

**I YEAR-I SEMESTER
COURSE CODE: 7MCE1C4**

CORE COURSE-IV-PRINCIPLES OF COMPILER DESIGN

Unit I

Introduction to Compilers: Compilers and Translators – Lexical analysis – Syntax analysis – Intermediate code generation – Optimization – code generation – Bookkeeping – Error handling – compiler writing tools.

Finite Automata and Lexical Analysis: The role of the lexical analyzer – the design of the lexical analyzers – Regular expressions – Finite automata – From regular expressions to finite automata – Minimizing the number of states of a DFA – A language for specifying lexical analyzers – Implementation of a lexical analyzer.

Unit II

The syntactic specification of Programming Languages: Context – free grammars – Derivations and parse trees – Capabilities of context – free grammars.

Basic Parsing Techniques: Parses – Shift – reduce parsing – Operator – precedence parsing – Top-down parsing – Predictive parsers.

Automatic construction of efficient parsers: LR parsers – Constructing SLR parsing tables – Constructing LALR parsing tables.

Unit III

Syntax – Directed translation: Syntax Directed translation schemes – Implementation of syntax – directed translators – Intermediate code – Postfix notation – Parse trees and syntax trees – Three – address code, quadruples, and triples – Translation of assignment statements – Boolean expressions – Statements that alter the flow of control – Postfix translations – Translation with a top-down parser.

Unit IV

Symbol Tables: The contents of a symbol table – Data structures for symbol tables – Representing scope information.

Run time storage administration: Implementation of a simple stack allocation scheme – Implementation of block – structured languages – Storage allocation in block – structured languages.

Error Detection and Recovery: Errors – lexical – phase errors – Syntactic phase errors – Semantic errors.

Unit V

Introduction to code optimization:- The principal sources of optimization – loop optimization– The DAG Representation of basic blocks.

Code generation: object programs – Problems in code generation – A machine model – A simple code generator – Register allocation and assignment – Code generation from DAG's –Peephole optimization.

Text Book:

1. “Principles of Compiler Design” by Alfred V. Aho Jeffrey D. Ullman, Narosa Publishing House, 1989 Reprint 2002

Books for Reference:

1. “Compiler Construction Principles and Practice”, by Dhamdhare D. M, 1981, Macmillan India.
2. “Compiler Design”, by Reinhard Wilhelm, Director Mauser, 1995, Addison Wesley.

UNIT I

OUTLINE

Introduction to Compilers:

- ❖ Compilers and Translators
- ❖ Lexical analysis
- ❖ Syntax analysis
- ❖ Intermediate code generation
- ❖ Optimization
- ❖ Code generation
- ❖ Bookkeeping
- ❖ Error handling
- ❖ Compiler writing tools.

Finite Automata and Lexical Analysis:

- ❖ The role of the lexical analyzer
- ❖ The design of the lexical analyzers
- ❖ Regular expressions
- ❖ Finite automata
- ❖ From regular expressions to finite automata
- ❖ Minimizing the number of states of a DFA
- ❖ A language for specifying lexical analyzers
- ❖ Implementation of a lexical analyzer.

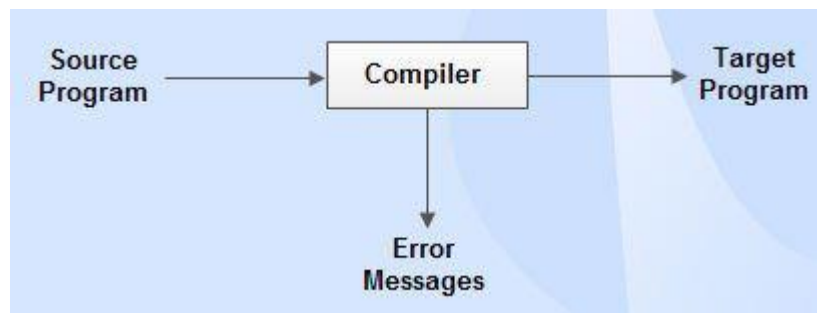
INTRODUCTION TO COMPILERS:

- A compiler is a translator that converts the high-level language into the machine language.
 - Compiler is used to show errors to the programmer.
 - The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
 - In the first part, the source program compiled and translated into the object program (low level language).
 - In the second part, object program translated into the target program through the assembler.
-

COMPILERS AND TRANSLATORS

COMPILER

Compiler is a translator which is used to convert programs in high-level language to low-level language. It translates the entire program and also reports the errors in source program encountered during the translation.



TRANSLATOR

- A program written in high-level language is called as source code. To convert the source code into machine code, translators are needed.
- A translator takes a program written in source language as input and converts it into a program in target language as output.
- It also detects and reports the error during translation.

ROLES OF TRANSLATOR ARE:

- Translating the high-level language program input into an equivalent machine language program.
 - Providing diagnostic messages wherever the programmer violates specification of the high-level language program.
-

LEXICAL ANALYSIS

- Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.
- A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.
- Lexical analysis consists of two stages of processing which are as follows:
 - Scanning
 - Tokenization

Token, Pattern and Lexeme

Token is a valid sequence of characters which are given by lexeme. In a programming language,

- keywords,
- constant,
- identifiers,
- numbers,
- operators and
- punctuations symbols

are possible tokens to be identified.

Pattern

- Pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token.
- It can be defined by regular expressions or grammar rules.

Lexeme

Lexeme is a sequence of characters that matches the pattern for a token
i.e., instance of a
token.

(eg.) $c = a + b * 5;$

Need of Lexical Analyzer

- ***Simplicity of design of compiler*** The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.
 - ***Compiler efficiency is improved*** Specialized buffering techniques for reading characters speed up the compiler process.
 - ***Compiler portability is enhanced***
-

SYNTAX ANALYSIS

- The parser has two functions.
- It checks that the tokens appearing in the input, which is the output of the lexical analyzer.
- It also imposes the token that is used by the subsequent phases of compiler.

For example, if a program PL/I program contains the expression

$A+/B$

Then after lexical analysis this expression appear to the system as the token sequence

$id+/id$

On seeing the /, the syntax analyzer should detect an error situation.

The second aspect of syntax analysis is to make explicit the hierarchical structure.

For example, the expression

$A/B*C$

It has two possible interpretations.

1. Divide A by B and then multiply by C
 2. Multiply B by C and then use the result to divide A.
-

INTERMEDIATE CODE GENERATION

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.

- **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation.

