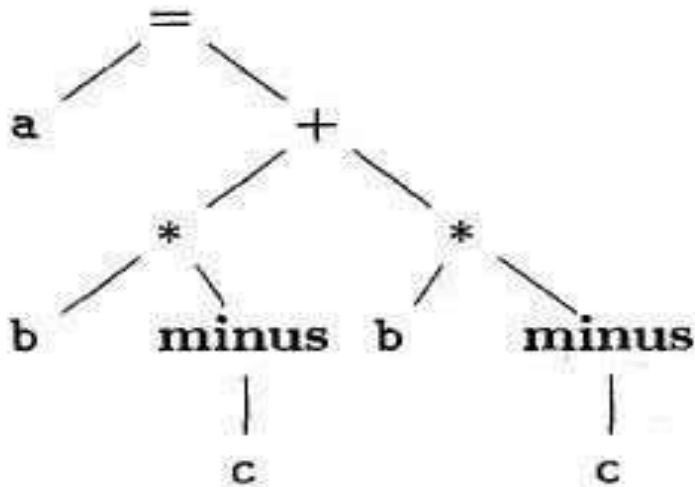
(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	. . .		

(b) Triples

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

- ➢ Quadruples
- ➢ Triples
- ➢ Indirect triples

Quadruples

- A quadruple is a record structure with four fields, which are, op, arg1, arg2 and result.

- The op field contains an internal code for the operator. The three-address statement x : =y op z is represented by placing y in arg1, z in arg2 and x in result.

Triples

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.

- The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table¬ or pointers into the triple structure (for temporary values).

Indirect Triples

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples

- Given input a : = b * - c + b * - c, the three-address code is as shown above.

- The synthesized attribute *S.code* represents the three-address code for the assignment *S.*

The non terminal *E* has two attributes :

1. *E.place*, the name that will hold the value of *E*, and

2. *E.code*, the sequence of three-address statements evaluating *E*.

PRODUCTION

SEMANTIC RULES

S

       while*E*d

o*S*$_1$

   S.begin :=   newlabel;

  S.after := newlabel;

code := gen(S.begin ':') ||

E.code ||

gen ('if' E.place '=' '0' 'goto' S.after)||

S$_1$.code ||

gen ('goto' S.begin) || gen

(S.after ':')

- 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
- Inherited attributes
    - An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings

Production

1)   L -> E n
2)   E -> E1
+ T
3)   E -> T
4)   T -> T1 *
F
5)   T -> F
6)   F -> (E)
7)   F -> digit

Semantic Rules

L.val = E.val
E.val = E1.val +
T.val
E.val = T.val
T.val = T1.val *
F.val T.val =
F.val
F.val = E.val
F.val =
digit.lexval

Production

1)   T -> FT'

2)   T' -> *FT'$_1$

3)   T' -> ε
1)   F -> digit

Semantic Rules

T'.inh = F.val
T.val = T'.syn T'$_1$.inh =
T'.inh*F.val T'.syn =
T'$_1$.syn T'.syn = T'.inh
F.val = F.val = digit.lexval

Syntax directed translation schemes:

- An SDT is a Context Free grammar with program fragments embedded within production bodies
- Those program fragments are called semantic actions
- They can appear at any position within production body
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order
- Typically SDT's are implemented during parsing without building a parse tree

Postfix translation schemes:

- Simplest SDDs are those that we can parse the grammar bottom-up and the SDD is s-attributed

- For such cases we can construct SDT where each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production

- SDT's with all actions at the right ends of the production bodies are called postfix SDT's

Example of postfix SDT

1) L -> E n   {print(E.val);}
2) E -> E1 + T   {E.val=E1.val+T.val;}
3) E -> T   {E.val = T.val;}
4) T -> T1 * F   {T.val=T1.val*F.val;}
5) T -> F   {T.val=F.val;}
6) F -> (E)   {F.val=E.val;}
7) F -> digit   {F.val=digit.lexval;}

- **Construction of Syntax Trees**

  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.

- S-Attributed Grammars are a class of attribute grammars characterized by having no inherited attributes. Inherited attributes, which must be passed down from parent nodes to children nodes of the abstract syntax tree during the semantic analysis of the parsing process, are a problem for bottom-up parsing because in bottom-up parsing, the parent nodes of the abstract syntax tree are created after creation of all of their children. YACC is based on the S-attributed approach.

L-Attributed definitions:

- A SDD is L-Attributed if the edges in dependency graph goes from Left to Right but not from Right to Left.

- More precisely, each attribute must be either

  - Synthesized

  - Inherited, but if there us a production A->X1X2...Xn and there is an inherited attribute Xi.a computed by a rule associated with this production, then the rule may only use:

- Inherited attributes associated with the head A Either inherited or synthesized attributes associated with the occurrences of symbols X1,Xz,...,Xi-1 located to the left of Xi

- Inherited or synthesized attributes associated with this occurrence of Xi itself, but in such a way that there is no cycle in the graph

- L-attributed grammars are a special type of attribute grammars. They allow the attributes to be evaluated in one left-to-right traversal of the abstract syntax tree. As a result, attribute evaluation in L-attributed grammars can be incorporated conveniently in top-down parsing. Many programming languages are L-attributed. Special types of compilers, the narrow compilers, are based on some form of L-attributed grammar.
- Any S-attributed grammar is also an L-attributed grammar.

- Attribute evaluation in S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing.

A SDT scheme is a context-free grammar with program fragments embedded within production bodies .The program fragments are called semantic actions and can appear at any position within the production body.

Any SDT can be implemented by first building a parse tree and then pre-forming the actions in a left-to-right depth first order. i.e during preorder traversal.

The use of SDT's to implement two important classes of SDD's

- 1. If the grammar is LR parsable, then SDD is S-attributed.

- 2. If the grammar is LL parsable, then SDD is L-attributed.

Postfix Translation Schemes
- The postfix SDT implements the desk calculator SDD with one change: the action for the first production prints the value. As the grammar is LR, and the SDD is S-attributed.

L →E n {print(E.val);}

E → E1 + T { E.val = E1.val + T.val }

E → E1 - T { E.val = E1.val - T.val }

E → T { E.val = T.val }

T → T1 * F { T.val = T1.val * F.val }

T → F { T.val = F.val }

F → (E) { F.val = E.val }

F → digit { F.val = digit.lexval }

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) }

Translation scheme for checking the type of statements:

1. Assignment statement: S→id: = E

2. Conditional statement: S→if E then S1

SDD for while-statements

$$S \rightarrow \textbf{while} \ (\ C \ ) \ S_1 \quad L_1 = new \ ();$$
$$L_2 = new \ ();$$
$$S_1.next = L1;$$
$$C.false = S.next;$$
$$C.true = L2;$$
$$S.code = \textbf{label} \parallel L1 \parallel C.code$$
$$\parallel \textbf{label} \parallel L2 \parallel S_1.code$$

SDT for while-statements

$$S \rightarrow \textbf{while} \ (\quad \{ \ L_1 = new \ (); \ L_2 = new \ ();$$
$$C.false = S.next; \ C.true = L2; \ \}$$
$$C \ ) \quad \{ \ S_1.next = L1; \ \}$$
$$S_1 \quad \{ \ S.code = \textbf{label} \parallel L1 \parallel C.code$$
$$\parallel \textbf{label} \parallel L2 \parallel S_1.code; \ \}$$

- Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.

- Boolean expressions are composed of the boolean operators (and, or, and not) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E1 relop E2, where E1 and E2 are arithmetic expressions.

Boolean Expressions

Example:
E->EorE | EandE |notE | (E) |id relop id | true | false

Methods of Translating Boolean Expressions:

- There are two principal methods of representing the value of a boolean expression. They are :

S->if E then S

S->if E then S else S S->while E

do S

Attributes:

  E.true: the label to which control flows if E is true. E.false: the

  label to while control flows if E is false E/S.code: three-address

  code for E/S

  S.next: the next three-address code following the three
   address code of S.

Functions to be used: || concatenate three address code

Flow-of-Control Statements

S->if E then S1 { E.true = newlabel; E.false = S.next;

S1.next = S.next; S.code = E.code || gen(E.true, ":") || S1.code; }

S->if E then S1 else S2 {

 E.true = newlabel; E.false =newlabel;S1.next = S.next;

 S2.next = S.next;
 S.code = E.code || gen(E.true, ":") ||S1.code|| gen("goto" S.next)
 ||
  gen(E.false ':') || S2.code; }

S->while E do S1 {

 S.begin = newlabel; E.true := newlabel;

 E.false = S.next; S1.next = S.begin;

 S.code = gen(S.begin ':') || E.code || gen(E.true, ':')

        || S1.code || gen('goto' S.begin); }
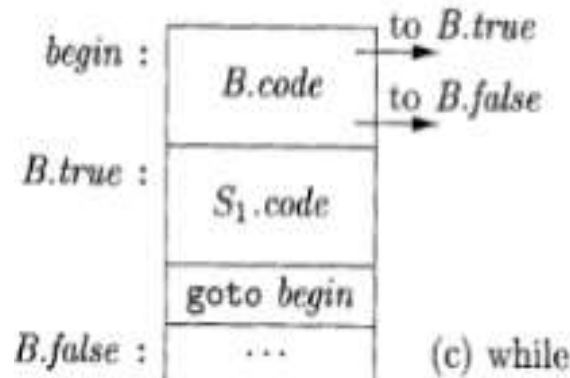
$$S \rightarrow \textbf{if } (B) S_1$$
$$S \rightarrow \textbf{if } (B) S_1 \textbf{ else } S_2$$
$$S \rightarrow \textbf{while } (B) S_1$$
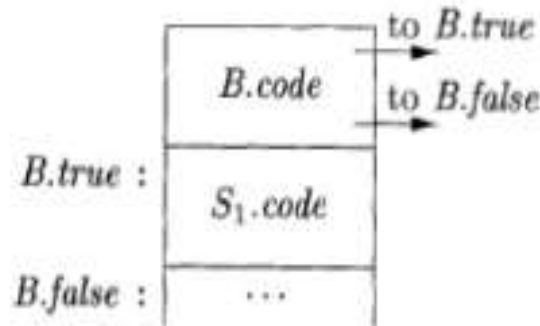
(a) if

(b) if-else

(c) while

$S \rightarrow \textbf{while } M_1 \ (\ B \ ) \ M_2 \ S_1$
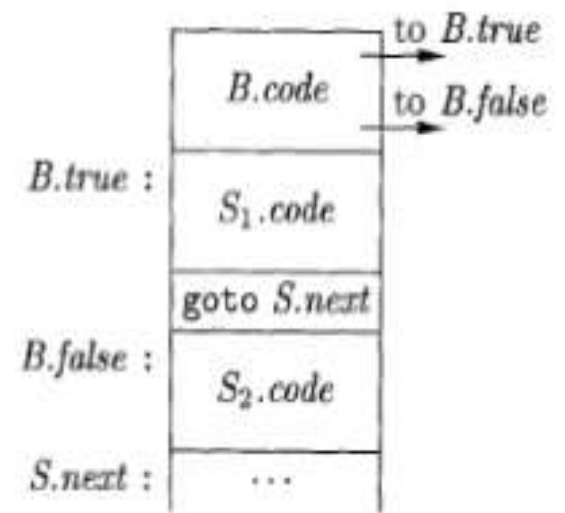
1) $S \rightarrow \textbf{if} \ (B) \ M \ S_1$ { $backpatch(B.truelist, \ M.instr);$
$S.nextlist \ = \ merge(B.falselist, \ S_1.nextlist);$ }

2) $S \rightarrow \textbf{if} \ (B) \ M_1 \ S_1 \ N \ \textbf{else} \ M_2 \ S_2$
{ $backpatch(B.truelist, \ M_1.instr);$
$backpatch(B.falselist, \ M_2.instr);$
$temp \ = \ merge(S_1.nextlist, \ N.nextlist);$
$S.nextlist \ = \ merge(temp, \ S_2.nextlist);$ }

3) $S \rightarrow \textbf{while} \ M_1 \ (B) \ M_2 \ S_1$
{ $backpatch(S_1.nextlist, \ M_1.instr);$
$backpatch(B.truelist, \ M_2.instr);$
$S.nextlist \ = \ B.falselist;$
$emit(\textnormal{'goto'} \ M_1.instr);$ }

4) $S \rightarrow \{ L \}$ { $S.nextlist \ = \ L.nextlist;$ }

5) $S \rightarrow A ;$ { $S.nextlist \ = \ \textbf{null};$ }

6) $M \rightarrow \epsilon$ { $M.instr \ = \ nextinstr;$ }

7) $N \rightarrow \epsilon$ { $N.nextlist \ = \ makelist(nextinstr);$
$emit(\textnormal{'goto} \ \_ \textnormal{'});$ }

8) $L \rightarrow L_1 \ M \ S$ { $backpatch(L_1.nextlist, \ M.instr);$
$L.nextlist \ = \ S.nextlist;$ }

9) $L \rightarrow S$ { $L.nextlist \ = \ S.nextlist;$ }

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, —Compilers–Principles, Techniques and Tools‖, Pearson Education, Low Price Edition, 2004

TYPE CHECKING AND RUN TIME ENVIRONMENT

The course will able the students to	
CLO13	Explain the role of different types of runtime environments and memory organization for implementation of programming languages.
CLO14	Differentiate static vs. dynamic storage allocation and the usage of activation records to manage program modules and their data.
CLO15	Understand the role of symbol table data structure in the construction of compiler.

- Type checking has the potential for catching errors in programs. In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element.

- 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. An implementation of a language is *strongly typed* if a compiler guarantees that the programs it accepts will run without type errors.

Rules for Type Checking:

- Type checking can take on two forms: synthesis and inference. *Type synthesis* builds up the type of an expression from the types of its sub expressions. It requires names to be declared before they are used. The type of $E1 + E_2$ is defined in terms of the types of $E1$ and E_2 • A typical rule for type synthesis has the form

- A compiler has to do semantic checks in addition to syntactic checks.
- Semantic Checks
  - Static – done during compilation
  - Dynamic – done during run-time
- Type checking is one of these static checking operations.
  - we may not do all type checking at compile-time.
  - Some systems also use dynamic type checking too.

- A type system is a collection of rules for assigning type expressions to the parts of a program.

- A type checker implements a type system.

- A sound type system eliminates run-time type checking for type errors.

- A programming language is strongly-typed, if every program its compiler accepts will execute without type errors.

  – In practice, some of type checking operations are done at run-time (so, most of the programming languages are not strongly-typed).

  – Ex: int x[100];
    … i will be between 0 and 99

    x[i] ➜ most of the compilers cannot guarantee that

- **A Simple Type Checking:**

  P → D;E

  D → D;D

  D → **id**:T   { addtype(id.entry,T.type) }

  T → char   { T.type=char }

  T → int T   { T.type=int }

  → real T   { T.type=real }

  → ↑T_1 { T.type=pointer(T_1.type) }

T → array[**intnum**] of T_1 {
T.type=array(1..intnum.val,T_1.type) }

Type Expressions:

- Types have structure, which we shall represent using *type expressions:* a type expression is either a basic type or is formed by applying an operator called a *type constructor* to a type expression. The sets of basic types and constructors depend on the language to be checked.

- A basic type is a type expression. Typical basic types for a language include boolean, char, integer, float, and void; the latter denotes "the absence of a value."

- A type name is a type expression.

- A type expression can be formed by applying the array type constructor to a number and a type expression.

- A record is a data structure with named fields. A type expression can be formed by applying the record type constructor to the field names and their types.

- A type expression can be formed by using the type constructor —>• for function types. We write s —»• t for "function from type s to type r."

- **Type Names and Recursive Types:**

- Once a class is defined, its name can be used as a type name in *C++* or Java; for example, consider Node in the program fragment

- p u b l i c class Node { • • • }

- p u b l i c   Node   n;

- Names can be used to define recursive types, which are needed for
data structures such as linked lists. The pseudo code for a list element

- class Cell { int info; Cell next; •••}

- A type system is a collection of rules that assign types to program constructs (more constraints added to checking the validity of the programs, violation of such constraints indicate errors)

- A languages type system species which operations are valid for which types

- Type systems provide a concise formalization of the semantic checking rules

- Statically typed languages: all or almost all type checking occurs at
  compilation time. (C, Java)

- Dynamically typed languages: almost all checking of types is done as
  part of program execution (Scheme)

- Un typed languages: no type checking (assembly, machine code)

Static Type Checking:

Type checking done at compile-time. When using these languages you are enforced to declare the type of variables before using them(compiler needs to know of which data type do the variable belongs to).

For example consider a statement in c++ int

a=10;

here compiler needs to know the data type of variable "a" before using it.

E.g., C,C++,JAVA,C# are some **Statically Typed Languages**

Type checking done at run-time. When using these languages you need not specify or declare the type of variable instead compiler itself figures out what type a variable is when you first assign it a value.