Code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;  
r2 = b + r1;  
a = r2
```

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms: quadruples and triples.

OPTIMIZATION

Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Optimization helps to:

- Reduce the space consumed and increases the speed of compilation.
- Manually analyzing datasets involves a lot of time. Hence we make use of software like Tableau for data analysis. Similarly manually performing the optimization is also tedious and is better done using a code optimizer.
- An optimized code often promotes re-usability.

Types of Code Optimization – The optimization process can be broadly classified into two types:

1. **Machine Independent Optimization** – This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.
2. **Machine Dependent Optimization** – Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

Phases of Optimization

There are generally two phases of optimization:

- **Global Optimization:**
Transformations are applied to large program segments that includes functions, procedures and loops.
- **Local Optimization:**
Transformations are applied to small blocks of statements. The local optimization is done prior to global optimization.

CODE GENERATION

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Example:

Consider the three address statement $x = y + z$. It can have the following sequence of codes:

MOV x, R₀

ADD y, R₀

Register and Address Descriptors:

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

A code-generation algorithm:

The algorithm takes a sequence of three-address statements as input. For each three address statement of the form $a := b \text{ op } c$ perform the various actions. These are as follows:

1. Invoke a function getreg to find out the location L where the result of computation $b \text{ op } c$ should be stored.
2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y'. If the value of y is not already in L then generate the instruction **MOV y', L** to place a copy of y in L.

3. Generate the instruction **OP z', L** where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptor.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of $x := y \text{ op } z$ those register will no longer contain y or z.

Generating Code for Assignment Statements:

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

1. $t := a - b$
2. $u := a - c$
3. $v := t + u$
4. $d := v + u$

Code sequence for the example is as follows:

Statement	Code Generated	Register descriptor Register empty	Address descriptor
$t := a - b$	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 R1
$d := v + u$	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

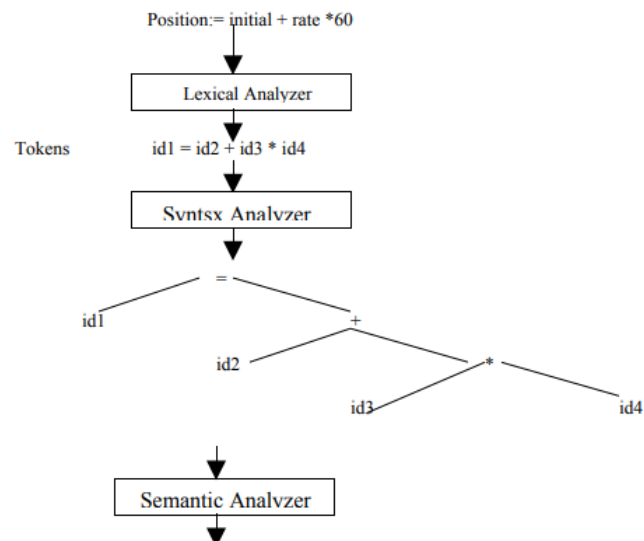
BOOKKEEPING

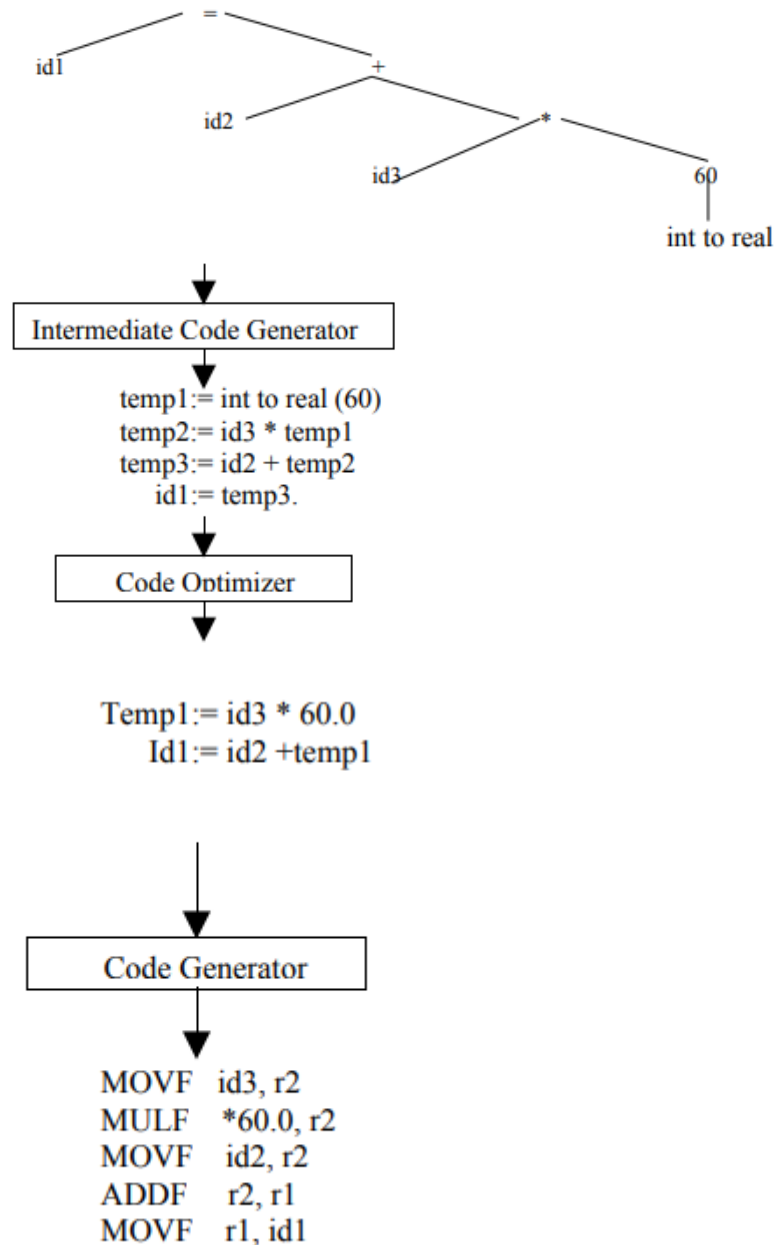
- A compiler needs to collect information about all the data objects that appear in the source program.

- The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers.
- The data structure used to record this information is called as Symbol Table.

ERROR HANDLING

- One of the most important functions of a compiler is the detection and reporting of errors in the source program.
- The error message should allow the programmer to determine exactly where the errors have occurred.
- Errors may occur in all or the phases of a compiler.
- Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg.
- Both of the table-management and error-Handling routines interact with all phases of the compiler.





COMPILER WRITING TOOLS

Some commonly used compiler-construction tools. It include,

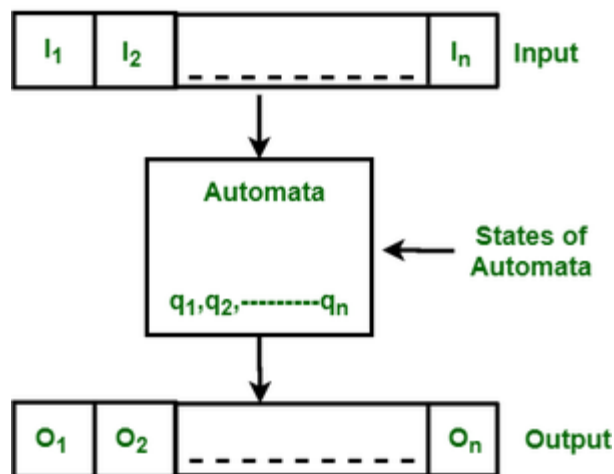
1. Parser generators. (e.g. yacc • Code generator generators)
 2. Scanner generators. (e.g. lex – The input to lex consists of a definition of each token as a regular expression)
 3. Syntax-directed translation engines.
 4. Automatic code generators.
 5. Data-flow analysis engines.
 6. Compiler-construction toolkits
-

FINITE AUTOMATA AND LEXICAL ANALYSIS:

Finite Automata

- Finite Automata(FA) is the simplest machine to recognize patterns.
- The finite automata or finite state machine is abstract machines which have five elements or tuple.
- It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol.
- Basically it is an abstract model of digital computer.

Following figure shows some essential features of a general automation.



The above figure shows following features of automata:

1. Input
2. Output
3. States of automata
4. State relation
5. Output relation

Lexical Analysis

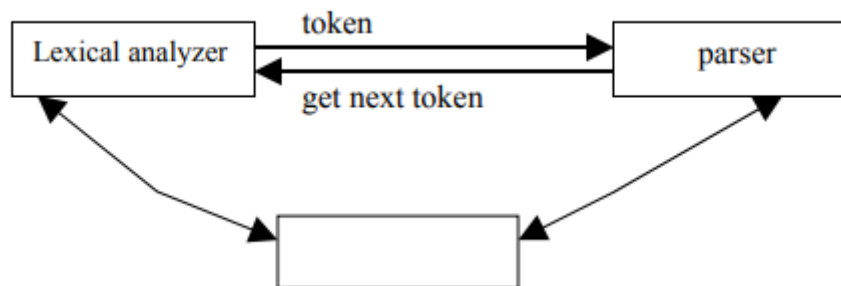
Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of **Tokens**.

- Lexical Analysis can be implemented with the Deterministic finite Automata.
 - The output is a sequence of tokens that is sent to the parser for syntax analysis
-

THE ROLE OF THE LEXICAL ANALYZER

Role of Lexical Analyzer

Role of Lexical Analyzer, the LA is the first phase of a compiler. Its main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.



- Upon receiving a 'get next token' command from the parser, the lexical analyzer reads the input character until it can identify the next token.
- The LA returns to the parser representation for the token it has found.
- The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.
- LA may also perform certain secondary tasks as the user interface.
- One such task is stripping out from the source program the commands and white spaces in the form of blank, tab and new line characters.
- Another is correlating error message from the compiler with the source program.

Input Buffering

- The LA scans the characters of the source program one at a time to discover tokens.

- Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

THE DESIGN OF THE LEXICAL ANALYZERS

There are two ways for designing a lexical analyzer, they are

1. Hand coding
2. Lexical analyzer generator

Hand Coding:

Programmer has to perform the following task Specify the tokens by writing regular expressions.

1. Construct Finite Automata equivalent to a regular expression
2. Recognize the tokens by constructed Finite Automata

two steps are done by lexical analyzer generator automatically.

Lexical Analyzer Generator

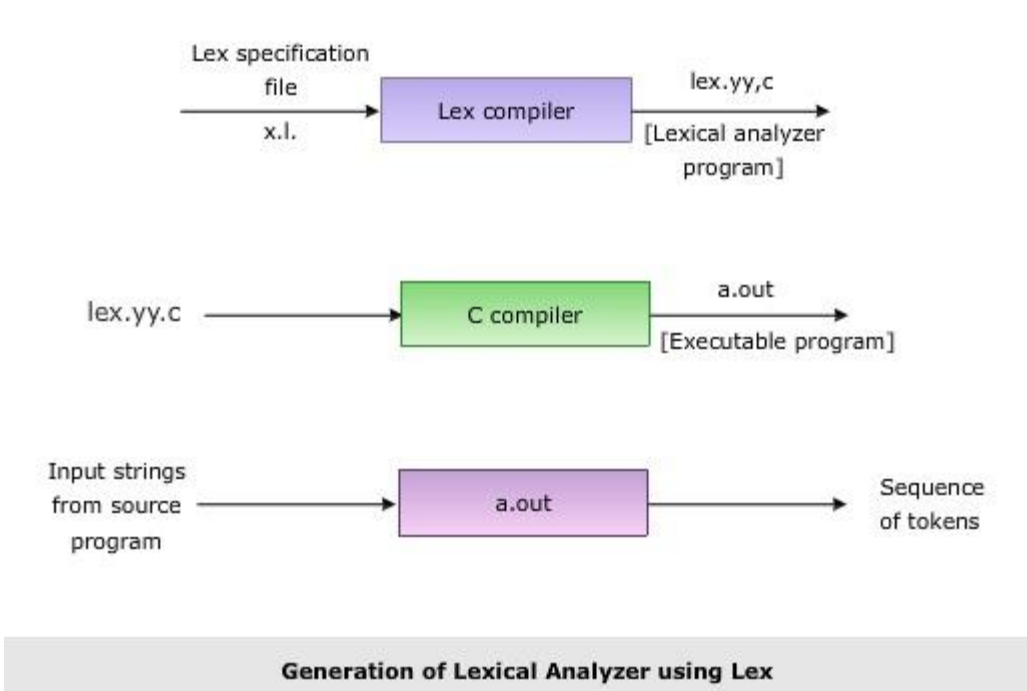
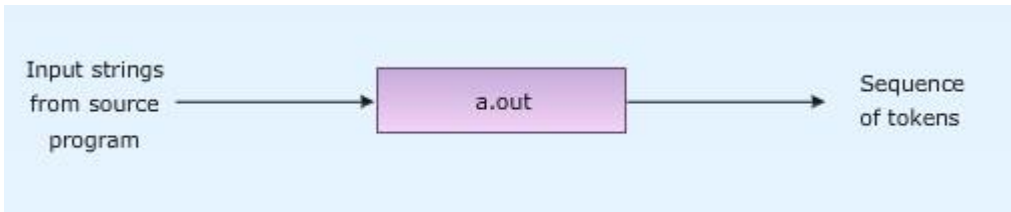
- Lexical Analyzer Generator introduce a tool called Lex, which allows one to specify a lexical analyzer by specifying regular expressions to describe pattern for tokens.
- The input for the lex tool is lex language.
- A program which is written in lex language will compile through lex compiler and produce a C Code called *lex.yy.c* always i.e,



Now, C code is compiled by C compiler and produce a file called a.out as always i.e.,



The C compiler output is a working lexical analyzer that can take a stream of input character and produce a stream of tokens i.e.,



REGULAR EXPRESSIONS

Regular expression is a formula that describes a possible set of string.

Component of regular expression

- X the character x
- .
- [x y z] any of the characters x, y, z,
- R? a R or nothing (=optionally as R)
- R* zero or more occurrences.....

R^+ one or more occurrences

R_1R_2 an R_1 followed by an R_2

R_2R_1 either an R_1 or an R_2 .

- A token is either a single string or one of a collection of strings of a certain type.
- If we view the set of strings in each token class as an language, we can use the regular expression notation to describe tokens.
- Consider an identifier, which is defined to be a letter followed by zero or more letters or digits.

In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the regular expression over alphabet.

- ϵ is a regular expression denoting $\{ \epsilon \}$, that is, the language containing only the empty string.
- For each 'a' in Σ , a is a regular expression denoting $\{ a \}$, the language with only one string consisting of the single symbol 'a'.
- If R and S are regular expressions, then

$(R) | (S)$ means $L_r \cup L_s$

$R.S$ means $L_r \cdot L_s$

R^* denotes L_r^*

FINITE AUTOMATA

Automation is defined as a system where information is transmitted and used for performing some functions without direct participation of man.

1, an automation in which the output depends only on the input is called automation without memory.

2, an automation in which the output depends on the input and state also is called as automation with memory.

3, an automation in which the output depends only on the state of the machine is called a Moore machine.

4. an automation in which the output depends on the state and input at any instant of time is called a mealy machine.

Description of Automata

- 1, an automata has a mechanism to read input from input tape,
- 2, any language is recognized by some automation, Hence these automation are basically language 'acceptors' or 'language recognizers'.

Types of Finite Automata

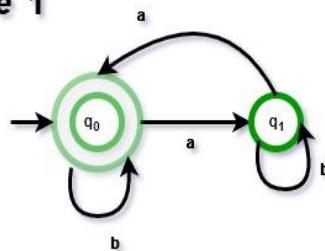
- Deterministic Automata
 - Non-Deterministic Automata
-

FROM REGULAR EXPRESSIONS TO FINITE AUTOMATA

We can convert them to finite automata.

- **Even number of a's :** The regular expression for even number of a's is $(b|ab^*ab^*)^*$. We can construct a finite automata as shown in Figure 1.

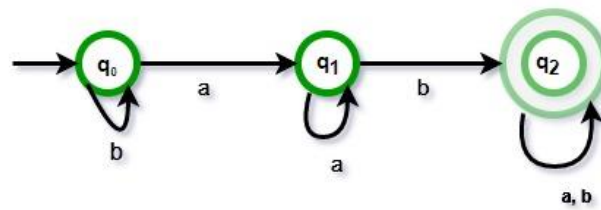
Figure 1



The above automata will accept all strings which have even number of a's. For zero a's, it will be in q_0 which is final state. For one 'a', it will go from q_0 to q_1 and the string will not be accepted. For two a's at any positions, it will go from q_0 to q_1 for 1st 'a' and q_1 to q_0 for second 'a'. So, it will accept all strings with even number of a's.

- **String with 'ab' as substring:** The regular expression for strings with 'ab' as substring is $(a|b)^*ab(a|b)^*$. We can construct finite automata as shown in Figure 2.

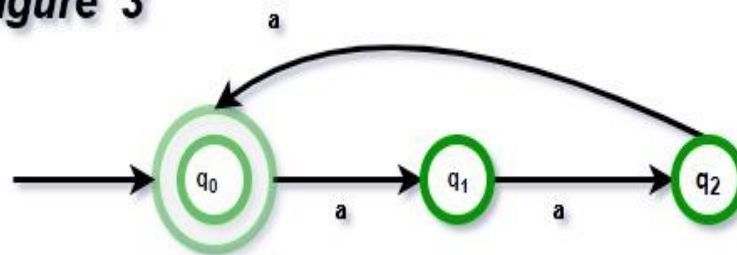
Figure 2



The above automata will accept all string which have 'ab' as substring. The automata will remain in initial state q_0 for b's. It will move to q_1 after reading 'a' and remain in same state for all 'a' afterwards. Then it will move to q_2 if 'b' is read. That means, the string has read 'ab' as substring if it reaches q_2 .

- **String with count of 'a' divisible by 3:** The regular expression for strings with count of a divisible by 3 is $\{a^{3n} \mid n \geq 0\}$. We can construct automata as shown in Figure 3.

Figure 3



The above automata will accept all string of form a^{3n} . The automata will remain in initial state q_0 for ϵ and it will be accepted. For string 'aaa', it will move from q_0 to q_1 then q_1 to q_2 and then q_2 to q_0 . For every set of three a's, it will come to q_0 , hence accepted. Otherwise, it will be in q_1 or q_2 , hence rejected.

Note: If we want to design a finite automata with number of a's as $3n+1$, same automata can be used with final state as q_1 instead of q_0 .

MINIMIZING THE NUMBER OF STATES OF A DFA

Minimization of DFA means reducing the number of states from given FA. Thus, we get the FSM (finite state machine) with redundant states after minimizing the FSM.

We have to follow the various steps to minimize the DFA. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1. $\delta(q, a) = p$

2. $\delta(r, a) = p$

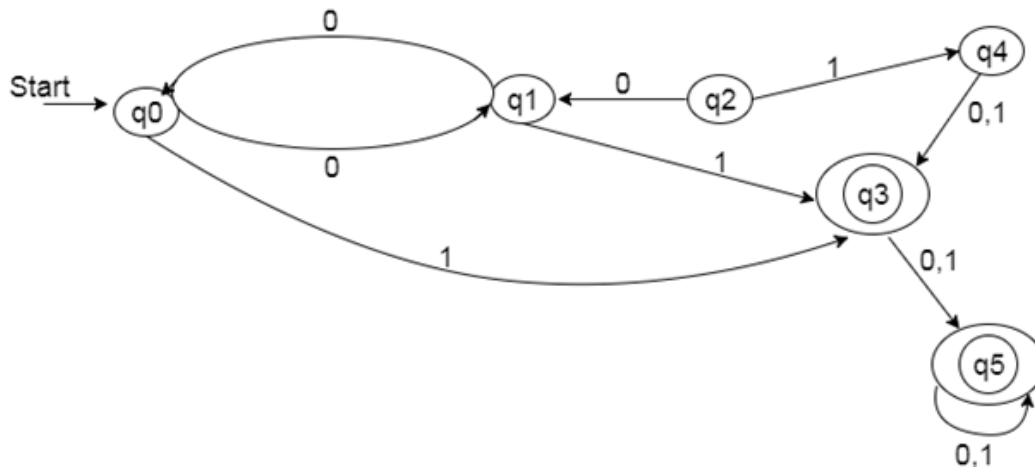
That means, find the two states which have the same value of a and b and remove one of them.

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Example:



Solution:

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states.

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q5	q5
*q5	q5	q5

Step 3: Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

State	0	1
q0	q1	q3
q1	q0	q3

2. Another set contains those rows, which starts from final states.

State	0	1
q3	q5	q5
q5	q5	q5

Step 4: Set 1 has no similar rows so set 1 will be the same.

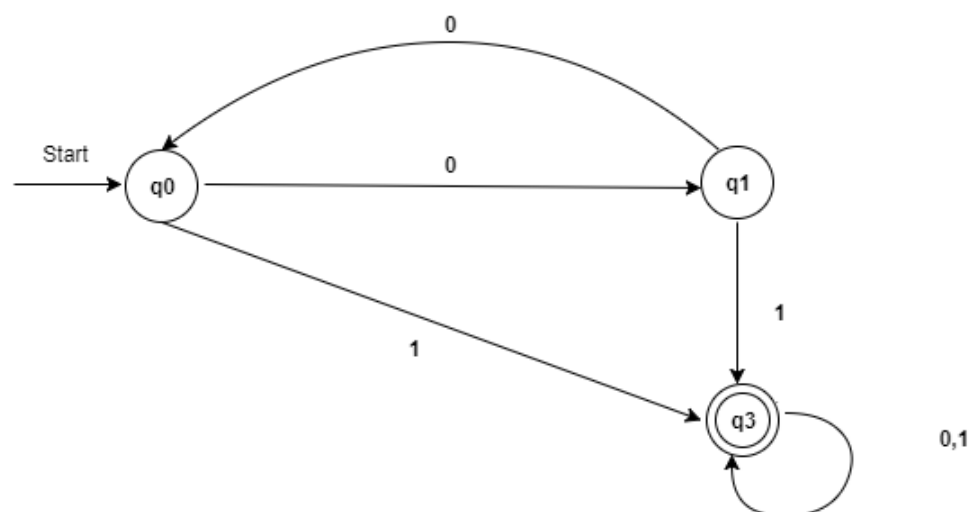
Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

State	0	1
q3	q3	q3

Step 6: Now combine set 1 and set 2 as:

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q3	q3

Now it is the transition table of minimized DFA.



A LANGUAGE FOR SPECIFYING LEXICAL ANALYZERS

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

- Lex
- YACC

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

Creating a lexical analyzer

- First, a specification of a lexical analyzer is prepared by creating a program `lex.l` in the Lex language. Then, `lex.l` is run through the Lex compiler to produce a C program `lex.yy.c`.
- Finally, `lex.yy.c` is run through the C compiler to produce an object program `a.out`, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

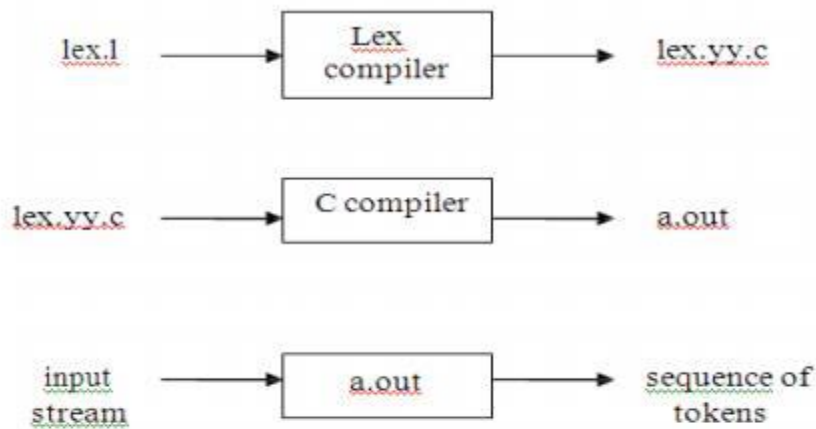


Fig1.11 Creating a lexical analyzer with lex

Lex Specification

A Lex program consists of three parts:

{ definitions }

%%

{ rules }

%%

{ user subroutines }

Definitions include declarations of variables, constants, and regular definitions

Ø **Rules** are statements of the form

p₁ {action₁}

p₂ {action₂}

...

p_n {action_n}

- where p_i is regular expression and action_i describes what action the lexical analyzer should take
- when pattern p_i matches a lexeme. Actions are written in C code.

→ **User subroutines** separately and loaded with the lexical analyzer. are auxiliary procedures needed by the actions. These can be compiled

YACC- YET ANOTHER COMPILER-COMPILER

- Yacc provides a general tool for describing the input to a computer program.
- The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized.

- Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.
-

IMPLEMENTATION OF A LEXICAL ANALYZER

Notation

- For convenience, we use a variation (allow userdefined abbreviations) in regular expression notation
- Union: $A + B \equiv A \mid B$
- Option: $A + \varepsilon \equiv A?$
- Range: $'a' + 'b' + \dots + 'z' \equiv [a-z]$
- Excluded range: complement of $[a-z] \equiv [^a-z]$

Regular Expressions in Lexical Specification

- Last lecture: a specification for the predicate $s \in L(R)$
- But a yes/no answer is not enough!
- Instead: partition the input into tokens
- We will adapt regular expressions to this goal

Regular Expressions \Rightarrow Lexical Spec. (1)

1. Select a set of tokens • Integer, Keyword, Identifier, Open Par, ...
2. Write a regular expression (pattern) for the lexemes of each token
 - Integer = digit +
 - Keyword = 'if' + 'else' + ...
 - Identifier = letter (letter + digit)*
 - OpenPar = '('

- ...

Regular Expressions \Rightarrow Lexical Spec. (2)

3. Construct R, matching all lexemes for all tokens

$R = \text{Keyword} + \text{Identifier} + \text{Integer} + \dots$

$= R_1 + R_2 + R_3 + \dots$

Facts: If $s \in L(R)$ then s is a lexeme

- Furthermore $s \in L(R_i)$ for some “i”
- This “i” determines the token that is reported

Regular Expressions \Rightarrow Lexical Spec. (3)

4. Let input be $x_1 \dots x_n$

- ($x_1 \dots x_n$ are characters)
- For $1 \leq i \leq n$ check $x_1 \dots x_i \in L(R)$?

5. It must be that $x_1 \dots x_i \in L(R_j)$ for some j (if there is a choice, pick a smallest such j)

6. Remove $x_1 \dots x_i$ from input and go to previous step

UNIT II

CHAPTER 4: The Syntactic Specification Of Programming Language

- Context Free Grammars
- Derivations And Parse Trees
- Capabilities Of Context Free Grammar

CHAPTER 5: Basic Parsing Techniques

- Parsers
- Shift reduce parsing
- Operator precedence parsing
- Top down parsing
- Predictive parsers

CHAPTER 6: Automatic construction of effective parsers

- LR Parsers
 - Constructing SLR parsing tables
 - Constructing LALR parsing tables
-

Context-Free Grammar

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four components:

- A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- 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.
- One of the non-terminals is designated as the start symbol (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Example

- We take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, $L = \{ w \mid w = w^R \}$ is not a regular language. But it can be described by means of CFG, as illustrated below:

- $G = (V, \Sigma, P, S)$

- Where:

$$V = \{ Q, Z, N \}$$

$$\Sigma = \{ 0, 1 \}$$

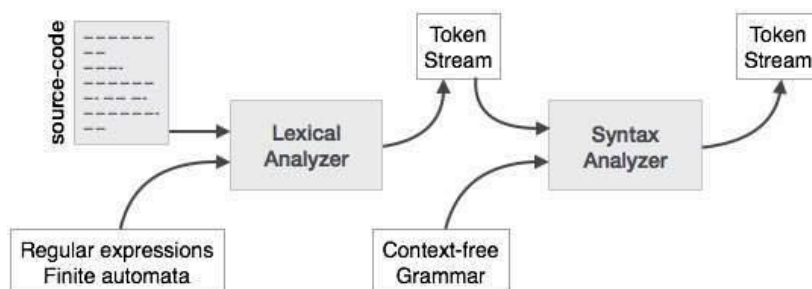
$$P = \{ Q \rightarrow Z \mid Q \rightarrow N \mid Q \rightarrow \epsilon \mid Z \rightarrow 0Q0 \mid N \rightarrow 1Q1 \}$$

$$S = \{ Q \}$$

- This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc

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**.



- This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.
- Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies.

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.

To decide which non-terminal to be replaced with production rule, we can have two options.

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.

Example

Production rules:

```
E → E + E
E → E * E
E → id
```

Input string: id + id * id

The left-most derivation is:

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

Notice that the left-most side non-terminal is always processed first.

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.

The right-most derivation is:

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

Parse Tree

- 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. Let us see this by an example from the last topic.
- We take the left-most derivation of $a + b * c$

- The left-most derivation is:

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

Capabilities of CFG

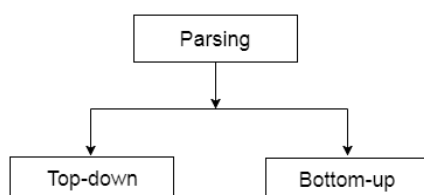
There are the various capabilities of CFG:

- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficient parser can be constructed automatically.
- Using the features of associativity & precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

CHAPTER 5

Parser

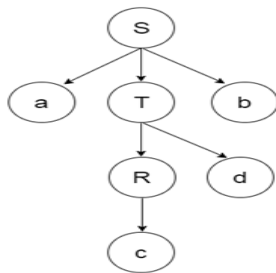
- Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.
- A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.
- Parsing is of two types: top down parsing and bottom up parsing.



Top down parsing

- The top down parsing is known as recursive parsing or predictive parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the top down parsing, the parsing starts from the start symbol and transform it into the input symbol.

Parse Tree representation of input string "acdb" is as follows:



Bottom up parsing

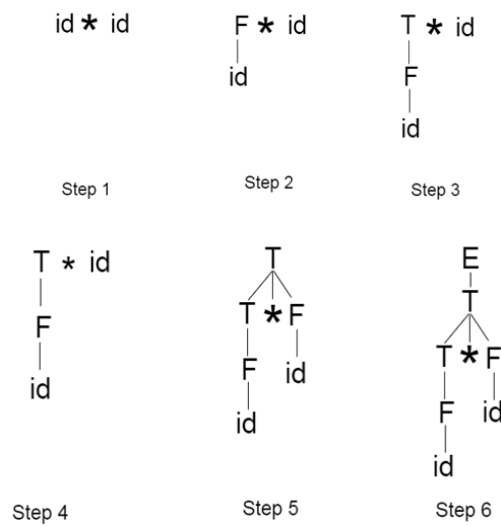
- Bottom up parsing is also known as shift-reduce parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

Example

Production

1. $E \rightarrow T$
2. $T \rightarrow T * F$
3. $T \rightarrow \text{id}$
4. $F \rightarrow T$
5. $F \rightarrow \text{id}$

Parse Tree representation of input string "id * id" is as follows:



Bottom up parsing is classified in to various parsing. These are as follows:

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. Table Driven LR Parsing
 - a) LR(1)
 - b) SLR(1)
 - c) CLR (1)
 - d) LALR(1)

Shift reduce parsing

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.

A String $\xrightarrow{\text{reduce to}}$ the starting symbol

- Shift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will be replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

Example:

Grammar:

- $S \rightarrow S+S$
- $S \rightarrow S-S$
- $S \rightarrow (S)$
- $S \rightarrow a$

Input string:

a1-(a2+a3)

Parsing table:

Stack contents	Input string	Actions
\$	a1-(a2+a3)\$	shift a1
\$a1	-(a2+a3)\$	reduce by $S \rightarrow a$
\$S	-(a2+a3)\$	shift -
\$S-	(a2+a3)\$	shift (
\$S-(a2+a3)\$	shift a2
\$S-(a2	+a3)\$	reduce by $S \rightarrow a$
\$S-(S	+a3) \$	shift +
\$S-(S+	a3) \$	shift a3
\$S-(S+a3) \$	reduce by $S \rightarrow a$
\$S-(S+S) \$	shift)
\$S-(S+S)	\$	reduce by $S \rightarrow S+S$
\$S-(S)	\$	reduce by $S \rightarrow (S)$
\$S-S	\$	reduce by $S \rightarrow S-S$
\$S	\$	Accept

There are two main categories of shift reduce parsing as follows:

1. Operator-Precedence Parsing
2. LR-Parser

Operator precedence parsing

- ❖ Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.
- ❖ A grammar is said to be operator precedence grammar if it has two properties:
 - No R.H.S. of any production has $a \in$.
 - No two non-terminals are adjacent.
- ❖ Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.
- ❖ There are the three operator precedence relations:
 - $a > b$ means that terminal "a" has the higher precedence than terminal "b".
 - $a < b$ means that terminal "a" has the lower precedence than terminal "b".
 - $a \doteq b$ means that the terminal "a" and "b" both have same precedence.

Precedence table:

	+	*	()	id	\$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(<	<	<	=	<	X
)	>	>	X	>	X	>
id	>	>	X	>	X	>
\$	<	<	<	X	<	X

Parsing Action

- Both end of the given input string, add the \$ symbol.
- Now scan the input string from left right until the > is encountered.
- Scan towards left over all the equal precedence until the first left most < is encountered.
- Everything between left most < and right most > is a handle.
- \$ on \$ means parsing is successful.

Example

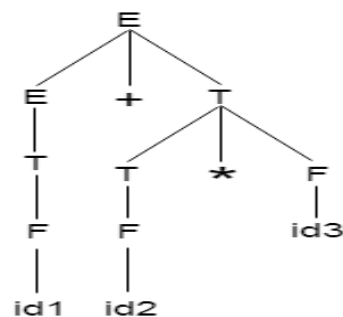
Grammar:

- $E \rightarrow E+T/T$
- $T \rightarrow T*F/F$
- $F \rightarrow id$

Given string:

w = id + id * id

Let us consider a parse tree for it as follows:



On the basis of above tree, we can design following operator precedence table:

	E	T	F	id	+	*	\$
E	X	X	X	X	\doteq	X	\triangleright
T	X	X	X	X	\triangleright	\doteq	\triangleright
F	X	X	X	X	\triangleright	\triangleright	\triangleright
id	X	X	X	X	\triangleright	\triangleright	\triangleright
+	X	\doteq	\triangleleft	\triangleleft	X	X	X
*	X	X	\doteq	\triangleleft	X	X	X
\$	\triangleleft	\triangleleft	\triangleleft	\triangleleft	X	X	X

Now let us process the string with the help of the above precedence table:

\$ < id1 > + id2 * id3 \$

\$ < F > + id2 * id3 \$

\$ < T > + id2 * id3 \$

\$ < E \doteq + < id2 > * id3 \$

\$ < E \doteq + < F > * id3 \$

\$ < E \doteq + < T \doteq * < id3 > \$

\$ < E \doteq + < T \doteq * \doteq F > \$

\$ < E \doteq + \doteq T > \$

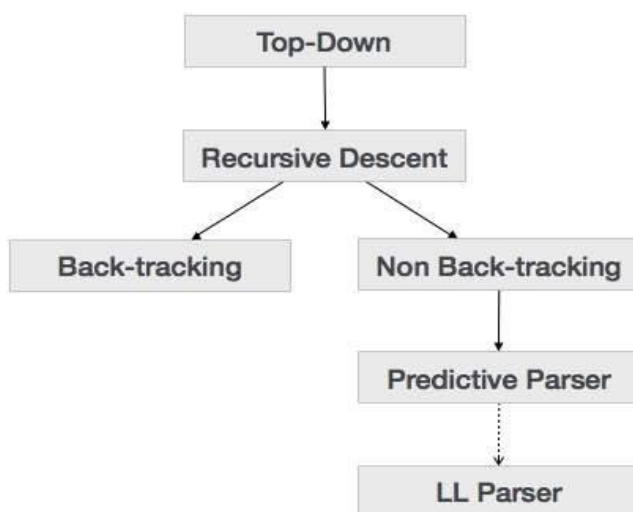
\$ < E \doteq + \doteq T > \$

\$ < E > \$

Accept.

Top-Down Parser

- ❖ We have learnt in the last chapter that the top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:



Recursive Descent Parsing

- ❖ Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.
- ❖ This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Back-tracking

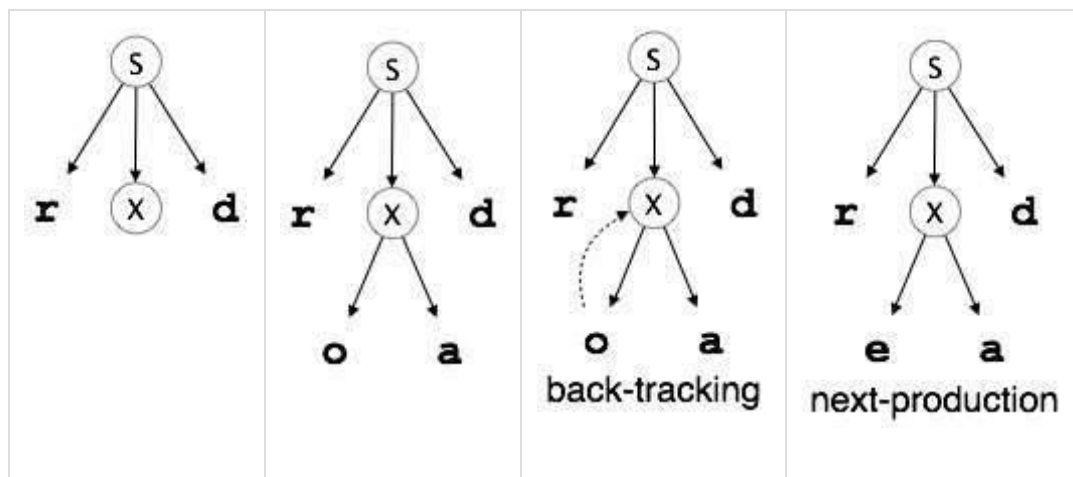
- ❖ Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

$S \rightarrow rXd|rZd$

$X \rightarrow oa|ea$

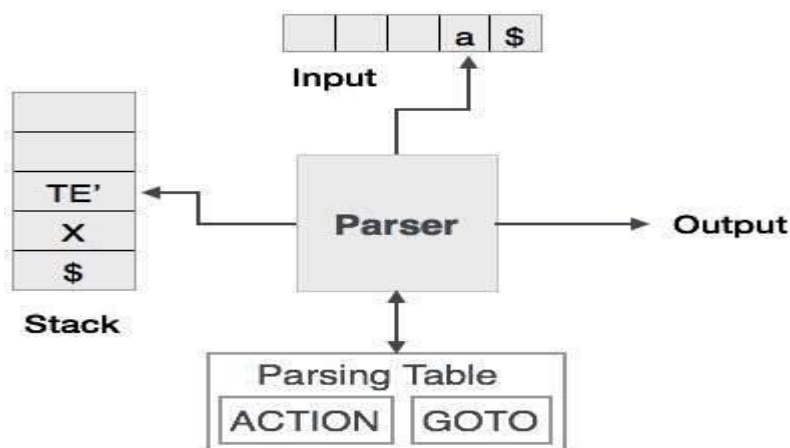
$Z \rightarrow ai$

- ❖ 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 ($S \rightarrow rXd$) 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 ($X \rightarrow oa$). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, ($X \rightarrow ea$).
- ❖ Now the parser matches all the input letters in an ordered manner. The string is accepted.

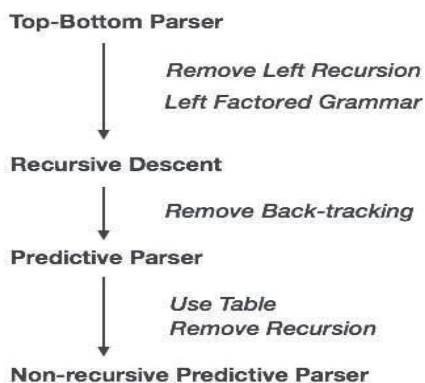


Predictive Parser

- ❖ Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.
- ❖ To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



- ❖ Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol \$ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.



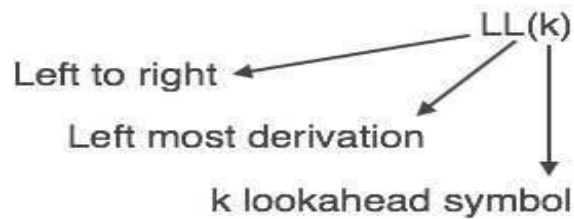
- ❖ In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

LL Parser

- ❖ An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL

grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

- ❖ LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally $k = 1$, so LL(k) may also be written as LL(1).



LL Parsing Algorithm

- ❖ We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.
- ❖ Given below is an algorithm for LL(1) Parsing:

Input:

string ω

parsing table M for grammar G

Output:

If $\omega \in L(G)$ then left-most derivation of ω ,
error otherwise.

InitialState: $\$S$ on stack (with S being start symbol)

$\omega\$$ in the input buffer

SET ip to point the first symbol of $\omega\$$.

repeat

let X be the top stack symbol and a the symbol pointed by ip.

if $X \in V_t$ or $\$$

if $X = a$

POP X and advance ip.

else

error()

endif

```

else      /* X is non-terminal */
if M[X,a]= X → Y1, Y2,...Yk
    POP X
    PUSH Yk,Yk-1,... Y1 /* Y1 on top */
Output the production X → Y1, Y2,...Yk
else
error()
endif
endif
until X = $      /* empty stack */

```

A grammar G is LL(1) if $A \rightarrow \alpha \mid \beta$ are two distinct productions of G :

- for no terminal, both α and β derive strings beginning with a .
- at most one of α and β can derive empty string.
- if $\beta \rightarrow t$, then α does not derive any string beginning with a terminal in FOLLOW(A).

Predictive Parsing

- ❖ The goal of predictive parsing is to construct a top-down parser that never backtracks. To do so, we must transform a grammar in two ways:
 1. eliminate left recursion, and
 2. perform left factoring.
- ❖ These rules eliminate most common causes for backtracking although they do not guarantee a completely backtrack-free parsing (called LL(1) as we will see later).

LR Parser

- ❖ LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.
- ❖ In the LR parsing, "L" stands for left-to-right scanning of the input.
- ❖ "R" stands for constructing a right most derivation in reverse.
- ❖ "K" is the number of input symbols of the look ahead used to make number of parsing decision.
- ❖ LR parsing is divided into four parts:
 - LR (0) parsing
 - SLR parsing,
 - CLR parsing
 - LALR parsing.

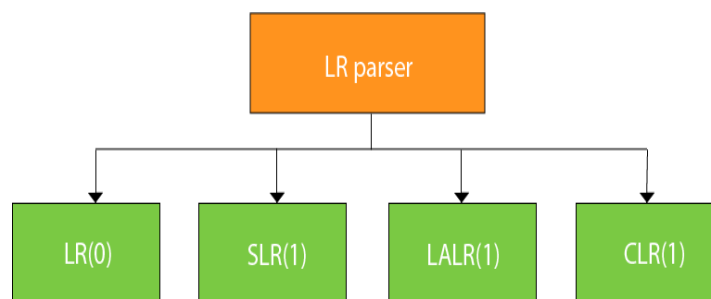


Fig: Types of LR parser

LR algorithm:

- The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.

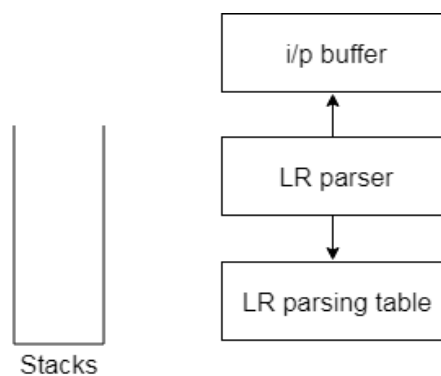


Fig: Block diagram of LR parser

- Input buffer is used to indicate end of input and it contains the string to be parsed followed by a \$ Symbol.
- A stack is used to contain a sequence of grammar symbols with a \$ at the bottom of the stack.
- Parsing table is a two dimensional array. It contains two parts: Action part and Go To part.

LR (1) Parsing

Various steps involved in the LR (1) Parsing:

- For the given input string write a context free grammar.
- Check the ambiguity of the grammar.
- Add Augment production in the given grammar.
- Create Canonical collection of LR (0) items.
- Draw a data flow diagram (DFA).
- Construct a LR (1) parsing table.

Augment Grammar

- Augmented grammar G' will be generated if we add one more production in the given grammar G . It helps the parser to identify when to stop the parsing and announce the acceptance of the input.

Example

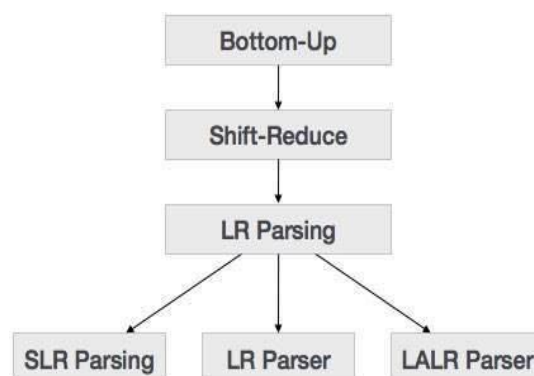
Given grammar

1. $S \rightarrow AA$
2. $A \rightarrow aA \mid b$

The Augment grammar G' is represented by

1. $S' \rightarrow S$
2. $S \rightarrow AA$
3. $A \rightarrow aA \mid b$

- Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



Shift-Reduce Parsing

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step:** The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step :** When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a

POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

LR Parser

- The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique.
- LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.
- There are three widely used algorithms available for constructing an LR parser:
- SLR(1) – Simple LR Parser:
 - Works on smallest class of grammar
 - Few number of states, hence very small table
 - Simple and fast construction
- LR(1) – LR Parser:
 - Works on complete set of LR(1) Grammar
 - Generates large table and large number of states
 - Slow construction
- LALR(1) – Look-Ahead LR Parser:
 - Works on intermediate size of grammar
 - Number of states are same as in SLR(1)

LR Parsing Algorithm

Here we describe a skeleton algorithm of an LR parser:

```
token=next_token()

repeat forever
  s = top of stack

  if action[s, token]="shift si" then
    PUSH token
    PUSH si
    token=next_token()