Now consider some statements in python:
str="Python"
str2=10

- A type checker for a simple language checks the type of each identifier. The type checker is a translation scheme that synthesizes the type of each expression from the types of its sub expressions. 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_erroris 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) }

Type equivalence

Name equivalence

Treat named types as basic types. Therefore two type expressions are *name equivalent* if and only if they are identical, that is if they can be represented by the same syntax tree, with the same labels.

Structural equivalence

Replace the named types by their definitions and recursively check the substituted trees.

Recursive Types:

In PASCAL a *linked list* is usually defined as
follows. type link

= ^ cell;

   cell = record
         info:
         type;
         next:
         link;

      end;

- The corresponding type graph has a cycle. So to decide structural equivalence of two types represented by graphs PASCAL compilers put a mark on each visited node (in order not to visit a node twice). In C, a linked listis usually defined as follows.

```
struct cell
 {
   int info;
   struct cell *next;
 };
```

To avoid cyclic graphs, C compilers

- Require type names to be declared before they are used, except for pointers to records.

- Use structural equivalence except for records for which they use name
  equivalence.

Type Conversions:

- Consider expressions like *x + i,* where *x* is of type float and *i* is of type integer.

- Since the representation of integers and floating-point numbers is different within a computer and different machine instructions are used for operations on integers and floats, the compiler may need to convert one of the operands of + to ensure that both operands are of the same type when the addition occurs.

- Suppose that integers are converted to floats when necessary, using a unary operator (f l o a t) .

For example:

- The integer 2 is converted to a float in the code for the expression 2*3.14:

- Type conversion rules vary from language to language. The rules for Java distinguish between widening conversions, which are intended to preserve information, and narrowing conversions, which can lose information.

- Conversion from one type to another is said to be implicit if it is done automatically by the compiler.

- Implicit type conversions, also called coercions,

Conversion between data types can be done in two ways by casting:
- Implicit casting

- Explicit casting

Implicit casting
- Implicit casting doesn't require a casting operator. This casting is normally used when converting data from smaller integral types to larger or derived types to the base type.
  int x = 123;

  double y = x;

- In the above statement, the conversion of data from int to double is done implicitly, in other words programmer don't need to specify any type operators.

Explicit casting

- Explicit casting requires a casting operator. This casting is normally used when converting a double to int or a base type to a derived type.

  double y = 123;

  int x = (int)y;

  In the above statement, we have to specify the type operator (int) when converting from double to int else the compiler will throw an error. You can learn more about casting here.

Conversion operators:

- Conversion operators help to cast user-defined types from one to the other much like the basic types. For implicit or explicit conversion, we have to create a static method in the corresponding class with method name as the type it returns including the keyword that says implicit or explicit.

Overloading of Functions and Operators

- – Enabling C++'s operators to work with class objects
- – Using traditional operators with user-defined objects
- – Requires great care; when overloading is misused, program difficult to understand

- – Examples of already overloaded operators

  - • Operator **<<** is both the stream-insertion operator and the bitwise left-shift operator
  - • **+** and **–**, perform arithmetic on multiple types

- – Compiler generates the appropriate code based on the manner in which the operator is used

Syntax:

- return-data-type operator symbol-of-operator
  (parameters)
  {
  //body of the function
  }

Example:

- void operator ++ ( )
  {
  body of function;
  }

- The value-number method can be applied to type expressions to resolve overloading based on argument types, efficiently. Since the signature for a function consists of the function name and the types of its arguments, overloading can be resolved based on signatures. However, it is not always possible to resolve overloading by looking only at the arguments of a function.

- An overloaded symbol has different meanings depending on its context. Overloading is resolved when a unique meaning is determined for each occurrence of a name. Overloading in Java can be resolved by looking only at the arguments of a function. The + operator in Java denotes either string concatenation or addition, depending on the types of its operands.

1. Procedure call: Source language issue

- 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 a procedure named
read array: Procedure
read array Var i: integer;
Begin
For i=1 to 9 do real(a[i])
End;

2. Activation tree: Source language issue

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

- a) Each node represents an activation of a procedure.
  b) The root represents the activation of the main program.
  c)The node for a is the parent of the node b if and only if control flows from activation a to b.
  d)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.

3. Control Stack: Source language issue

- A control stack used to keep track of live procedure activations.

- The idea is to push the node for activation onto the control stack as the activation begins and to pop the node when the activation ends.

4. The scope of declaration: Source language issue

A declaration is a syntactic construct that associates information with a name.

- A declaration 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.

5. Bindings of names: Source language issue

- Even if each time name declared once in a program, the same name may denote different data objects at runtime.

- "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 bound to s.

- This association referred as a binding of x.

- The compiler demands for a block of memory to an operating system. The compiler utilizes this block of memory executing the compiled program. This block of memory called runtime **storage organization.**
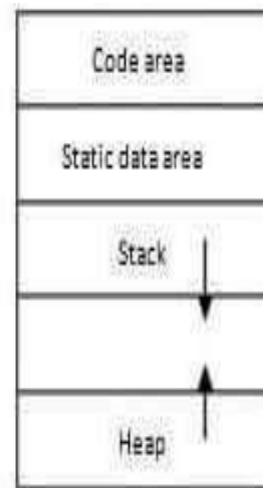- The runtime storage is subdivided to hold code and data such as the generated target code and Data objects.

| Code area |
| Static data area |
| Stack |
| |
| Heap |

Fig : Typical subdivision of run time memory into code and data areas

- Heap area is the area of runtime storage in which the other information stored. For time example memory for some data items allocated under the program control. Memory required for these data items obtained from this heap area.

- Moreover, A stack is used to manage the active procedure. Managing of active procedures means when a call occurs then execution of activation interrupted and information about the status of the stack is saved on the stack. When the control returns from the call this suspended activation resumed after storing the values of relevant registers.

- **Activation Records: Storage organization**
- Various field of activation record is as follows:

1.Temporary values: The temporary variables needed during the evaluation of expressions. Such variables stored in the temporary field of activation record stored.

2.Local variables: The local data is a data that is local to the execution
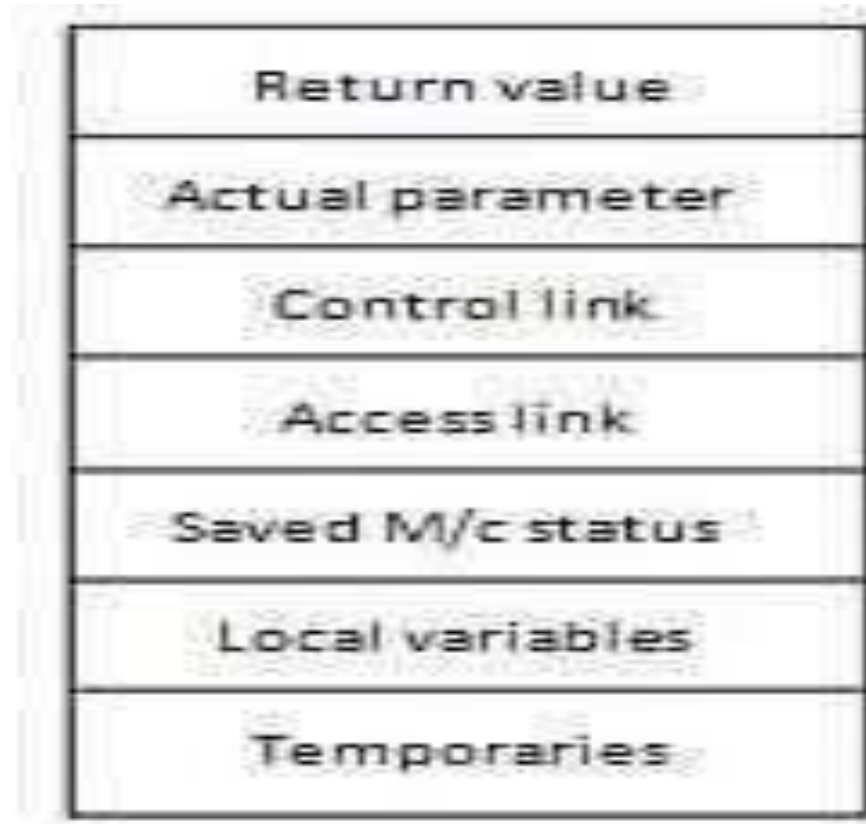procedure stored in this field of activation record.

Fig   Activation Record

- Saved machine registers: This field holds the information regarding the status of a machine just before the procedure called. This field contains the registers and program counter.

- Control link: This field is optional. So it points to the activation record of the calling procedure. So This link also called dynamic link.

Similarly, Access link:

- This field is also optional. It refers to the nonlocal data in another activation record. This field also called static link field.

Actual parameters: This field holds the information about the actual parameters. Also, these actual parameters passed to the called procedure.

Return values: This field used to store the result of a function call.

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 de-allocates 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, every time 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.

- **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, which consists 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).

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 procedure called and have a size that depends on the parameters of the call.

- **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 activation ends.

2. A called activation outlives the caller.

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

- Scope rules determine the treatment of non-local names

- A common rule is lexical scoping or static scoping (most languages use lexical scoping)

- In some cases, when a procedure refer to variables that are not local to it, then such variables are called nonlocal variables

- There are two types of scope rules, for the non-local names. They are

1. Static scope
2. Dynamic scope

- Suppose procedure p at depth np refers to a non-local *a* at depth na, then storage for *a* can be found as

    – follow (np-na) access links from the record at the top of the stack
    – after following (np-na) links we reach procedure for which *a* is local

- Therefore, address of a non local *a* in procedure p can be stored in symbol table as

    (np-na, offset of *a* in record of activation having *a*)

- **Static Scope or Lexical Scope**

- Lexical scope is also called static scope. In this type of scope, the scope is verified by examining the text of the program.

- Examples: PASCAL, C and ADA are the languages that use the static scope rule.

- These languages are also called block structured languages

Example:

{

Declaration statements

....

}

- A block statement contains its own data declarations

- Blocks can be nested

- The property is referred to as block structured

- Scope of the declaration is given by most closely nested rule

  - The scope of a declaration in block B includes B

  - If a name X is not declared in B

    then an occurrence of X is in the scope of declarator X in B' such that

    - B' has a declaration of X

    - B' is most closely nested around B

- Blocks are simpler to handle than procedures

- Use stack for memory allocation

- Allocate space for complete procedure body at one time

Lexical scope without nested procedures:

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated
  at compile time
- Any name non-local to one procedure is non-local to all procedures
- In absence of nested procedures use stack allocation
- Storage for non locals is allocated statically
- A non local name must be local to the top of the stack
- Stack allocation of non local has advantage:
  - Non locals have static allocations
  - Procedures can be passed/returned as parameters

How to setup access links?

- suppose procedure p at depth np calls procedure x at depth nx.

- The code for setting up access links depends upon whether the called
procedure is nested within the caller.

  - np < nx

    Called procedure is nested more deeply than p. Therefore, x must be declared in p. The access link in the called procedure must point to the access link of the activation just below it

  - np ≥ nx

    From scoping rules enclosing procedure at the depth 1,2,... ,nx-1 must be same. Follow np-(nx-1) links from the caller, we reach the most recent activation of the procedure that encloses both called and calling procedure

Displays:

- Faster access to non locals

- Uses an array of pointers to activation records

- Non locals at depth i is in the activation record pointed to by d[i]

Justification for Displays:

- Suppose procedure at depth j calls procedure at depth i

- Case j < i then i = j + 1
  - called procedure is nested within the caller
  - first j elements of display need not be changed
  - set d[i] to the new activation record

- Case j ≥ I
  - enclosing procedure at depthes 1…i-1 are same and are left un- disturbed
  - old value of d[i] is saved and d[i] points to the new record
  - display is correct as first i-1 records are not disturbed

Parameter Passing:

- The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

r-value

- The value of an expression is called its r-value.

- r-values can always be assigned to some other variable.

l-value

- The location of memory (address) where an expression is stored is known as the l-value of that expression.

Formal Parameters:

- Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

Actual Parameters:

- Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Call by value:

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

- used in Pascal and C

- formal is treated just like a local name

- caller evaluates the actual parameters and places r-value in the storage for formals

- call has no effect on the activation record of caller

Call by reference (call by address):

- the caller passes a pointer to each location of actual parameters

- if actual parameter is a name then l-value is passed

- if actual parameter is an expression then it is evaluated in a new location and the address of that location is passed

Copy restore (copy-in copy-out, call by value result):

- actual parameters are evaluated, r-values are passed by call by value, l-values are determined before the call

- when control returns, the current r-values of the formals are copied into l-values of the locals

Call by name (used in Algol):

- names are copied

- local names are different from names of calling procedure

  swap(i,a[i])

  temp = i
  i = a[i]
  a[i] = temp

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

<symbol name, type, attribute>

For example, if a symbol table has to store information about the following variable declaration:

static int interest;

Implementation:

A symbol table can be implemented in one of the following ways:

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

Store the following information about identifiers.

- The name (as a string).

- The data type.

- The block level.

- Its scope (global, local, or parameter).

- Its offset from the base pointer (for local variables and parameters only).

Operations:

1. insert()

2. lookup()

The install() function will insert a new symbol into the symbol table.

Each symbol has a block level.

– Block level 1 = Keywords.

– Block level 2 = Global variables.

– Block level 3 = Parameters and local variables.

install() will create an IdEntry object and store it in the table.

- Whenever a symbol is encountered, we must look it up in the symbol table.
- If it is the first encounter, then idLookup() will return null.
- If it is not the first encounter, then idLookup() will return a reference to the IdEntry for that identifier found in the table.
- Once we have the IdEntry object, we may add information to it.
- Since a variable should be declared when it first appears,
  - If the parser is parsing a declaration, then it expects idLookup() to return null.
  - If the parser is not parsing a declaration, then it expects idLookup() to return non-null.
  - In each case, anything else is an error.

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.

  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 global symbol table has been searched for the name.

- Storage is usually taken from heap
- Allocated data is retained until de-allocated
- Allocation can be either explicit or implicit

  - Pascal : explicit allocation and de-allocation by new() and dispose()

  - Lisp : implicit allocation when cons is used, and de-allocation through garbage collection

Dynamic Storage Allocation:

new(p);   p^.key:=k;   p^.info:=i;

- **Garbage :** unreachable cells
  - Lisp does garbage collection
  - Pascal and C do not

- head^.next := nil;

- **Dangling reference**
  - dispose(head^.next)

Explicit Allocation of Fixed Sized Blocks

- Link the blocks in a list

- Allocation and de-allocation can be done with very little overhead

- blocks are drawn from contiguous area of storage

- An area of each block is used as pointer to the next block

- A pointer available points to the first block

- Allocation means removing a block from the available list

- De-allocation means putting the block in the available list

- Compiler routines need not know the type of objects to be held in the blocks

- Each block is treated as a variant record

Explicit Allocation of Variable Size Blocks:

- Storage can become fragmented

- Situation may arise

    - If program allocates five blocks

    - then de-allocates second and fourth block

- Fragmentation is of no consequence if blocks are of fixed size

- Blocks can not be allocated even if space is available

Implicit De-allocation:

- Requires co-operation between user program and run time system

- Run time system needs to know when a block is no longer in use

- Implemented by fixing the format of storage blocks

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, —Compilers–Principles, Techniques and Tools‖, Pearson Education, Low Price Edition, 2004

CODE OPTIMIZATION AND CODE GENERATOR

The course will able the students to

CLO16	Learn the code optimization techniques to improve the performance of a program in terms of speed & space.
CLO17	Implement the global optimization using data flow analysis such as basic blocks and DAG.
CLO18	Understand the code generation techniques to generate target code.
CLO19	Design and implement a small compiler using a software engineering approach.
CLO20	Apply the optimization techniques to intermediate code and generate machine code

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.

Function preserving transformations examples:

- Common sub expression elimination

- Copy propagation,

- Dead-code elimination

- Constant folding

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

t1: = 4*i t2:

= a [t1] t3: =

4*j

t4: = 4*i

t5: = n

t6: = b [t4] +t5

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

t1: = 4*i t2: =

a [t1] t3: = 4*j

t5: = n

t6: = b [t1] +t5

- The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the 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 Elimination

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

Example:

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

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

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.

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- Induction-variable elimination: which we apply to replace variables from inner loop.
- Reduction in strength: which replaces and expensive operation by a cheaper one, such as a multiplication by an addition

There are two types of basic block optimizations. They are :

1. Structure-Preserving Transformations
2. Algebraic Transformations

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

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

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 specification 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.

Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- Code motion, which moves code outside a loop.

- Induction-variable elimination, which we apply to replace variables from inner loop.

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

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 */

Loops are usually processed inside out. For example consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. 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 we cannot get rid of either j or t4 completely; t4 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.

•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^2 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.

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.