  elseif action[s, token]="reduce A::= $\beta$ " then
    POP  $2*|\beta|$  symbols
    s = top of stack
    PUSH A
```

PUSH goto[s,A]

elseif action[s, token]="accept"then

return

else

error()

LL vs. LR

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Canonical Collection of LR(0) items

- An LR (0) item is a production G with dot at some position on the right side of the production.
- LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.
- In the LR (0), we place the reduce node in the entire row.

Example

Given grammar:

1. $S \rightarrow AA$
2. $A \rightarrow aA \mid b$

Add Augment Production and insert ' \cdot ' symbol at the first position for every production in G

1. $S' \rightarrow \cdot S$
2. $S \rightarrow \cdot AA$
3. $A \rightarrow \cdot aA$
4. $A \rightarrow \cdot b$

I0 State:

- Add Augment production to the I0 State and Compute the Closure
- $I0 = \text{Closure}(S' \rightarrow \cdot S)$
- Add all productions starting with S in to I0 State because " \cdot " is followed by the non-terminal. So, the I0 State becomes

I0 = $S' \rightarrow \cdot S$

$S \rightarrow \cdot AA$

- Add all productions starting with "A" in modified I0 State because " \cdot " is followed by the non-terminal. So, the I0 State becomes.

I0 = $S' \rightarrow \cdot S$

$S \rightarrow \cdot AA$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

I1 = Go to (I0, S) = closure ($S' \rightarrow S \cdot$) = $S' \rightarrow S \cdot$

- Here, the Production is reduced so close the State.

I1 = $S' \rightarrow S \cdot$

I2 = Go to (I0, A) = closure ($S \rightarrow A \cdot A$)

- Add all productions starting with A in to I2 State because " \cdot " is followed by the non-terminal. So, the I2 State becomes

I2 = $S \rightarrow A \cdot A$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

- Go to (I2, a) = Closure ($A \rightarrow a \cdot A$) = (same as I3)
- Go to (I2, b) = Closure ($A \rightarrow b \cdot$) = (same as I4)

I3 = Go to (I0, a) = Closure ($A \rightarrow a \cdot A$)

- Add productions starting with A in I3.

$A \rightarrow a \cdot A$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

- Go to (I3, a) = Closure ($A \rightarrow a \bullet A$) = (same as I3)

Go to (I3, b) = Closure ($A \rightarrow b \bullet$) = (same as I4)

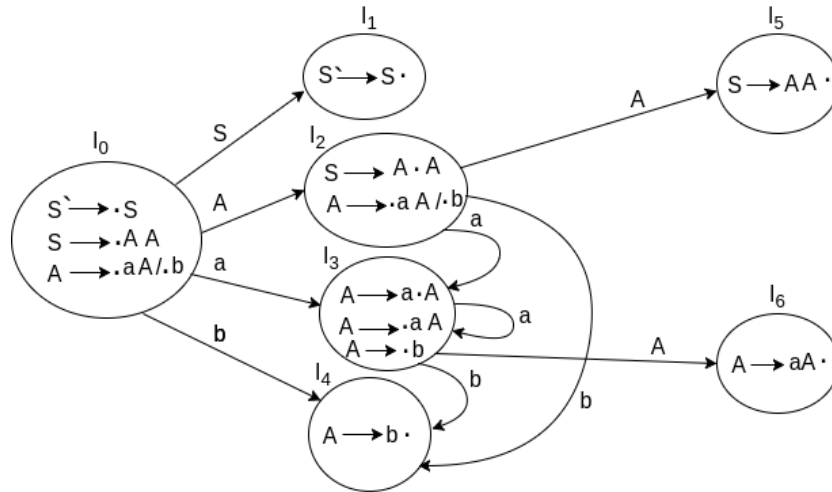
I4= Go to (I0, b) = closure ($A \rightarrow b \bullet$) = $A \rightarrow b \bullet$

I5= Go to (I2, A) = Closure ($S \rightarrow AA \bullet$) = $SA \rightarrow A \bullet$

I6= Go to (I3, A) = Closure ($A \rightarrow aA \bullet$) = $A \rightarrow aA \bullet$

Drawing DFA:

The DFA contains the 7 states I0 to I6.



LR(0) Table

- If a state is going to some other state on a terminal then it correspond to a shift move.
- If a state is going to some other state on a variable then it correspond to go to move.
- If a state contain the final item in the particular row then write the reduce node completely.

States	Action			Go to	
	a	b	\$	A	S
I ₀	S3	S4		2	1
I ₁			accept		
I ₂	S3	S4		5	
I ₃	S3	S4		6	
I ₄	r3	r3	r3		
I ₅	r1	r1	r1		
I ₆	r2	r2	r2		

Explanation:

- I0 on S is going to I1 so write it as 1.
- I0 on A is going to I2 so write it as 2.
- I2 on A is going to I5 so write it as 5.
- I3 on A is going to I6 so write it as 6.
- I0, I2 and I3 on a are going to I3 so write it as S3 which means that shift 3.
- I0, I2 and I3 on b are going to I4 so write it as S4 which means that shift 4.

- I4, I5 and I6 all states contains the final item because they contain • in the right most end. So rate the production as production number.

Productions are numbered as follows:

$$S \rightarrow AA \quad \dots (1)$$

$$A \rightarrow aA \quad \dots (2)$$

$$A \rightarrow b \quad \dots (3)$$

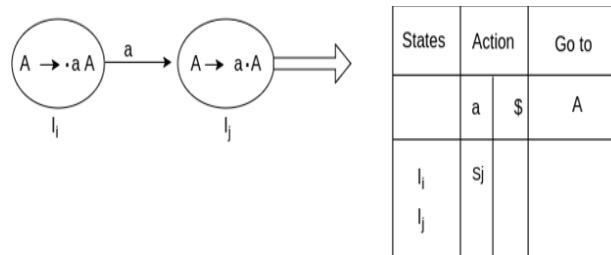
- I1 contains the final item which drives ($S' \rightarrow S\bullet$), so action {I1, \$} = Accept.
- I4 contains the final item which drives $A \rightarrow b\bullet$ and that production corresponds to the production number 3 so write it as r3 in the entire row.
- I5 contains the final item which drives $S \rightarrow AA\bullet$ and that production corresponds to the production number 1 so write it as r1 in the entire row.
- I6 contains the final item which drives $A \rightarrow aA\bullet$ and that production corresponds to the production number 2 so write it as r2 in the entire row.

SLR (1) Parsing

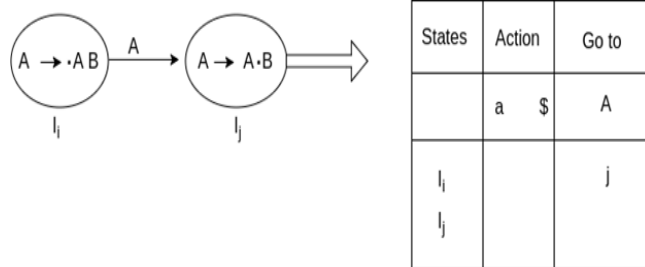
- SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. The only difference is in the parsing table. To construct SLR (1) parsing table, we use canonical collection of LR (0) item.
- In the SLR (1) parsing, we place the reduce move only in the follow of left hand side.
- Various steps involved in the SLR (1) Parsing:
- For the given input string write a context free grammar
 - Check the ambiguity of the grammar
 - Add Augment production in the given grammar
 - Create Canonical collection of LR (0) items
 - Draw a data flow diagram (DFA)
 - Construct a SLR (1) parsing table

SLR (1) Table Construction

- ❖ The steps which use to construct SLR (1) Table is given below:
- ❖ If a state (I_i) is going to some other state (I_j) on a terminal then it corresponds to a shift move in the action part.



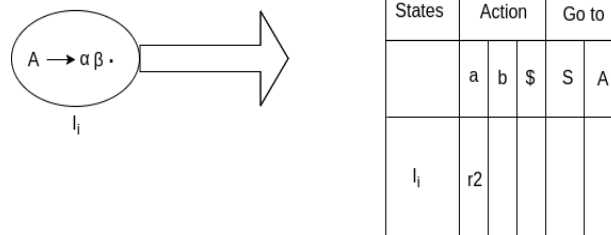
- ❖ If a state (I_i) is going to some other state (I_j) on a variable then it correspond to go to move in the Go to part.



- ❖ If a state (I_i) contains the final item like $A \rightarrow ab\bullet$ which has no transitions to the next state then the production is known as reduce production. For all terminals X in FOLLOW (A), write the reduce entry along with their production numbers.

Example

1. $S \rightarrow \bullet Aa$
 2. $A \rightarrow \alpha\beta\bullet$
1. Follow(S) = { \$ }
 2. Follow (A) = { a }



SLR (1) Grammar

$S \rightarrow E$
 $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow id$

- ❖ Add Augment Production and insert '•' symbol at the first position for every production in G

$S' \rightarrow \bullet E$
 $E \rightarrow \bullet E + T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T * F$
 $T \rightarrow \bullet F$
 $F \rightarrow \bullet id$

I0 State:

- ❖ Add Augment production to the I0 State and Compute the Closure

I0 = Closure ($S' \rightarrow \bullet E$)

- ❖ Add all productions starting with E in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = $S' \rightarrow \bullet E$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

- ❖ Add all productions starting with T and F in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0 = $S' \rightarrow \bullet E$

$E \rightarrow \bullet E + T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet id$

I1 = Go to (I0, E) = closure ($S' \rightarrow E \bullet$, $E \rightarrow E \bullet + T$)

I2 = Go to (I0, T) = closure ($E \rightarrow T \bullet T$, $T \rightarrow \bullet * F$)

I3 = Go to (I0, F) = Closure ($T \rightarrow F \bullet$) = $T \rightarrow F \bullet$

I4 = Go to (I0, id) = closure ($F \rightarrow id \bullet$) = $F \rightarrow id \bullet$

I5 = Go to (I1, +) = Closure ($E \rightarrow E + \bullet T$)

- ❖ Add all productions starting with T and F in I5 State because "." is followed by the non-terminal. So, the I5 State becomes

I5 = $E \rightarrow E + \bullet T$

$T \rightarrow \bullet T * F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet id$

- ❖ Go to (I5, F) = Closure ($T \rightarrow F \bullet$) = (same as I3)
Go to (I5, id) = Closure ($F \rightarrow id \bullet$) = (same as I4)

I6 = Go to (I2, *) = Closure ($T \rightarrow T * \bullet F$)

- ❖ Add all productions starting with F in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

I6 = $T \rightarrow T * \bullet F$

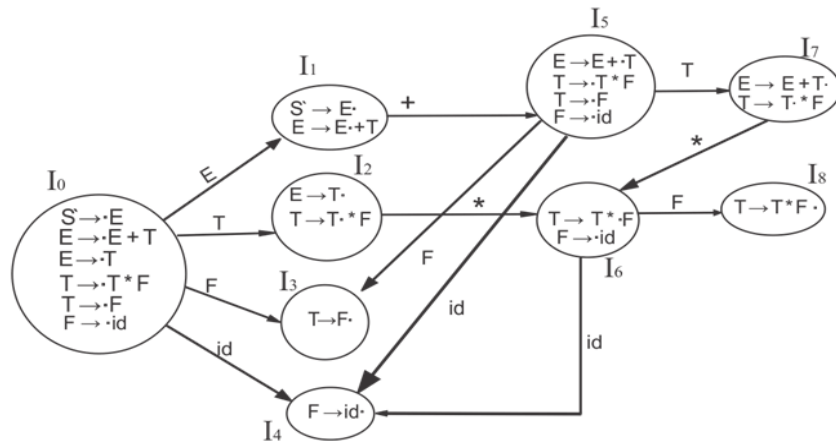
$F \rightarrow \bullet id$

❖ Go to (I6, id) = Closure ($F \rightarrow id \bullet$) = (same as I4)

I7 = Go to (I5, T) = Closure ($E \rightarrow E + T \bullet$) = $E \rightarrow E + T \bullet$

I8 = Go to (I6, F) = Closure ($T \rightarrow T * F \bullet$) = $T \rightarrow T * F \bullet$

Drawing DFA:



SLR (1) Table

States	Action				Go to		
	id	+	*	\$	E	T	F
I ₀	S ₄				1	2	3
I ₁		S ₅		Accept			
I ₂		R ₂	S ₆	R ₂			
I ₃		R ₄	R ₄	R ₄			
I ₄		R ₅	R ₅	R ₅			
I ₅	S ₄					7	3
I ₆	S ₄						8
I ₇		R ₁	S ₆	R ₁			
I ₈		R ₃	R ₃	R ₃			

Explanation:

First (E) = First (E + T) \cup First (T)

First (T) = First (T * F) \cup First (F)

First (F) = {id}

First (T) = {id}

First (E) = {id}

Follow (E) = First (+T) \cup {\$} = {+, \$}

Follow (T) = First (*F) \cup First (F)

= {*, +, \$}

Follow (F) = {*, +, \$}

- I1 contains the final item which drives $S \rightarrow E \bullet$ and follow (S) = { $\$$ }, so action {I1, $\$$ } = Accept
 - I2 contains the final item which drives $E \rightarrow T \bullet$ and follow (E) = { $+$, $\$$ }, so action {I2, $+$ } = R2, action {I2, $\$$ } = R2
 - I3 contains the final item which drives $T \rightarrow F \bullet$ and follow (T) = { $+$, $*$, $\$$ }, so action {I3, $+$ } = R4, action {I3, $*$ } = R4, action {I3, $\$$ } = R4
 - I4 contains the final item which drives $F \rightarrow id \bullet$ and follow (F) = { $+$, $*$, $\$$ }, so action {I4, $+$ } = R5, action {I4, $*$ } = R5, action {I4, $\$$ } = R5
 - I7 contains the final item which drives $E \rightarrow E + T \bullet$ and follow (E) = { $+$, $\$$ }, so action {I7, $+$ } = R1, action {I7, $\$$ } = R1
 - I8 contains the final item which drives $T \rightarrow T * F \bullet$ and follow (T) = { $+$, $*$, $\$$ }, so action {I8, $+$ } = R3, action {I8, $*$ } = R3, action {I8, $\$$ } = R3.
-

LALR PARSING

MOTIVATION

- The LALR (Look Ahead-LR) parsing method is between SLR and Canonical LR both in terms of power of parsing grammars and ease of implementation.
- This method is often used in practice because the tables obtained by it are considerably smaller than the Canonical LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar.
- The same is almost true for SLR grammars, but there are a few constructs that can not be handled by SLR techniques.

CONSTRUCTING LALR PARSING TABLES

- **CORE:** A core is a set of LR (0) (SLR) items for the grammar, and an LR (1) (Canonical LR) grammar may produce more than two sets of items with the same core.
- The core does not contain any look ahead information.

Example: Let s1 and s2 are two states in a Canonical LR grammar.

$$S1 - \{C \rightarrow c.C, c/d; C \rightarrow .cC, c/d; C \rightarrow .d, c/d\}$$

$$S1 - \{C \rightarrow c.C, \$; C \rightarrow .cC, \$; C \rightarrow .d, \$\}$$

- These two states have the same core consisting of only the production rules without any look ahead information.

CONSTRUCTION IDEA:

1. Construct the set of LR (1) items.
2. Merge the sets with common core together as one set, if no conflict (shift-shift or shift-reduce) arises.
3. If a conflict arises it implies that the grammar is not LALR.
4. The parsing table is constructed from the collection of merged sets of items using the same algorithm for LR (1) parsing.

ALGORITHM:

Input: An augmented grammar G' .

Output: The LALR parsing table actions and goto for G' .

Method:

1. Construct $C = \{I_0, I_1, I_2, \dots, I_n\}$, the collection of sets of LR(1) items.
2. For each core present in among these sets, find all sets having the core, and replace these sets by their union.
3. Parsing action table is constructed as for Canonical LR.
4. The goto table is constructed by taking the union of all sets of items having the same core. If J is the union of one or more sets of LR (1) items, that is, $J = I_1 \cup I_2 \cup \dots \cup I_k$, then the cores of $\text{goto}(I_1, X)$, $\text{goto}(I_2, X)$, ..., $\text{goto}(I_k, X)$ are the same as all of them have same core. Let K be the union of all sets of items having same core as $\text{goto}(I_1, X)$. Then $\text{goto}(J, X) = K$.

EXAMPLE

GRAMMAR:

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

STATES:

- $I_0 : S' \rightarrow .S, \$$
 $S \rightarrow .CC, \$$
 $C \rightarrow .cC, c/d$
 $C \rightarrow .d, c/d$
- $I_1 : S' \rightarrow S., \$$
- $I_2 : S \rightarrow C.C, \$$
 $C \rightarrow .Cc, \$$

C -> .d, \$

- I3: C -> c. C, c /d

C -> .Cc, c /d

C -> .d, c /d

- I4: C -> d., c /d

- I5: S -> CC., \$

- I6: C -> c.C, \$

C -> .cC, \$

C -> .d, \$

- I7: C -> d., \$

- I8: C -> cC., c /d

- I9: C -> cC., \$

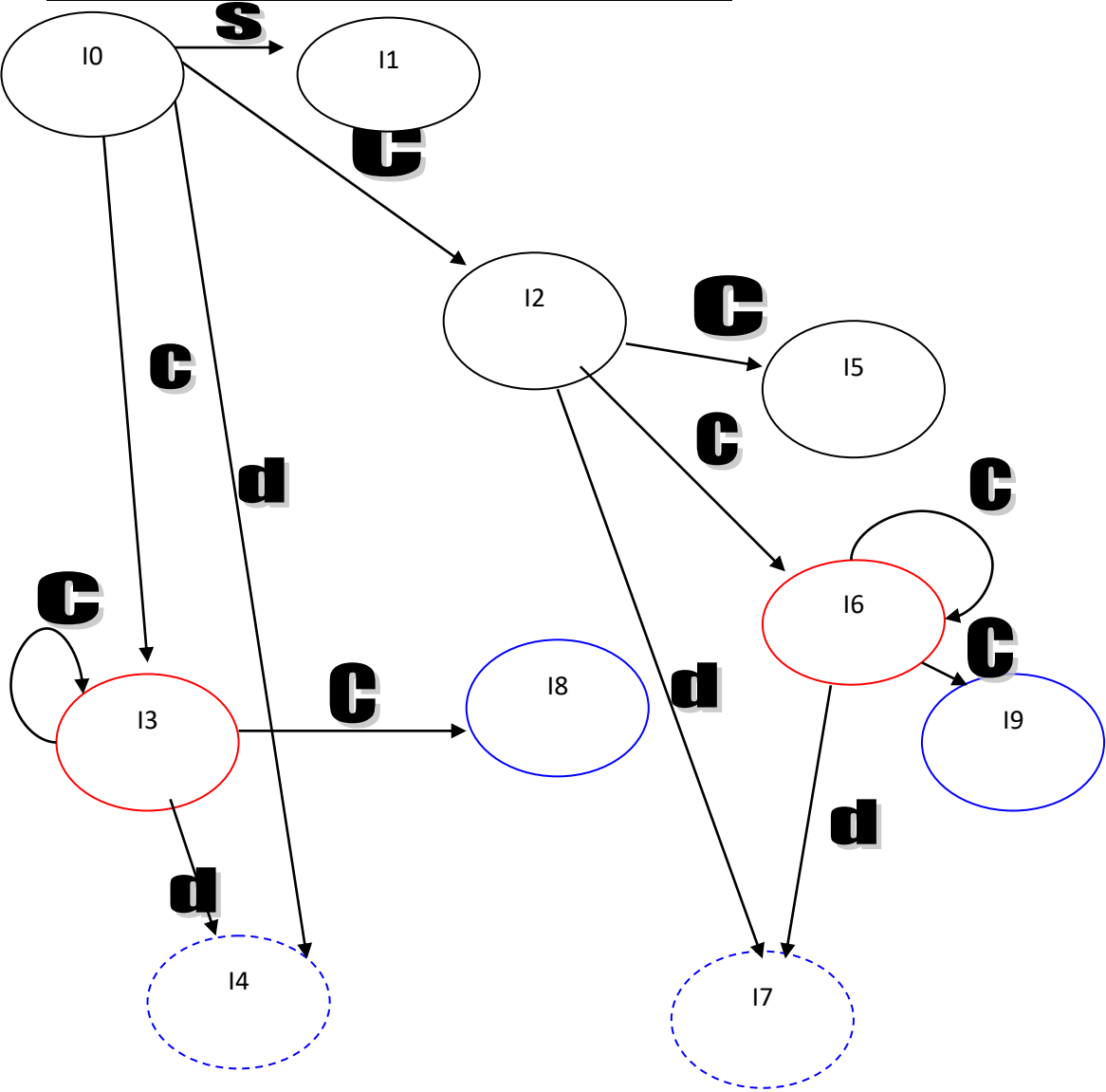
CANONICAL PARSING TABLE:

STATE	Actions			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		

NOTE: For goto graph see the construction used in Canonical LR.

LALR PARSING TABLE:

START	Actions			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		



- Showing states with same core with same colour which get merged in conversion from LR(1) to LALR.
- States merged together: 3 and 6

4 and 7

8 and 9

UNIT III

CHAPTER : 7

- ❖ Syntax Directed Translation Schemes
 - ❖ Implementation Of Syntax
 - ❖ Directed Translators
 - ❖ Intermediate Code
 - ❖ Postfix Notation
 - ❖ Parse Tree And Syntax Trees
 - ❖ Three Address Code,Quadruples And Triples
 - ❖ Translation Of Assignment Statements
 - ❖ Boolean Expressions
 - ❖ Statements That Alter The Flow Of Control
 - ❖ Postfix Translations
 - ❖ Translation With Top Down Parser
-

CHAPTER 7

Syntax directed translation

- ❖ In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules.
- ❖ So we can say that

Grammar + semantic rule = SDT (syntax directed translation)

- ❖ In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
- ❖ In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
- ❖ In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Example

Production	Semantic Rules
$E \rightarrow E + T$	$E.val := E.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T * F$	$T.val := T.val + F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (F)$	$F.val := F.val$
$F \rightarrow \text{num}$	$F.val := \text{num.lexval}$

E.val is one of the attributes of E.

Syntax directed translation scheme

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example

Production	Semantic Rules
$S \rightarrow E \$$	{ printE.VAL }
$E \rightarrow E + E$	{E.VAL := E.VAL + E.VAL }
$E \rightarrow E * E$	{E.VAL := E.VAL * E.VAL }
$E \rightarrow (E)$	{E.VAL := E.VAL }
$E \rightarrow I$	{E.VAL := I.VAL }
$I \rightarrow I \text{ digit}$	{I.VAL := 10 * I.VAL + LEXVAL }
$I \rightarrow \text{digit}$	{ I.VAL:= LEXVAL }

Implementation of Syntax directed translation

- ❖ Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.
- ❖ SDT is implementing by parse the input and produce a parse tree as a result.

Example

Parse tree for SDT:

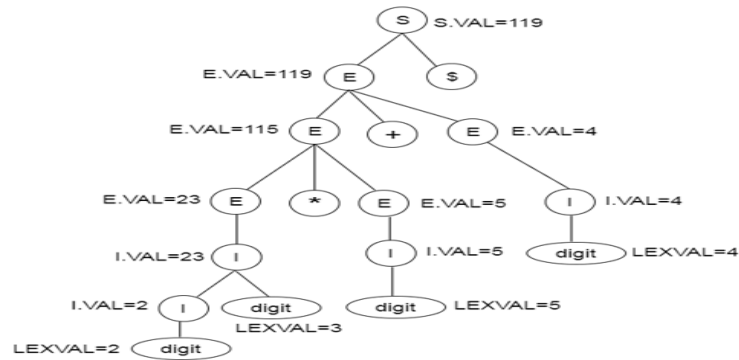


Fig: Parse tree

Syntax Directed Translation

Background :

- ❖ Parser uses a CFG(Context-free-Grammer) to validate the input string and produce output for next phase of the compiler. Output could be either a parse tree or abstract syntax tree.
- ❖ Now to interleave semantic analysis with syntax analysis phase of the compiler, we use Syntax Directed Translation.
- ❖ **Definition**
Syntax Directed Translation are augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes.
- ❖ Syntax directed translation rules use 1) lexical values of nodes, 2) constants & 3) attributes associated to the non-terminals in their definitions.
- ❖ The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order.
- ❖ In many cases, translation can be done during parsing without building an explicit tree.

Example

$E \rightarrow E+T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow \text{INT} \mid \text{LIT}$

- ❖ This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any. In this example we will focus on evaluation of the given expression, as we don't have any semantic assertions to check in this very basic example.

$E \rightarrow E+T \quad \{ E.val = E.val + T.val \} \quad PR\#1$

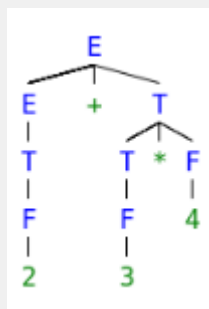
$E \rightarrow T \quad \{ E.val = T.val \} \quad PR\#2$

$T \rightarrow T * F \quad \{ T.val = T.val * F.val \} \quad PR\#3$

$T \rightarrow F \quad \{ T.val = F.val \} \quad PR\#4$

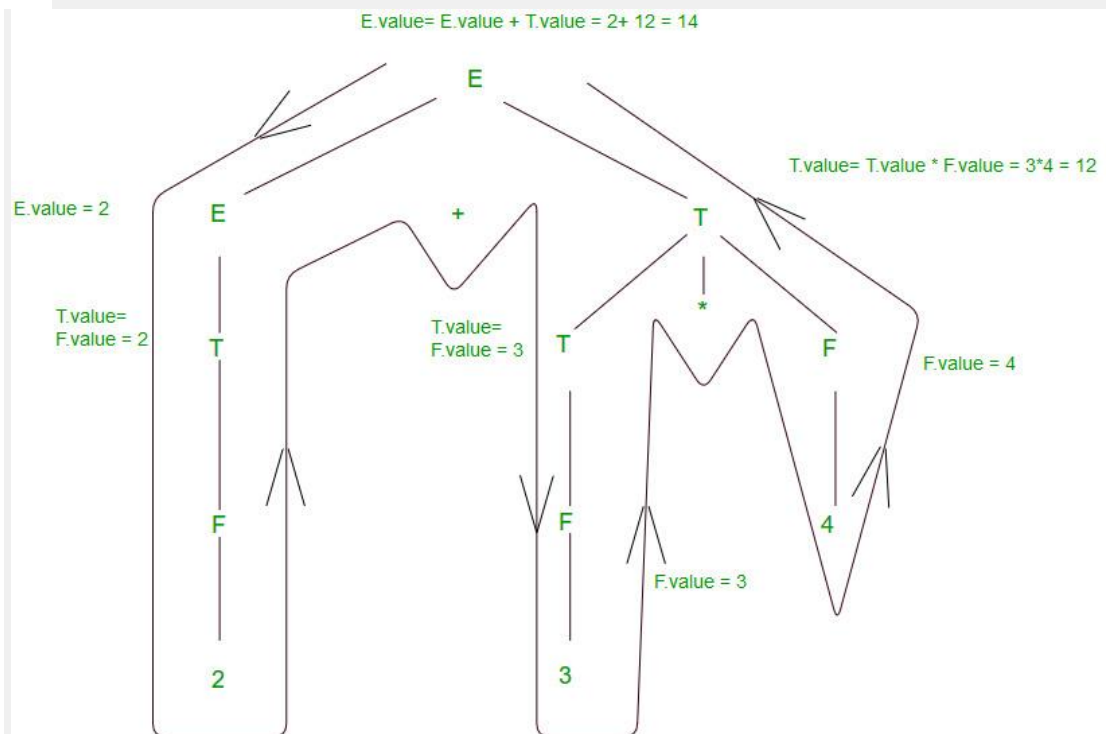
$F \rightarrow INTLIT \quad \{ F.val = INTLIT.lexval \} \quad PR\#5$

- ❖ For understanding translation rules further, we take the first SDT augmented to $[E \rightarrow E+T]$ production rule. The translation rule in consideration has val as attribute for both the non-terminals – E & T.
- ❖ Right hand side of the translation rule corresponds to attribute values of right side nodes of the production rule and vice-versa. Generalizing, SDT are augmented rules to a CFG that associate 1) set of attributes to every node of the grammar and 2) set of translation rules to every production rule using attributes, constants and lexical values.
- ❖ Let's take a string to see how semantic analysis happens – $S = 2+3*4$. Parse tree corresponding to S would be



- ❖ To evaluate translation rules, we can employ one depth first search traversal on the parse tree. This is possible only because SDT rules don't impose any specific order on evaluation until children attributes are computed before parents for a grammar having all synthesized attributes.

- ❖ Otherwise, we would have to figure out the best suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals. For better understanding, we will move bottom up in left to right fashion for computing translation rules of our example.



- ❖ Above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children attributes are computed before parents, as discussed above.
- ❖ Right hand side nodes are sometimes annotated with subscript 1 to distinguish between children and parent.

Additional Information

- ❖ **Synthesized Attributes** are such attributes that depend only on the attribute values of children nodes.

Thus $[E \rightarrow E+T \{ E.val = E.val + T.val \}]$ has a synthesized attribute val corresponding to node E . If all the semantic attributes in an augmented grammar are synthesized, one depth first search traversal in any order is sufficient for semantic analysis phase.

- ❖ **Inherited Attributes** are such attributes that depend on parent and/or siblings attributes.

Thus $[E_p \rightarrow E+T \{ E_p.val = E.val + T.val, T.val = E_p.val \}]$, where E & E_p are

same production symbols annotated to differentiate between parent and child, has an inherited attribute val corresponding to node T.

:

Intermediate code

- ❖ Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.

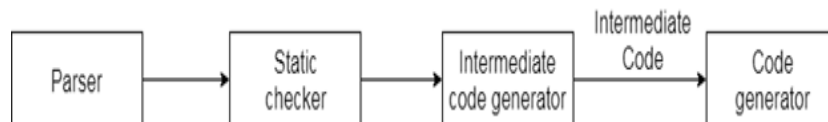


Fig: Position of intermediate code generator

- If the compiler directly translates source code into the machine code without generating intermediate code then a full native compiler is required for each new machine.
- The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine.
- Intermediate code generator receives input from its predecessor phase and semantic analyzer phase. It takes input in the form of an annotated syntax tree.
- Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.

Intermediate representation

- ❖ Intermediate code can be represented in two ways:

1. High Level intermediate code:

- ❖ **HH**High level intermediate code can be represented as source code. To enhance performance of source code, we can easily apply code modification. But to optimize the target machine, it is less preferred.

2. Low Level intermediate code

- ❖ Low level intermediate code is close to the target machine, which makes it suitable for register and memory allocation etc. it is used for machine-dependent optimizations.

Postfix Notation

- Postfix notation is the useful form of intermediate code if the given language is expressions.
- ❖ Postfix notation is also called as 'suffix notation' and 'reverse polish'.
- ❖ Postfix notation is a linear representation of a syntax tree.
- ❖ In the postfix notation, any expression can be written unambiguously without parentheses.
- ❖ The ordinary (infix) way of writing the sum of x and y is with operator in the middle: $x * y$. But in the postfix notation, we place the operator at the right end as $xy *$.
- ❖ In postfix notation, the operator follows the operand.

Example

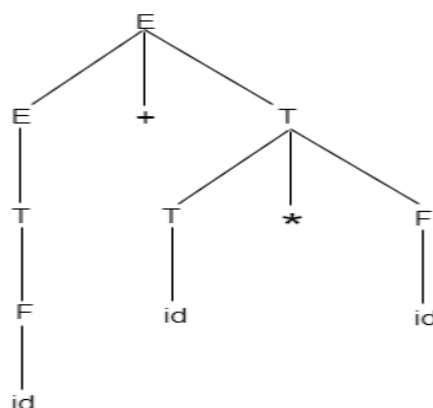
Production

- $E \rightarrow E1 \text{ op } E2$
- $E \rightarrow (E1)$
- $E \rightarrow \text{id}$

Semantic Rule	Program fragment
$E.\text{code} = E1.\text{code} \parallel E2.\text{code} \parallel \text{op}$	print op
$E.\text{code} = E1.\text{code}$	
$E.\text{code} = \text{id}$	print id

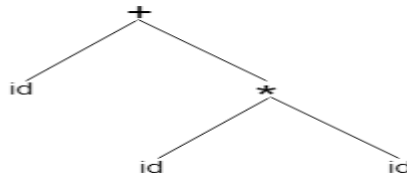
Parse tree and Syntax tree

- ❖ When you create a parse tree then it contains more details than actually needed. So, it is very difficult to compiler to parse the parse tree. Take the following parse tree as an example:

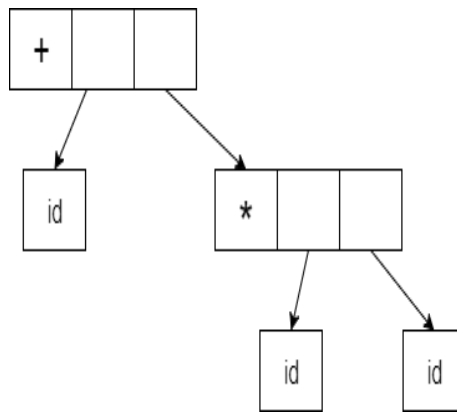


- ❖ In the parse tree, most of the leaf nodes are single child to their parent nodes.
- ❖ In the syntax tree, we can eliminate this extra information.
- ❖ Syntax tree is a variant of parse tree. In the syntax tree, interior nodes are operators and leaves are operands.
- ❖ Syntax tree is usually used when represent a program in a tree structure.

A sentence **id + id * id** would have the following syntax tree:



Abstract syntax tree can be represented as:



- ❖ Abstract syntax trees are important data structures in a compiler. It contains the least unnecessary information.
- ❖ Abstract syntax trees are more compact than a parse tree and can be easily used by a compiler.

Three address code

- ❖ Three-address code is an intermediate code. It is used by the optimizing compilers.
- ❖ In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- ❖ Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

Example

Given Expression:

$$a := (-c * b) + (-c * d)$$

Three-address code is as follows:

$$t_1 := -c$$

$$t_2 := b * t_1$$

$$t_3 := -c$$

$$t_4 := d * t_3$$

$$t_5 := t_2 + t_4$$

$$a := t_5$$

- ❖ **t** is used as registers in the target program.
- ❖ The three address code can be represented in two forms: **quadruples** and **triples**.

Quadruples

- ❖ The quadruples have four fields to implement the three address code. The field of quadruples contains the name of the operator, the first source operand, the second source operand and the result respectively.

Operator
Source 1
Source 2
Destination

Fig: Quadruples field

Example

- $a := -b * c + d$
- Three-address code is as follows:
 - $t_1 := -b$
 - $t_2 := c + d$
- $t_3 := t_1 * t_2$
- $a := t_3$

These statements are represented by quadruples as follows:

	Operator	Source 1	Source 2	Destination
(0)	uminus	b	-	t ₁
(1)	+	c	d	t ₂
(2)	*	t ₁	t ₂	t ₃
(3)	:=	t ₃	-	a

Triples

- ❖ The triples have three fields to implement the three address code. The field of triples contains the name of the operator, the first source operand and the second source operand.
- ❖ In triples, the results of respective sub-expressions are denoted by the position of expression. Triple is equivalent to DAG while representing expressions.

Operator
Source 1
Source 2

Fig: Triples field

Example:

a := -b * c + d

Three address code is as follows:

t₁ := -b t₂ := c + d t₃ := t₁ * t₂ a := t₃

These statements are represented by triples as follows:

	Operator	Source 1	Source 2
(0)	uminus	B	-
(1)	+	C	d
(2)	*	(0)	(1)
(3)	:=	(2)	-

Translation of Assignment Statements

- ❖ In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.

Consider the grammar

- $S \rightarrow id := E$
- $E \rightarrow E1 + E2$
- $E \rightarrow E1 * E2$
- $E \rightarrow (E1)$
- $E \rightarrow id$

The translation scheme of above grammar is given below:

Production rule	Semantic actions
$S \rightarrow id := E$	<pre>{ p = look_up(id.name); If p ≠ nil then Emit (p = E.place) Else Error; }</pre>
$E \rightarrow E1 + E2$	<pre>{ E.place = newtemp(); Emit (E.place = E1.place '+' E2.place) }</pre>
$E \rightarrow E1 * E2$	<pre>{ E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) }</pre>
$E \rightarrow (E1)$	<pre>{ E.place = E1.place }</pre>
$E \rightarrow id$	<pre>{ p = look_up(id.name); If p ≠ nil then Emit (p = E.place) Else Error; }</pre>

- The p returns the entry for id.name in the symbol table.

Boolean expressions

- ❖ Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

Consider the grammar

$E \rightarrow E \text{ OR } E$
 $E \rightarrow E \text{ AND } E$
 $E \rightarrow \text{NOT } E$
 $E \rightarrow (E)$
 $E \rightarrow \text{id relop id}$
 $E \rightarrow \text{TRUE}$
 $E \rightarrow \text{FALSE}$

- The relop is denoted by <, >, <=, >=.
- The AND and OR are left associated. NOT has the higher precedence then AND and lastly OR.

Production rule	Semantic actions
$E \rightarrow E1 \text{ OR } E2$	<pre>{E.place = newtemp(); Emit (E.place ':=' E1.place 'OR' E2.place) }</pre>
$E \rightarrow E1 \text{ AND } E2$	<pre>{E.place = newtemp(); Emit (E.place ':=' E1.place 'AND' E2.place) }</pre>
$E \rightarrow \text{NOT } E1$	<pre>{E.place = newtemp(); Emit (E.place ':=' 'NOT' E1.place) }</pre>
$E \rightarrow (E1)$	<pre>{E.place = E1.place}</pre>
$E \rightarrow \text{id relop id2}$	<pre>{E.place = newtemp(); Emit ('if' id1.place relop.op id2.place 'goto' nextstar + 3); EMIT (E.place ':=' '0') EMIT ('goto' nextstat + 2)</pre>

	<pre> EMIT (E.place ':=' '1') </pre>
$E \rightarrow \text{TRUE}$	<pre> {E.place := newtemp(); Emit (E.place ':=' '1') } </pre>
$E \rightarrow \text{FALSE}$	<pre> {E.place := newtemp(); Emit (E.place ':=' '0') } </pre>

- ❖ The EMIT function is used to generate the three address code and the newtemp() function is used to generate the temporary variables.
- ❖ The $E \rightarrow \text{id rel op id2}$ contains the next_state and it gives the index of next three address statements in the output sequence.
- ❖ Here is the example which generates the three address code using the above translation scheme:

- $p > q \text{ AND } r < s \text{ OR } u > r$
 - 100: **if** $p > q$ **goto** 103
 - 101: $t1 := 0$
 - 102: **goto** 104
 - 103: $t1 := 1$
 - 104: **if** $r > s$ **goto** 107
 - 105: $t2 := 0$
 - 106: **goto** 108
 - 107: $t2 := 1$
 - 108: **if** $u > v$ **goto** 111
 - 109: $t3 := 0$
 - 110: **goto** 112
 - 111: $t3 := 1$
 - 112: $t4 := t1 \text{ AND } t2$
 - 113: $t5 := t4 \text{ OR } t3$
-

Statements that alter the flow of control

- ❖ The goto statement alters the flow of control. If we implement goto statements then we need to define a LABEL for a statement. A production can be added for this purpose:

- $S \rightarrow \text{LABEL} : S$
- $\text{LABEL} \rightarrow \text{id}$

- ❖ In this production system, semantic action is attached to record the LABEL and its value in the symbol table.
- ❖ Following grammar used to incorporate structure flow-of-control constructs:

- $S \rightarrow \text{if } E \text{ then } S$
- $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
- $S \rightarrow \text{while } E \text{ do } S$
- $S \rightarrow \text{begin } L \text{ end}$
- $S \rightarrow A$
- $L \rightarrow L ; S$
- $L \rightarrow S$

- ❖ Here, S is a statement, L is a statement-list, A is an assignment statement and E is a Boolean-valued expression.

Translation scheme for statement that alters flow of control

- We introduce the marker non-terminal M as in case of grammar for Boolean expression.
- This M is put before statement in both if then else. In case of while-do, we need to put M before E as we need to come back to it after executing S.
- In case of if-then-else, if we evaluate E to be true, first S will be executed.
- After this we should ensure that instead of second S, the code after the if-then else will be executed. Then we place another non-terminal marker N after first S.

The grammar is as follows:

- $S \rightarrow \text{if } E \text{ then } M S$
- $S \rightarrow \text{if } E \text{ then } M S \text{ else } M S$
- $S \rightarrow \text{while } M E \text{ do } M S$
- $S \rightarrow \text{begin } L \text{ end}$
- $S \rightarrow A$

- $L \rightarrow L ; M S$
- $L \rightarrow S$
- $M \rightarrow \epsilon$
- $N \rightarrow \epsilon$

The translation scheme for this grammar is as follows:

Production rule	Semantic actions
$S \rightarrow \text{if } E \text{ then } M S_1$	BACKPATCH (E.TRUE, M.QUAD) S.NEXT = MERGE (E.FALSE, S1.NEXT)
$S \rightarrow \text{if } E \text{ then } M_1 S_1 \text{ else } M_2 S_2$	BACKPATCH (E.TRUE, M1.QUAD) BACKPATCH (E.FALSE, M2.QUAD) S.NEXT = MERGE (S1.NEXT, N.NEXT, S2.NEXT)
$S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$	BACKPATCH (S1.NEXT, M1.QUAD) BACKPATCH (E.TRUE, M2.QUAD) S.NEXT = E.FALSE GEN (goto M1.QUAD)
$S \rightarrow \text{begin } L \text{ end}$	S.NEXT = L.NEXT
$S \rightarrow A$	S.NEXT = MAKELIST ()
$L \rightarrow L ; M S$	BACKPATHCH (L1.NEXT, M.QUAD) L.NEXT = S.NEXT
$L \rightarrow S$	L.NEXT = S.NEXT
$M \rightarrow \epsilon$	M.QUAD = NEXTQUAD
$N \rightarrow \epsilon$	N.NEXT = MAKELIST (NEXTQUAD) GEN (goto_)

Postfix Translation

- ❖ In a production $A \rightarrow \alpha$, the translation rule of $A.CODE$ consists of the concatenation of the CODE translations of the non-terminals in α in the same order as the non-terminals appear in α .
- ❖ Production can be factored to achieve postfix form.

Postfix translation of while statement

The production

- $S \rightarrow \text{while } M1 \text{ E } \text{do } M2 \text{ S1}$

Can be factored as:

- $S \rightarrow C \text{ S1}$
- $C \rightarrow W \text{ E } \text{do}$
- $W \rightarrow \text{while}$

Production Rule	Semantic Action
$W \rightarrow \text{while}$	$W.QUAD = \text{NEXTQUAD}$
$C \rightarrow W \text{ E } \text{do}$	$C \text{ W E do}$
$S \rightarrow C \text{ S1}$	$\text{BACKPATCH}(S1.NEXT, C.QUAD)$ $S.NEXT = C.FALSE$ $\text{GEN}(\text{goto } C.QUAD)$

A suitable transition scheme would be

Postfix translation of for statement

The production

- $S \text{ for } L = E1 \text{ step } E2 \text{ to } E3 \text{ do } S1$

Can be factored as

- $F \rightarrow \text{for } L$
- $T \rightarrow F = E1 \text{ by } E2 \text{ to } E3 \text{ do}$
- $S \rightarrow T \text{ S1}$

- The Emit function is used for appending the three address code to the output file. Otherwise it will report an error.
- The newtemp() is a function used to generate new temporary variables.
- E.place holds the value of E.

Translation With A Top Down Parser

- Any translation done by top down parser can be done in a bottom up parser also.
- But in certain situations, translation with top down parser is advantageous as risks. such as placing a marker non terminal can be avoided
- Semantic routines can be called in the middle of productions in top down parser. So the location of $a[i]$ can be computed at the run time by evaluating the formula $i * \text{width} + c$ where c is (base A low * width) which is evaluated at compile time.
- Intermediate code generator should produce the code to evaluate this formula $i * \text{width} + c$ (one multiplication and one addition operation)
- A two dimensional array can be stored in either row –major(row by row) or column major(column by column)
- Most of the programming languages use row major based method.
- The location of $A[i_1, i_2]$ is based $A + (i_1 - \text{row}) * n_2 + i_2 - \text{low}_2) * \text{width}$.

UNIT IV

CHAPTER :8 - Symbol Table

- ❖ The content of the symbol table
- ❖ Data structures for symbol tables
- ❖ Representing scope information

CHAPTER :9 -Run Time Storage Administration

- ❖ Implementation of simple stack allocation scheme
- ❖ Implementation of block structured languages
- ❖ Storage allocation in block structured language

CHAPTER :10 -Error Detection And Recovery

- ❖ Errors
 - ❖ Lexical Phase errors
 - ❖ Syntactic phase errors
 - ❖ Semantic errors.
-

CHAPTER 8

The contents of a symbol table

Symbol Table

- Symbol table is an important data structure used in a compiler.
- Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases.

Symbol Table entries – Each entry in symbol table is associated with attributes that support compiler in different phases.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information used by compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Operations of Symbol table – The basic operations defined on a symbol table include:

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

The symbol table used for following purposes:

- It is used to store the name of all entities in a structured form at one place.
- It is used to verify if a variable has been declared.
- It is used to determine the scope of a name.
- It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.
- A symbol table can either be linear or a hash table. Using the following format, it maintains the entry for each name.

1. <symbol name, type, attribute>

- For example, suppose a variable store the information about the following variable declaration:
- **static int** salary
- then, it stores an entry in the following format:
- <salary, **int**, **static**>
- The clause attribute contains the entries related to the name.

Implementation

- ❖ The symbol table can be implemented in the unordered list if the compiler is used to handle the small amount of data.

- ❖ A symbol table can be implemented in one of the following techniques:
 - Linear (sorted or unsorted) list
 - Hash table
 - Binary search tree
- ❖ Symbol table are mostly implemented as hash table.

Operations

The symbol table provides the following operations:

Insert ()

- Insert () operation is more frequently used in the analysis phase when the tokens are identified and names are stored in the table.
- The insert() operation is used to insert the information in the symbol table like the unique name occurring in the source code.
- In the source code, the attribute for a symbol is the information associated with that symbol. The information contains the state, value, type and scope about the symbol.
- The insert () function takes the symbol and its value in the form of argument.

For example:

`int x;`

Should be processed by the compiler as:

`insert (x, int)`

lookup()

In the symbol table, lookup() operation is used to search a name. It is used to determine:

- The existence of symbol in the table.
- The declaration of the symbol before it is used.
- Check whether the name is used in the scope.
- Initialization of the symbol.
- Checking whether the name is declared multiple times.

The basic format of lookup() function is as follows:

- lookup (symbol)
- This format is varies according to the programming language

Activation Record

- Control stack is a run time stack which is used to keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed.

- When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped.
- Activation record is used to manage the information needed by a single execution of a procedure.
- An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

The diagram below shows the contents of activation records:

Return value
Actual Parameters
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries

Return Value: It is used by calling procedure to return a value to calling procedure.

Actual Parameter: It is used by calling procedures to supply parameters to the called procedures.

Control Link: It points to activation record of the caller.

Access Link: It is used to refer to non-local data held in other activation records.

Saved Machine Status: It holds the information about status of machine before the procedure is called.

Local Data: It holds the data that is local to the execution of the procedure.

Temporaries: It stores the value that arises in the evaluation of an expression.

Data structure for symbol table

Following are commonly used data structure for implementing symbol table :-

1. **List** –

- In this method, an array is used to store names and associated information.
- A pointer “**available**” is maintained at end of all stored records and new names are added in the order as they arrive
- To search for a name we start from beginning of list till available pointer and if not found we get an error “**use of undeclared name**”
- While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. “**Multiple defined name**”
- Insertion is fast $O(1)$, but lookup is slow for large tables – $O(n)$ on average
- Advantage is that it takes minimum amount of space.

2. **Linked List** –

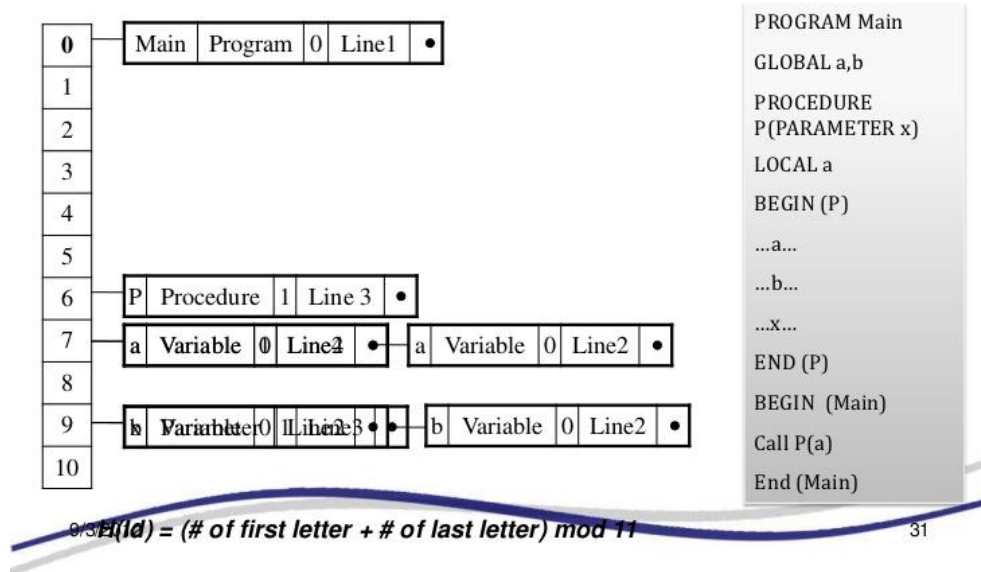
- This implementation is using linked list. A link field is added to each record.
- Searching of names is done in order pointed by link of link field.
- A pointer “**First**” is maintained to point to first record of symbol table.
- Insertion is fast $O(1)$, but lookup is slow for large tables – $O(n)$ on average

3. **Hash Table** –

- In hashing scheme two tables are maintained – a hash table and symbol table and is the most commonly used method to implement symbol tables..
- A hash table is an array with index range: 0 to $\text{tablesize} - 1$. These entries are pointer pointing to names of symbol table.
- To search for a name we use hash function that will result in any integer between 0 to $\text{tablesize} - 1$.
- Insertion and lookup can be made very fast – $O(1)$.
- Advantage is quick search is possible and disadvantage is that hashing is complicated to implement.

HASH TABLE - EXAMPLE

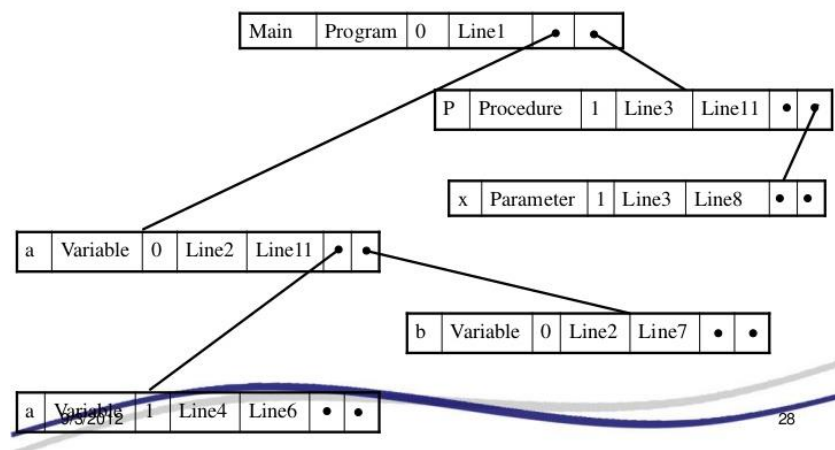
M	n	a	b	P	x
77	110	97	98	80	120



4. Binary Search Tree –

- Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.
- All names are created as child of root node that always follow the property of binary search tree.
- Insertion and lookup are $O(\log_2 n)$ on average.

BINARY TREE



Representing Scope Information

In the source program, every name possesses a region of validity, called the scope of that name.

The rules in a block-structured language are as follows:

1. If a name declared within block B then it will be valid only within B.

2. If B1 block is nested within B2 then the name that is valid for block B2 is also valid for B1 unless the name's identifier is re-declared in B1.

- These scope rules need a more complicated organization of symbol table than a list of associations between names and attributes.
- Tables are organized into stack and each table contains the list of names and their associated attributes.
- Whenever a new block is entered then a new table is entered onto the stack. The new table holds the name that is declared as local to this block.
- When the declaration is compiled then the table is searched for a name.
- If the name is not found in the table then the new name is inserted.
- When the name's reference is translated then each table is searched, starting from the each table on the stack.

For example:

```
int x;  
void f(int m) {  
    float x, y;  
    {  
        int i, j;  
        int u, v;  
    }  
}  
int g(int n)  
{  
    bool t;  
}
```

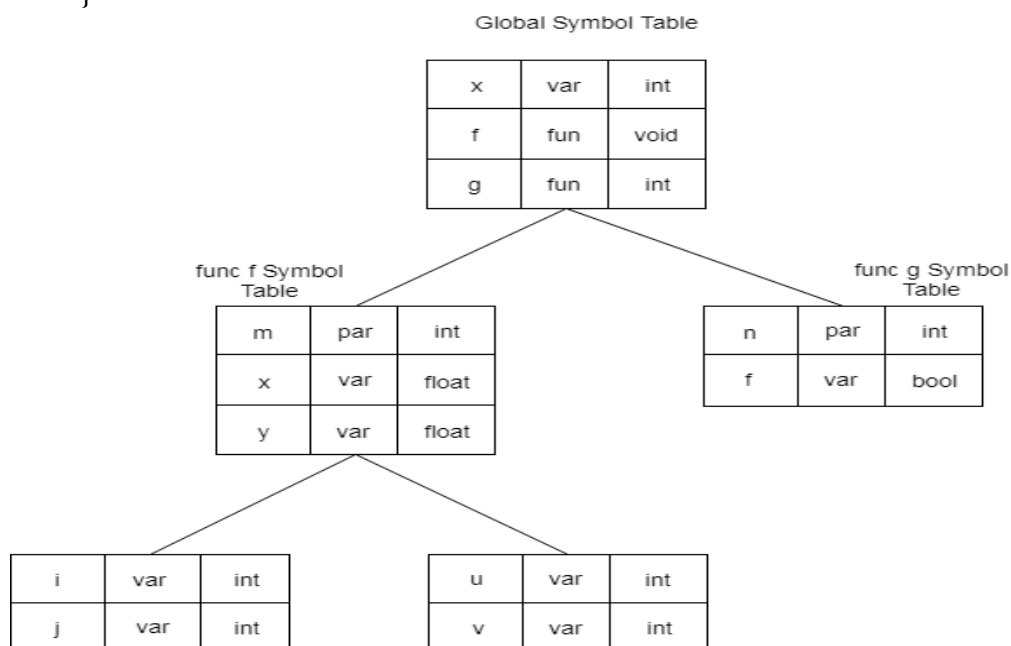


Fig: Symbol table organization that complies with static scope information rules

CHAPTER 9:

RUN-TIME STORAGE MANAGEMENT

- ❖ The information which required during an execution of a procedure is kept in a block of storage called an activation record. The activation record includes storage for names local to the procedure.
 - ❖ We can describe address in the target code using the following ways:
 - Static allocation
 - Stack allocation
 - ❖ In static allocation, the position of an activation record is fixed in memory at compile time.
 - ❖ In the stack allocation, for each execution of a procedure a new activation record is pushed onto the stack. When the activation ends then the record is popped.
 - ❖ For the run-time allocation and deallocation of activation records the following three-address statements are associated:
 - Call
 - Return
 - Halt
 - 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
-

Implementation Of A Simple Stack Allocation:

STACK ALLOCATION OF SPACE

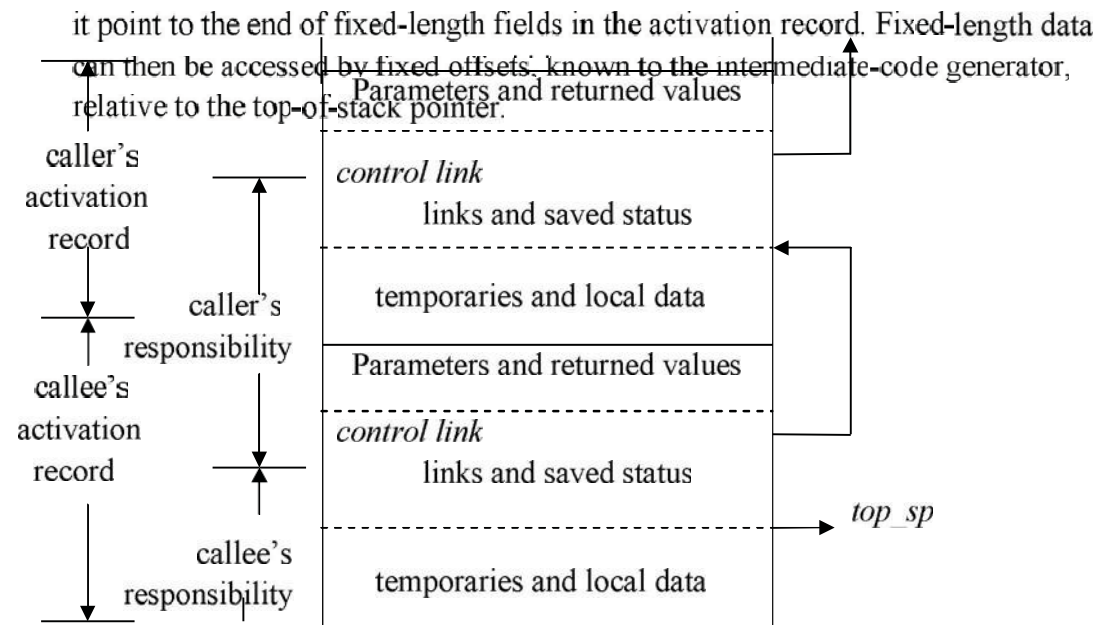
- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is 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).
- When designing calling sequences and the layout of activation records, the following

principles are helpful:

- Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.
- the control link, the access link, and the machine status fields.
- Fixed length items are generally placed in the middle. Such items typically include
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to have

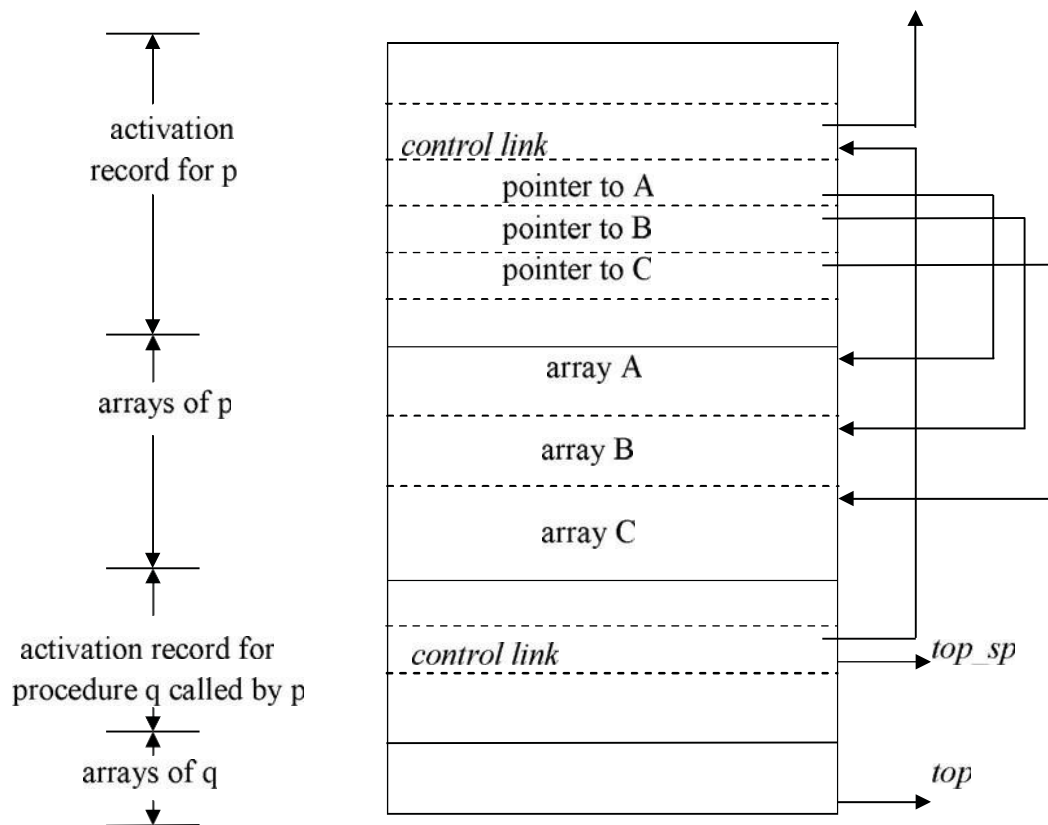


Division of tasks between caller and callee

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

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.



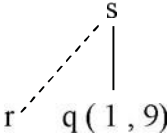
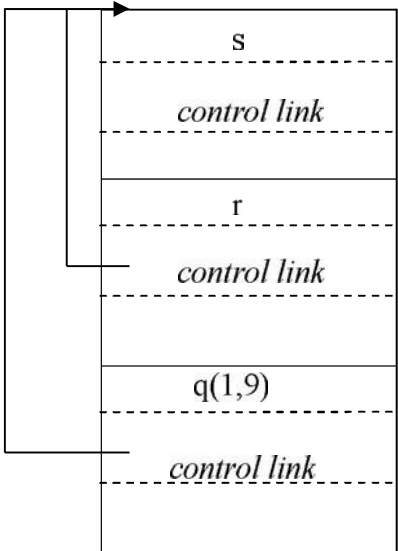
Access to dynamically allocated arrays

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

HEAP ALLOCATION

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

1. The values of local names must be retained when an activation ends.
 2. A called activation outlives the caller.
- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
 - Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

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

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

Implementation of block structured languages

- ❖ Basic block contains a sequence of statement. The flow of control enters at the beginning of the statement and leave at the end without any halt (except may be the last instruction of the block).
- ❖ The following sequence of three address statements forms a basic block:

- ❖ $t1 := x * x$
- ❖ $t2 := x * y$
- ❖ $t3 := 2 * t2$
- ❖ $t4 := t1 + t3$
- ❖ $t5 := y * y$
- ❖ $t6 := t4 + t5$

Basic block construction:

Algorithm: Partition into basic blocks

Input: It contains the sequence of three address statements

Output: it contains a list of basic blocks with each three address statement in exactly one block

Method: First identify the leader in the code. The rules for finding leaders are as follows:

- The first statement is a leader.
- Statement L is a leader if there is an conditional or unconditional goto statement like:
if....goto L or goto L
- Instruction L is a leader if it immediately follows a goto or conditional goto statement like: if goto B or goto B
- ❖ For each leader, its basic block consists of the leader and all statement up to. It doesn't include the next leader or end of the program.
- ❖ Consider the following source code for dot product of two vectors a and b of length 10:

code for the above source program is given below:

B1

(1) prod := 0

(2) $i := 1$

B2

1. (3) $t1 := 4 * i$
2. (4) $t2 := a[t1]$
3. (5) $t3 := 4 * i$
4. (6) $t4 := b[t3]$
5. (7) $t5 := t2 * t4$
6. (8) $t6 := \text{prod} + t5$
7. (9) $\text{prod} := t6$
8. (10) $t7 := i + 1$
9. (11) $i := t7$
10. (12) if $i \leq 10$ goto (3)

❖ Basic block B1 contains the statement (1) to (2)

1. Basic block begin
2. $\text{prod} := 0;$
3. $i := 1;$
4. do begin
5. $\text{prod} := \text{prod} + a[i] * b[i];$
6. $i := i + 1;$
7. end
8. while $i \leq 10$
9. end

The three address B2 contains the statement (3) to (12)

Optimization of Basic Blocks:

- ❖ Optimization process can be applied on a basic block. While optimization, we don't need to change the set of expressions computed by the block.
- ❖ There are two type of basic block optimization. These are as follows:
 1. Structure-Preserving Transformations
 2. Algebraic Transformations

Structure preserving transformations:

The primary Structure-Preserving Transformation on basic blocks is as follows:

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

(a) Common sub-expression elimination:

- In the common sub-expression, you don't need to be computed it over and over again. Instead of this you can compute it once and kept in store from where it's referenced when encountered again.

$a := b + c$

$b := a - d$

$c := b + c$

$d := a - d$

- In the above expression, the second and forth expression computed the same expression. So the block can be transformed as follows:

$a := b + c$

$b := a - d$

$c := b + c$

$d := b$

(b) Dead-code elimination

- It is possible that a program contains a large amount of dead code.
- This can be caused when once declared and defined once and forget to remove them in this case they serve no purpose.
- Suppose the statement $x := y + z$ appears in a block and x is dead symbol that means it will never subsequently used. Then without changing the value of the basic block you can safely remove this statement.

(c) Renaming temporary variables

- A statement $t := b + c$ can be changed to $u := b + c$ where t is a temporary variable and u is a new temporary variable. All the instance of t can be replaced with the u without changing the basic block value.

(d) Interchange of statement

- Suppose a block has the following two adjacent statements:
 $t1 := b + c$
 $t2 := x + y$
- These two statements can be interchanged without affecting the value of block when value of $t1$ does not affect the value of $t2$.

2. Algebraic transformations:

- In the algebraic transformation, we can change the set of expression into an algebraically equivalent set. Thus the expression $x := x + 0$ or $x := x * 1$ can be eliminated from a basic block without changing the set of expression.
- Constant folding is a class of related optimization. Here at compile time, we evaluate constant expressions and replace the constant expression by their values. Thus the expression $5 * 2.7$ would be replaced by 13.5 .
- Sometimes the unexpected common sub expression is generated by the relational operators like $<=$, $>=$, $<$, $>$, $+$, $=$ etc.
- Sometimes associative expression is applied to expose common sub expression without changing the basic block value. if the source code has the assignments

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

The following intermediate code may be generated:

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

CHAPTER 10

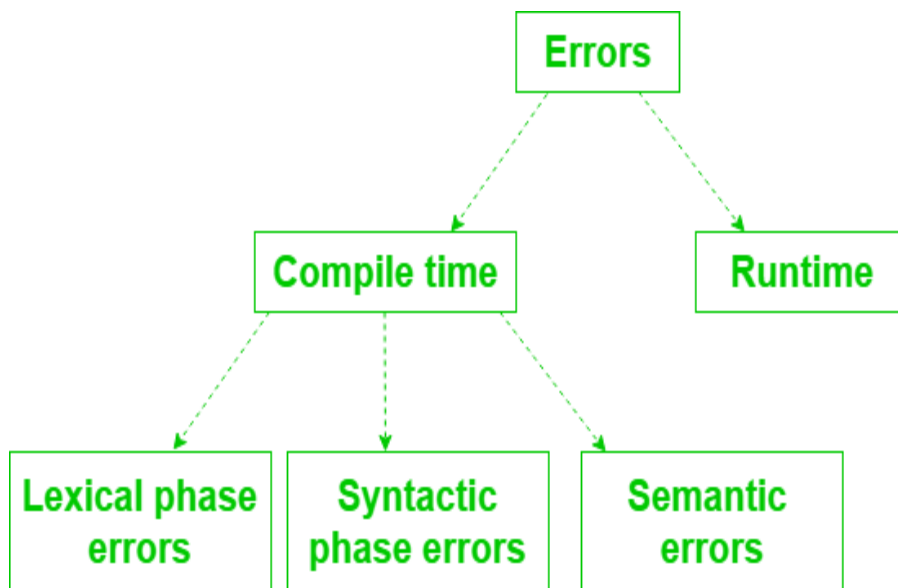
Errors :

- In this phase of compilation, all possible errors made by the user are detected and reported to the user in form of error messages. This process of locating errors and reporting it to user is called **Error Handling process**.

Functions of Error handler

- Detection
- Reporting
- Recovery

Classification of Errors



Lexical PhaseError

- ❖ During the lexical analysis phase this type of error can be detected.
- ❖ Lexical error is a sequence of characters that does not match the pattern of any token.

Lexical phase error is found during the execution of the program.

Lexical phase error can be:

- Spelling error.
- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters.
- To remove the character that should be present.
- To replace a character with an incorrect character.
- Transposition of two characters.

Example:

```
Void main()
{
    int x=10, y=20;
    char * a;
    a= &x;
    x= 1xab;
}
```

- ❖ In this code, 1xab is neither a number nor an identifier. So this code will show the lexical error.

Error recovery:

Panic Mode Recovery

- In this method, successive characters from the input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as; or }
- Advantage is that it is easy to implement and guarantees not to go to infinite loop
- Disadvantage is that a considerable amount of input is skipped without checking it for additional error

SyntaxPhase Error

- ❖ During the syntax analysis phase, this type of error appears. Syntax error is found during the execution of the program.
- ❖ Some syntax error can be:
 - Error in structure
 - Missing operators
 - Unbalanced parenthesis

- ❖ When an invalid calculation enters into a calculator then a syntax error can also occurs. This can be caused by entering several decimal points in one number or by opening brackets without closing them.

For example 1: Using "=" when "==" is needed.

```
if (number=200)
    count << "number is equal to 20";
else
    count << "number is not equal to 200"
```

The following warning message will be displayed by many compilers:

- ❖ **Syntax Warning:** assignment operator used in if expression line 16 of program firstprog.cpp
- ❖ In this code, if expression used the equal sign which is actually an assignment operator not the relational operator which tests for equality.
- ❖ Due to the assignment operator, number is set to 200 and the expression number=200 are always true because the expression's value is actually 200. For this example the correct code would be:

```
16 if (number==200)
```

Example 2: Missing semicolon:

```
int a = 5      // semicolon is missing
```

Compiler message:

```
ab.java:20: ';' expected
int a = 5
```

Example 3: Errors in expressions:

```
x = (3 + 5; // missing closing parenthesis ')'
y = 3 + * 5; // missing argument between '+' and '*'
```

Error recovery:

1. Panic Mode Recovery

- In this method, successive characters from input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as ; or }
- Advantage is that it's easy to implement and guarantees not to go to infinite loop
- Disadvantage is that a considerable amount of input is skipped without checking it for additional errors

2. Statement Mode recovery

- In this method, when a parser encounters an error, it performs necessary correction on remaining input so that the rest of input statement allows the parser to parse ahead.
- The correction can be deletion of extra semicolons, replacing comma by semicolon or inserting missing semicolon.
- While performing correction, utmost care should be taken for not going in infinite loop.
- Disadvantage is that it finds difficult to handle situations where actual error occurred before point of detection.

3. Error production

- If user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.
- If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
- Disadvantage is that it's difficult to maintain.

4. Global Correction

- The parser examines the whole program and tries to find out the closest match for it which is error free.
 - The closest match program has less number of insertions, deletions and changes of tokens to recover from erroneous input.
 - Due to high time and space complexity, this method is not implemented practically.
-

Semantic Error

- ❖ During the semantic analysis phase, this type of error appears. These types of error are detected at compile time.
- ❖ Most of the compile time errors are scope and declaration error. **For example:** undeclared or multiple declared identifiers. Type mismatched is another compile time error.
- ❖ The semantic error can arise using the wrong variable or using wrong operator or doing operation in wrong order.

Some semantic error can be:

- Incompatible types of operands
- Undeclared variable
- Not matching of actual argument with formal argument

Example 1: Use of a non-initialized variable:

```
int i;  
void f (int m)  
{  
    m=t;  
}
```

In this code, t is undeclared that's why it shows the semantic error.

Example 2: Type incompatibility:

```
int a = "hello";    // the types String and int are not compatible
```

Example 3: Errors in expressions:

```
String s = "...";  
int a = 5 - s;    // the - operator does not support arguments of type String
```

Error recovery

- If error “**Undeclared Identifier**” is encountered then, to recover from this a symbol table entry for corresponding identifier is made.
- If data types of two operands are incompatible then, automatic type conversion is done by the compiler.
- Attention reader! Don't stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the [CS Theory Course](#) at a student-friendly price and become industry ready.

UNIT V

CHAPTER 11:- Introduction To Code Optimization

- ❖ The principal sources of optimization
- ❖ Loop optimization
- ❖ The dag representation of basic blocks

CHAPTER 12:- Code Generation

- ❖ Object programs
 - ❖ Problems in code generation
 - ❖ A machine model
 - ❖ A simple code generator
 - ❖ Register allocation and assignment
 - ❖ Code generation from DAG's
 - ❖
 - ❖ Peephole optimization
-

CHAPTER 11:-

PRINCIPAL SOURCES OF OPTIMIZATION

- ❖ 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

The other transformations come up primarily when global optimizations are performed.

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

Common Sub expressions elimination:

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

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

Example:

```
i=0;
```

```
if(i=1)
```

```
{
```

```
  a=b+5;
```

```
}
```

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

Constant folding:

- ❖ Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

- $a=3.14157/2$ can be replaced by
- $a=1.570$ there by eliminating a division operation.

Loop Optimizations:

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

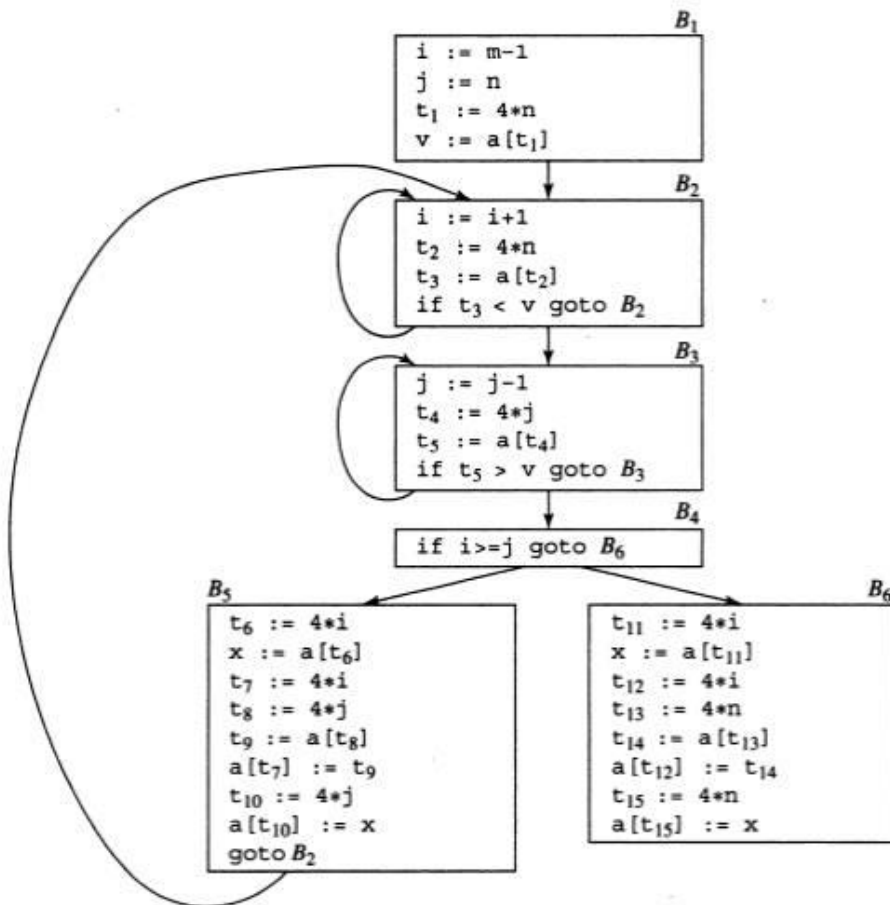


Fig. 5.2 Flow graph

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 $\text{limit}-2$ is a loop-invariant computation in the following while-statement:

- `while (i <= limit-2) /* statement does not change limit*/`

- ❖ Code motion will result in the equivalent of

- `t = limit-2;`

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

Induction Variables :

- ❖ Loops are usually processed inside out. For example consider the loop around B3. Note that the values of j and $t4$ 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 in Fig.5.3 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.

Example:

- ❖ As the relationship $t4 := 4*j$ surely holds after such an assignment to $t4$ in Fig. and $t4$ is not changed elsewhere in the inner loop around B3, it follows that just after the statement $j := j-1$ the relationship $t4 := 4*j-4$ must hold. We may therefore replace the assignment $t4 := 4*j$ by $t4 := t4-4$. The only problem is that $t4$ does not have a value when we enter

block B3 for the first time. Since we must maintain the relationship $t_4=4*j$ on entry to the block B3, we place an initialization of t_4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in Fig.5.3.

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

Reduction In Strength:

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

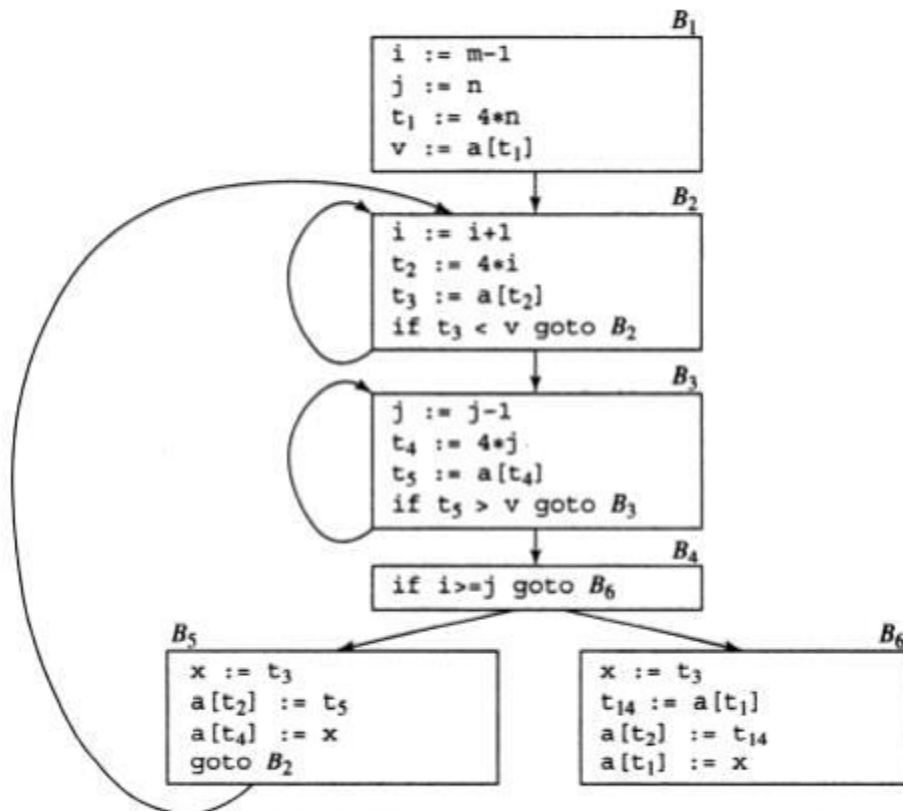


Fig. 5.3 B5 and B6 after common subexpression elimination

Fig. 5.3 B5 and B6 after common subexpression elimination

Loop Optimization

- ❖ Loop optimization is most valuable machine-independent optimization because program's inner loop takes bulk to time of a programmer.
- ❖ If we decrease the number of instructions in an inner loop then the running time of a program may be improved even if we increase the amount of code outside that loop.
- ❖ For loop optimization the following three techniques are important:
 1. Code motion
 2. Induction-variable elimination
 3. Strength reduction

1.Code Motion:

- ❖ Code motion is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

For example

In the while statement, the limit-2 equation is a loop invariant equation.

```
while (i<=limit-2)  /*statement does not change limit*/
```

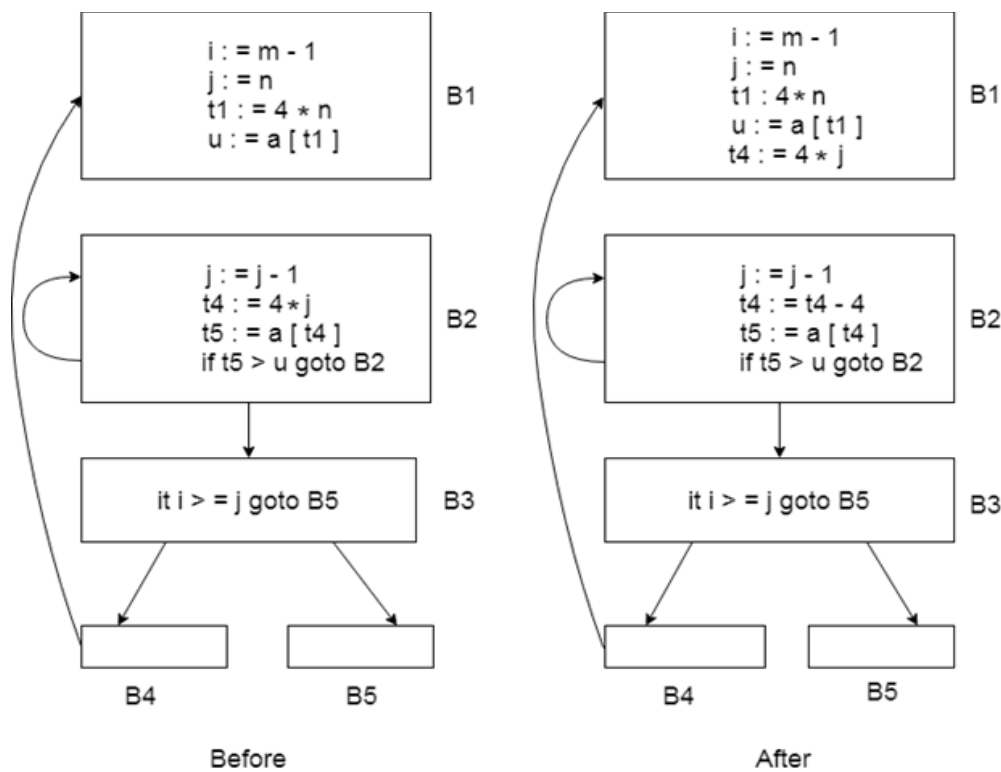
After code motion the result is as follows:

```
a= limit-2;
```

```
while(i<=a)  /*statement does not change limit or a*/
```

2.Induction-Variable Elimination

- ❖ Induction variable elimination is used to replace variable from inner loop.
- ❖ It can reduce the number of additions in a loop. It improves both code space and run time performance.



- ❖ In this figure, we can replace the assignment $t4 := 4*j$ by $t4 := t4 - 4$. The only problem which will be arose that $t4$ does not have a value when we enter block B2 for the first time. So we place a relation $t4 = 4*j$ on entry to the block B2.

3.Reduction in Strength

- ❖ Strength reduction is used to replace the expensive operation by the cheaper once on the target machine.
- ❖ Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.
- ❖ Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

Example:

1. **while** ($i < 10$)
2. {
3. $j = 3 * i + 1$;
4. $a[j] = a[j] - 2$;
5. $i = i + 2$;
6. }
7. After strength reduction the code will be:
8. $s = 3 * i + 1$;
9. **while** ($i < 10$)
10. {