- 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.
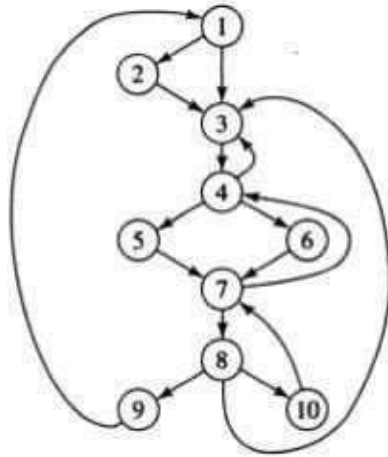
(a) Flow graph
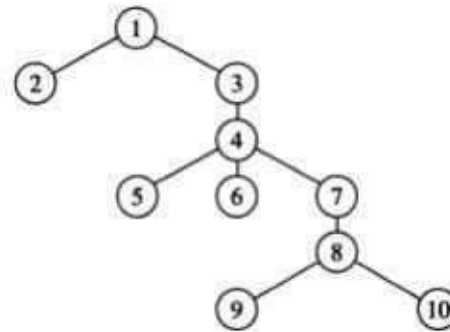
(b) Dominator tree

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.

Natural Loops:

One application of dominator information is in determining the loops of a flow graph suitable for improvement. There are two essential properties of loops:

A loop must have a single entrypoint, 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 headerOne 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 ofedges 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.

- 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.

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 theother. 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 single loop.
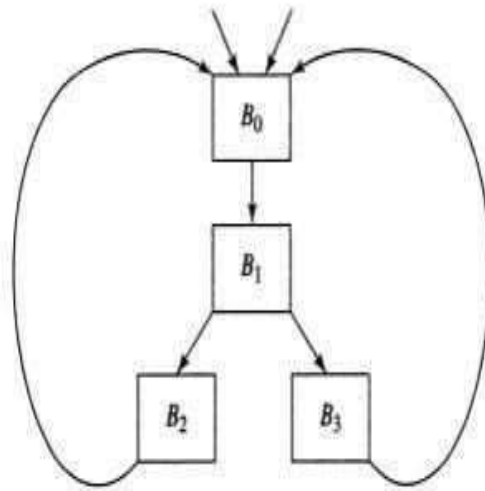
Two loops with the same header

Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheader. 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
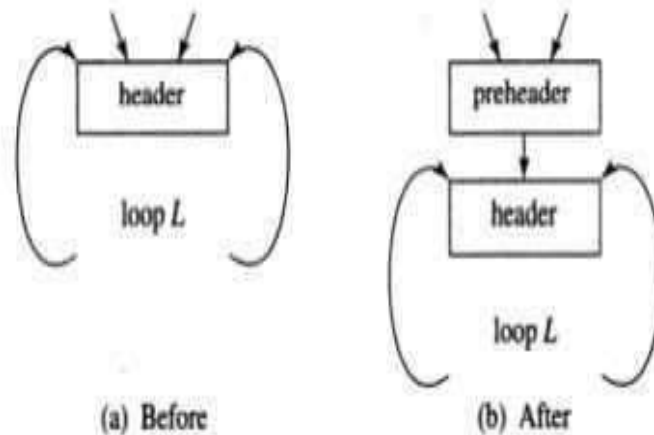
Reducible flow graphs:

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

1. There are no jumps into the middle of loops from outside

2. 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.

Peephole Optimization:

A simple but effective technique for locally improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining as short sequence of target instructions(called the peephole) and replacing these instructions by a shorter or faster sequence. Whenever possible.It is a characteristics of peephole optimization that each improvement may spawn opportunities for additional improvements.

Characteristics of peephole optimization:

- Redundant-instruction elimination
- Flow-of-controloptimization
- Algebraic simplifications
- Use of machine idioms

At source code level, the following can be done by the user:

int add_ten(int x)	int add_ten(int x)	int add_ten(int x)	int add_ten(int x)

int y, z;
y = 10;
z = x + y;
return z;
}
	{
	  int y;
	  y = 10;
	  y = x + y;
	  return y;
	}
		{
		  int y = 10;
		  return x + y;
		}
			{
			  return x + 10;
			}

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

MOV x, R0

MOV R0, R1

We can delete the first instruction and re-write the sentence as:

MOV x,R1

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
Return x+10;
Printf("value of x is %d",x);
}
```

In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...
MOV    R1,
R2    GOTO
L1

...
L1 : GOTO L2
L2 : INC R1
```

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.
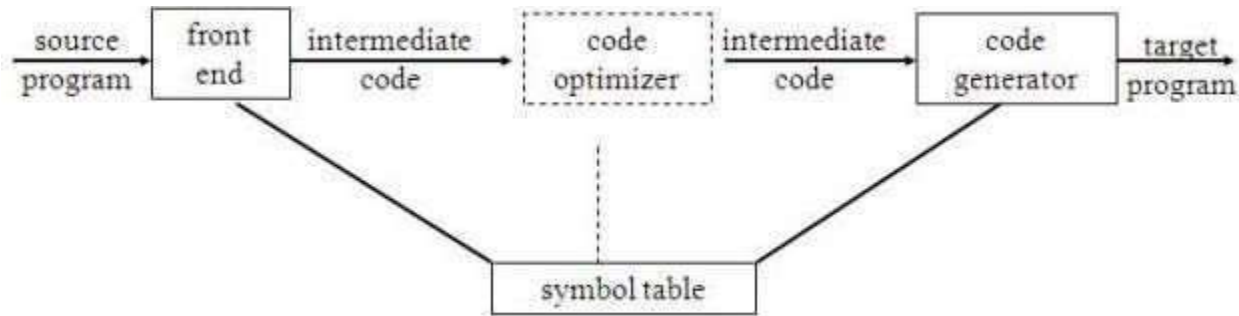
Fig: Position of code generator

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

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 representation can be :

  a. Linear representation such as postfix notation
  b. Three address representation such as quadruples
  c. Virtual machine representation such as stack machine code
  d. Graphical representations such 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 be error-free.

The output of the code generator is the target program. Like the intermediate code, this output may take on a variety of forms:

- Absolute machine language

- Relocatable machine language

- Assembly language

Absolute machine language:

It can be placed in a fixed location in memory and immediately executed. Eg:

"student-job " compilers such as WATIV and PL/C.

Relocatable machine language:

A set of relocatable object modules can be linked together and loaded for execution by a linking loader.we gain a great deal of flexibility in being able to compile subroutines separately and to call other previousily compiled programs from an object module.

Memory management:

Mapping names in the source program to addresses of data objects in runtime memory is done cooperatively by the front end and the code generator.

If machine code is being generated,labels in three-address statements have to be converted to addresses of instructions.this process is analogous to the "back patching" technique.

Labels in three-address statements have to be converted to addresses of
Instructions.

Instructions involving register operands are shorter and faster than those involving operands in memory. The use of registers is subdivided into two subproblems :

1. Register allocation
2. Register assignment

Certain machine requires register pairs(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

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.

Familiarity with the   target machine and its instruction set is a word. prerequisite for designing a good code generator.

The target computer is a byte-addressable machine with 4 bytes to a

It has n general-purpose registers, R0, R1, . . . , Rn-1. It

 has two-address instructions of the form:

op source, destination

where, op is an op-code, and source and destination are datafields. It has the following op-codes :

MOV(move source to destination)
ADD(add source to destination)
SUB(substract source from destination)

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

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

For example : MOV R0, M stores contents of Register R0 into memory location M.

Instruction cost = 1+cost for source and destination address modes.

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 occupies only one word of memory.

The three-address statement a : = b + c can be implemented by many different instruction sequences :

    MOV b, R0
    ADD c, R0   cost = 6
    MOV R0, a

    MOV b, a
    ADD c, a   cost = 6

Assuming R0, R1 and R2 contain the addresses of a, b, and c :

MOV *R1, *R0
ADD *R2, *R0   cost = 2

Assuming R1 and R2 contain the values of b and c,respectively and that the value of b is not needed after the assignment, we can use:

Add R2,R1
Mov R1,a   cost=3

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

Run-time Storage Management:

Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure. The two standard storage allocation strategies are:

1. Static allocation
2. Stack allocation

In static allocation, the position of an activation record in memory is fixed
at compile time.
In stack allocation, a new activation record is pushed onto the stack for each
execution of a procedure. The record is popped when the activation ends.

The following three-address statements are associated with the run-time allocation and deallocation of activation records:

Call,

Return,

Halt, and

Action, a placeholder for other statements.

We assume that the run-time memory is divided into areas for:

1. Code
2. Static data
3. Stack

Static allocation:

The codes needed to implement static allocation are as follows:

MOV #here+20,callee.static_area	/*It saves return address*/
GOTO callee.code_area	/*It transfers control to the target code for the calledprocedure*/

where,
callee.static_area - Address of the activation record
callee.code_area - Address of the first instruction for called
procedure #here + 20 - Literal return address which is the address
of the instruction following GOTO.