```

11.      j=s;
12.      a[j]= a[j]-2;
13.      i=i+2;
14.      s=s+6;
15.      }

```

In the above code, it is cheaper to compute $s=s+6$ than $j=3 * i$

DAG representation for basic blocks

A DAG for basic block is a directed acyclic graph with the following labels on nodes:

1. The leaves of graph are labeled by unique identifier and that identifier can be variable names or constants.
 2. Interior nodes of the graph is labeled by an operator symbol.
 3. Nodes are also given a sequence of identifiers for labels to store the computed value.
- DAGs are a type of data structure. It is used to implement transformations on basic blocks.
 - DAG provides a good way to determine the common sub-expression.
 - It gives a picture representation of how the value computed by the statement is used in subsequent statements.

Algorithm for construction of DAG

Input: It contains a basic block

Output: It contains the following information:

- Each node contains a label. For leaves, the label is an identifier.
- Each node contains a list of attached identifiers to hold the computed values.

Case (i) $x := y \text{ OP } z$

Case (ii) $x := \text{OP } y$

Case (iii) $x := y$

Method:

Step 1:

- If y operand is undefined then create node(y).
- If z operand is undefined then for case(i) create node(z).

Step 2:

- For case(i), create node(OP) whose right child is node(z) and left child is node(y).
- For case(ii), check whether there is node(OP) with one child node(y).

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

Output:

- For node(x) delete x from the list of identifiers. Append x to attached identifiers list for the node n found in step 2. Finally set node(x) to n.

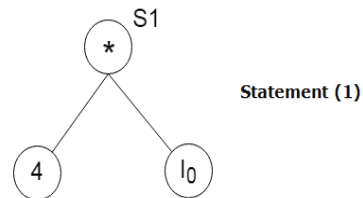
Example:

Consider the following three address statement:

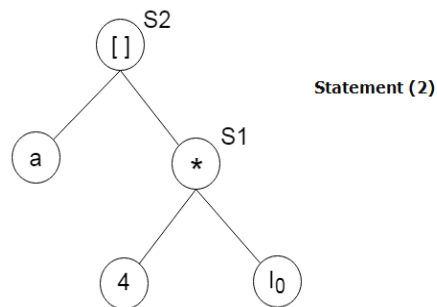
1. $S1 := 4 * i$
2. $S2 := a[S1]$
3. $S3 := 4 * i$
4. $S4 := b[S3]$
5. $S5 := s2 * S4$
6. $S6 := \text{prod} + S5$
7. $\text{Prod} := s6$
8. $S7 := i + 1$
9. $i := S7$
10. **if** $i \leq 20$ **goto** (1)

Stages in DAG Construction:

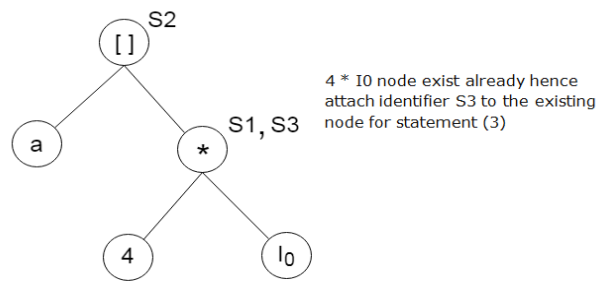
(a)



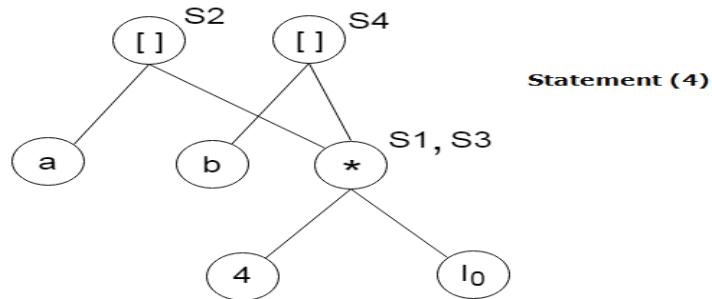
(b)



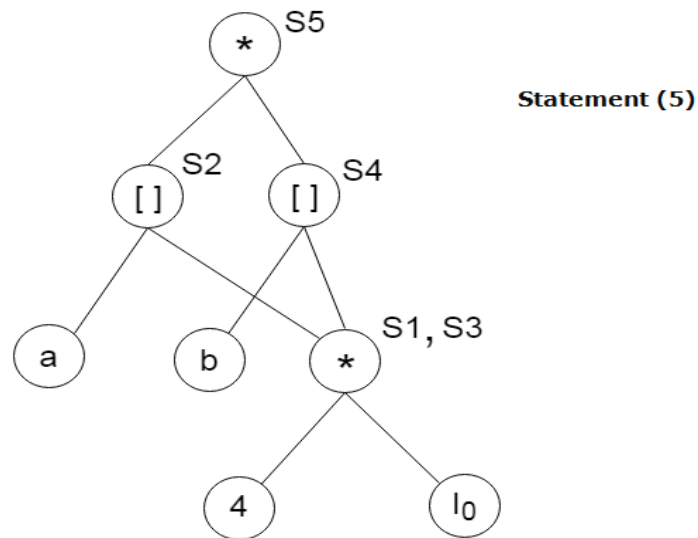
(c)



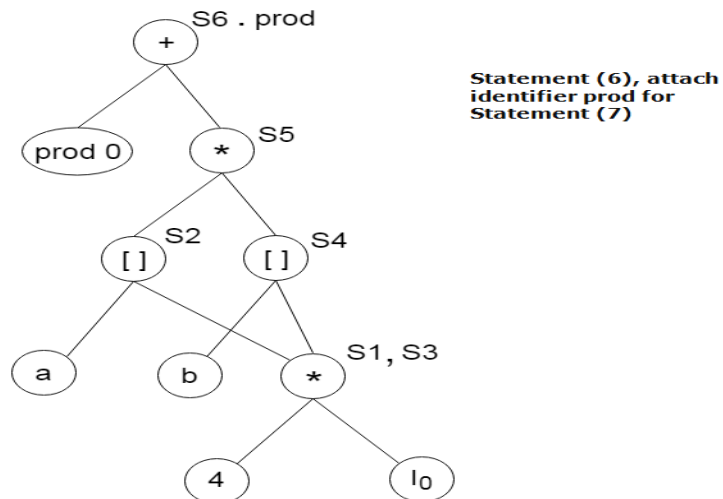
(d)



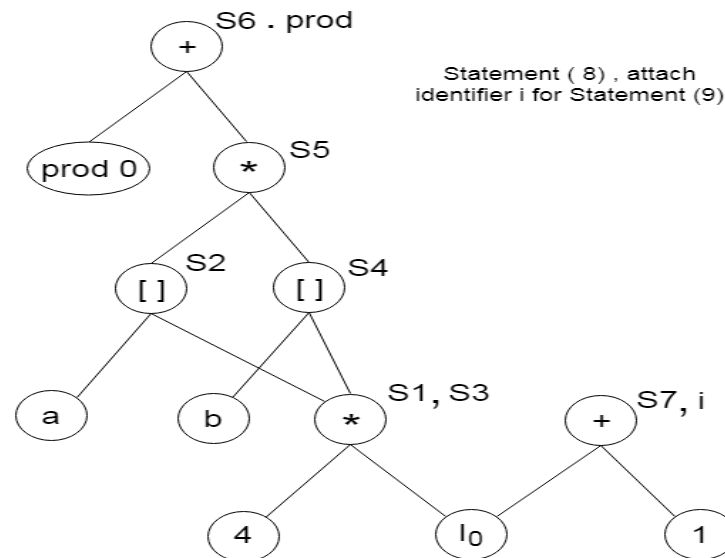
(e)



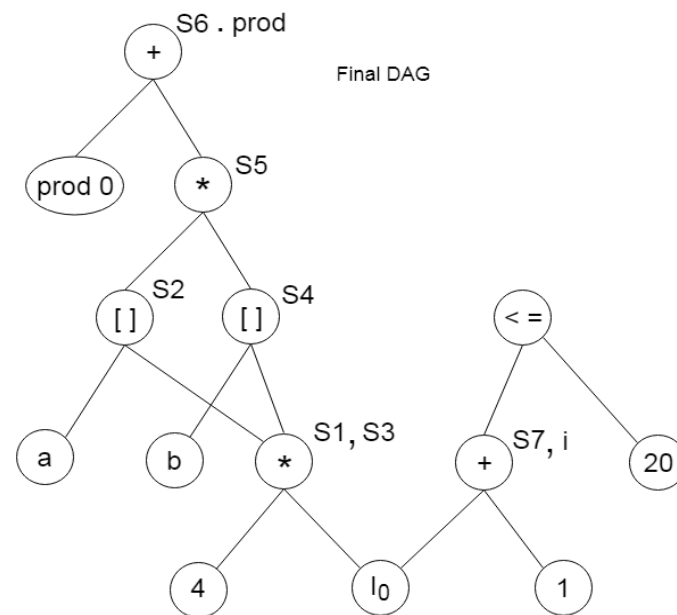
(f)



(g)



(h)



Global data flow analysis

- ❖ To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.
- ❖ Certain optimization can only be achieved by examining the entire program. It can't be achieved by examining just a portion of the program.
- ❖ For this kind of optimization user defined chaining is one particular problem.
- ❖ Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

- ❖ Based on the local information a compiler can perform some optimizations. For example, consider the following code:

```
x = a + b;
```

```
x = 6 * 3
```

- In this code, the first assignment of x is useless. The value computer for x is never used in the program.
- At compile time the expression 6*3 will be computed, simplifying the second assignment statement to x = 18;
- Some optimization needs more global information. For example, consider the following code:

```
▪ a = 1;  
▪ b = 2;  
▪ c = 3;  
▪ if (...) x = a + 5;  
▪ else x = b + 4;  
▪ c = x + 1;
```

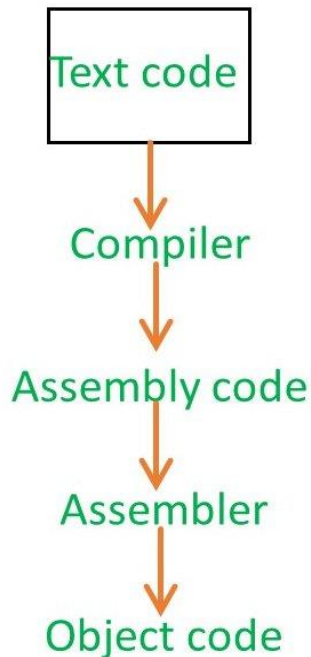
In this code, at line 3 the initial assignment is useless and x + 1 expression can be simplified as 7.

- ❖ But it is less obvious that how a compiler can discover these facts by looking only at one or two consecutive statements. A more global analysis is required so that the compiler knows the following things at each point in the program:
 - Which variables are guaranteed to have constant values
 - Which variables will be used before being redefined
 - ❖ Data flow analysis is used to discover this kind of property. The data flow analysis can be performed on the program's control flow graph (CFG).
 - ❖ The control flow graph of a program is used to determine those parts of a program to which a particular value assigned to a variable might propagate.
-

CHAPTER 12

Object programs

- Let assume that, you have a c program, then you give the C program to compiler and compiler will produce the output in assembly code. Now, that assembly language code will give to the assembler and assembler is going to produce you some code. That is known as **Object Code**.



- But, when you compile a program, then you are not going to use both compiler and assembler. You just take the program and give it to the compiler and compiler will give you the directly executable code. The compiler is actually combined inside the assembler along with loader and linker. So all the module kept together in the compiler software itself. So when you calling gcc, you are actually not just calling the compiler, you are calling the compiler, then assembler, then linker and loader.
- Once you call the compiler, then your object code is going to present in Hard-disk. This object code contains various part –

Header
Text segment
Data segment
Relocation information
Symbol table
Debugging information

OBJECT CODE

Problems In Code Generation

The following issues arise during the code generation phase:

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

1. Input to code generator:

- ❖ The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
- ❖ Intermediate 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.
- e. • 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.

2. Target program:

- The output of the code generator is the target program. The output may be : a. Absolute machine language
- It can be placed in a fixed memory location and can be executed immediately- b. Relocatable machine language
- .
- It allows subprograms to be compiled separately.c. Assembly language
- Code generation is made easier.

3. Memory management:

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

- if $i > j$, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j . When i is processed, the machine locations for all instructions that forward jumps to i are filled.

4. Instruction selection:

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

- $a := b + c$
- $d := a + e$ (a)
- MOV b,R0
- ADD c,R0
- MOV R0,a (b)
- MOV a,R0
- ADD e,R0
- MOV R0,d

5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory. The use of registers is subdivided into two subproblems :
- Register allocation - the set of variables that will reside in registers at a point in the program is selected.
- Register assignment - the specific register that a value picked•
- Certain machine requires even-odd register pairs for some operands and results.
For example , consider the division instruction of the form :D x, y

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

6. Evaluation order

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

Machine model

- The target computer is a type of byte-addressable machine. It has 4 bytes to a word.
- The target machine has n general purpose registers, R0, R1, ..., Rn-1. It also has two-address instructions of the form:

op source, destination

Where, op is used as an op-code and source and destination are used as a data field.

- It has the following op-codes:

ADD (add source to destination)

SUB (subtract source from destination)

MOV (move source to destination)

- The source and destination of an instruction can be specified by the combination of registers and memory location with address modes.

MODE	FORM	ADDRESS	EXAMPLE	ADDED COST
absolute	M	M	Add R0, R1	1
register	R	R	Add temp, R1	0
indexed	c(R)	C+ contents(R)	ADD 100 (R2), R1	1
indirect register	*R	contents(R)	ADD * 100	0

indirect indexed	*c(R)	contents(c+ contents(R))	(R2), R1	1
literal	#c	c	ADD #3, R1	1

- Here, cost 1 means that it occupies only one word of memory.
- Each instruction has a cost of 1 plus added costs for the source and destination.
- Instruction cost = 1 + cost is used for source and destination mode.

Example:

1. Move register to memory $R0 \rightarrow M$

2. Indirect indexed mode:

MOV *4(R0), M

cost = $1+1+1$ (since one word **for** memory location M, one word **for** result of *4(R0) and one **for** instruction)

3. Literal Mode:

MOV #1, R0

cost = $1+1+1 = 3$ (one word **for** constant 1 and one **for** instruction)

A Simple Code Generator

- ❖ Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Example:

- ❖ Consider the three address statement $x := y + z$. It can have the following sequence of codes:

MOV x, R₀

ADD y, R₀

Register and Address Descriptors:

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

A code-generation algorithm:

The algorithm takes a sequence of three-address statements as input. For each three address statement of the form $a := b \text{ op } c$ perform the various actions. These are as follows:

1. Invoke a function `getreg` to find out the location L where the result of computation $b \text{ op } c$ should be stored.
2. Consult the address description for y to determine y' . If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction **MOV y' , L** to place a copy of y in L .
3. Generate the instruction **OP z' , L** where z' is used to show the current location of z . if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L . If x is in L then update its descriptor and remove x from all other descriptor.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of $x := y \text{ op } z$ those register will no longer contain y or z .

Generating Code for Assignment Statements:

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

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

Code sequence for the example is as follows:

Statement	Code Generated	Register descriptor Register empty	Address descriptor
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 R1
d := v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

REGISTER ALLOCATION AND ASSIGNMENT

Local register allocation

- ❖ Register allocation is only within a basic block. It follows top-down approach.
 - ❖ Assign registers to the most heavily used variables
- Traverse the block

- Count uses
- Use count as a priority function
- Assign registers to higher priority variables first

Advantage

- Heavily used values reside in registers

Disadvantage

- Does not consider non-uniform distribution of uses

Need of global register allocation

- Local allocation does not take into account that some instructions (e.g. those in loops) execute more frequently. It forces us to store/load at basic block endpoints since each block has no knowledge of the context of others.
- To find out the live range(s) of each variable and the area(s) where the variable is used/defined global allocation is needed. Cost of spilling will depend on frequencies and locations of uses.
- Register allocation depends on:
 - ❖ Size of live range
 - ❖ Number of uses/definitions
 - ❖ Frequency of execution
 - ❖ Number of loads/stores needed.
 - ❖ Cost of loads/stores needed.

Register allocation by graph coloring

- Global register allocation can be seen as a graph coloring problem.

Basic idea:

- Identify the live range of each variable
- Build an interference graph that represents conflicts between live ranges (two nodes are connected if the variables they represent are live at the same moment)
- Try to assign as many colors to the nodes of the graph as there are registers so that two neighbors have different colors

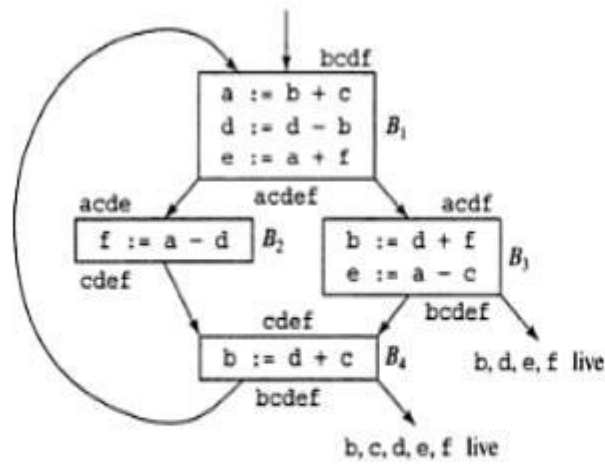


Fig 4.3 Flow graph of an inner loop

Fig 4.3 Flow graph of an inner loop

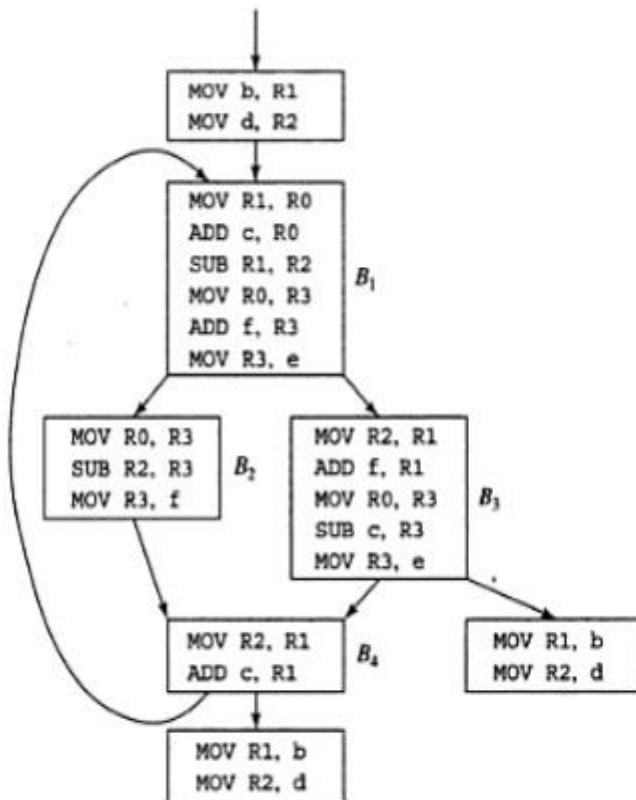


Fig 4.4 Code sequence using global register assignment

Fig 4.4 Code sequence using global register assignment

Register allocation And Assignment

- ❖ Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:

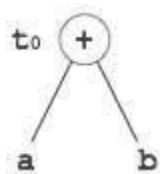
- ❖ **Register allocation:** In register allocation, we select the set of variables that will reside in register.
- ❖ **Register assignment:** In Register assignment, we pick the register that contains variable.

Code Generation From Directed Acyclic Graph

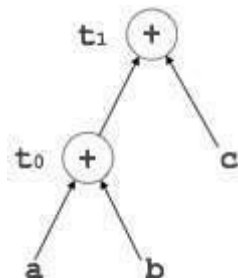
- Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

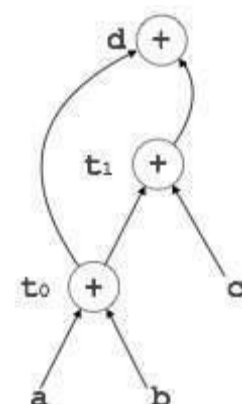
$t_0 = a + b$
 $t_1 = t_0 + c$
 $d = t_0 + t_1$



$[t_0 = a + b]$



$[t_1 = t_0 + c]$



$[d = t_0 + t_1]$

Peephole Optimization

- ❖ This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination

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

<pre>intadd_ten(int x) { int y, z; y =10; z = x + y; return z; }</pre>	<pre>intadd_ten(int x) { int y; y =10; y = x + y; return y; }</pre>	<pre>intadd_ten(int x) { int y =10; return x + y; }</pre>	<pre>intadd_ten(int x) { return x +10; }</pre>
------------------------------------------------------------------------	---------------------------------------------------------------------	-----------------------------------------------------------	------------------------------------------------

- ❖ 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

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

Example:

<pre>voidadd_ten(int x) { return x +10; printf("valueof x is%d", x); }</pre>

- ❖ 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.

Flow of control optimization

- ❖ 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
```

- ❖ In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...  
MOV R1, R2  
GOTO L2  
...  
L2 : INC R1
```

Algebraic expression simplification

- ❖ There are occasions where algebraic expressions can be made simple. For example, the expression $a = a + 0$ can be replaced by a itself and the expression $a = a + 1$ can simply be replaced by $\text{INC } a$.

Strength reduction

- ❖ There are operations that consume more time and space. Their ‘strength’ can be reduced by replacing them with other operations that consume less time and space, but produce the same result.
- ❖ For example, $x * 2$ can be replaced by $x \ll 1$, which involves only one left shift. Though the output of $a * a$ and a^2 is same, a^2 is much more efficient to implement.