Implementation of return statement:

A return from procedure callee is implemented by : GOTO *callee.static_area

This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Stack allocation:

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, the position of activation record is stored in register so words in activation records can be accessed as offsets from the value in this register.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

MOV #stackstart , SP /* initializes stack */
Code for the first procedure HALT /*
terminate execution */

Implementation of Call statement:

```
ADD #caller.recordsize, SP   /* increment stack pointer */
MOV #here + 16, *SP/*Save return address */
GOTO callee.code_area
where,
caller.recordsize        - size of the activation record
#here + 16 - address of the instruction following the GOTO
```

Implementation of Return statement:

```
GOTO *0 ( SP )        /*return to the caller */
SUB #caller.recordsize, SP   /* decrement SP and restore to previous value */
```

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

The following sequence of three-address statements forms a basic block t1 : = a * a

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

Basic Block Construction

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

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

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:

    The first statement is a leader.

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

    Any statement that immediately follows a goto or conditional goto statementis a leader.

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

Consider the following source code for dot product of two vectors:

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

The three-address code for the above source program is given as (1)
prod := 0
(2) i := 1
(3) t1 := 4* I
(4) t2 := a[t1] /*compute a[i] */
(5) t3 := 4* I
(6) t4 := b[t3] /*compute
(7) b[i] */ t5 := t2*t4
(8) t6 := prod+t5
(9) prod := t6
(10)t7 := i+1
(11)i := t7
(12)if i<=20 goto (3)
Basic block 1: Statement (1) to (2)
Basic block 2: Statement (3) to (12)

A number of transformations can be applied to a basic block without expressions computed by the block. Two important classes of transformation are :

- Structure-preserving transformations
- Algebraic transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

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

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.

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 is 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 construction 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 a basic block is transformed to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

Two statements

t1:=b+c

t2:=x+y

can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

Basic Blocks:

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

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

Basic Block Construction

The following three-address statements are associated with the run- time allocation and Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

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

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:

   a. The first statement is a leader.

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

   c. Any statement that immediately follows a goto or conditional goto statementis a leader.

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

Consider the following source code for dot product of two vectors: The three- address code for the above source program is given as :

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

Basic block 1: Statement (1) to (2)
Basic block 2: Statement (3) to (12)

Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without expressions computed by the block. Two important classes of transformation are :

- Structure-preserving transformations

- Algebraic transformations

Flow Graphs:

Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.The nodes of the flow graph are basic blocks. It has a distinguished initial node.

E.g.: Flow graph for the vector dot product is given as follows:

```
prod : = 0
i : = 1
```

```
t1 : = 4 * i
t2 : = a [ t1 ]
t3 : = 4 * i
t4 : = b [ t3 ]
t5 : = t2 * t4
t6 : = prod + t5
prod : = t6
t7 : = i + 1 i : = t7
if i <= 20 goto B2
```
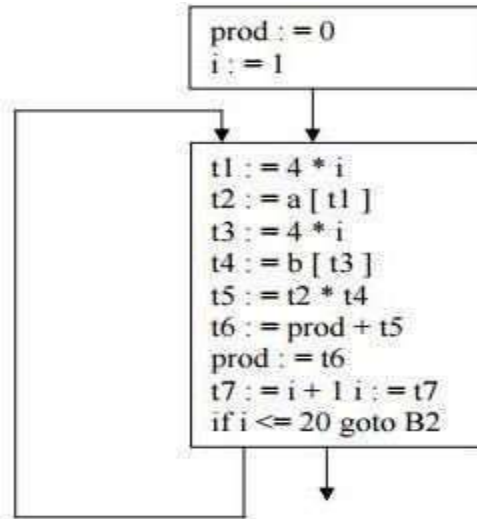
Flow graph for program

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

Register and Address Descriptors:

•A register descriptor is used to keep track of what is currently in each registers. The registerdescriptors 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.

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:

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

2. 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.

3. 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.

4. 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:

Code sequence for the example is:

$$t := a - b$$
$$u := a - c$$
$$v := t + u$$
$$d := v + u$$

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, R0 SUB b, R0	R0 contains t	t in R0
u := a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0

d := v + u	ADD R1, R0
	
	 MOV R0, d
		R0 contains d	d in R0
			 d in R0 and memory

Generating Code for Indexed Assignments:

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

Statements	Code Generated	Cost
a := b[i]	MOV b(Ri), R	2
a[i] := b	MOV b, a(Ri)	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 *Rp, a	2
*p : = a	MOV a, *Rp	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	MOV y, R0

if x < 0 goto z ADD

z, R0

MOV R0,x

CJ< z

Register Allocation and Assignment:

•Global Register Allocation

•Usage Counts

•Register Assignment for Outer Loops

•Register Allocation by Graph Coloring

Global Register Allocation:

The code generation algorithm use registers to hold values for the duration of a single basic block. However, all live variables were stored at the end of each block. To save some of these stores and corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally). Since programs spend most of their time in inner loops, a natural approach to global register assignment is to try to keep a frequently used value in a fixed register throughout a loop.

For the time being, assume that we know the loop structure of a flow graph, and that we know what values computed in a basic block are used outside that block.

Usage Counts:

In this section we shall assume that the savings to be realized by keeping a variable x in a register for the duration of a loop L is one unit of cost for each reference to x if x is already in a register. However, if we use the approach to generate code for a block, there is a good chance that after x has been computed in a block it will remain in a register if there are subsequent uses of x in that block. Thus we count a savings of one for each use of x in loop L that is not preceded by an assignment to x in the same block. We also save two units if we can avoid a store of x at the end of a block. Thus, if x is allocated a register, we count a savings of two for each block in loop L for which x is live on exit and in which x is assigned a value.

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B)$$

Example :

Consider the the basic blocks in the inner loop depicted in Fig, where jump and conditional jump statements have been omitted. Assume registers RO, Rl, and R2 are allocated to hold values throughout the loop. Variables live on entry into and on exit from each block are shown in Fig for convenience, immediately above and below each block, respectively. There are some subtle points about live variables that we address in the next chapter. For example, notice that both e and f are live at the end of B1, but of these, only e is live on entry to B2 and only f on entry to B3. In general, the variables live at the end of a block are the union of those live at the beginning of each of its successor blocks.
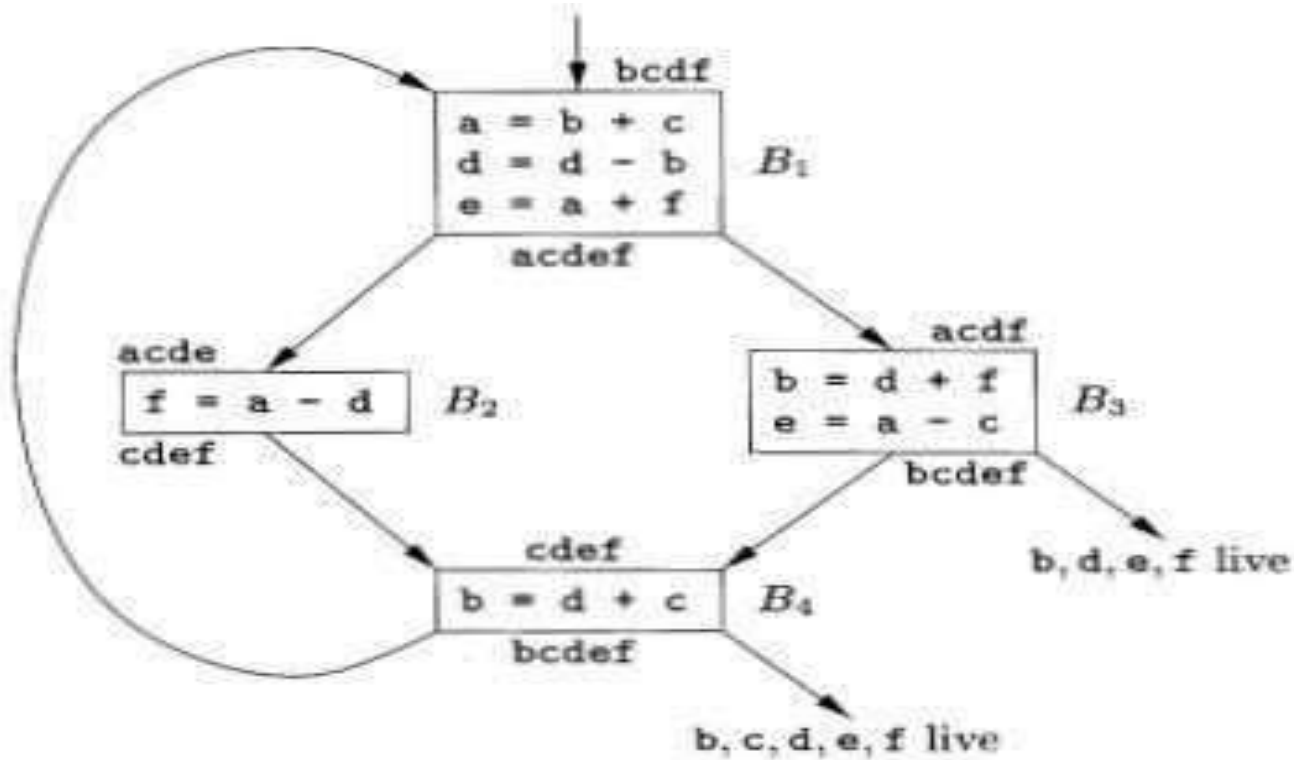
Flow graph of an inner loop
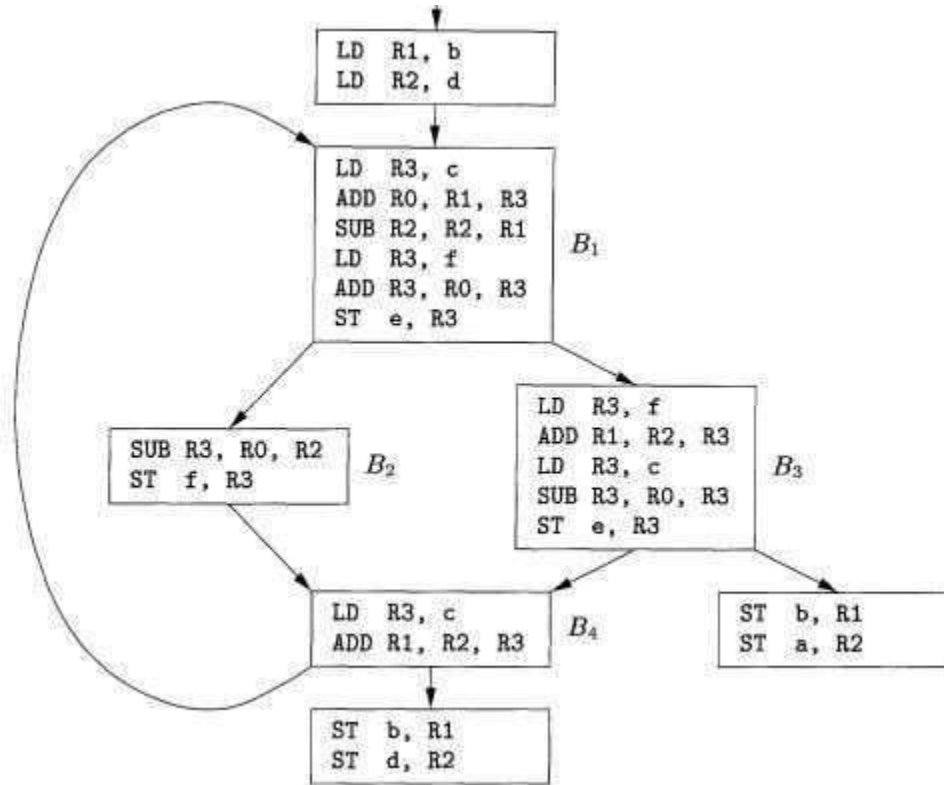
Code sequence using global register assignment

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.

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

Step 1:

    If y is undefined then create node(y).

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

Step 2:

    For the case(i), create a node(OP) whose left child is node(y) and right child is node(z). (Checking for common sub expression). Let n be this node.

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

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

Step 3:

    Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

Consider the block of three- address statements in

```
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)
```

(a) Statement (1)

(b) Statement (2)

(c)

node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

(d) Statement (4)

(e) Statement (5)

(f)

Statement (6), attach identifier prod for Statement (7)
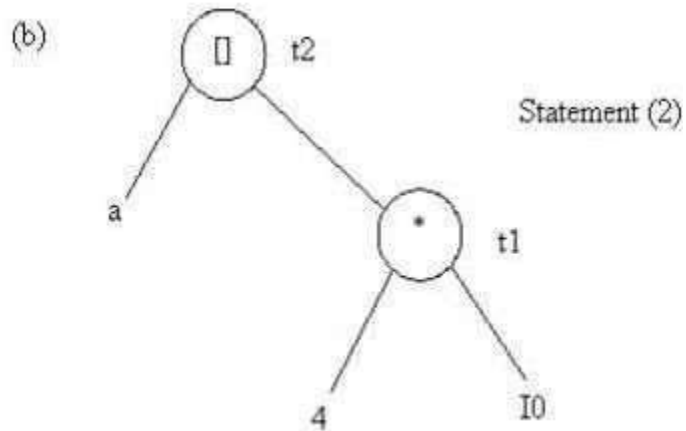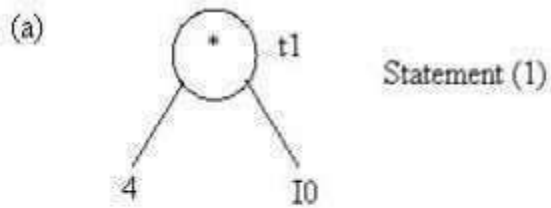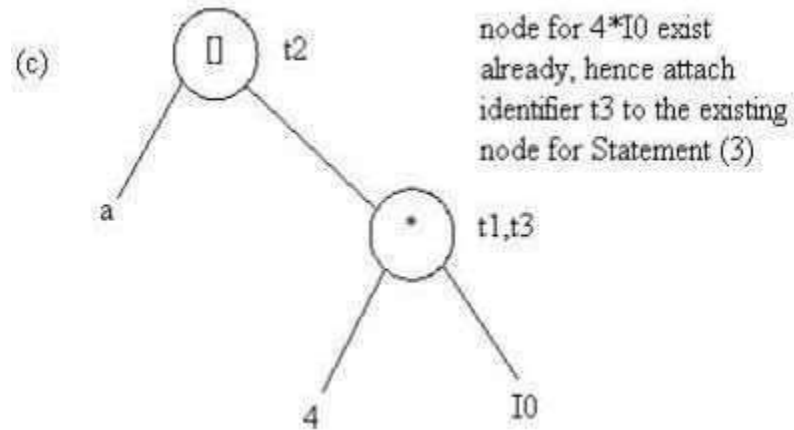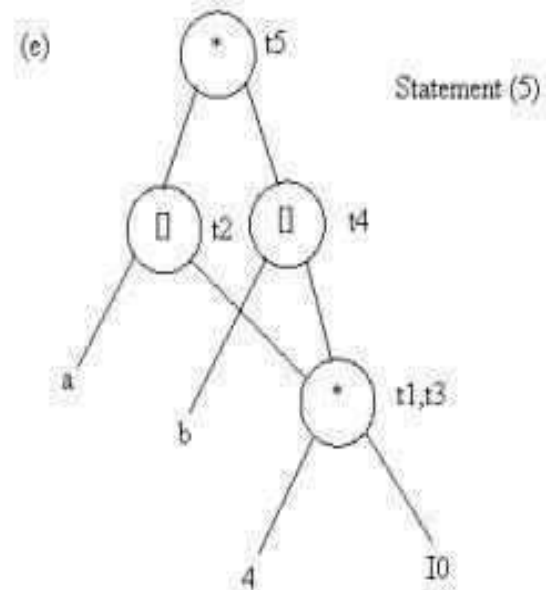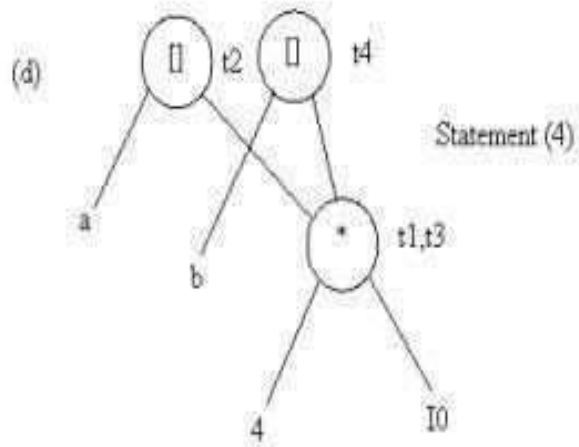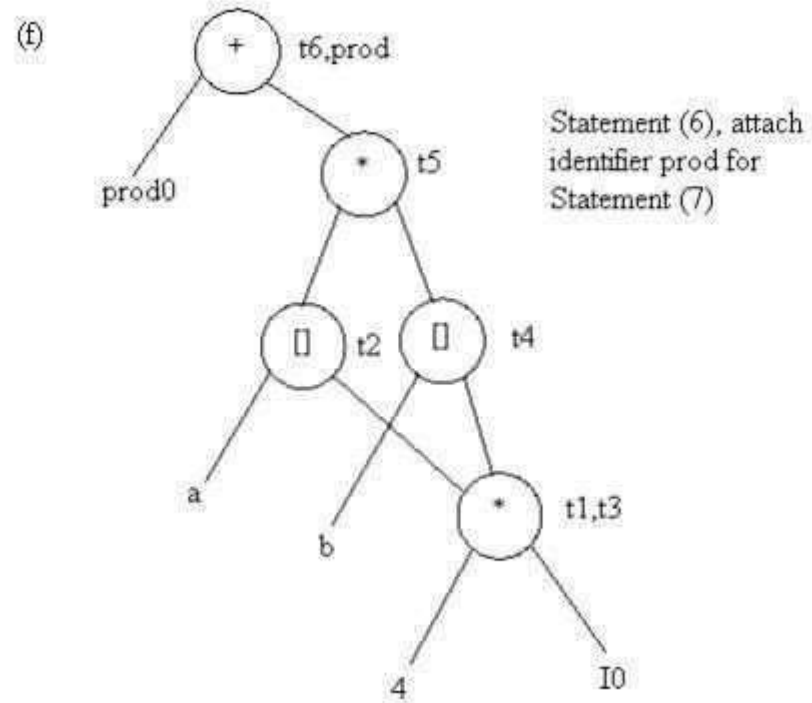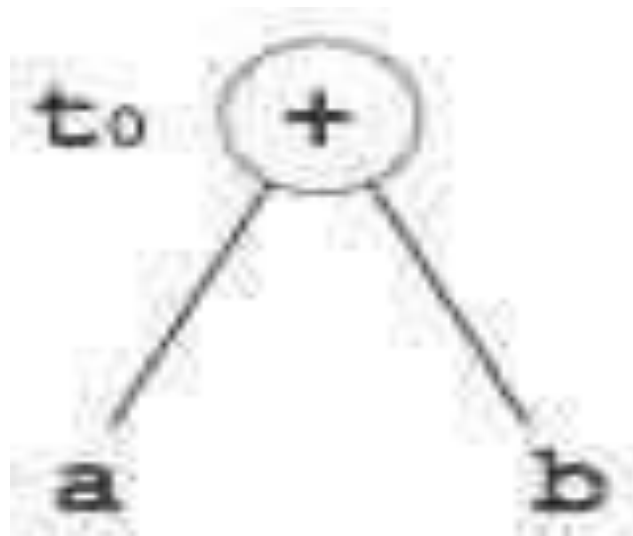
t0 = a + b
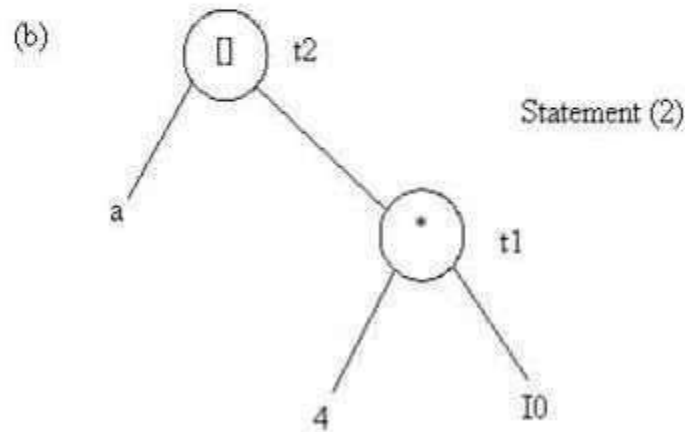t1 = t0 + c
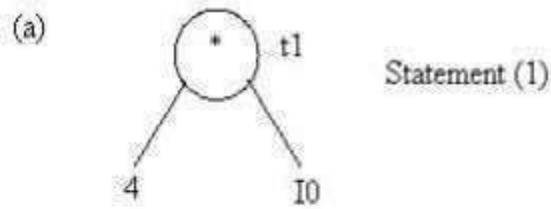d = t0 + t1

$[t_0 = a + b]$
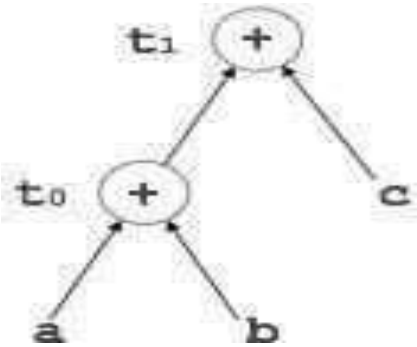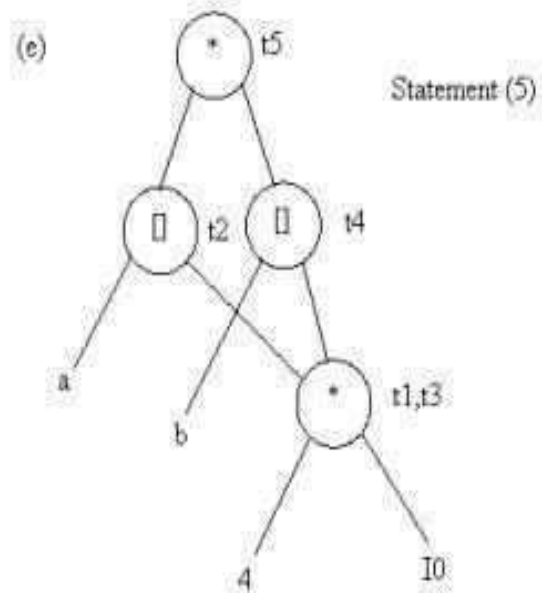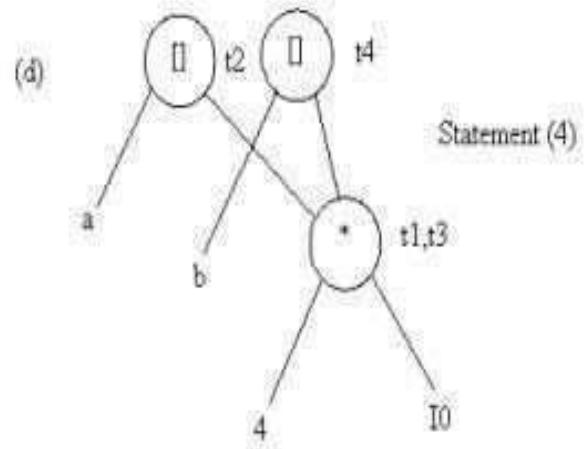
$[t_1 = t_0 + c]$

$[d = t_0 + t_1]$

Consider the block of three- address statements in

```
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)
```
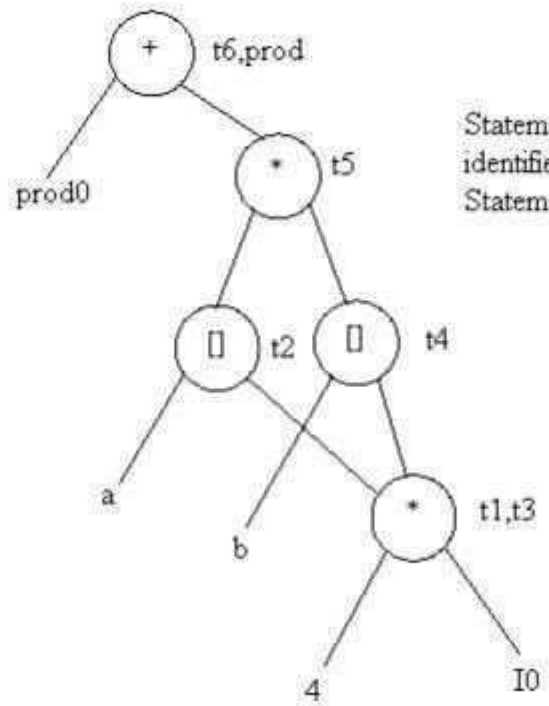
(a) * t1   Statement (1)
4   I0

(b) □ t2   Statement (2)
a   * t1
4   I0

(c) □ t2   node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)
a   * t1,t3
4   I0

(d) Statement (4)

(e) Statement (5)

(f)

+   t6,prod

prod0

*   t5

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

[] t2   [] t4

a

b

*   t1,t3

4   I0

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, —Compilers–Principles, Techniques and Tools‖, Pearson Education, Low Price Edition, 2004